GURUNANAK INSTITUTE OF TECHNOLOGY

157/F, Nilgunj Road, Panihati

Kolkata -700114

Website: www.gnit.ac.in

Email: info.gnit@jisgroup.org

Approved by A.I.C.T.E., New Delhi

Affiliated to MAKAUT, West Bengal

ONLINE COURSE WARE

Paper Name: Data Structure and Algorithm

Paper Code: FT 605 A

Credit: 3

Module I

Linear Data structures

Array and Linked List

Lecture 1

Introduction:

Data, information, knowledge and wisdom:

Data, information, knowledge and wisdom are closely related concepts, but each has its own role in relation to the other, and each term has its own meaning. Data is collected and analyzed; data only becomes information suitable for making decisions once it has been analyzed in some fashion. Knowledge is derived from extensive amounts of experience dealing with information on a subject. For example, the height of Mount Everest is generally considered data. The height can be recorded precisely with an altimeter and entered into a database. This data may be included in a book along with other data on Mount Everest to describe the mountain in a manner useful for those who wish to make a decision about the best method to climb it. Using an understanding based on experience climbing mountains to advise persons on the way to reach Mount Everest's peak may be seen as "knowledge". Some complement the series "data", "information" and "knowledge" with "wisdom", which would mean the status of a person in possession of certain "knowledge" who also knows under which circumstances is good to use it.
Data is the least abstract concept, information the next least, and knowledge the most abstract. Data becomes information by interpretation; e.g., the height of Mount Everest is generally considered "data", a book on Mount Everest geological characteristics may be considered "information", and a climber's guidebook containing practical information on the best way to reach Mount Everest's peak may be considered "knowledge".

Data can be organized in many ways. Data structure is the logical and mathematical model of a particular organization of data. It is used to represent data in the memory of the computer so that the processing of data can be done in easier way

The data structures can be of the following types:

1. Linear Data structures

2. Non-Linear Data Structures

1. Linear Data structures- In these data structures the elements form a sequence. Such as Arrays, Linked Lists Stacks and Queues are linear data structures.

2. Non-Linear Data Structures- In these data structures the elements do not form a sequence. Such as Trees and Graphs are non-linear data structures.

Data Type

Data type is a way to classify various types of data such as integer, string, etc. which determines the values that can be used with the corresponding type of data, the type of operations that can be performed on the corresponding type of data. There are two data types −

- Built-in Data Type
- Derived Data Type

Built-in Data Type

Those data types for which a language has built-in support are known as Built-in Data types. For example, most of the languages provide the following built-in data types.

- Integers
- Boolean (true, false)
- Floating (Decimal numbers)
- Character and Strings

Derived Data Type

Those data types which are implementation independent as they can be implemented in one or the other way are known as derived data types. These data types are normally built by the combination of primary or built-in data types and associated operations on them. For example −

- List
- Array
- Stack
- Queue

Basic Operations

The data in the data structures are processed by certain operations. The particular data structure chosen largely depends on the frequency of the operation that needs to be performed on the data structure.

- Traversing
- Searching
- Insertion
- Deletion
- Sorting
- Merging

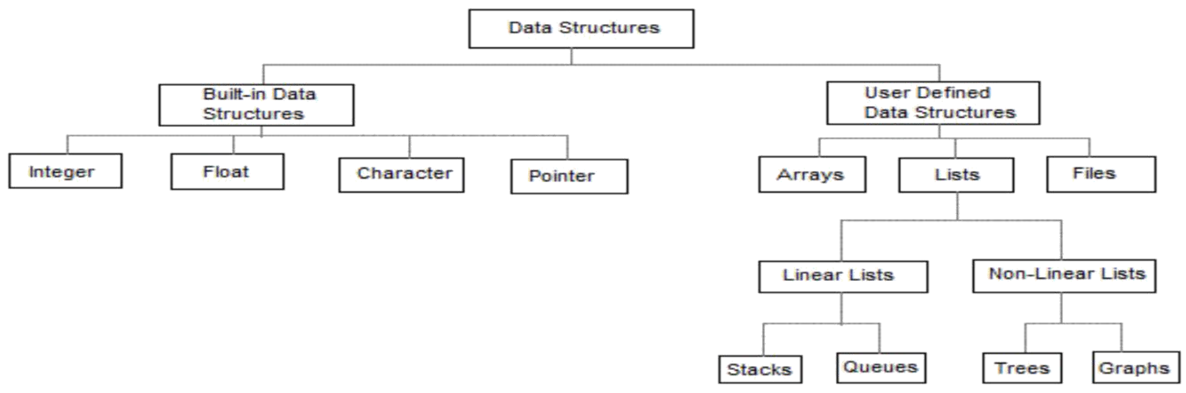
Basic types of Data Structures

As we discussed above, anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as **Primitive Data Structures**.

Then we also have some complex Data Structures, which are used to store large and connected data. Some examples of **Abstract Data Structure** are:

- Linked List
- Tree
- Graph
- Stack, Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required. We will look into these data structures in more details in our later lessons.



INTRODUCTION TO DATA STRUCTURES

Definition of ADT

An ADT is a mathematical model of a data structure that specifies the type of data stored, the operations supported on them, and the types of parameters of the operations. An ADT specifies what each operation does, but not how it does it. Typically, an ADT can be implemented using one of many different data structures. A useful first step in deciding what data structure to use in a program is to specify an ADT for the program.

In general, the steps of building ADT to data structures are:

1. Understand and clarify the nature of the target information unit.

2. Identify and determine which data objects and operations to include in the models.

3. Express this property somewhat formally so that it can be understood and communicate well.

4. Translate this formal specification into proper language. In C++, this becomes a .h file. In Java, this is called "user interface".

5. Upon finalized specification, write necessary implementation. This includes storage scheme and operational detail. Operational detail is expressed as separate functions (methods).

Data Structures: ADT and its Implementation

Now let us consider two alternate data structures for the above ADT: (1) an unordered array of records and (2) an ordered array of records, ordered by IDs. These different data structures greatly influence the implementation details and how fast and efficient the program runs.

First let us look at using an unordered array. Assume that the student records are stored in an array with no particular order. We can use a variable n to keep track of the number of students currently in the array.

- ADD: Simply take the record and store it in slot n of the array and increment n. This takes constant amount of time since the time is independent of n.

- SEARCH: Since the array is not ordered, we have to scan through the whole array to find the requested record. The result could be either the record is found or the record doesn't exist. The time taken to perform the search is proportional to n and the worst case scenario is n. Of course, if the record I am looking for happens to be the fist item in the array, it only takes constant time. However, when determining the running time of an algorithm, we are often interested in worst case analysis.

- DELETE: This operation requires us to first search for the given record. Once it is found, the algorithm can simply replace it by the last record and decrement n. Once the record is found, it only takes a constant amount of time to delete it. But time spent searching for the record is the same as above, proportional to n. Therefore, in all, this operation also takes time proportional to n, in the worst case.

Now let us consider using an ordered array. Assume that the student records are sorted in an array, with an increasing order of student IDs. We can also use a variable n to keep track of the number of students currently in the array.

- ADD: Since the array is sorted, we first need to find out where should we insert the record. This can be done by scanning through the array, comparing the current record in the array with the record we want to insert, and finding the smallest index i of the record whose ID is larger than the new ID. Then the new record should be put to the ith slot in the array. To do that, all the records in slots i, i+1, i+2, ..., n need to be moved down one slot to create an empty slot for the new record. We can then put the new record into the ith slot and increment n. This operation takes time proportional to n.

- SEARCH: Since the array of records is sorted by IDs, we can use a binary search to find the given record. In general, a binary search algorithm first compares the given ID with the ID in the middle of the array (with index n/2 or (n+1)/2). Then the algorithm branches based on different conditions: if the two IDs are the same, we find the record; if the given ID is smaller, the algorithm continues to search the first half of the current array, ignoring the second half; otherwise, the algorithm continues to search the second half of the current array, ignoring the first half. The operation time is log2n. Finding the time for binary search will be discussed in the future.

- DELETE: This operation requires us to first search for the given record. This can be done in log2n time as shown above. Since the array is ordered, we need to fill in the empty slot i with a record, which means the records in slots i+1, i+2, ..., n need to be moved up one slot to cover the empty one. The time is also proportional to n. In all, this operation takes time proportional to n, in the worst case.

Lecture 2

Algorithm:

An Algorithm is a finite set of instructions that, if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

1. Input. Zero or more quantities are externally supplied.

2. Output. At least one quantity is produced.
3. Definiteness. Each instruction is clear and unambiguous.
4. Finiteness. If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. Effectiveness. Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in criterion 3; it also must be feasible.

Algorithm is not the complete code or program, it is just the core logic (solution) of a problem, which can be expressed either as an informal high level description as **pseudocode** or using a **flowchart**.

An algorithm is said to be efficient and fast, if it takes less time to execute and consumes less memory space. The performance of an algorithm is measured on the basis of following properties:

1. Time Complexity
2. Space Complexity

Space Complexity

Its the amount of memory space required by the algorithm, during the course of its execution. Space complexity must be taken seriously for multi-user systems and in situations where limited memory is available.

Total memory space needed by the program is the sum of following two memories:

(1) Fixed size Memory: It contains the space required for simple variables, constants, instructions and fixed size structured variable such as array.

(2) Variable size Memory: It contains the space required for structured variable to which memory is allocated run time. It also contains space required while function is calling itself.

An algorithm generally requires space for following components:

- **Instruction Space:** It's the space required to store the executable version of the program. This space is fixed, but varies depending upon the number of lines of code in the program.
- **Data Space:** It's the space required to store all the constants and variables value.
- **Environment Space:** It's the space required to store the environment information needed to resume the suspended function.

Time Complexity

The amount of time needed by a program to complete its execution is known as Time complexity.

The measurement of time is done in terms of number of instructions executed by the program during its execution.

Thus Time Complexity depends on the Size of the program and type of the algorithm being used.

Best case Time Complexity

The measurement of minimum time that is required by an algorithm to complete its execution is known as Best Case Time Complexity.

Time complexity of particular algorithm can be calculated by providing different input values to the algorithm.

Consider an example of sorting N elements. If we supply input values that is already sorted, an algorithm required less time to sort them. This is known as Best case time complexity.

However best case time complexity does not guarantee that the algorithm always execute within this time for different input values.

Average case Time Complexity

The measurement of average time that is required by an algorithm to complete its execution is known as Average Case Time Complexity.

Time complexity of particular algorithm can be calculated by providing different input values to the algorithm.

Consider an example of sorting N elements. Average time complexity can be calculated by measuring the time required to complete the execution of an algorithm for different input values and then calculate the average time required to sort N elements.

Worst case Time Complexity

The measurement of maximum time that is required by an algorithm to complete its execution is known as Worst Case Time Complexity.

Time complexity of particular algorithm can be calculated by providing different input values to the algorithm.

Consider an example of sorting N elements. If we supply input values that is in reverse order, an algorithm required maximum time to sort them. This is known as worst case time complexity.

Thus, worst case time complexity always guarantees that the algorithm will always execute within this time for different input values.

Asymptotic Notations

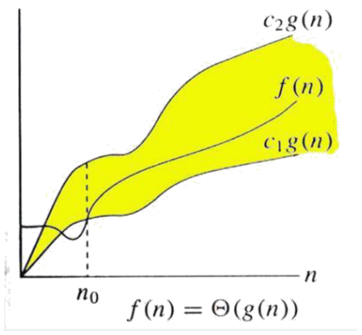
Asymptotic Notations are used to describe the complexity of an algorithm. Complexity of an algorithm indicates how much time needed by an algorithm to complete its execution for given set of input data.

The same problem can be solved using different algorithms. In order to select the best algorithm for a problem, we need to determine how much time the different algorithm will take to run and then select the better algorithm.

There are various Asymptotic Notations are available to describe complexity of an algorithm. Which are

1. Big-O Notation
2. Big-Omega Notation
3. Big-Theta Notation
4. Little-o Notation
5. Little-omega Notation

We have discussed Asymptotic Analysis, and Worst, Average and Best Cases of Algorithms. The main idea of asymptotic analysis is to have a measure of efficiency of algorithms that doesn't depend on machine specific constants, and doesn't require algorithms to be implemented and time taken by programs to be compared. Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis. The following 3 asymptotic notations are mostly used to represent time complexity of algorithms.



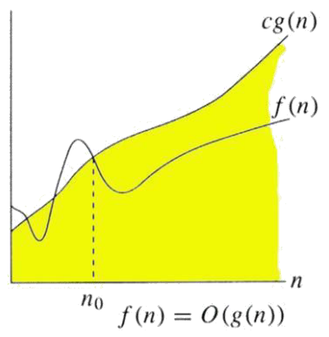
1) Θ Notation: The theta notation bounds functions from above and below, so it defines exact asymptotic behavior. A simple way to get Theta notation of an expression is to drop low order terms and

ignore leading constants. For example, consider the following expression. 3n3 + 6n2 + 6000 = Θ(n3)

Dropping lower order terms is always fine because there will always be a n0 after which Θ(n3) has higher values than Θn2) irrespective of the constants involved. For a given function g(n), we denote Θ(g(n)) is following set of functions.

Θ(g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that
0 <= c1*g(n) <= f(n) <= c2*g(n) for all n >= n0}

The above definition means, if f(n) is theta of g(n), then the value f(n) is always between c1*g(n) and c2*g(n) for large values of n (n >= n0). The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.



2) Big O Notation: The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time.

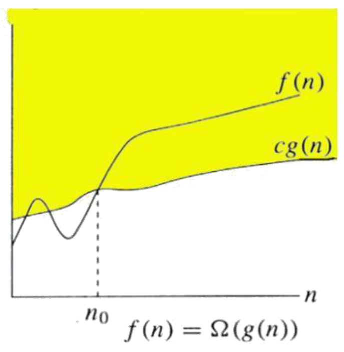
If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:

1. The worst case time complexity of Insertion Sort is Θ(n^2).

2. The best case time complexity of Insertion Sort is Θ(n).

The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

O(g(n)) = { f(n): there exist positive constants c
and n0 such that 0 <= f(n) <= cg(n) for
all n >= n0}



3) Ω Notation: Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.

Ω Notation< can be useful when we have lower bound on time complexity of an algorithm. As discussed in the previous post, the best case performance of an algorithm is generally not useful, the Omega notation is the least used notation among all three.

For a given function g(n), we denote by Ω(g(n)) the set of functions.

Ω (g(n)) = {f(n): there exist positive constants c and
n0 such that 0 <= cg(n) <= f(n) for
all n >= n0}.

Let us consider the same Insertion sort example here. The time complexity of Insertion Sort can be written as Ω(n), but it is not a very useful information about insertion sort, as we are generally interested in worst case and sometimes in average case.

Big-O notation is most popular.

Big - O Notation is used to describe the Time complexity of an algorithm. It means how much time is needed by an algorithm to complete its execution for the input size of N. For Example a sorting algorithm take longer time to sort 5000 elements than 50. Following are commonly used Orders of an algorithm.

(1) O(1): An algorithm that will always execute in the same time regardless of the size of the input data is having complexity of O(1).

(2) O(n): An algorithm whose performance is directly proportional to the size of the input data is having complexity of O(n). It is also known as linier complexity. If an algorithm uses looping structure over the data then it is having linier complexity of O(n). Linier Search is an example of O(n).

(3) O(n2): An algorithm whose performance is directly proportional to the square of the size of the input data is having complexity of O(n2). If an algorithms uses nested looping structure over the data then it is having quadratic complexity of O(n2). Bubble sort, Selection Sort are the example of O(n2).

(4) O(logn): An algorithm in which during each iteration the input data set is partitioned into to sub parts is having complexity of O(logn). Quick Sort, Binary Search are the example of O(logn) complexity.

Lecture 3

Array

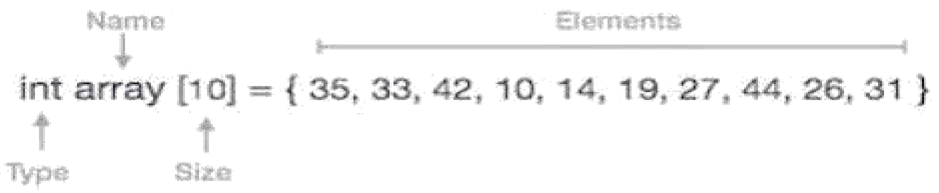
Array is a container which can hold a fix number of items and these items should be the same type. Most of the data structures make use of arrays to implement their algorithms.

Following are the important terms to understand the concept of Array.

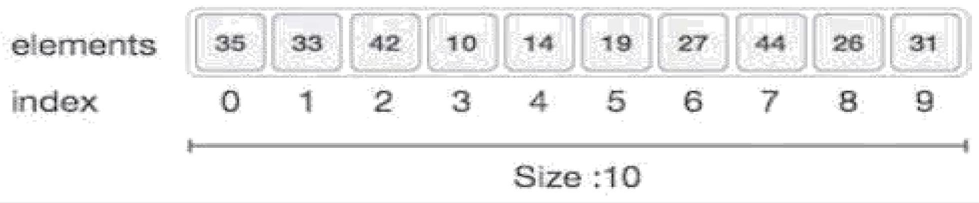
- **Element** − Each item stored in an array is called an element.
- **Index** − Each location of an element in an array has a numerical index, which issued to identify the element.

Array Representation:

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



As per the above illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 8 which means it can store 8 elements.
- Each element can be accessed via its index. For example, we can fetch an elementat index 6 as 9.

Basic Operations:

Following are the basic operations supported by an array.

- **Traverse** − Prints all the array elements one by one.
- **Insertion** − Adds an element at the given index.
- **Deletion** − Deletes an element at the given index.
- **Search** − Searches an element using the given index or by the value.
- **Update** − Updates an element at the given index.

In C, when an array is initialized with size, then it assigns defaults values to its elementsin following order.

Data Type	Default Value
bool	false
char	0
int	0
float	0.0
double	0.0f
void	
wchar_t	0

Insertion Operation:

Insert operation is to insert one or more data elements into an array. Based on therequirement, a new element can be added at the beginning, end, or any given index of array.

Here, we see a practical implementation of insertion operation, where we add data at the end of the array −

Algorithm

Let **Array** be a linear unordered array of **MAX** elements.

Example

Result

Let **LA** be a Linear Array (unordered) with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm where ITEM is inserted into the Kth position of LA −

```
1. Start
2. Set J=N
3. Set N = N+1
4. Repeat steps 5 and 6 while J >= K
5. Set LA[J+1] = LA[J]
6. Set J = J-1
7. Set LA[K] = ITEM
8. Stop
```

Array Insertions:

In the previous section, we have learnt how the insertion operation works. It is not alwaysnecessary that an element is inserted at the end of an array. Following can be a situation with array insertion −

- Insertion at the beginning of an array
- Insertion at the given index of an array
- Insertion after the given index of an array
- Insertion before the given index of an array

Insertion at the Beginning of an Array:

When the insertion happens at the beginning, it causes all the existing data items to shiftone step downward. Here, we design and implement an algorithm to insert an element at the beginning of an array.

Algorithm

We assume **A** is an array with **N** elements. The maximum numbers of elements it can store is defined by **MAX**. We shall first check if an array has any empty space to store any element and then we proceed with the insertion process.

```
begin
IF N = MAX, return
ELSE
    N = N + 1
    For All Elements in A
        Move to next adjacent location
    A[FIRST] = New_Element
end
```

Insertion at the Given Index of an Array:

In this scenario, we are given the exact location (**index**) of an array where a new data element (**value**) needs to be inserted. First we shall check if the array is full, if it is not, then we shall move all data elements from that location one step downward. This will make room for a new data element.

Algorithm

We assume **A** is an array with **N** elements. The maximum numbers of elements it can store is defined by **MAX**.

```
begin
IF N = MAX, return
ELSE
    N = N + 1
    SEEK Location index
    For All Elements from A[index] to A[N]
        Move to next adjacent location
    A[index] = New_Element
end
```

Insertion after the Given Index of an Array:

In this scenario we are given a location (**index**) of an array after which a new data element(**value**) has to be inserted. Only the seek process varies, the rest of the activities are the same as in the previous example.

Algorithm

We assume **A** is an array with **N** elements. The maximum numbers of elements it can store is defined by **MAX**.

```
begin
IF N = MAX, return
ELSE
    N = N + 1
    SEEK Location index
    For All Elements from A[index + 1] to A[N]
        Move to next adjacent location
    A[index + 1] = New_Element
End
```

Insertion before the Given Index of an Array:

In this scenario we are given a location **(index)** of an array before which a new data element (**value**) has to be inserted. This time we seek till **index-1**, i.e., one location ahead of the given index. Rest of the activities are the same as in the previous example.

Algorithm:

We assume A is an array with N elements. The maximum numbers of elements it can store is defined by MAX.

```
begin
IF N = MAX, return
ELSE
    N = N + 1
    SEEK Location index
    For All Elements from A[index - 1] to A[N]
        Move to next adjacent location
    A[index - 1] = New_Element
End
```

Deletion Operation:

Deletion refers to removing an existing element from the array and re-organizing allelements of an array.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**.
Following is the algorithm to delete an element available at the Kth position of LA.

```
1. Start
2. Set J=K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J-1] = LA[J]
5. Set J = J+1
6. Set N = N-1
7. Stop
```

Traverse Operation:

The method of processing each element in the array exactly once is known as Traversal.

In array,traversal starts from first element in the array and ends at the last element of the array.

Algorithm

```
Step 1: START = 0

Step 2: Repeat Step3 while (START<N)

Step 3: Read A [START]

        START = START + 1
```

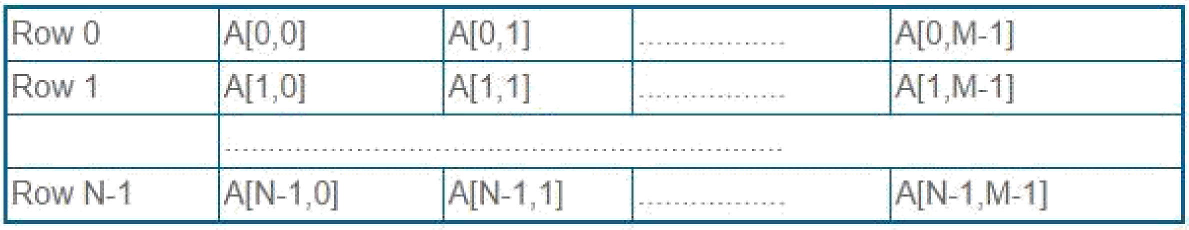
Lecture 4

Row Major Order Representation of Array:

Row Major Order is a method of representing multi dimension array in sequential memory.

In this method elements of an array are arranged sequentially row by row. Thus elements of first row occupies first set of memory locations reserved for the array, elements of second row occupies the next set of memory and so on.

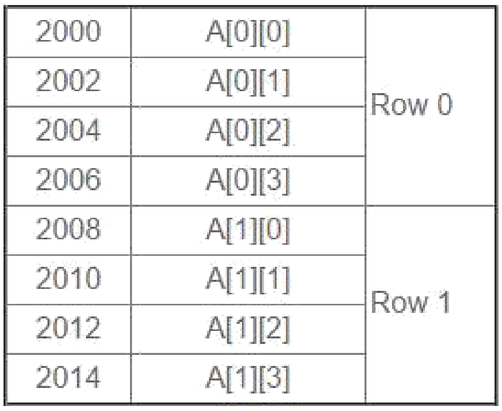
Consider a Two Dimensional Array consist of N rows and M columns. It can be stored sequentially in memory row by row as shown below:



Row 0	A[0,0]	A[0,1]	....	A[0,M-1]
Row 1	A[1,0]	A[1,1]	....	A[1,M-1]
	....			
Row N-1	A[N-1,0]	A[N-1,1]	....	A[N-1,M-1]

Example:

Consider following example in which a two dimensional array consist of two rows and four columns is stored sequentially in row major order as:



2000	A[0][0]	Row 0
2002	A[0][1]	
2004	A[0][2]	
2006	A[0][3]	
2008	A[1][0]	Row 1
2010	A[1][1]	
2012	A[1][2]	
2014	A[1][3]	

The Location of element A[i, j] can be obtained by evaluating expression:

LOC (A [i, j]) = Base_Address + W [M (i) + (j)]

Here,

Base_Address is the address of first element in the array.
W is the word size. It means number of bytes occupied by each element.
N is number of rows in array.
M is number of columns in array.

Suppose we want to calculate the address of element A [1, 2].
It can be calculated as follow:

Here,

Base_Address = 2000, W= 2, M=4, N=2, i=1, j=2

LOC (A [i, j]) = Base_Address + W [M (i) + (j)]
LOC (A[1, 2]) = 2000 + 2 *[4*(1) + 2]
= 2000 + 2 * [4 + 2]
= 2000 + 2 * 6
= 2000 + 12
= 2012

Column Major Order Representation of Array:

Column Major Order is a method of representing multi dimension array in sequential memory.

In this method elements of an array are arranged sequentially column by column. Thus elements of first column occupies first set of memory locations reserved for the array, elements of second column occupies the next set of memory and so on.

Consider a Two Dimensional Array consist of N rows and M columns. It can be stored sequentially in memory column by column as shown below:

Column 0	A[0,0]	A[1,0]		A[N-1,0]
Column 1	A[0,1]	A[1,1]		A[N-1,1]
Column N-1	A[0,M-1]	A[1,M-1]		A[N-1,M-1]

Example:

Consider following example in which a two dimensional array consist of two rows and four columns is stored sequentially in Column Major Order as:

2000	A[0][0]	Column 0
2002	A[1][0]	
2004	A[0][1]	Column 1
2006	A[1][1]	
2008	A[0][2]	Column 2
2010	A[1][2]	
2012	A[0][3]	Column 3
2014	A[1][3]	

The Location of element A[i, j] can be obtained by evaluating expression:

LOC (A [i, j]) = Base_Address + W [N (j) + (i)]

Here,

Base_Address is the address of first element in the array.
W is the word size. It means number of bytes occupied by each element.
N is number of rows in array.
M is number of columns in array.

Suppose we want to calculate the address of element A [1, 2].
It can be calculated as follow:

Here,
Base_Address = 2000, W= 2, M=4, N=2, i=1, j=2

```
LOC (A [i, j]) =        Base_Address + W [N (j) + (i)]
LOC (A[1, 2]) =         2000 + 2 *[2*(2) + 1]
              =         2000 + 2 * [4 + 1]
              =         2000 + 2 * 5
              =         2000 + 10
              =         2010
```

Polynomial Representation using Array:

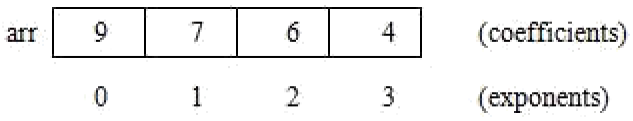
- Representation
- Addition
- Multiplication

Representation of a Polynomial:

A polynomial is an expression that contains more than two terms. A term is made up of coefficient and exponent. An example of polynomial is

P(x) = $4x^3+6x^2+7x+9$

A polynomial thus may be represented using arrays or linked lists. Array representation assumes that the exponents of the given expression are arranged from 0 to the highest value (degree), which is represented by the subscript of the array beginning with 0. The coefficients of the respective exponent are placed at an appropriate index in the array. The array representation for the above polynomial expression is given below:



arr	9	7	6	4	(coefficients)
	0	1	2	3	(exponents)

Sparse Matrix:

A matrix that has minimum number of non-zero elements is called a sparse matrix i.e. very few elements are sparsely distributed in the matrix.

Example: An array of order 3 x 4 containing sparsely located non-zero elements.

	0	**1**	**2**	**3**
0	0	25	0	7
1	0	0	2	8
2	0	0	1	0

(Order 3 x 4)

In the above example, out of 12 elements only 5 elements contain non zero values.
Such a matrix is called sparse matrix.

Representation of Sparse Matrix:

- **3-Tuple Representation**
- **List Representation**

3-Tuple Representation: An array of three columns is required. The size of the array is the number of non-zero elements plus one. First row is called the header row that contains total number of rows, total number of columns and non-zero element's value i.e.

Header Row: Row | Column | Element

	rows	**columns**	**Elements (non zero)**
0	3	4	5
1	0	1	25
2	0	3	7
3	1	2	2
4	1	3	8
5	2	2	1

Sparse Matrix representation for the array given above

Number of rows: 3

Number of columns: 4

Number of elements: 5

Order of Sparse Matrix = (5+1) x 3 = 6 x 3.

List Representation: Each row is converted to a node in linked representation where each node contains, row subscripts, column subscript and non-zero element. The first node returns the total number of rows, columns and elements.

Example:

3|4|5 -> 0|1|25 -> 0|3|7 -> 1|2|2 -> 1|3|8 -> 2|2|1 -> NULL

Lecture 5

Linked List Concepts:

A linked list is a non-sequential collection of data items. It is a dynamic data structure. For every data item in a linked list, there is an associated pointer that would give the memory location of the next data item in the linked list.
The data items in the linked list are not in consecutive memory locations. They may be anywhere, but the accessing of these data items is easier as each data item contains the address of the next data item.

Advantages of linked lists:

Linked lists have many advantages. Some of the very important advantages are:

1. Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.
2. Linked lists have efficient memory utilization. Here, memory is not pre--allocated. Memory is allocated whenever it is required and it is de-allocated (removed) when it is no longer needed.
3. Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.
4. Many complex applications can be easily carried out with linked lists.

Disadvantages of linked lists:

1. It consumes more space because every node requires an additional pointer to store address of the next node.
2. Searching a particular element in list is difficult and also time consuming.

Types of Linked Lists:

Basically we can put linked lists into the following four items:

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

A singly linked list is one in which all nodes are linked together in some sequential manner. Hence, it is also called as linear linked list.
A doubly linked list is one in which all nodes are linked together by multiple links which helps in accessing both the successor node (next node) and predecessor node (previous node) from any arbitrary node within the list. Therefore each node in a double linked list has two link fields (pointers) to point to the left node (previous) and the right node (next). This helps to traverse in forward direction and backward direction.

A circular linked list is one, which has no beginning and no end. A single linked list can be made a circular linked list by simply storing address of the very first node in the link field of the last node.

Array	**Linked List**
Size of an array is fixed	Size of a list is not fixed
Memory is allocated from stack	Memory is allocated from heap
It is necessary to specify the number of elements during declaration (i.e., during	It is not necessary to specify the number of elements during declaration

compile time).	(i.e., memory is allocated during run time).
It occupies less memory than a linked list for the same number of elements.	It occupies more memory.
Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.	Inserting a new element at any position can be carried out easily.

Comparison between array and linked list:

Applications of linked list:

1. Linked lists are used to represent and manipulate polynomial. Polynomials are expression containing terms with non-zero coefficient and exponents. For example:
   $P(x) = a_0 X^n + a_1 X^{n\ 1} + \ldots + a_{n-1} X + a_n$
2. Represent very large numbers and operations of the large number such as addition, multiplication and division.
3. Linked lists are to implement stack, queue, trees and graphs.
4. Implement the symbol table in compiler construction

Singly Linked List:

A linked list allocates space for each element separately in its own block of memory called a "node". The list gets an overall structure by using pointers to connect all its nodes together like the links in a chain. Each node contains two fields; a "data" field to store whatever element, and a "next" field which is a pointer used to link to the next node. Each node is allocated in the heap using malloc(), so the node memory continues to exist until it is explicitly de-allocated using free(). The front of the list is a pointer to the "start" node.

The beginning of the linked list is stored in a "**start**" pointer which points to the first A

single linked list is shown in the following figure.

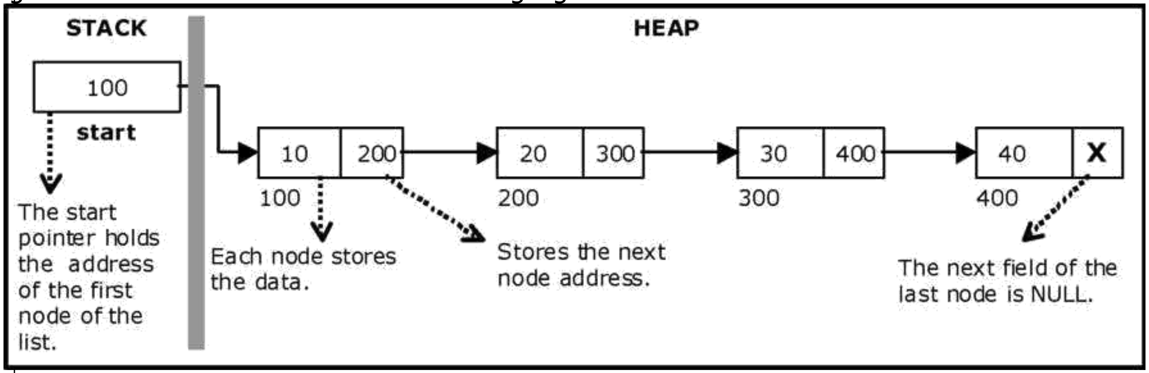

STACK
HEAP
100
start
10
200
20
300
30
400
40
X
100
200
300
400
The start pointer holds the address of the first node of the list.
Each node stores the data.
Stores the next node address.
The next field of the last node is NULL.

Figure: Single Linked List

node. The first node contains a pointer to the second node. The second node contains a pointer to the third node and so on. The last node in the list has its next field set to NULL to mark the end of the list. Code can access any node in the list by starting at the **start** and following the next pointers.

The **start** pointer is an ordinary local pointer variable, so it is drawn separately on the left top to show that it is in the stack. The list nodes are drawn on the right to show that they are allocated in the heap.

Implementation of Singly Linked List:

Before writing the code to build the above list, we need to create a **start** node**,** used to create and access other nodes in the linked list. The following structure definition will do (see the following figure):

- Creating a structure with one data item and a next pointer which will be pointing next node of the list. This is called as self-referential structure.
- Initialize the start pointer to be NULL.

```
struct slinklist
{
        int data;
        struct slinklist* next;
};
typedef struct slinklist node;
node *start=NULL;
```

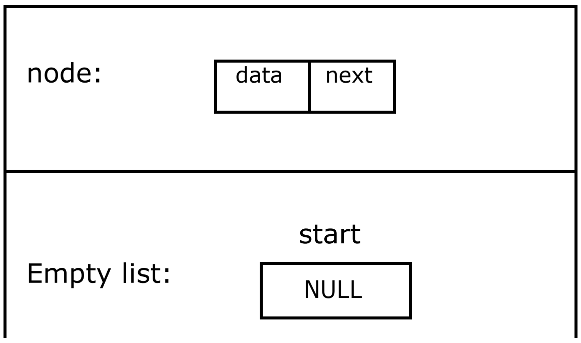


Figure: Structure definition, single link node and empty list

Lecture 6

The basic operations in a singly linked list are:

- Creation
- Insertion
- Deletion
- Traversing

Creating a node for Singly Linked List:

Creating a singly linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function. The function getnode(), is used for creating a node, after allocating memory for the structure of type node, the information for the item (i.e., data) has to be read from the user, set next field to NULL and finally returns the address of the node. The following figure illustrates the creation of a node for single linked list.

```
node* getnode()
{
        node* newnode;
        newnode = (node *) malloc(sizeof(node));
        printf("\n Enter data: ");
        scanf("%d", &newnode -> data);
        newnode -> next = NULL; return
        newnode;
}
```

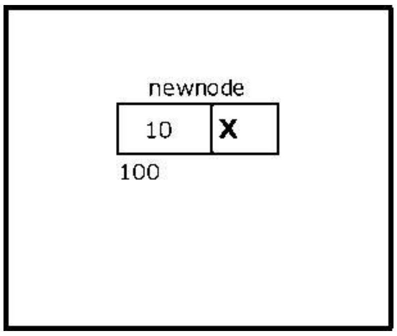


Figure: A new node with a value of 10

Creating a Singly Linked List with 'n' number of nodes:

The following steps are to be followed to create 'n' number of nodes:

- Get the new node using getnode().
  newnode = getnode();

- If the list is empty, assign new node as start.
  start = newnode;

- If the list is not empty, follow the steps given below:
  - The next field of the new node is madeto point the first node (i.e.start node) in the list by assigning the address of the first node.
  - The start pointer is made to point the new node by assigning the address of the new node.

- Repeat the above steps 'n' times.

The following figure shows 4 items in a single linked list stored at different locations in memory.

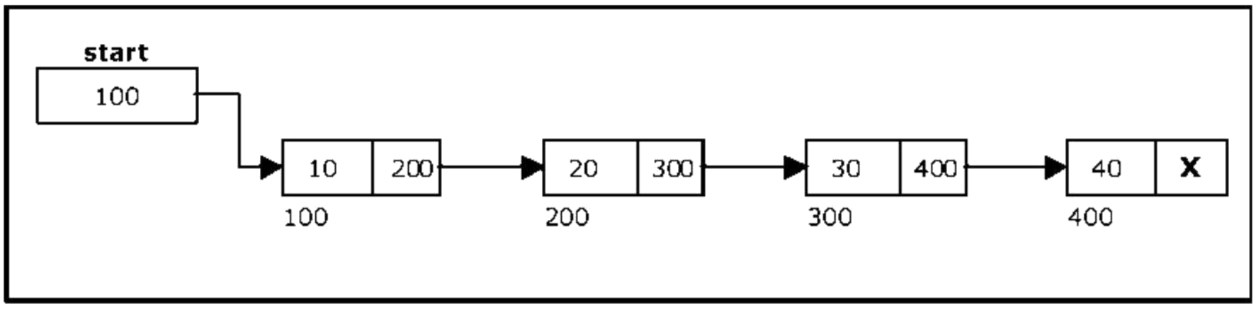


Figure: Singly Linked List with 4 nodes

The function createlist(), is used to create 'n' number of nodes:

```
void createlist(int n)
{int i;

        node *newnode;
        node *temp;
        for( i = 0; i    n ; i+ +)
        {
                newnode = getnode();
                if(star = = NULL)
                {start = newnode;

                }
                else
                {temp = start;
                        while(temp -> next != NULL)
                        temp = temp -> next; temp ->
                        next = newnode;

                }
        }
}
```

One of the most primitive operations that can be done in a singly linked list is the insertion of a node. Memory is to be allocated for the new node (in a similar way that is done while creating a list) before reading the data. The new node will contain empty data field and empty next field. The data field of the new node is then stored with the information read from the user. The next field of the new node is assigned to NULL. The new node can then be inserted at three different places namely:

- Inserting a node at the beginning
- Inserting a node at the end
- Inserting a node at intermediate position

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the list:

- Get the new node using getnode()
  newnode = getnode();

- If the list is empty then *start = newnode.*

- If the list is not empty, follow the steps given below: newnode -> next = start;

The following figure shows inserting a node into the single linked list at the beginning

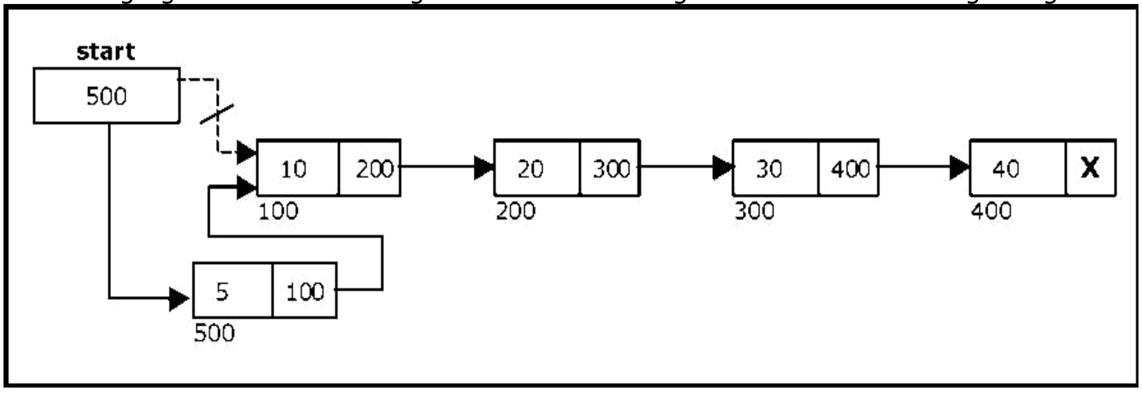


Figure: Inserting a node at the beginning

start = newnode;

The function insert_at_beg(), is used for inserting a node at the beginning

```
void insert_at_beg()
{
        node *newnode;
        newnode = getnode();
        if(start == NULL)
        {
                start = newnode;
        }
        else
        {
                newnode -> next =
                start; start = newnode;
        }
}
```

The following steps are followed to insert a new node at the end of the list:

- Get the new node using getnode()

  newnode = getnode();

- If the list is empty then *start = newnode.*
- If the list is not empty follow the steps given below:

  temp = start;

  while(temp -> next != NULL)

  temp = temp -> next; temp

  -> next = newnode;

The following figure shows inserting a node into the single linked list at the end.

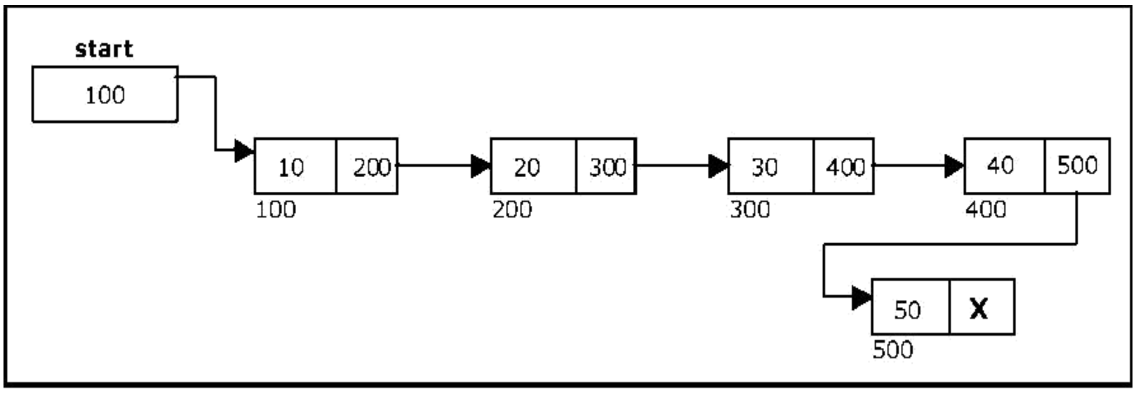


Figure: Inserting a node at the end

The function insert_at_end(), is used for inserting a node at the end.

```
void insert_at_end()
{
        node        *newnode,
        *temp;  newnode  =
        getnode();    if(start
        == NULL)
        {
                start = newnode;
        }
        else
        {
                temp = start;
                while(temp -> next !=
                NULL) temp = temp ->
                next; temp -> next =
                newnode;
        }
}
```

Inserting a node at intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

- Get the new node using getnode()
  newnode = getnode();

Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.

- Store the starting address (which is in start pointer) in temp and prevpointers. Then traverse the temp pointer upto the specified position followed by prev pointer.

- After reaching the specified position, follow the steps given below:

```
prev -> next = newnode;
newnode -> next = temp;
```

- Let the intermediate position be 3.

The following figure shows inserting a node into the single linked list at a specified intermediate position other than beginning and end.

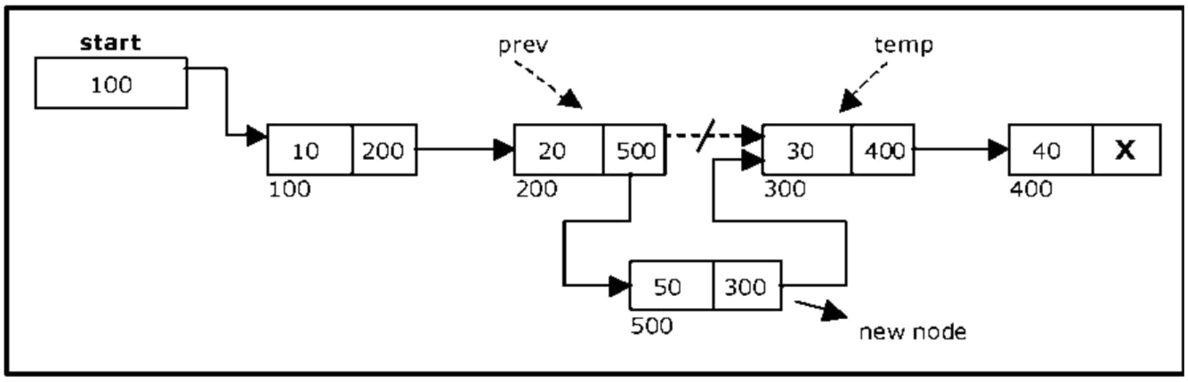


Figure: Inserting a node at an intermediate position

The function insert_at_mid(), is used for inserting a node in the intermediate position.

```
void insert_at_mid()
{
        node *newnode, *temp,
        *prev; int pos, nodectr, ctr =
        1; newnode = getnode();
        printf("\n Enter the position:
        "); scanf("%d", &pos);
        nodectr = countnode(start);
        if(pos > 1 && pos < nodectr)
        {
                temp = prev = start;
                while(ctr < pos)
                {
                        prev = temp;
                        temp = temp -> next;
                        ctr++;
                }
                prev -> next = newnode;
                newnode -> next = temp;
        }
        else
        {
                printf("position %d is not a middle position", pos);
        }
}
```

Lecture 7

Another primitive operation that can be done in a singly linked list is the deletion of a node. Memory is to be released for the node to be deleted. A node can be deleted from the list from three different places namely.

- Deleting a node at the beginning
- Deleting a node at the end
- Deleting a node at intermediate position

Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:

```
temp = start;
start = start -> next;
free(temp);
```

The following figure shows deleting a node at the beginning of a single linked list.

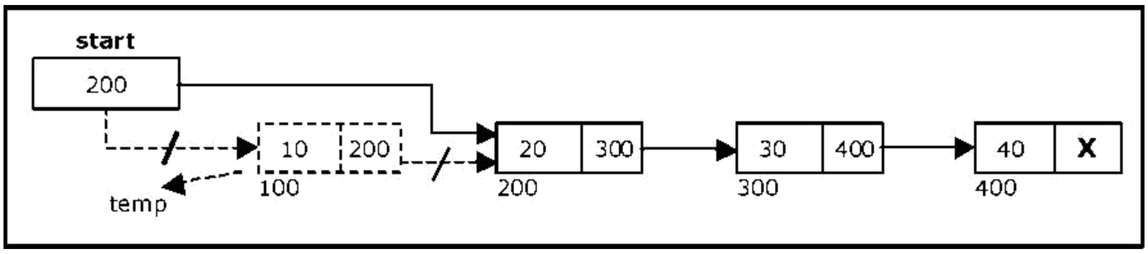


Figure: Deleting a node at the beginning

The function delete_at_beg(), is used for deleting the first node in the

list.

```
void delete_at_beg()
{
        node *temp;
        if(start == NULL)
        {
                printf("\n No nodes are
                exist.."); return ;
        }
        else
        {
                temp = start;
                start = temp -> next;
                free(temp);
                printf("\n Node deleted ");
        }
}
```

The following steps are followed to delete a node at the end of the list:

- If list is empty then display 'Empty List' message.

- If the list is not empty, follow the steps given below:

```
temp = prev = start;
while(temp -> next != NULL)
{
        prev = temp;
        temp = temp -> next;
}
prev -> next = NULL;
```

The following figure shows deleting a node at the end of a single linked list.

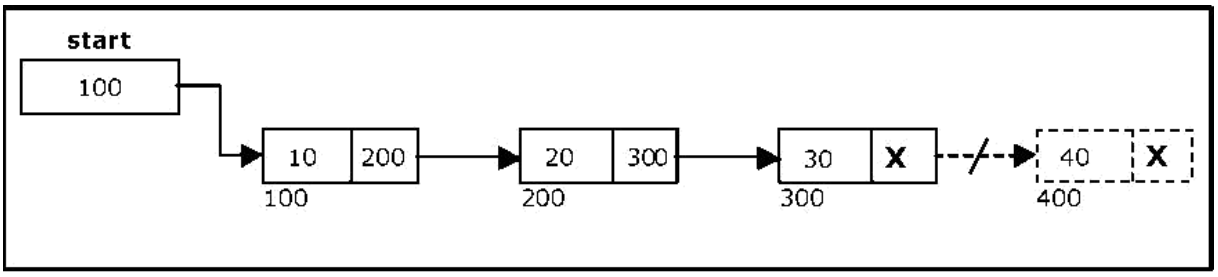


Figure: Deleting a node at the end

```
free(temp);
```

The function delete_at_last(), is used for deleting the last node in the list. *void delete_at_last()*

```
{
        node *temp, *prev;
        if(start == NULL)
        {
                printf("\n Empty List..");
                return ;
        }
        else
        {
                temp = start;
                prev = start;
                while(temp -> next != NULL)
                {
                        prev = temp;
                        emp = temp -> next;
                }
                prev -> next = NULL;
                free(temp);
                printf("\n Node deleted ");
        }
}
```

Deleting a node at the beginning:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two node).

- If list is empty then display 'Empty List' message
- If the list is not empty, follow the steps given below

```
                if(pos > 1 && pos < nodectr)
                {
                        temp = prev = start;
                        ctr = 1;
                        while(ctr < pos)
                        {
                                prev = temp;
                                temp = temp -> next;
                                ctr+ + ;
                        }
                        prev -> next = temp ->
                        next; free(temp);
                        printf("\n node deleted..");
                }
```

The following figure shows deleting a node at a specified intermediate position other than beginning and end from a single linked list.

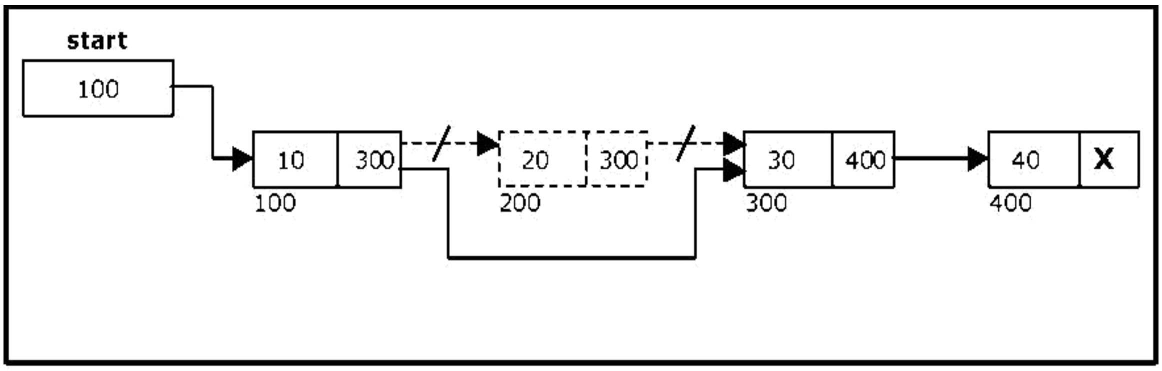


Figure: Deleting a node at an intermediate position

The function delete_at_mid(), is used for deleting the intermediate node in the list.

```
        void delete_at_mid()
        {
                int ctr = 1, pos, nodectr;
                node *temp, *prev;
                if(start == NULL)
                {
                        printf("\n Empty List..");
                        return ;
                }
                else
                {
                        printf("\n Enter position of node to
                        delete: "); scanf("%d", &pos);
                        nodectr = countnode(start);
                        if(pos > nodectr)
                        {
                                printf("\nThis node doesnot exist");
                        }
                        if(pos > 1 && pos < nodectr)
                        {
                                temp = prev = start;
                                while(ctr < pos)
                                {
                                        prev = temp;
                                        temp = temp ->
                                        next;
                                        ctr ++;
                                }
                        prev -> next = temp ->
                        next; free(temp);
                        printf("\n Node deleted..");
                }
                else
                {
                        printf("\n Invalid
                        position.."); getch();
                }

        }
}
```

Traversal and displaying a list (Left to Right):

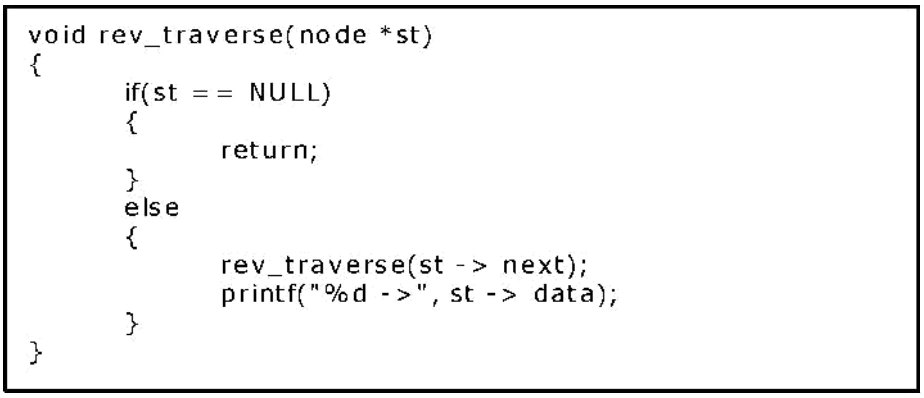
To display the information, you have to traverse (move) a linked list, node by node from the first node, until the end of the list is reached. Traversing a list involves the following steps:

- Assign the address of start pointer to a temp pointer
- Display the information from the data field of each node

The function *traverse()* is used for traversing and displaying the information stored in the list from left to right.

```
void traverse()
{
        node *temp;
        temp = start;
        printf("\n The contents of List (Left to Right):
        \n"); if(start == NULL )
                printf("\n Empty List");
        else
        {
                while (temp != NULL)
                {
                        printf("%d ->", temp ->
                        data); temp = temp ->
                        next;
                }
        }
        printf("X");
}
```

Alternatively there is another way to traverse and display the information. That is in reverse order. The function rev_traverse(), is used for traversing and displaying the information stored in the list from right to left.



```
void rev_traverse(node *st)
{
        if(st == NULL)
        {
                return;
        }
        else
        {
                rev_traverse(st -> next);
                printf("%d ->", st -> data);
        }
}
```

Counting the Number of Nodes:
The following code will count the number of nodes exist in the list using *recursion.*

```
int countnode(node *st)
{
        if(st == NULL)
                return 0;
        else
                return(1 + countnode(st -> next));
}
```

Lecture 8

Doubly Linked List:

A doubly linked list is a two-way list in which all nodes will have two links. This helps in accessing both successor node and predecessor node from the given node position. It provides bi-directional traversing. Each node contains three fields:

- Left link.
- Data.
- Right link.

The left link points to the predecessor node and the right link points to the successor node. The data field stores the required data.

Many applications require searching forward and backward thru nodes of a list. For example searching for a name in a telephone directory would need forward and backward scanning thru a region of the whole list.

The basic operations in a doubly linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

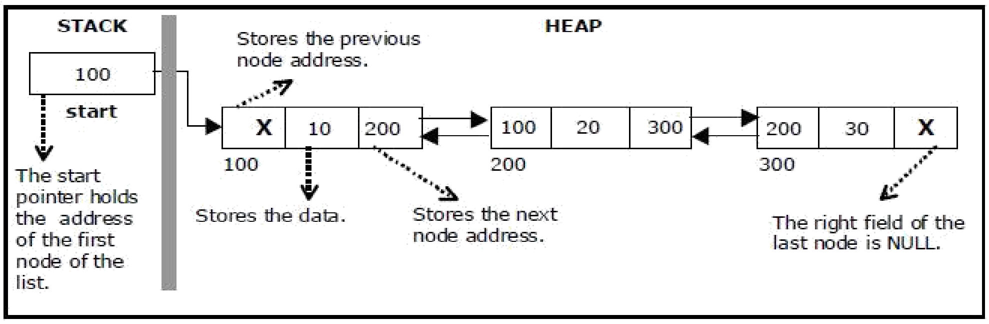

STACK
HEAP
100
start
The start pointer holds the address of the first node of the list.
Stores the previous node address.
X
10
200
100
20
300
200
30
X
100
200
300
Stores the data.
Stores the next node address.
The right field of the last node is NULL.

Figure: A doubly linked list

```
struct dlinklist
{
            struct dlinklist *left;
            int data;
            struct slinklist*right;
};
typedef struct dlinklist node;
node *start=NULL;
```

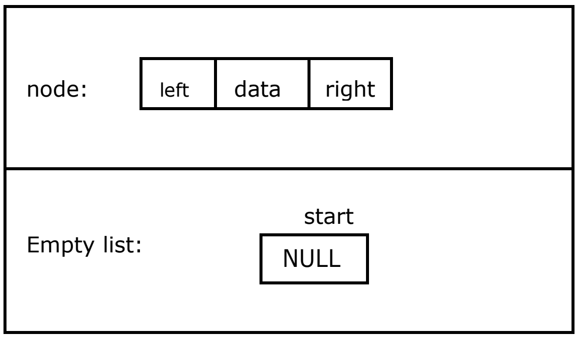


Figure: Structure definition, double link node and empty list

The beginning of the doubly linked list is stored in a "**start**" pointer which points to the first node. The first node's left link and last node's right link is set to NULL.

Creating a node for Doubly Linked List:

Creating a double linked list starts with creating a node. Sufficient memory has to be allocated for creating a node. The information is stored in the memory, allocated by using the malloc() function. The function getnode(), is used for creating a node, after allocating memory for the structure of type node, the information for the item (i.e., data) has to be read from the user and set left field to NULL and right field also set to NULL

```
node* getnode()
{
        node* newnode;
        newnode = (node *) malloc(sizeof(node));
        printf("\n Enter data: ");
        scanf("%d", &newnode -> data);
        newnode -> left = NULL;
        newnode -> right = NULL;
        return newnode;
}
```

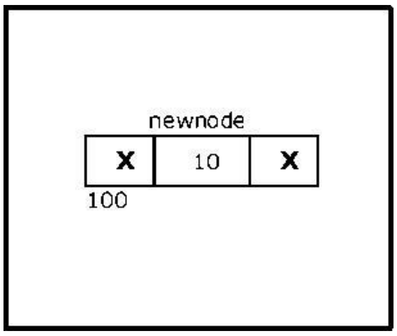


Figure: A new node with a value of 10

Creating a Doubly Linked List with n number of nodes:

The following steps are to be followed to create 'n' number of nodes:

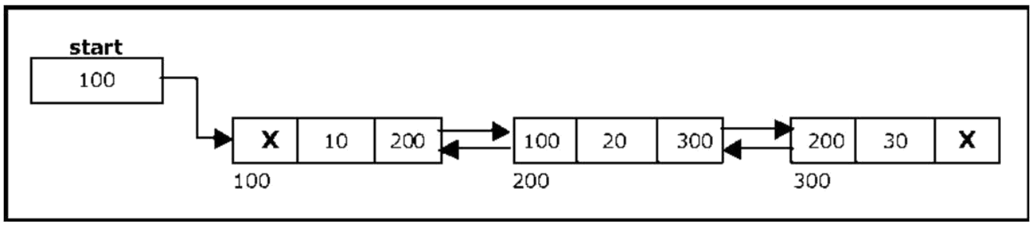
- Get the new node using getnode().
  newnode =getnode();
- If the list is empty then *start = newnode.*
- If the list isnot empty, follow the steps given below:
  - The left field of the new node is made to point the previous node.
  - The previous nodes right field must be assigned with address of the new node.
- Repeat the above steps 'n' times.

```
void createlist(int n)
{int i;

        node *newnode;
        node *temp;
        for( i  = 0; i <  n; i++)
        {
                newnode = getnode();
                if(start = = NULL)
                {start = newnode;

                }
                else
                {temp = start; while(temp -
                        > right)
                                temp = temp -> right;
                        temp -> right = newnode;
                        newnode -> left = temp;

                }
        }
}
```



The following figure shows 3 items in a double linked list stored at different locations.

Figure: Doubly Linked List with 3 nodes

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the

list:

- Get the new node using getnode().

  newnode=getnode();

- If the list is empty then *start = newnode.*
- If the list is not empty, follow the steps given below:

  newnode -> right = start;
  start -> left = newnode;
  start = newnode;

The function dbl_insert_beg(), is used for inserting a node at the beginning. The following figure shows inserting a node into the double linked list at the beginning.

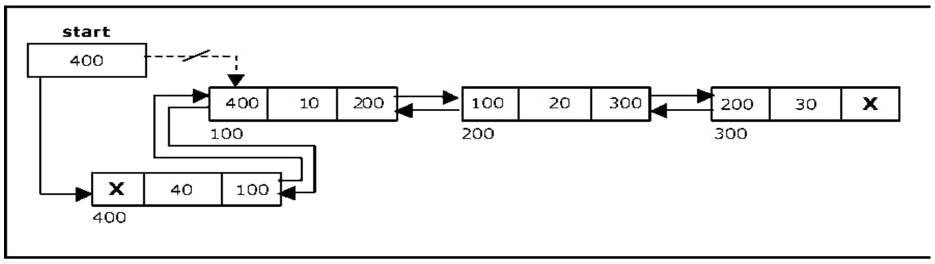


Figure: Inserting a node at the beginning

Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list:

- Get the new node using getnode()

```
newnode=getnode();
```

- If the list is empty then *start = newnode.*
- If the list is not empty follow the steps given below:

```
temp = start;
while(temp -> right != NULL)
temp = temp -> right;
temp -> right =
newnode;
newnode -> left = temp;
```

The function dbl_insert_end(), is used for inserting a node at the end. The following figure shows inserting a node into the double linked list at the end.

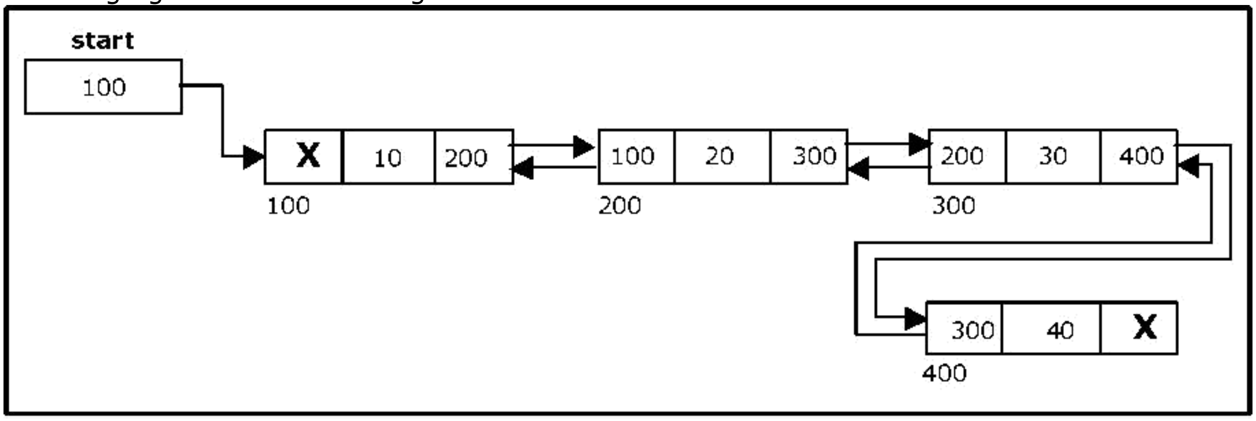


Figure: Inserting a node at the end

Inserting a node at an intermediate position:

The following steps are followed, to insert a new node in an intermediate position in the list:

- Get the new node using getnode().
  newnode=getnode();
- Ensure that the specified position is in between first node and last node. If not, specified position is invalid. This is done by countnode() function.
- Store the starting address (which is in start pointer) in temp and prev pointers. Then traverse the temp pointer upto the specified position followed by prev pointer.
- After reaching the specified position, follow the steps given below:

```
newnode -> left = temp;
newnode -> right = temp -> right;
temp -> right -> left =
newnode;
temp -> right = newnode;
```

The function dbl_insert_mid(), is used for inserting a node in the intermediate position. The following figure shows inserting a node into the double linked list at a specified intermediate position other than beginning and end.

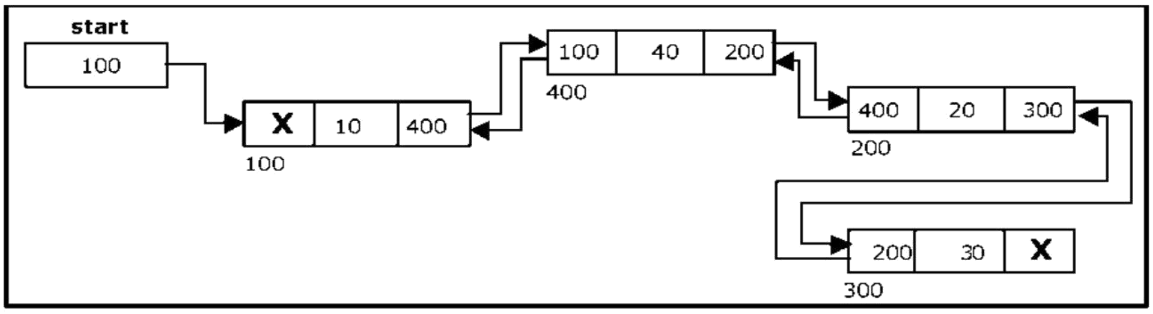


Figure: Inserting a node at an intermediate position

Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:

```
temp = start;
start = start -> right;
start -> left = NULL;
free(temp);
```

The function dbl_delete_beg(), is used for deleting the first node in the list. The following figure shows deleting a node at the beginning of a double linked list.

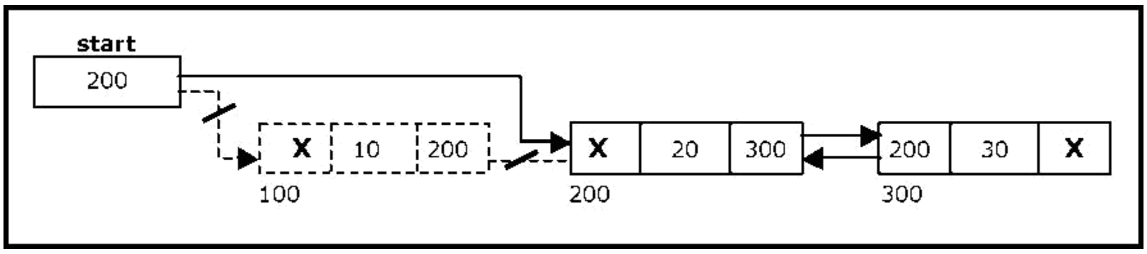


Figure: Deleting a node at beginning

Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

- If list is empty then display 'Empty List' message
- If the list is not empty, follow the steps given below:

```
temp = start;
while(temp -> right != NULL)
{
        temp = temp -> right;
}
temp -> left -> right =
NULL; free(temp);
```

The function dbl_delete_last(), is used for deleting the last node in the list. The following figure shows deleting a node at the end of a double linked list.

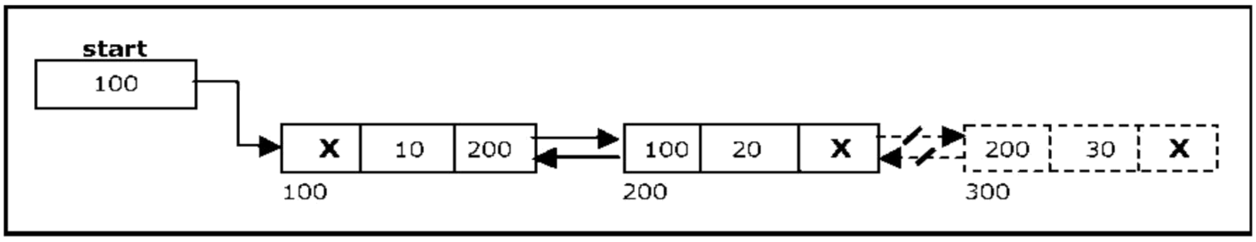


Figure: Deleting a node at the end

Deleting a node at Intermediate Position:

The following steps are followed, to delete a node from an intermediate position in the list (List must contain more than two nodes).

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:
    - Get the position of the node to delete.
    - Ensure that the specified position is in between first node and last node. If not, specified position is invalid.
    - Then perform the following steps:

```
if(pos >1&& pos < nodectr)
{
        temp = start;
          i= 1;
        while(i < pos)
        {
                temp = temp -> right;
                i++;
        }
        temp -> right -> left = temp ->
        left;
        temp -> left -> right = temp ->
        right; free(temp);
        printf("\n node deleted..");
}
```

The function delete_at_mid(), is used for deleting the intermediate node in the list. The following figure shows deleting a node at a specified intermediate position other than beginning and end from a double linked list.

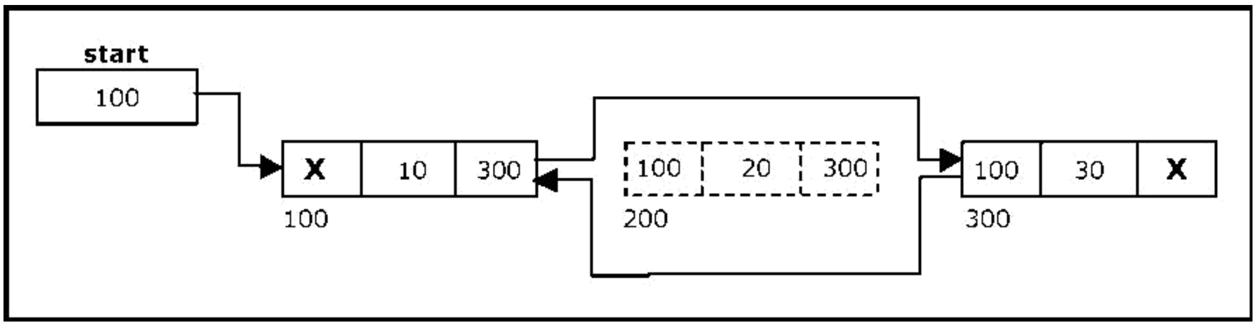


Figure: Deleting a node at an intermediate position

Traversal and displaying a list (Left to Right):

To display the information, you have to traverse the list, node by node from the first node, until the end of the list is reached. The function *traverse_left_right()* is used for traversing and displaying the information stored in the list from left to right.

The following steps are followed, to traverse a list from left to right:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:

```
temp = start;
while(temp != NULL)
{
        print temp ->
        data;
        temp = temp -> right;
}
```

Traversal and displaying a list (Right to Left):

To display the information from right to left, you have to traverse the list, node by node from the first node, until the end of the list is reached. The function *traverse_right_left()* is used for traversing and displaying the information stored in the list from right to left. The following steps are followed, to traverse a list from right to left:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given

```
below: temp = start;
while(temp -> right !=
NULL) temp = temp -> right;
while(temp != NULL)
{
        print temp -
        > data;
        temp = temp -
        > left;
}
```

Counting the Number of Nodes:

The following code will count the number of nodes exist in the list (using recursion).

```
int countnode(node *start)
{
            if(start == NULL)
                    return 0;
            else
                    return(1 + countnode(start ->right ));
}
```

Lecture 9

Circular Linked List:

It is just a single linked list in which the link field of the last node points back to the address of the first node. A circular linked list has no beginning and no end. It is necessary to establish a special pointer called *start* pointer always pointing to the first node of the list. Circular linked lists are frequently used instead of ordinary linked list because many operations are much easier to implement. In circular linked list no null pointers are used, hence all pointers contain valid address.

A circular linked list is shown in the following figure.

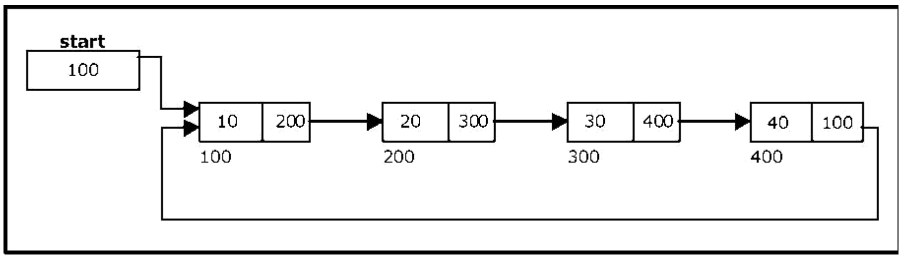


Figure: Circular Linked List

The basic operations in a circular single linked list are:

- Creation.
- Insertion.
- Deletion.
- Traversing.

Creating a circular Linked List with 'n' number of nodes:

The following steps are to be followed to create 'n' number of nodes:

- Get the new node using getnode().

```
newnode = getnode();
```

- If the list is empty, assign new node as start.

```
start = newnode;
```

- If the list is not empty, follow the steps givenbelow:

```
temp = start;
while(temp -> next != NULL)
        temp = temp -> next;
temp -> next =
newnode;
```

- Repeat the above steps 'n' times.

- newnode -> next = start;

The function createlist(), is used to create 'n' number of nodes.

Inserting a node at the beginning:

The following steps are to be followed to insert a new node at the beginning of the circular list:

- Get the new node using getnode().

```
newnode = getnode();
```

- If the list is empty, assign new node as start.

```
start = newnode;
newnode -> next =
start;
```

- If the list is not empty, follow the steps given below:

```
last = start;
while(last -> next !=
start)
        last = last ->
next; newnode -> next
= start;
start = newnode;
last -> next = start;
```

The function cll_insert_ beg(), is used for inserting a node at the beginning. The following figure shows inserting a node into the circular linked list at the beginning.

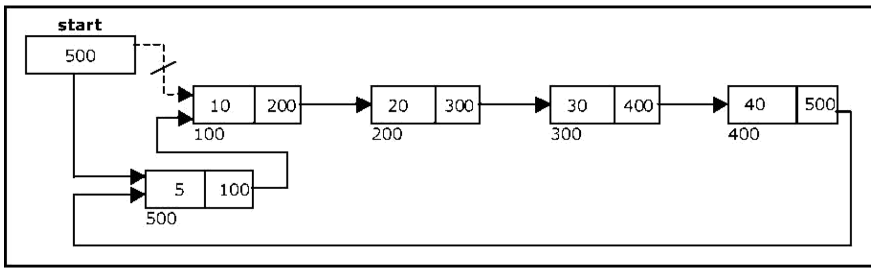


Figure: Inserting a node at the beginning

Inserting a node at the end:

The following steps are followed to insert a new node at the end of the list:

- Get the new node using getnode().

```
newnode = getnode();
```

- If the list is empty, assign new node as start.

```
start = newnode;
newnode -> next = start;
```

- If the list is not empty follow the steps given below:

```
temp = start;
while(temp -> next != start)
        temp = temp -> next;
temp -> next =
newnode; newnode ->
next = start;
```

The function cll_insert_end(), is used for inserting a node at the end. The following figure shows inserting a node into the circular linked list at the end.

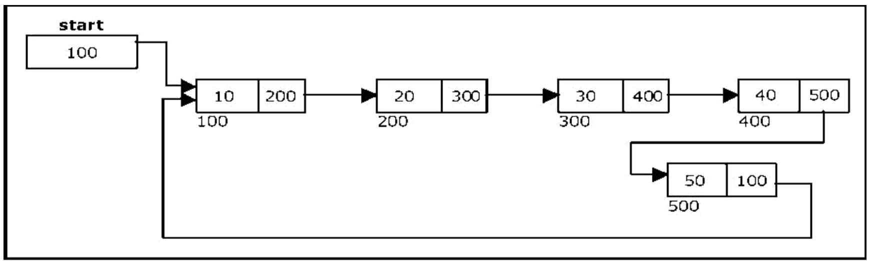


Figure: Inserting a node at the end

Deleting a node at the beginning:

The following steps are followed, to delete a node at the beginning of the list:

- If the list is empty, display a message 'Empty List'.
- If the list is not empty, follow the steps given below:

```
last = temp = start;
while(last -> next != start)
        last = last -> next;
start = start -> next;
last -> next = start;
```

- After deleting the node, if the list is empty then *start = NULL.*

The function cll_delete_beg(), is used for deleting the first node in the list. The following figure shows deleting a node at the beginning of a circular linked list.

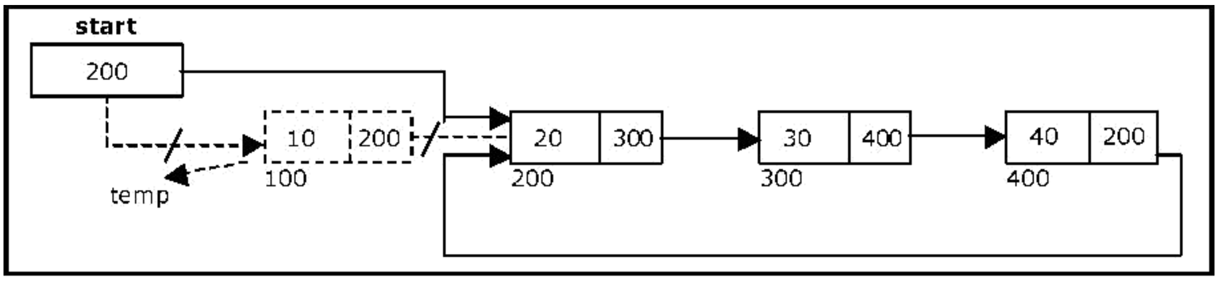


Figure: Deleting a node at the beginning

Deleting a node at the end:

The following steps are followed to delete a node at the end of the list:

- If the list is empty, display a message 'Empty List'.
- If the list is not empty, follow the steps given below:

```
temp = start;
prev = start;
while(temp -> next != start)
{
        prev = temp;
        temp = temp -> next;
}

prev -> next = start;
```

- After deleting the node, if the list is empty then *start = NULL.*

The function cll_delete_last(), is used for deleting the last node in the list. The following figure shows deleting a node at the end of a circular linked list.

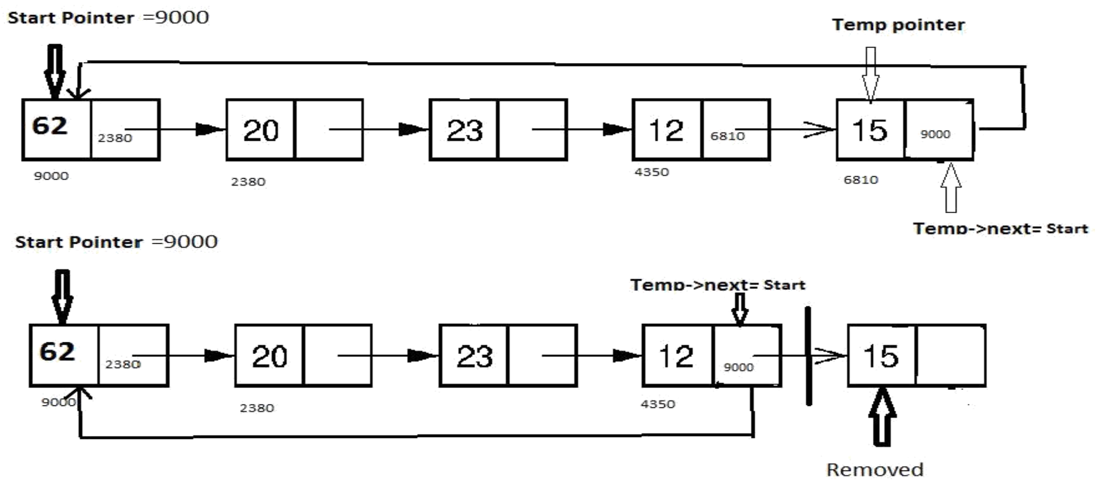


Figure: Deleting a node at the end

Traversing a circular single linked list from left to right:

The following steps are followed, to traverse a list from left to right:

- If list is empty then display 'Empty List' message.
- If the list is not empty, follow the steps given below:

```
temp = start;
do {
        printf("%d ", temp ->
        data);
        temp = temp -> next;
} while(temp != start);
```

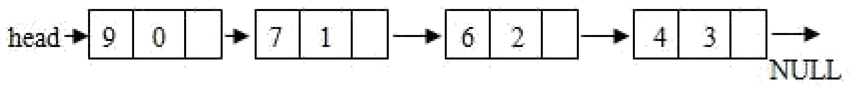
Lecture 10

A polynomial represented using a linked list

A structure may be defined such that it contains two parts- one is the coefficient and second is the corresponding exponent. The structure definition may be given as shown below:

```
struct polynomial
{
        int coefficient;
        int exponent;
        struct polynomial *next;
};
```

Thus the above polynomial may be represented using linked list as shown below:



Some examples of polynomials are:

5x2 + 3x + 1

12x3 - 4x

5x4 - 8x3 + 2x2 + 4x1 + 9x0

It is not necessary to write terms of the polynomials in decreasing order of degree. In other words the two polynomials 1 + x and x + 1 are equivalent.

The computer implementation requires implementing polynomials as a list of pairs of coefficient and exponent. Each of these pairs will constitute a structure, so a polynomial will be represented as a list of structures. A linked list structure that represents polynomials 5x4 - 8x3 + 2x2 + 4x1 + 9x0 illustrates in the following figure.

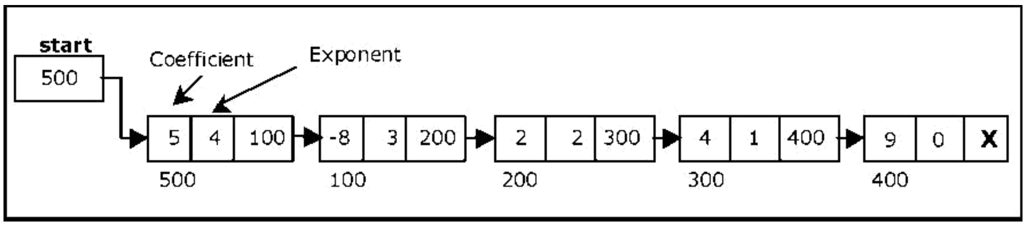


Figure 26. Singly Linked List for the polynomial F(x) =5x4 - 8x3 + 2x2 + 4x1 + 9x0

Addition of two Polynomials:

For adding two polynomials using arrays is straightforward method, since both the arrays may be added up element wise beginning from 0 to n-1, resulting in addition of two polynomials. Addition of two polynomials using linked list requires comparing the exponents, and wherever the exponents are found to be same, the coefficients are added up. For terms with different exponents, the complete term is simply added to the result thereby making it a part of addition result.

To add two polynomials follow the following steps:

- Read two polynomials.
- Add them.
- Display the resultant polynomial.

Multiplication of two Polynomials:

Multiplication of two polynomials however requires manipulation of each node such that the exponents are added up and the coefficients are multiplied. After each term of first polynomial is operated upon with each term of the second polynomial, then the result has to be added up by comparing the exponents and adding the coefficients for similar exponents and including terms as such with dissimilar exponents in the result.

To add two polynomials follow the following steps:

- Read two polynomials.
- Multiply them.
- Display the resultant polynomial.

Module II

Linear Data structures

Stack and Queue

Lecture 11

STACK:

A stack is a list of elements in which an element may be inserted or deleted only at one end, called the top of the stack. Stacks are sometimes known as LIFO (last in, first out) lists.
As the items can be added or removed only from the top i.e. the last item to be added to a stack is the first item to be removed. The two basic operations associated with stacks are:

- *Push:* is the term used to insert an element into a stack.
- *Pop:* is the term used to delete an element from a stack.

"Push" is the term used to insert an element into a stack. "Pop" is the term used to delete an element from the stack.

All insertions and deletions take place at the same end, so the last element added to the stack will be the first element removed from the stack. When a stack is created, the stack base remains fixed while the stack top changes as elements are added and removed. The most accessible element is the top and the least accessible element is the bottom of the stack.

Figure: Examples of stack

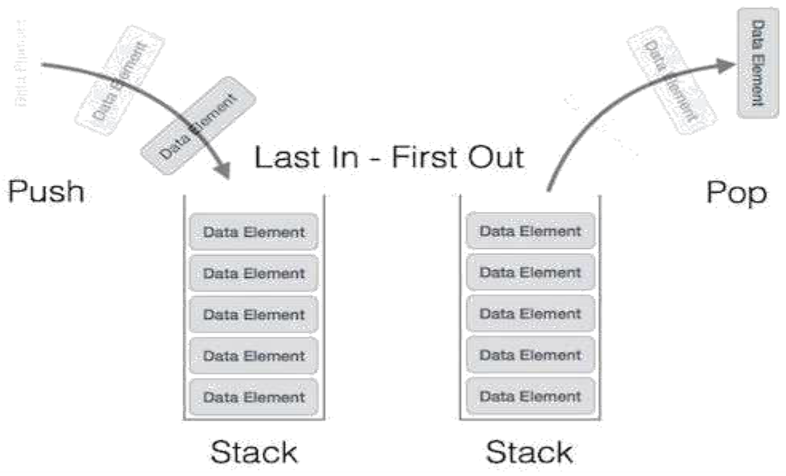


Figure: A stack and its operations

Representation of Stack:

Let us consider a stack with 6 elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack. If we attempt to add new element beyond the maximum size, we will encounter a *stack overflow* condition. Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a *stack underflow* condition.

When an element is added to a stack, the operation is performed by push(). The following figure shows the creation of a stack and addition of elements using push().

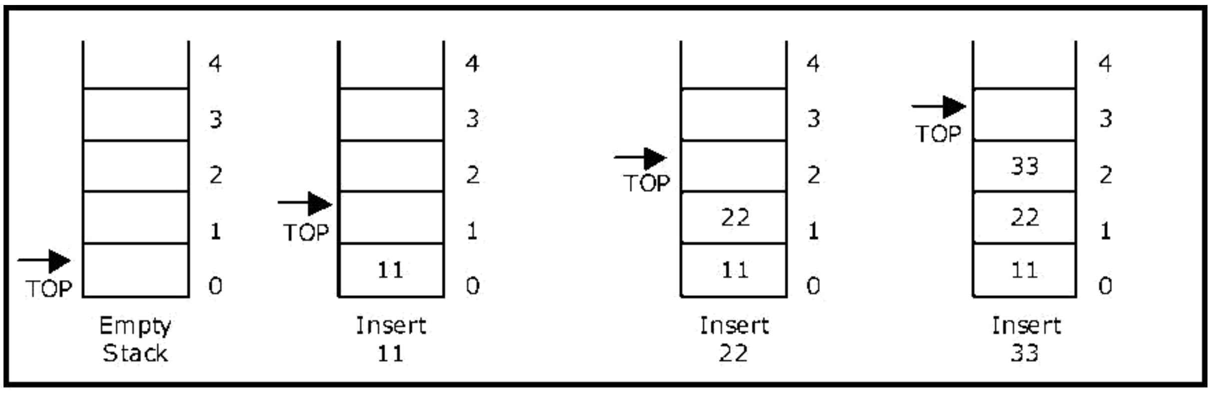


Figure: Push operations on stack

When an element is taken off from the stack, the operation is performed by pop(). The following figure shows a stack initially with three elements and shows the deletion of elements using pop().

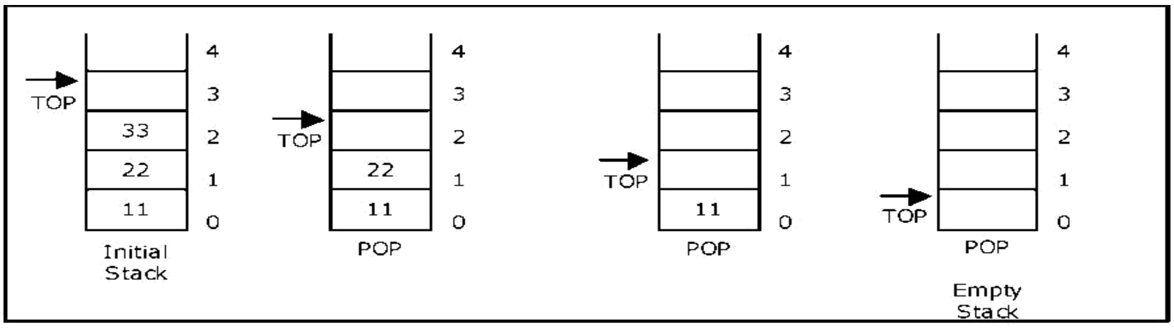


Figure: Pop operations on stack

Linked List Implementation of Stack:

We can represent a stack as a linked list. In a stack push and pop operations are performed at one end called top. We can perform similar operations at one end of list using top pointer. The linked stack looks as shown in the following figure.

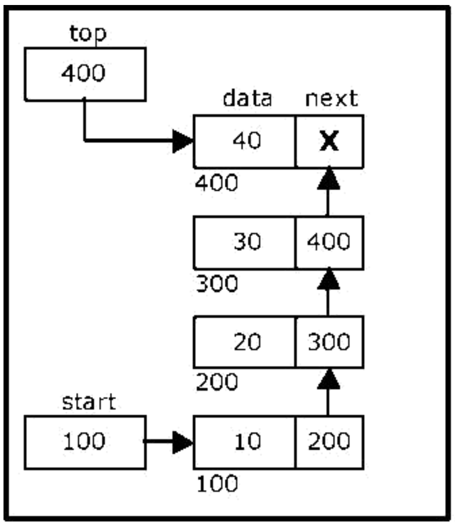


Figure: Linked Stack Representation

Algebraic Expressions:

An algebraic expression is a legal combination of operators and operands. Operand is the quantity on which a mathematical operation is performed. Operand may be a variable like x, y, z or a constant like 5, 4, 6 etc. Operator is a symbol which signifies a mathematical or logical operation between the operands. Examples of familiar operators include +, -, *, /, ^ etc.

An algebraic expression can be represented using three different notations. They are infix, postfix and prefix notations:

Infix: It is the form of an arithmetic expression in which we fix (place) thearithmetic operator in between the two operands.

Example: (A + B) * (C - D)

Prefix: It is the form of an arithmetic notation in which we fix (place) the arithmeticoperator before (pre) its two operands. The prefix notation is called as polish notation (due to the polish mathematician Jan Lukasiewicz in the year 1920).

Example: * + A B - C D

Postfix: It is the form of an arithmetic expression in which we fix (place) thearithmetic operator after (post) its two operands. The postfix notation is called as suffix notation and is also referred to reverse polish notation.

Example: A B + C D - *

The three important features of postfix expression are:

1. The operands maintain the same order as in the equivalent infix

   expression.
2. The parentheses are not needed to designate the expression un-ambiguously.

While evaluating the postfix expression the priority of the operators is no longer relevant.

The following table briefly tries to show the difference in all three notations –

Sr. No.	Infix Notation	Prefix Notation	Postfix Notation
1	a + b	+ a b	a b +
2	(a + b) * c	* + a b c	a b + c *
3	a * (b + c)	* a + b c	a b c + *
4	a / b + c / d	+ / a b / c d	a b / c d / +
5	(a + b) * (c + d)	* + a b + c d	a b + c d + *
6	((a + b) * c) - d	- * + a b c d	a b + c * d -

Parsing Expressions:

As we have discussed, it is not a very efficient way to design an algorithm or program to parse infix notations. Instead, these infix notations are first converted into either postfix or prefix notations and then computed.

To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

Precedence:

When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example –

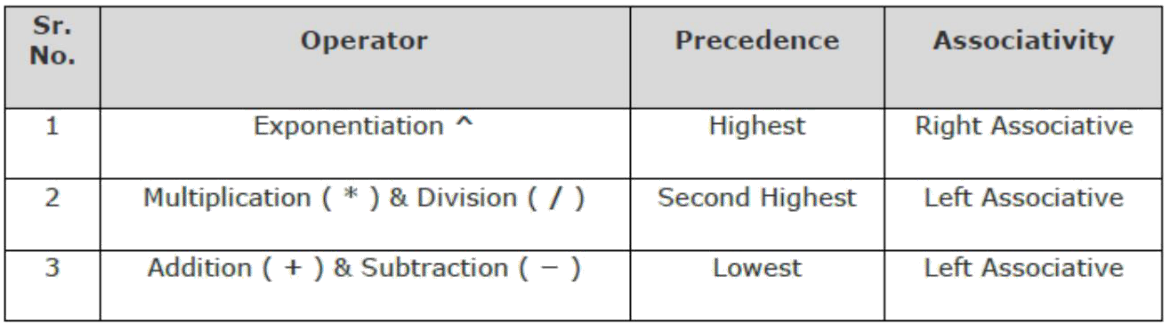
$$a + b * c \Rightarrow a + (b * c)$$

As multiplication operation has precedence over addition, b * c will be evaluated first. A table of operator precedence is provided later.

Associativity:

Associativity describes the rule where operators with the same precedence appear in an expression. For example, in expression a+b−c, both + and – have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both + and − are left associative, so the expression will be evaluated as **(a+b)−c**.

Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest) –



Sr. No.	Operator	Precedence	Associativity
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (*) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (−)	Lowest	Left Associative

The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis. For example –

In **a+b*c**, the expression part **b*c** will be evaluated first, with multiplication asprecedence over addition. We here use parenthesis for **a+b** to be evaluated first, like **(a+b)*c**.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations

- **push()** − Pushing (storing) an element on the stack.
- **pop()** − Removing (accessing) an element from the stack.

- **Algorithm for PUSH Operation**

```
Step 1: If TOP >= SIZE – 1 then
            Write —Stack is Overflowll
Step 2: TOP = TOP + 1
Step 3: STACK [TOP] = X
```

- **Algorithm for POP Operation**

```
Step 1: If TOP = -1 then
            Write —Stack is Underflowll
Step 2: Return STACK [TOP]
Step 3: TOP = TOP - 1
```

Lecture 12

Converting expressions using Stack:

Let us convert the expressions from one type to another. These are done as follows:

1. Infix to postfix
2. Infix to prefix
3. Postfix to infix
4. Postfix to prefix
5. Prefix to infix
6. Prefix to postfix

We will only do the first one.

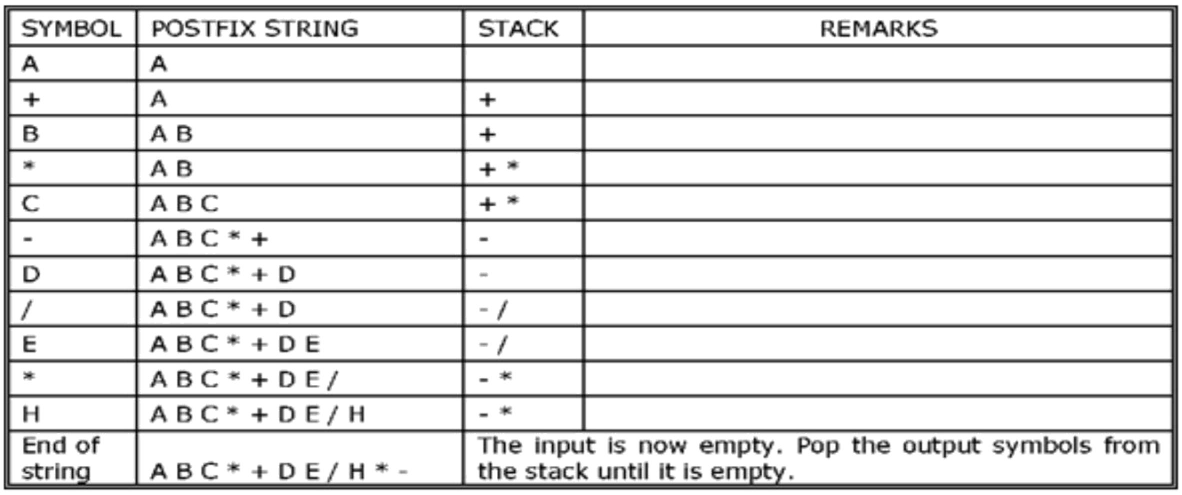
Conversion from infix to postfix:

Procedureto convert from infix expression to postfix expression is as follows:

1. Scan the infix expression from left to right.
2. If the scanned symbol is left parenthesis, push it onto the stack.
3. If the scanned symbol is an operand, then place directly in the postfix expression (output).
4. If the symbol scanned is a right parenthesis, then go on popping all the items from the stack and place them in the postfix expression till we get the matching left parenthesis.
5. If the scanned symbol is an operator, then go on removing all the operators from the stack and place them in the postfix expression, if and only if the precedence of the operator which is on the top of the stack is greater than (or greater than or equal) to the precedence of the scanned operator and push the scanned operator onto the stack otherwise, push the scanned operator onto the stack.

Example 1:

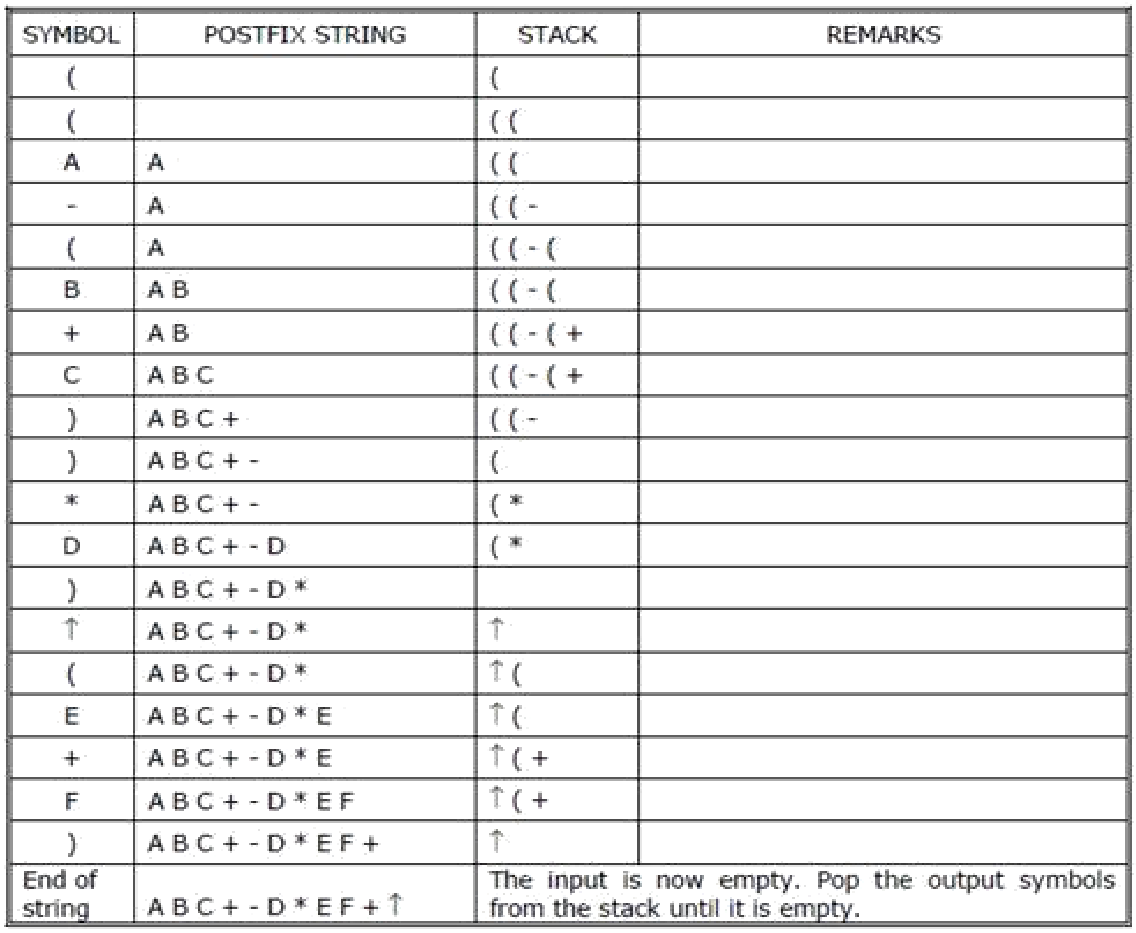
Convert the following infix expression A + B * C - D / E * H into its equivalent postfix expression.



SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	
B	A B	+	
*	A B	+ *	
C	A B C	+ *	
-	A B C * +	-	
D	A B C * + D	-	
/	A B C * + D	- /	
E	A B C * + D E	- /	
*	A B C * + D E /	- *	
H	A B C * + D E / H	- *	
End of string	A B C * + D E / H * -	The input is now empty. Pop the output symbols from the stack until it is empty.	

Example 2:

Convert ((A – (B + C)) * D) ↑(E + F) infix expression to postfix form:



SYMBOL	POSTFIX STRING	STACK	REMARKS
(		(	
(		((	
A	A	((	
-	A	((-	
(	A	((- (	
B	A B	((- (	
+	A B	((- (+	
C	A B C	((- (+	
)	A B C +	((-	
)	A B C + -	(	
*	A B C + -	(*	
D	A B C + - D	(*	
)	A B C + - D *		
↑	A B C + - D *	↑	
(	A B C + - D *	↑ (	
E	A B C + - D * E	↑ (	
+	A B C + - D * E	↑ (+	
F	A B C + - D * E F	↑ (+	
)	A B C + - D * E F +	↑	
End of string	A B C + - D * E F + ↑	The input is now empty. Pop the output symbols from the stack until it is empty.	

Evaluation of postfix expression:

The postfix expression is evaluated easily by the use of a stack. When a number is seen, it is pushed onto the stack; when an operator is seen, the operator is applied to the two numbers that are popped from the stack and the result is pushed onto the stack. When an expression is given in postfix notation, there is no need to know any precedence rules; this is our obvious advantage.

Postfix Expression Evaluation using Stack Data Structure

A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using Stack data structure we can use the following steps...

1. Read all the symbols one by one from left to right in the given Postfix Expression
2. If the reading symbol is operand, then push it on to the Stack.
3. If the reading symbol is operator (+ , - , * , / etc.,), then perform TWO pop operations and store the two popped oparands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand2 and operand1 and push result back on to the Stack.
4. Finally! Perform a pop operation and display the popped value as final result.

Example:

Evaluate the following postfix expression: 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
↑	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

Applications of stacks:

- Stack is used by compilers to check for balancing of parentheses, brackets and braces.
- Stack is used to evaluate a postfix expression.

- Stack is used to convert an infix expression into postfix/prefix form.
- In recursion, all intermediate arguments and return values are stored on the processor's stack.
- During a function call the return address and arguments are pushed onto a stack and on return they are popped off.

Lecture 13

QUEUE:

A queue is another special kind of list, where items are inserted at one end called the rear and deleted at the other end called the front. Another name for a queue is a―FIFO‖ or ―First-in-first-out‖ list.

The operations for a queue are analogues to those for a stack, the difference is that the insertions go at the end of the list, rather than the beginning. We shall use the following operations on queues:

- *enqueue*: which inserts an element at the end of the queue.
- *dequeue*: which deletes an element at the start of the queue.

Figure: A real-world example of queue

Representation of Queue:

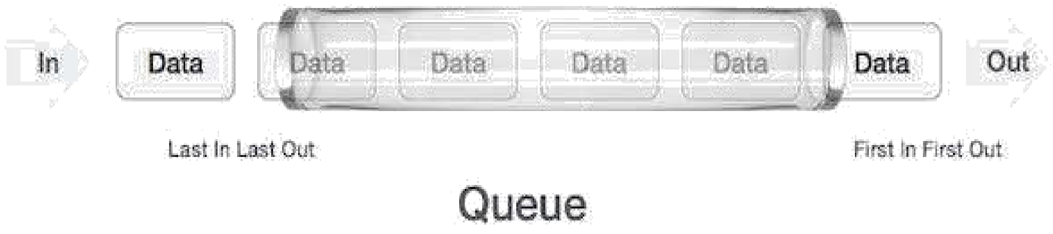
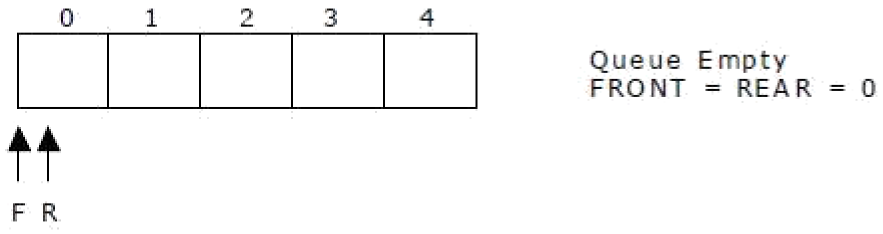
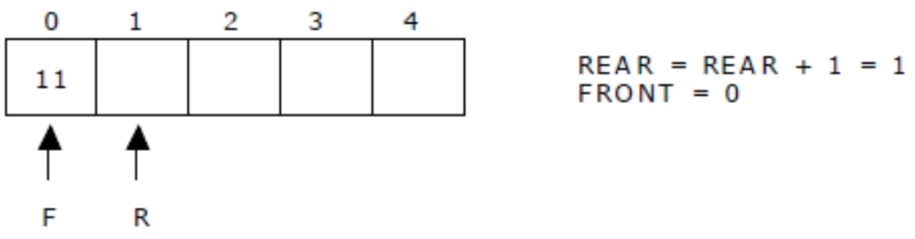


Figure: A queue representation

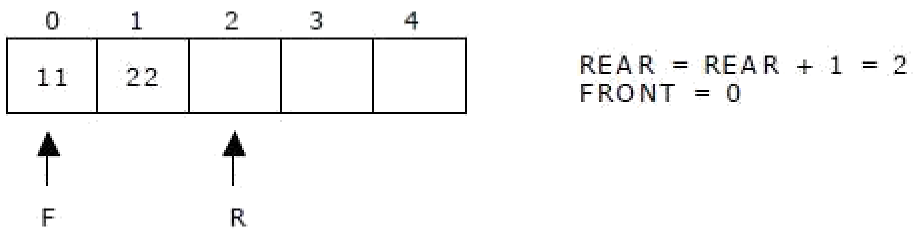
Let us consider a queue, which can hold maximum of five elements. Initially the queueis empty.



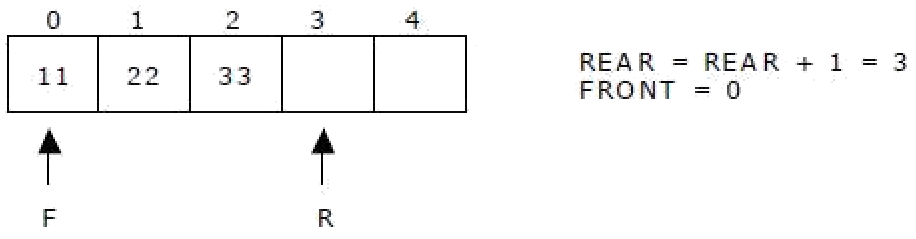
Now, insert 11 to the queue. Then queue status will be:



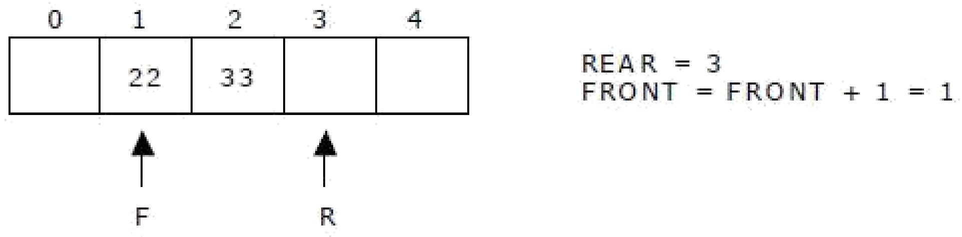
Next, insert 22 to the queue. Then the queue status is:



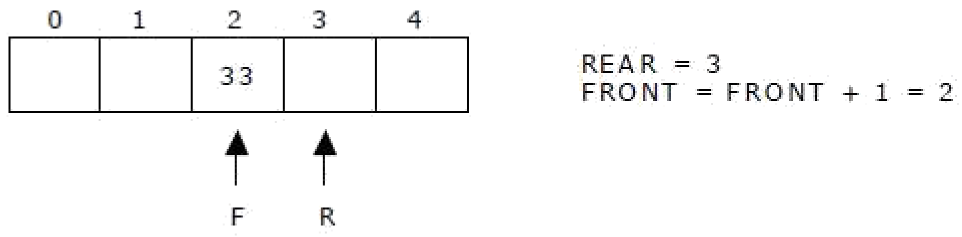
Again insert another element 33 to the queue. The status of the queue is:



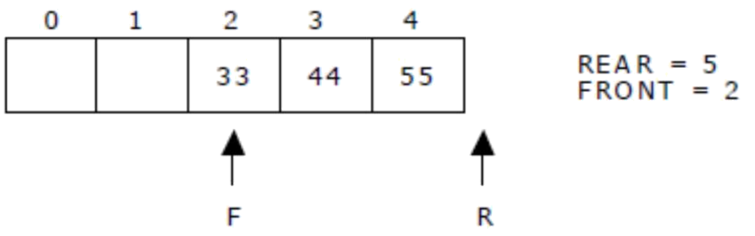
Now, delete an element. The element deleted is the element at the front of the queue. So the status of the queue is:



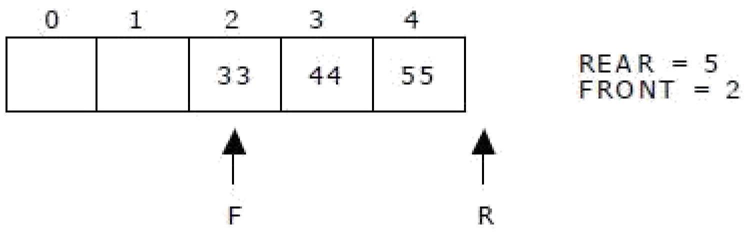
Again, delete an element. The element to be deleted is always pointed to by the FRONTpointer. So, 22 is deleted. The queue status is as follows:



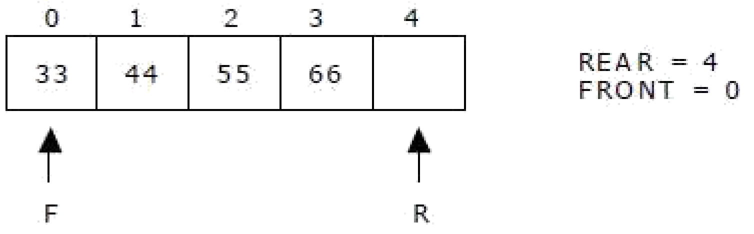
Now, insert new elements 44 and 55 into the queue. The queue status is:



Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:



Now it is not possible to insert an element 66 even though there are two vacant positions in the linear queue. To overcome this problem the elements of the queue are to be shifted towards the beginning of the queue so that it creates vacant position at the rear end. Then the FRONT and REAR are to be adjusted properly. The element 66 can be inserted at the rear end. After this operation, the queue status is as follows:



This difficulty can overcome if we treat queue position with index 0 as a position that comes after position with index 4 i.e., we treat the queue as a **circular queue.**

Linked List Implementation of Queue:

We can represent a queue as a linked list. In a queue data is deleted from the front end and inserted at the rear end. We can perform similar operations on the two ends of a list. We use two pointers front and rear for our linked queue implementation. The linked queue looks as shown in the following figure:

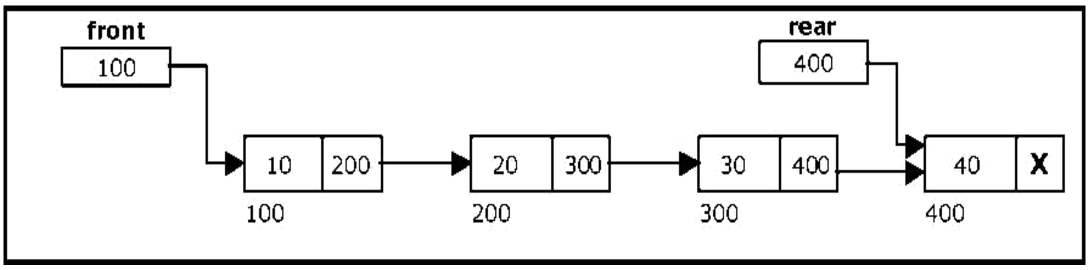


Figure: Linked Queue representation

Applications of Queue:

- It is used to schedule the jobs to be processed by the CPU.
- When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out (FIFO) basis.
- Breadth first search uses a queue data structure to find an element from a graph.

Insert Operation

Insert operation is used to insert an element into Queue. In order to insert an element into Queue first we have to check weather space is available in the Queue or not. If Queue is full then we cannot insert an element into Queue. If value of REAR variable is greater than or equal to SIZE – 1 then we cannot insert an element into Queue. This condition is known as ―Overflow‖. If Queue is not overflow then we can insert an element into Queue. First we have to increment the value of REAR variable by one and then insert an element into Queue. If the element which is inserted in Queue is first element then we have to set the value of FRONT variable to 0.

```
Step 1: If REAR >= SIZE – 1 then
                Write ―Queue is Overflow‖
Step 2: REAR = REAR + 1
Step 3: QUEUE [REAR] = X
Step 4: If FRONT = -1 then
                FRONT = 0
```

Delete Operation

Delete operation is used to delete an element from Queue. In order to delete an element from Queue first we have to check weather Queue is empty or not. If Queue is empty then we cannot delete an element from Queue. This condition is known as ―Underflow‖. If Queue is not underflow then we can delete element from Queue. After deleting element from Queue we have to set the value of FRONT and REAR variables according to the elements in the Queue.

```
Step 1: If FRONT = -1 then
        Write ―Queue is Underflow‖
Step 2: Return QUEUE [FRONT]
Step 3: If FRONT = REAR then
            FRONT = 0
            REAR = 0
            Else
            FRONT = FRONT + 1
```

Lecture 14

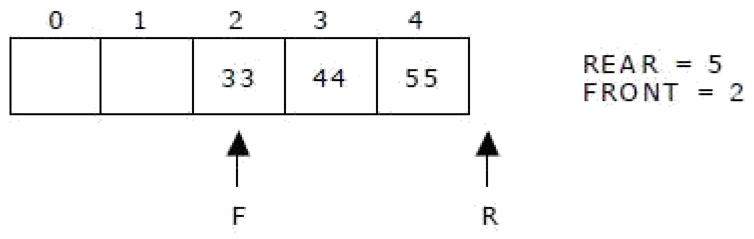
Circular Queue:

A more efficient queue representation is obtained by regarding the array Q[MAX] as circular. Any number of items could be placed on the queue. This implementation of a queue is called a circular queue because it uses its storage array as if it were a circle instead of a linear list.

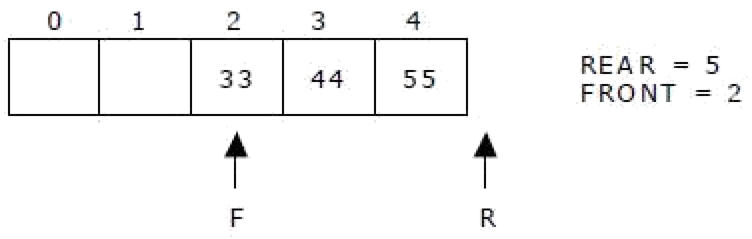
There are two problems associated with linear queue. They are:

- Time consuming: linear time to be spent in shifting the elements to the beginning of the queue.
- Signalling queue full: even if the queue is having vacant position.

For example, let us consider a linear queue status as follows:



Next insert another element, say 66 to the queue. We cannot insert 66 to the queue asthe rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:

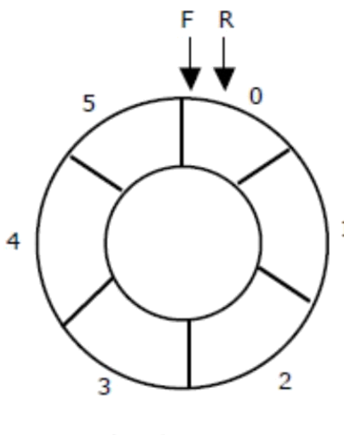


This difficulty can be overcome if we treat queue position with index zero as a position that comes after position with index four then we treat the queue as a **circular queue.**

In circular queue if we reach the end for inserting elements to it, it is possible to insert new elements if the slots at the beginning of the circular queue are empty.

Representation of Circular Queue:

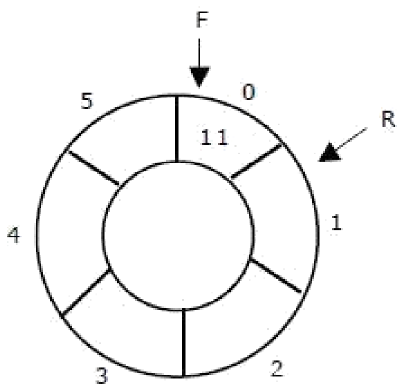
Let us consider a circular queue, which can hold maximum (MAX) of six elements. Initially the queue is empty.



Circular Queue

Queue Empty
MAX = 6
FRONT = REAR = 0
COUNT = 0

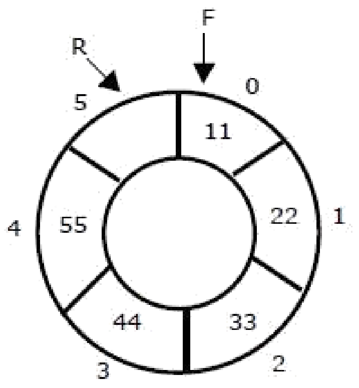
Now, insert 11 to the circular queue. Then circular queue status will be:



Circular Queue

FRONT = 0
REAR = (REAR + 1) % 6 = 1
COUNT = 1

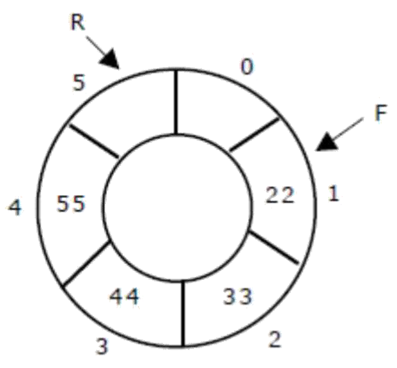
Insert new elements 22, 33, 44 and 55 into the circular queue. The circular queue status is:



Circular Queue

FRONT = 0
REAR = (REAR + 1) % 6 = 5
COUNT = 5

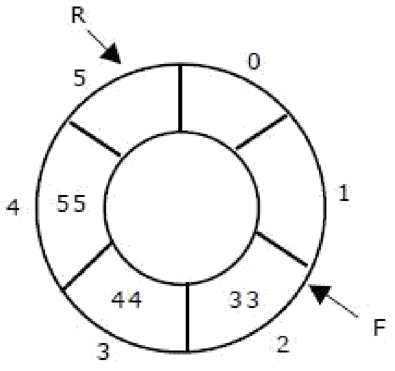
Now, delete an element. The element deleted is the element at the front of the circularqueue. So, 11 is deleted. The circular queue status is as follows:



Circular Queue

```
FRONT = (FRONT + 1) % 6 = 1
REAR = 5
COUNT = COUNT - 1 = 4
```

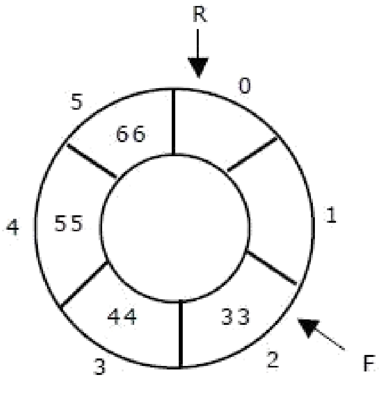
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The circular queue status is as follows:



Circular Queue

```
FRONT = (FRONT + 1) % 6 = 2
REAR = 5
COUNT = COUNT - 1 = 3
```

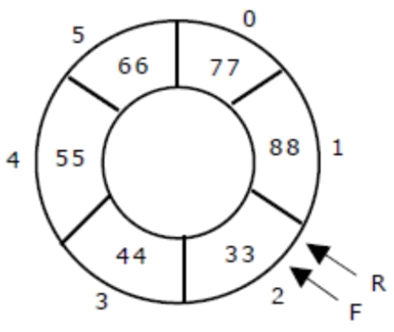
Again, insert another element 66 to the circular queue. The status of the circular queue is:



Circular Queue

```
FRONT = 2
REAR = (REAR + 1) % 6 = 0
COUNT = COUNT + 1 = 4
```

Now, insert new elements 77 and 88 into the circular queue. The circular queue status is:



Circular Queue

FRONT = 2, REAR = 2
REAR = REAR % 6 = 2
COUNT = 6

Now, if we insert an element to the circular queue, as COUNT = MAX we cannot add the element to circular queue. So, the circular queue is *full*.

Algorithm for Insert Operation

```
Step 1: If REAR = SIZE-1 then
REAR = 0
Else
REAR=REAR + 1
Step 2: If FRONT = REAR then
Write ("Circular Queue Overflow")
Step 3: CQ[REAR]=X
Step 4: If FRONT = -1 then
FRONT=0
```

Algorithm for Delete Operation

```
Step 1: If FRONT = -1 then
Write ("Circular Queue Underflow")
Step 2: Return (CQ [FRONT])
Step 3: If FRONT = REAR then
FRONT=REAR=-1
Step 4: If FRONT = SIZE-1 then
FRONT=0
Else
FRONT=FRONT+1
```

DeQueue:

In the preceding section we saw that a queue in which we insert items at one end and from which we remove items at the other end. In this section we examine an extension of the queue, which provides a means to insert and remove items at both ends of the queue. This data structure is a *dequeue*. The word *dequeue* is an acronym derived from *double-ended queue*. The following figure shows the representation of a dequeue.

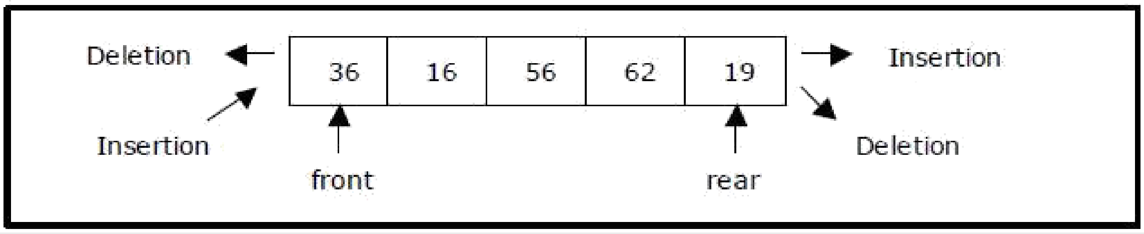


Figure 9. Representation of a deque

A deque provides four operations. Figure 4.6 shows the basic operations on a deque.

- enqueue_front: insert an element at front.
- dequeue_front: delete an element at front.
- enqueue_rear: insert element at rear.
- dequeue_rear: delete element at rear.

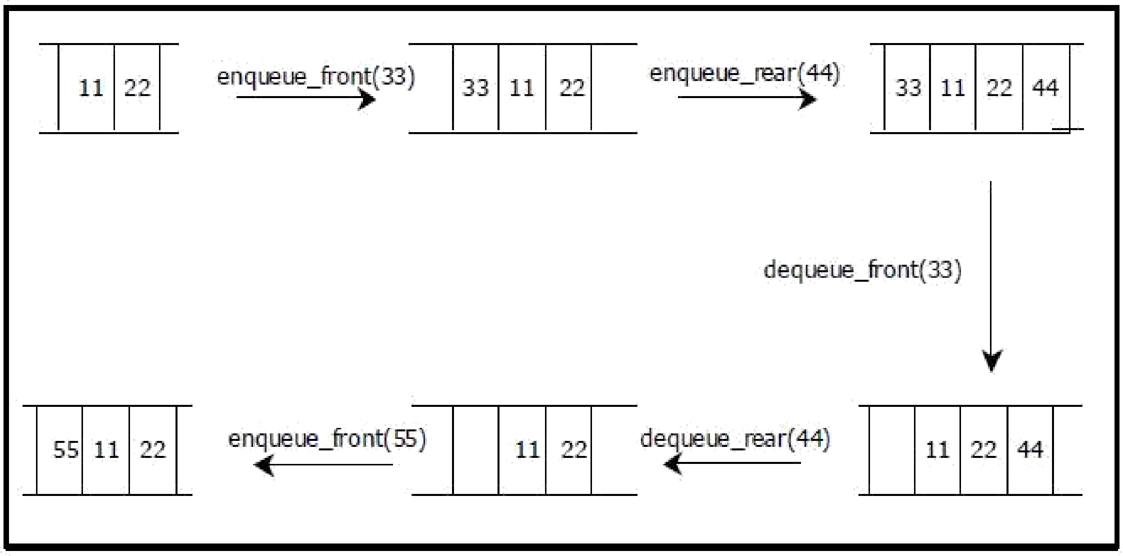


Figure: Basic operations on deque

There are two variations of deque. They are:

- Input restricted deque (IRD)
- Output restricted deque (ORD)

An Input restricted deque is a deque, which allows insertions at one end but allows deletions at both ends of the list.

An output restricted deque is a deque, which allows deletions at one end but allows insertions at both ends of the list.

Algorithm to add an element into DeQueue

Assumptions: pointer f,r and initial values are -1,-1

Q[] is an array

max represent the size of a queue

enq_front

```
step1. Start
step2. Check the queue is full or not as if (f <>
step3. If false update the pointer f as f= f-1
step4. Insert the element at pointer f as Q[f] = element
step5. Stop
```

enq_back

```
step1. Start
step2. Check the queue is full or not as if (r == max-1) if yes queue is full
step3. If false update the pointer r as r= r+1
step4. Insert the element at pointer r as Q[r] =
element step5. Stop
```

Algorithm to delete an element from the DeQueue

deq_front

```
step1. Start
step2. Check the queue is empty or not as if (f == r) if yes queue is empty
step3. If false update pointer f as f = f+1 and delete element at position f
as element = Q[f]
step4. If ( f== r) reset pointer f and r as f=r=-1
step5. Stop
```

deq_back

```
step1. Start
step2. Check the queue is empty or not as if (f == r) if yes queue is empty
step3. If false delete element at position r as element = Q[r] step4.
Update pointer r as r = r-1
step5. If (f == r ) reset pointer f and r as f = r= -1
step6. Stop
```

Lecture 15

Recursion

Introduction to Recursion:

A function is recursive if a statement in the body of the function calls itself. Recursion is the process of defining something in terms of itself. For a computer language to be recursive, a function must be able to call itself.

For example, let us consider the function factorial() shown below, which computes the factorial of an integer.

```
#include <stdio.h>
int factorial (int);
main()
{
        intnum, fact;
        printf (“Enter a positive integer value: ");
        scanf (“%d”, &num);
        fact = factorial (num);
        printf ("\n Factorial of %d =%5d\n", num, fact);
}
int factorial (int n)
{
        int result;
        if (n == 0)
                return (1);
        else
                result = n * factorial (n-1);
        return (result);
}
```

A non-recursive or iterative version for finding the factorial is as follows:

```
factorial (int n)
{
        inti, result = 1;
        if (n == 0)
                return (result);
        else
        {
                for (i=1; i<=n; i++)
                        result = result * i;
        }
        return (result);
}
```

The operation of the non-recursive version is clear as it uses a loop starting at 1 and ending at the target value and progressively multiplies each number by the moving product.

When a function calls itself, new local variables and parameters are allocated storage on the stack and the function code is executed with these new variables from the start. A recursive call does not make a new copy of the function. Only the arguments and variables are new. As each recursive call returns, the old local variables and parameters are removed from the stack and execution resumes at the point of the function call inside the function.

When writing recursive functions, you must have an exit condition somewhere to force the function to return without the recursive call being executed. If you do not have an exit condition, the recursive function will recurse forever until you run out of stack space and indicate error about lack of memory, or stack overflow.

What is base condition in recursion?

In recursive program, the solution to base case is provided and solution of bigger problem is expressed in terms of smaller problems.

```
int fact(int n)
{
   if (n < = 1) // base case
      return 1;
   else
      return n*fact(n-1);
}
```

In the above example, base case for n < = 1 is defined and larger value of number can be solved by converting to smaller one till base case is reached.

Differences between recursion and iteration:

- Both involve repetition
- Both involve a termination test
- Both can occur infinitely

Iteration	Recursion
Iteration explicitly user a repetition structure.	Recursion achieves repetition through repeated function calls.
Iteration terminates when the loop continuation.	Recursion terminates when a base case is recognized.
Iteration keeps modifying the counter until the loop continuation condition fails.	Recursion keeps producing simple versions of the original problem until the base case is reached.
Iteration normally occurs within a loop so the extra memory assigned is omitted.	Recursion causes another copy of the function and hence a considerable memory space’s occupied.
It reduces the processor’s operating time.	It increases the processor’s operating time.

Factorial of a given number:

The operation of recursive factorial function is as follows:

Start out with some natural number N (in our example, 5). The recursive definition is:

n = 0, 0 ! = 1	*Base Case*
n> 0, n ! = n * (n - 1) !	*Recursive Case*

Recursion Factorials:

5! =5 * 4! = 5 *___ = ____ = __	factr(5) = 5 * factr(4)
4! = 4 *3! = 4 *___ = ___ = __	factr(4) = 4 * factr(3)
3! = 3 * 2! = 3 * ___ = ___ = __	factr(3) = 3 * factr(2)
2! = 2 * 1! = 2 * ___ = ___ = __	factr(2) = 2 * factr(1)
1! = 1 * 0! = 1 * __ = __ = __	factr(1) = 1 * factr(0)
0!=1	factr(0)=__

5! = 5*4! = 5*4*3! = 5*4*3*2! = 5*4*3*2*1! = 5*4*3*2*1*0! = 5*4*3*2*1*1 =120

We define 0! to equal 1, and we define factorial N (where N > 0), to be N * factorial (N-1). All recursive functions must have an exit condition, that is a state when it does not recurse upon itself. Our exit condition in this example is when N = 0.

Tracing of the flow of the factorial () function:

When the factorial function is first called with, say, N = 5, here is what happens:

FUNCTION:
Does N = 0? No
Function Return Value = 5 * factorial (4)

At this time, the function factorial is called again, with N = 4.

FUNCTION:
Does N = 0? No
Function Return Value = 4 * factorial (3)

At this time, the function factorial is called again, with N = 3.

FUNCTION:
Does N = 0? No
Function Return Value = 3 * factorial (2)

At this time, the function factorial is called again, with N = 2.

FUNCTION:
Does N = 0? No

Function Return Value = 2 * factorial (1)

At this time, the function factorial is called again, with N = 1.

FUNCTION:
Does N = 0? No
Function Return Value = 1 * factorial (0)

At this time, the function factorial is called again, with N = 0.

FUNCTION:
Does N = 0? Yes
Function Return Value = 1

Now, we have to trace our way back up! See, the factorial function was called six times. At any function level call, all function level calls above still exist! So, when we have N = 2, the function instances where N = 3, 4, and 5 are still waiting for their return values.

So, the function call where N = 1 gets retraced first, once the final call returns 0. So, the function call where N = 1 returns 1*1, or 1. The next higher function call, where N = 2, returns 2 * 1 (1, because that's what the function call where N = 1 returned). You just keep working up the chain.

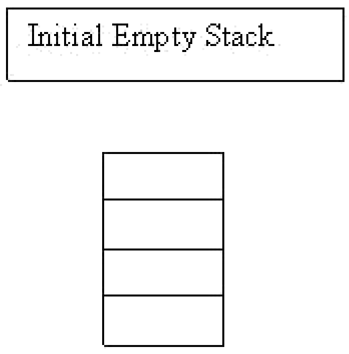
When N = 2, 2 * 1, or 2 was returned.
When N = 3, 3 * 2, or 6 was returned.
When N = 4, 4 * 6, or 24 was returned.
When N = 5, 5 * 24, or 120 was returned.

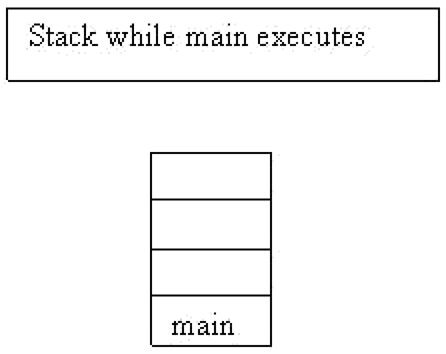
And since N = 5 was the first function call (hence the last one to be recalled), the value 120 is returned.

Use of Stack:

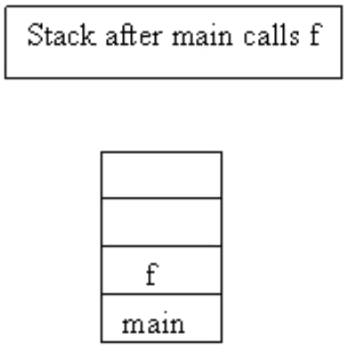
Let us explain the program stack and then show how that applies to recursion.

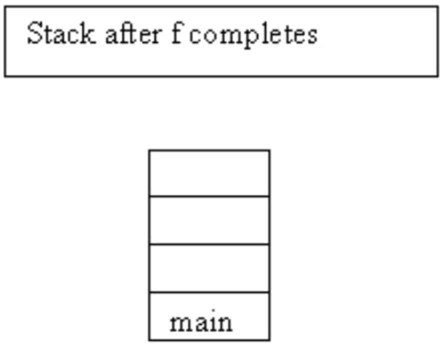
Every executable’s main is loaded into a program stack at the beginning of execution. It remains there until it completes, at which time it is popped off of the stack.



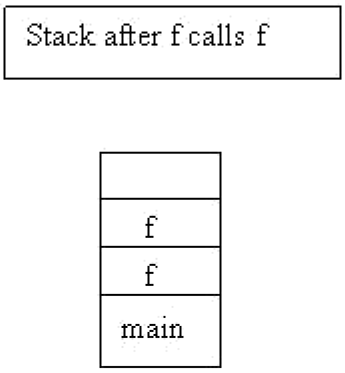


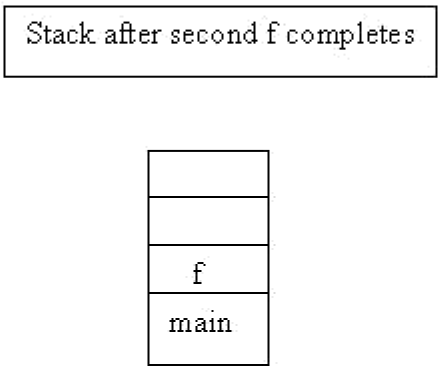
If main calls a function, that function is loaded onto the top of the stack and remains there until it is complete at which point it is popped off of the stack.





Now, a recursive function calls itself. That means another instance of the function will be placed on the stack and will remain there until the function completes.

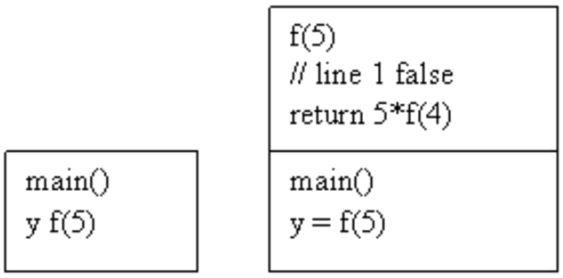




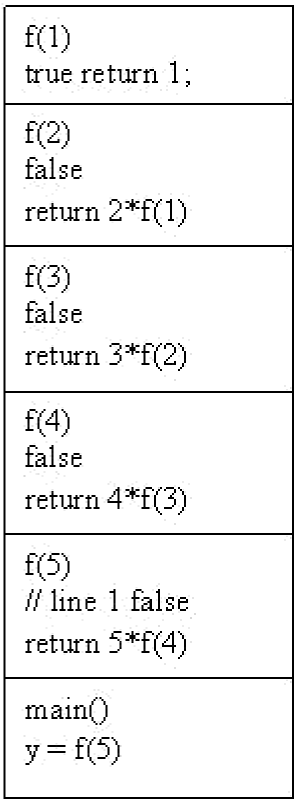
You need to look at a recursive function in terms of the program stack. Let’s use the factorial again as an example. Factorial of 5 is 5x4x3x2x1 = 120 and this can be implemented recursively.

```
int f(int x)
{
            if(x == 1)
      return 1; return       // line 1
             f(x-1)*x;       // line 2
}
void main()
{
      int y = f(5);   // main call
      // y should get 120
}
```

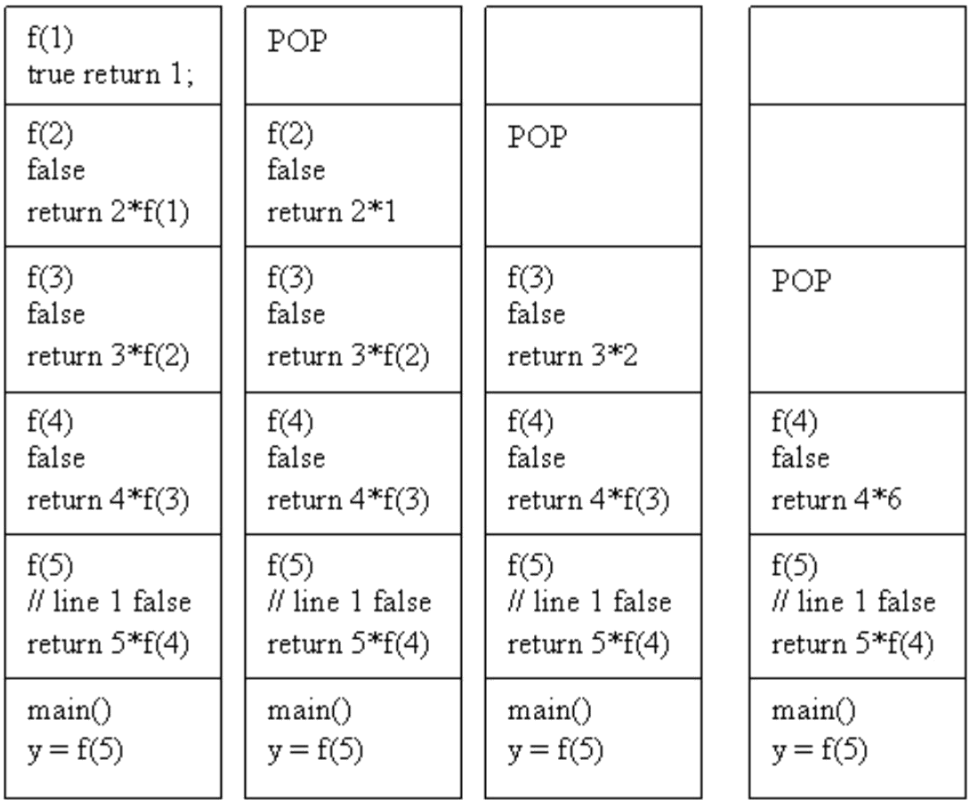
So let’s watch the stack and see what happens.

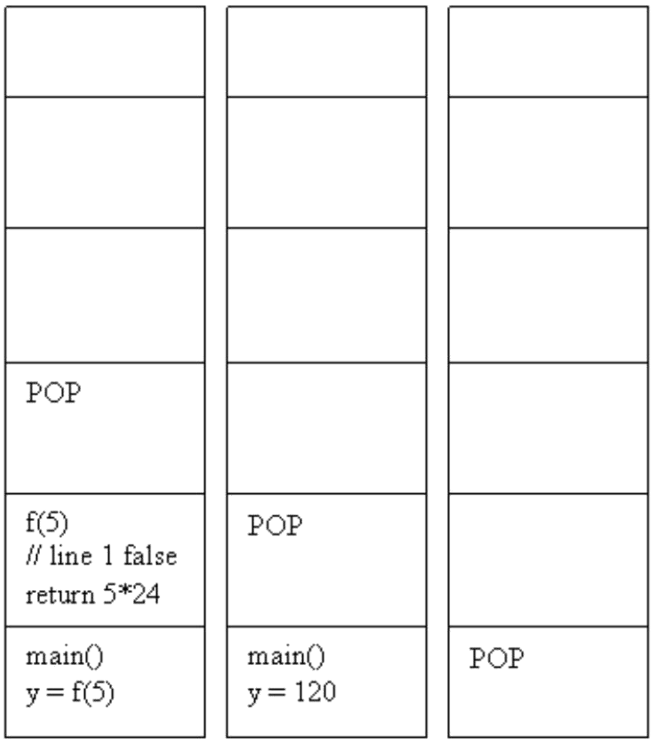


main calls the function f with a value of 5, so on the stack we get f(5). Line 1 is false so we go to line 2 which calls f(4), etc. Note that f(4) must complete before f(5) can complete. f(5) will complete by returning 5*f(4). The stack will look like:



So at this point none of the functions have yet returned! The first to return will be f(1) which will return 1. Then f(2) will return 2. Then f(3) will return 6. As in:


f(1)
true return 1;
f(2)
false
return 2*f(1)
f(3)
false
return 3*f(2)
f(4)
false
return 4*f(3)
f(5)
// line 1 false
return 5*f(4)
main()
y = f(5)
POP
f(2)
false
return 2*1
f(3)
false
return 3*f(2)
f(4)
false
return 4*f(3)
f(5)
// line 1 false
return 5*f(4)
main()
y = f(5)
POP
f(3)
false
return 3*2
f(4)
false
return 4*f(3)
f(5)
// line 1 false
return 5*f(4)
main()
y = f(5)
POP
f(4)
false
return 4*6
f(5)
// line 1 false
return 5*f(4)
main()
y = f(5)


POP
f(5)
// line 1 false
return 5*24
main()
y = f(5)
POP
main()
y = 120
POP

Tail Recursion:

Tail Recursion is another form of linear recursion, where the function makes a recursive call as its very last operation. Note that, there is a difference between last operation and last statement.

Therefore, a function will be said tail recursive if the recursive call is the last thing the function performs. When a function uses tail recursion, it can be converted to the non-recursive one, by iterating through the recursive calls rather than calling them explicitly. Reversing array elements could be a good example of tail recursion. The recursive solution of reversing array elements problem is illustrated as below.

Suppose, we are given an array of decimal integers, and we have to reverse the order of elements in which they are stored. We can solve this problem by using linear recursion. Reversal of an array can be achieved by swapping the first and last elements and then recursively reversing the remaining elements in the array.

```
#include <stdio.h>
#define SIZE 10

voidrecursiveRev (int *, int, int);

int main ()
{
        intarr[SIZE] = {1, 2, 3, 4, 5, 6, 7, 8, 9 ,
        10}; inti;
        recursiveRev (arr, 0, SIZE-1);

        printf ("Printing array elements after reversing
        them...\n"); for (i = 0; i< SIZE; i++)
                printf ("%d\t", arr[i]);
}

voidrecursiveRev (int * arr, inti, int j)
{
        if(i< j)
        {
                inttmp;
                tmp = arr[i];
                arr[i] = arr[j];
                arr[j] = tmp;
                recursiveRev (arr, i+1, j-1);
         }
}
```
OUTPUT
```
Printing array elements after reversing them...
10      9       8       7       6       5       4       3       2       1
```

Lecture 16

The Towers of Hanoi:

In the game of Towers of Hanoi, there are three towers labelled 1, 2, and 3. The game starts with n disks on tower A. For simplicity, let n is 3. The disks are numbered from 1 to 3, and without loss of generality we may assume that the diameter of each disk is the same as its number. That is, disk 1 has diameter 1 (in some unit of measure), disk 2 has diameter 2, and disk 3 has diameter 3. All three disks start on tower A in the order 1, 2, 3. The objective of the game is to move all the disks in tower 1 to entire tower 3 using tower 2. That is, at no time can a larger disk be placed on a smaller disk.

The following figure, illustrates the initial setup of towers of Hanoi. The next figure, illustrates the final setup of towers of Hanoi.

The rules to be followed in moving the disks from tower 1 tower 3 using tower 2 are as follows:

- Only one disk can be moved at a time.
- Only the top disc on any tower can be moved to any other tower.
- A larger disk cannot be placed on a smaller disk.

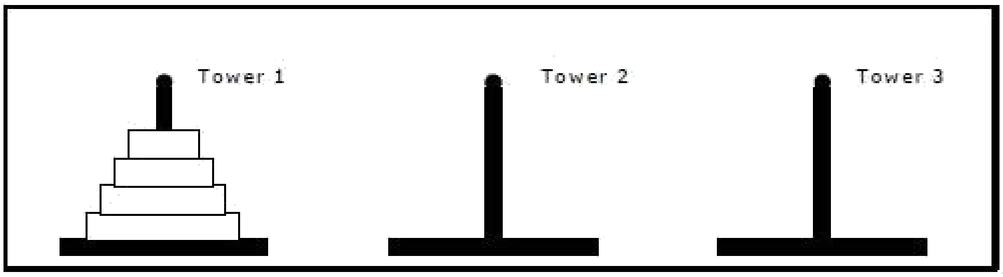


Figure 1. Initial Setup of Towers of Hanoi

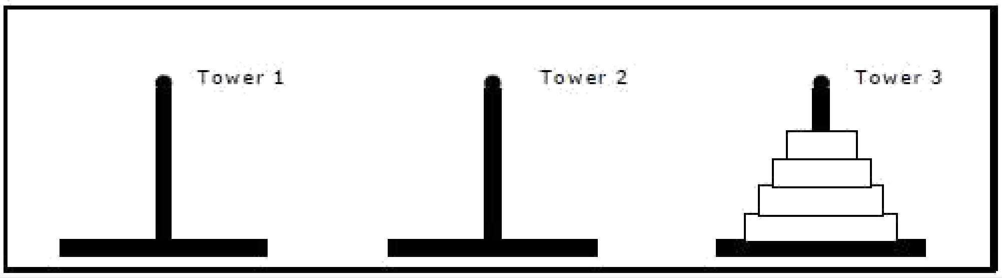


Figure: Final Setup of Towers of Hanoi

The towers of Hanoi problem can be easily implemented using recursion. To move the largest disk to the bottom of tower 3, we move the remaining n – 1 disks to tower 2 and then move the largest disk to tower 3. Now we have the remaining n – 1 disks to be moved to tower 3. This can be achieved by using the remaining two towers. We can also use tower 3 to place any disk on it, since the disk placed on tower 3 is the largest disk and continue the same operation to place the entire disks in tower 3 in order.

FUNCTION MoveTower(*disk*, *source*, *dest*, *spare*):

```
IF disk == 0, THEN:
    move disk from source to dest
ELSE:
    MoveTower(disk - 1, source, spare, dest)   // Step 1 above
    move disk from source to dest              // Step 2 above
    MoveTower(disk - 1, spare, dest, source)   // Step 3 above
END IF
```

The program that uses recursion to produce a list of moves that shows how to accomplish the task of transferring the n disks from tower 1 to tower 3 is as follows:

```
#include <stdio.h>
#include <conio.h>
void towers_of_hanoi (int n, char *a, char *b, char *c);
int cnt=0;
int main (void)
{
        int n;
        printf("Enter number of discs: ");
        scanf("%d",&n);
        towers_of_hanoi (n, "Tower 1", "Tower 2", "Tower
        3"); getch();
}
void towers_of_hanoi (int n, char *a, char *b, char *c)
{
        if (n == 1)
        {
                ++cnt;
                printf ("\n%5d: Move disk 1 from %s to %s", cnt, a,
                c); return;
        }
        else
        {
                towers_of_hanoi (n-1, a, c, b);
                ++cnt;
                printf ("\n%5d: Move disk %d from %s to %s", cnt, n, a,
                c); towers_of_hanoi (n-1, b, a, c); return;

        }
}
```

Output of the program:

RUN 1:

```
Enter the number of discs: 3

1: Move disk 1 from tower 1 to tower 3.
2: Move disk 2 from tower 1 to tower 2.
3: Move disk 1 from tower 3 to tower 2.
4: Move disk 3 from tower 1 to tower 3.
5: Move disk 1 from tower 2 to tower 1.
6: Move disk 2 from tower 2 to tower 3.
7: Move disk 1 from tower 1 to tower 3.
```

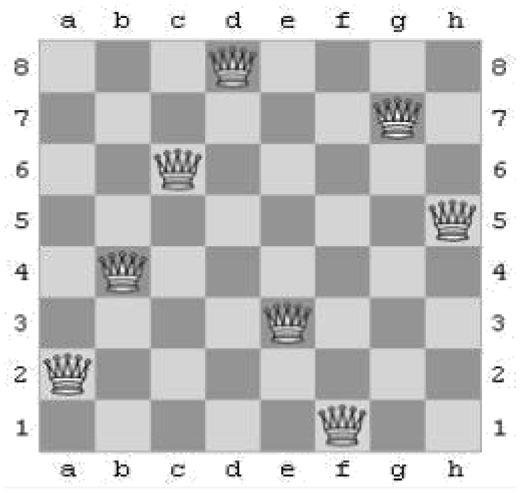
RUN 2:

```
Enter the number of discs: 4

1: Move disk 1 from tower 1 to tower 2.
2: Move disk 2 from tower 1 to tower 3.
3: Move disk 1 from tower 2 to tower 3.
4: Move disk 3 from tower 1 to tower 2.
5: Move disk 1 from tower 3 to tower 1.
6: Move disk 2 from tower 3 to tower 2.
7: Move disk 1 from tower 1 to tower 2.
8: Move disk 4 from tower 1 to tower 3.
9: Move disk 1 from tower 2 to tower 3.
10: Move disk 2 from tower 2 to tower 1.
11: Move disk 1 from tower 3 to tower 1.
12: Move disk 3 from tower 2 to tower 3.
13: Move disk 1 from tower 1 to tower 2.
14: Move disk 2 from tower 1 to tower 3.
15: Move disk 1 from tower 2 to tower 3.
```

Eight Queens Problem

It is the puzzle of placing eight queens on the chess board of 8×8 so that no two queens attack each other. It means no two queens share the same row, column and diagonal. Later it is extended to nxn queens that are placing n queens on nxn board and excludes with 2×2 and 3×3 boards.



It can be done in different combinations, Say for example eight queens can be placed on 8×8 board in 92 combinations. Now we will see how we can solve the problem recursively.

Solving eight-queen problem in C:

```
#include<stdio.h>

intq[50],n;

void printQueens();
int isConsistent(int);
```

```
void enumerate(int);

int main()
{
        printf("Enter the size of board:");
        scanf("%d",&n);
        if(n==2||n==3)
        {
                printf("Can't play the game..");
                return0;
        }
        enumerate(0);
        return0;
}
void enumerate(intc)
{
        inti;
        if(c == n)
                printQueens();
        else
        {
                for(i = 0; i< n; i++)
                {
                        q1 = i;
                        if(isConsistent(c))
                        enumerate(c+1);
                }
        }
}
void printQueens()
{
        staticintcount=1;
        inti,j;
        printf("Combination %d\n",count++);
        for(i=0;i<n;i++)
        {
                for(j=0;j<n;j++)
                        if(q[i]==j)
                                printf("Q ");
                        else
                                printf("* ");
                        printf("\n");
        }
        printf("\n\n");
}
int isConsistent(intc)
{
        inti;
        for(i = 0; i< c; i++)
        {
                if(q[i] == q1)
                        return0;   // same column
                if((q[i] - q1) == (c - i))
                        return0;   // same major diagonal
                if((q1 - q[i]) == (c - i))
                        return0;   // same minor diagonal
        }
        return1;
}
```

Execution:

```
Enter the size of board: 1
Combination 1
Q
```

Execution:

```
Enter the size of board: 3
Can't play the game..
```

Execution:

```
Enter the size of board: 4
Combination 1

* Q * *
* * * Q
Q * * *
* * Q *

Combination 2

* * Q *
Q * * *
* * * Q
* Q * *
```

Execution:

```
Enter the size of board: 8
————
————
Combination 91

* * * * * * * Q
* * Q * * * * *
Q * * * * * * *
* * * * * Q * *
* Q * * * * * *
* * * * Q * * *
* * * * * * Q *
* * * Q * * * *

Combination 92

* * * * * * * Q
* * * Q * * * *
Q * * * * * * *
* * Q * * * * *
* * * * * Q * *
* Q * * * * * *
* * * * * * Q *
* * * * Q * * *
```

Module III

Non- Linear Data structures

Trees & Graphs

Lecture 17

TREES:

A tree is hierarchical collection of nodes. One of the nodes, known as the root, is at the top of the hierarchy. Each node can have at most one link coming into it. The node where the link originates is called the parent node. The root node has no parent. The links leaving a node (any number of links are allowed) point to child nodes. Trees are recursive structures. Each child node is itself the root of a subtree. At the bottom of the tree are leaf nodes, which have no children.

Trees represent a special case of more general structures known as graphs. In a graph, there is no restrictions on the number of links that can enter or leave a node, and cycles may be present in the graph. The figure 5.1.1 shows a tree and a non-tree.

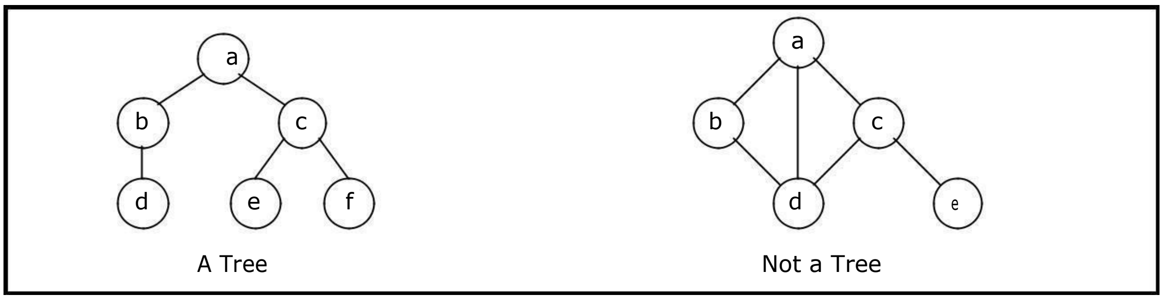


Figure 5.1.1 A Tree and a not a tree

In a tree data structure, there is no distinction between the various children of a node i.e., none is the "first child" or "last child". A tree in which such distinctions are made is called an **ordered tree**, and data structures built on them are called **ordered tree data structures**. Ordered trees are by far the commonest form of tree data structure.

BINARY TREE:

In general, tree nodes can have any number of children. In a binary tree, each node can have at most two children. A binary tree is either **empty** or consists of a node called the **root** together with two binary trees called the **left subtree** and the **right subtree**.

A tree with no nodes is called as a **null** tree. A binary tree is shown in figure 5.2.1.

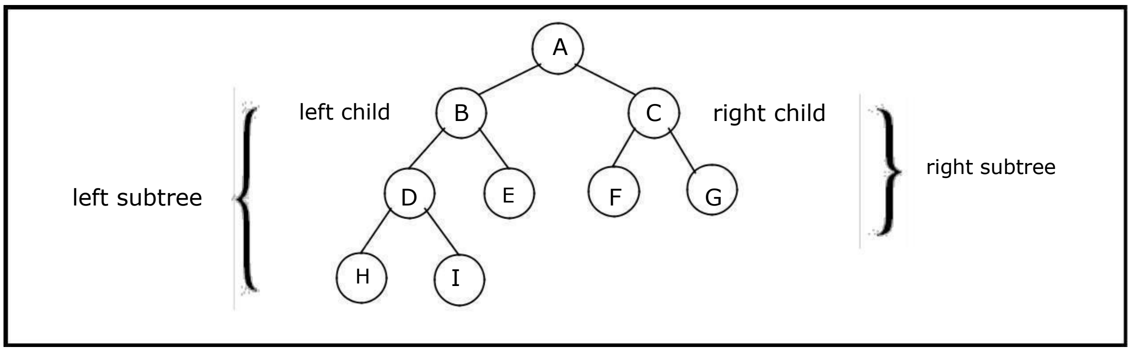


Figure 5.2.1. Binary Tree

Binary trees are easy to implement because they have a small, fixed number of child links. Because of this characteristic, binary trees are the most common types of trees and form the basis of many important data structures.

Tree Terminology:

Leaf node

A node with no children is called a *leaf* (or *external node*). A node which is not a leaf is called an *internal node*.

Path

A sequence of nodes $n_1, n_2, \ldots, n_k$, such that n_i is the parent of $n_i + 1$ for i = 1, 2,. . ., k - 1. The length of a path is 1 less than the number of nodes on the path. Thus there is a path of length zero from a node to itself.

For the tree shown in figure 5.2.1, the path between A and I is A, B, D, I.

Siblings

The children of the same parent are called siblings.

For the tree shown in figure 5.2.1, F and G are the siblings of the parent node C and H and I are the siblings of the parent node D.

Ancestor and Descendent

If there is a path from node A to node B, then A is called an ancestor of B and *B* is called a descendent of A.

Subtree

Any node of a tree, with all of its descendants is a subtree.

Level

The level of the node refers to its distance from the root. The root of the tree has level O, and the level of any other node in the tree is one more than the level of its parent. For example, in the binary tree of Figure 5.2.1 node F is at level 2 and node H is at level 3. *The maximum number of nodes at any level is* 2^n.

Height

The maximum level in a tree determines its height. The height of a node in a tree is the length of a longest path from the node to a leaf. The term depth is also used to denote height of the tree. The height of the tree of Figure 5.2.1 is 3.

Depth

The depth of a node is the number of nodes along the path from the root to that node. For instance, node „C" in figure 5.2.1 has a depth of 1.

Assigning level numbers and Numbering of nodes for a binary tree:

The nodes of a binary tree can be numbered in a natural way, level by level, left to right. The nodes of a complete binary tree can be numbered so that the root is assigned the number 1, a left child is assigned twice the number assigned its parent, and a right child is assigned one more than twice the number assigned its parent. For example, see Figure 5.2.2.

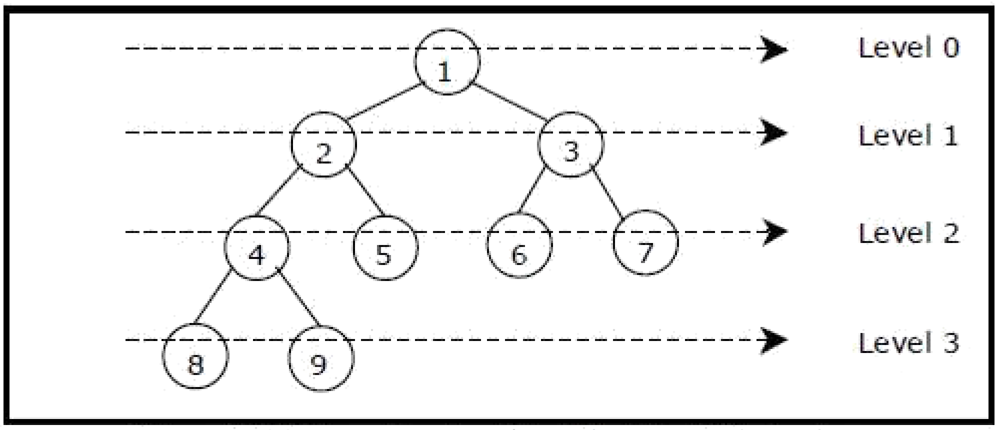


Figure 5.2.2. Level by level numbering of binary tree

Properties of binary trees:

Some of the important properties of a binary tree are as follows:

1. If *h* = height of a binary tree, then
    a. Maximum number of leaves = 2^h
    b. Maximum number of nodes = $2^{h+1} - 1$
2. If a binary tree contains m nodes at level l, it contains at most 2m nodes at level l + 1.
3. Since a binary tree can contain at most one node at level 0 (the root), it can contain at most 2^l node at level l.
4. The total number of edges in a full binary tree with n node is n - 1.

Strictly Binary tree:

If every non-leaf node in a binary tree has nonempty left and right subtrees, the tree is termed as strictly binary tree. Thus the tree of figure 5.2.3(a) is strictly binary. A strictly binary tree with n leaves always contains 2n - 1 nodes.

Full Binary tree:

A full binary tree of height h has all its leaves at level h. Alternatively; All non leaf nodes of a full binary tree have two children, and the leaf nodes have no children.

A full binary tree with height *h* has $2^{h + 1} - 1$ nodes. A full binary tree of height h is a *strictly binary tree* all of whose leaves are at level h. Figure 5.2.3(d) illustrates the full binary tree containing 15 nodes and of height 3.

A full binary tree of height h contains 2^h leaves and, $2^h - 1$ non-leaf nodes.

Thus by induction, total number of nodes (*tn*) = $\sum_{l=0}^{h} 2^{l} = 2^{h+1} - 1$.

For example, a full binary tree of height 3 contains $2^{3+1} - 1 = 15$ nodes.

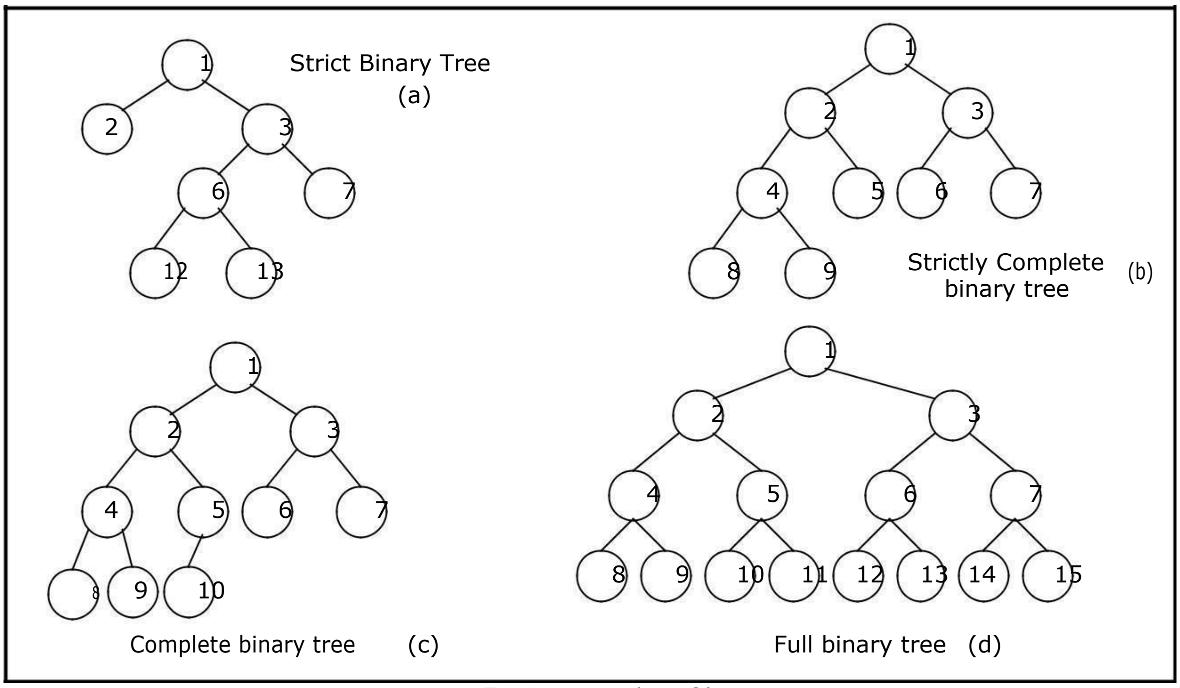


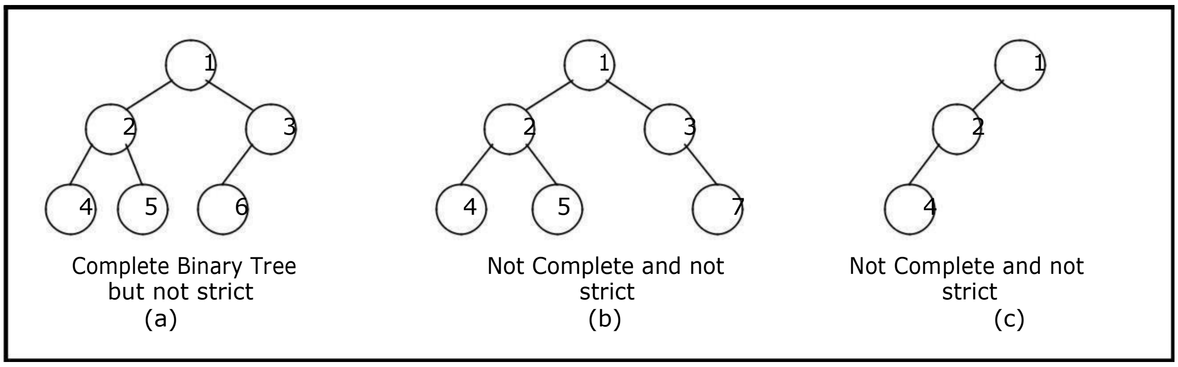
Figure 5.2.3. Examples of binary trees

Complete Binary tree:

A binary tree with *n* nodes is said to be **complete** if it contains all the first *n* nodes of the above numbering scheme. Figure 5.2.4 shows examples of complete and incomplete binary trees.

A complete binary tree of height h looks like a full binary tree down to level h-1, and the level h is filled from left to right.

A complete binary tree with n leaves that is *not strictly* binary has 2n nodes. For example, the tree of Figure 5.2.3(c) is a complete binary tree having 5 leaves and 10 nodes.



Complete Binary Tree but not strict
(a)
Not Complete and not strict
(b)
Not Complete and not strict
(c)

Figure 5.2.4. Examples of complete and incomplete binary trees

Internal and external nodes:

We define two terms: Internal nodes and external nodes. An internal node is a tree node having at least one–key and possibly some children. It is some times convenient to have another types of nodes, called an external node, and pretend that all null child links point to such a node. An external node doesn"t exist, but serves as a conceptual place holder for nodes to be inserted.

We draw internal nodes using circles, with letters as labels. External nodes are denoted by squares. The square node version is sometimes called an extended binary tree. A binary tree with n internal nodes has n+1 external nodes. Figure 5.2.6 shows a sample tree illustrating both internal and external nodes.

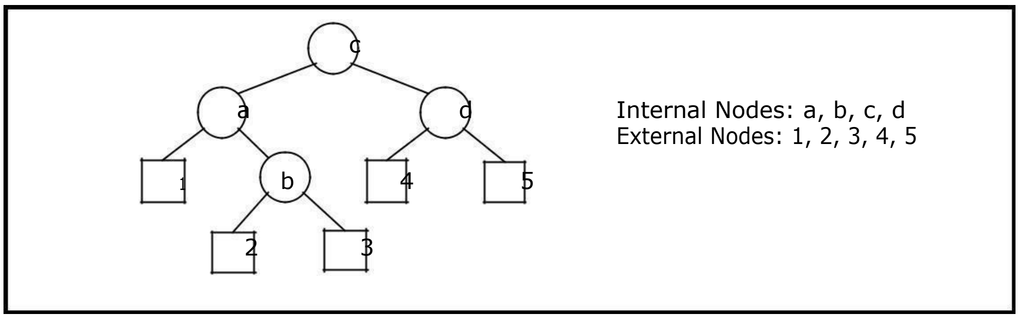


Figure 5.2.6. Internal and external nodes

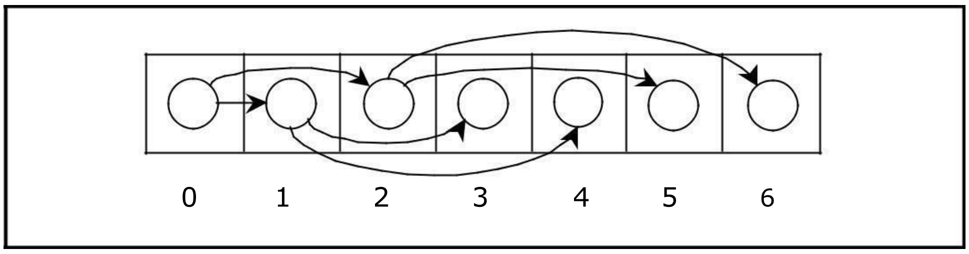
Data Structures for Binary Trees:

1. Arrays; especially suited for complete and full binary trees.
2. Pointer-based.

Array-based Implementation:

Binary trees can also be stored in arrays, and if the tree is a complete binary tree, this method wastes no space. In this compact arrangement, if a node has an index *i*, its children are found at indices 2*i*+1 and 2*i*+2, while its parent (if any) is found at index *floor((i-1)/2)* (assuming the root of the tree stored in the array at an index zero).

This method benefits from more compact storage and better locality of reference, particularly during a preorder traversal. However, it requires contiguous memory, expensive to grow and wastes space proportional to $2^h - n$ for a tree of height *h* with *n* nodes.



Linked Representation (Pointer based):

Array representation is good for complete binary tree, but it is wasteful for many other binary trees. The representation suffers from insertion and deletion of node from the middle of the tree, as it requires the moment of potentially many nodes to reflect the change in level number of this node. To overcome this difficulty we represent the binary tree in linked representation.
In linked representation each node in a binary has three fields, the left child field denoted as *LeftChild*, data field denoted as *data* and the right child field denoted as *RightChild*. If any sub-tree is empty then the corresponding pointer"s LeftChild and RightChild will store a NULL value. If the tree itself is empty the root pointer will store a NULL value.

The advantage of using linked representation of binary tree is that:

- Insertion and deletion involve no data movement and no movement of nodes except the rearrangement of pointers.

The disadvantages of linked representation of binary tree includes:-

- Given a node structure, it is difficult to determine its parent node.
- Memory spaces are wasted for storing NULL pointers for the nodes, which have no subtrees.

The structure definition, node representation empty binary tree is shown in figure 5.2.6 and the linked representation of binary tree using this node structure is given in figure 5.2.7.

```
struct binarytree
{
        struct binarytree *LeftChild;
        int data;
        struct binarytree *RightChild;
};

typedef struct binarytree node;

node *root = NULL;
```

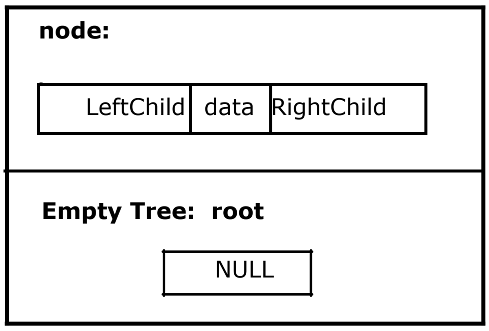


Figure 5.2.6. Structure definition, node representation and empty tree

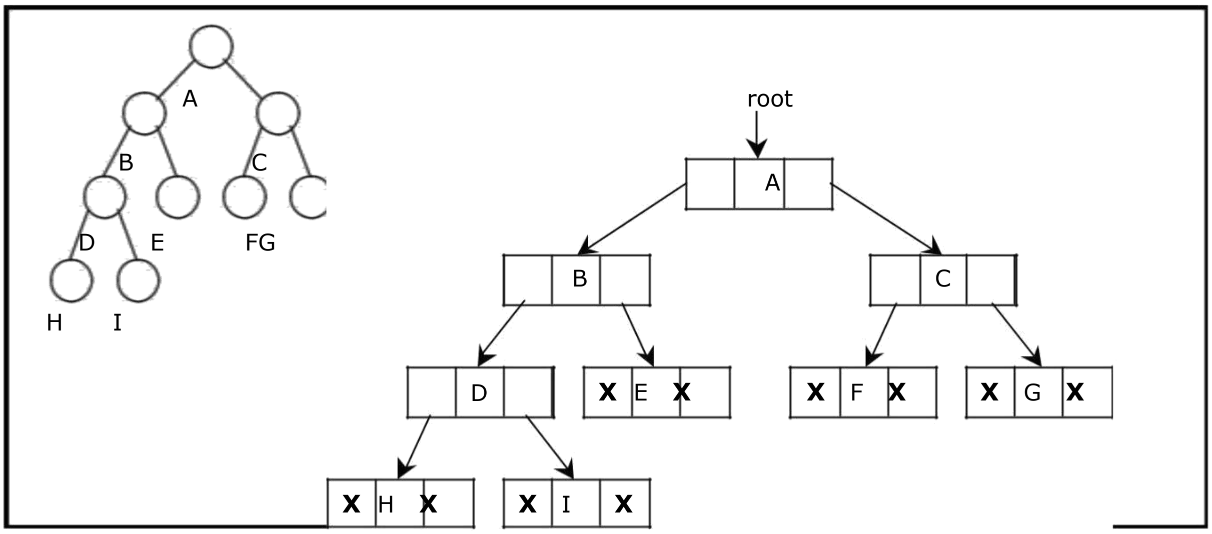


Figure 5.2.7. Linked representation for the binary tree

Lecture 18

Binary Tree Traversal Techniques:

A tree traversal is a method of visiting every node in the tree. By visit, we mean that some type of operation is performed. For example, you may wish to print the contents of the nodes.

There are four common ways to traverse a binary tree:

1. *Preorder*
2. *Inorder*
3. *Postorder*
4. *Level order*

In the first three traversal methods, the left subtree of a node is traversed before the right subtree. The difference among them comes from the difference in the time at which a root node is visited.

Recursive Traversal Algorithms:

Inorder Traversal:

In the case of inorder traversal, the root of each subtree is visited after its left subtree has been traversed but before the traversal of its right subtree begins. The steps for traversing a binary tree in inorder traversal are:

1. Visit the left subtree, using inorder.
2. Visit the root.
3. Visit the right subtree, using inorder.

The algorithm for inorder traversal is as follows:

```
void inorder(node *root)
{
        if(root != NULL)
        {
                inorder(root->lchild);

                print root -> data;
                inorder(root->rchild);
        }
}
```

Preorder Traversal:

In a preorder traversal, each root node is visited before its left and right subtrees are traversed. Preorder search is also called backtracking. The steps for traversing a binary tree in preorder traversal are:

1. Visit the root.
2. Visit the left subtree, using preorder.
3. Visit the right subtree, using preorder.

The algorithm for preorder traversal is as follows:

```
void preorder(node *root)
{
        if( root != NULL )
        {
                print root -> data;
                preorder (root -> lchild);
                preorder (root -> rchild);
        }
}
```

Postorder Traversal:

In a postorder traversal, each root is visited after its left and right subtrees have been traversed. The steps for traversing a binary tree in postorder traversal are:

1. Visit the left subtree, using postorder.
2. Visit the right subtree, using postorder
3. Visit the root.

The algorithm for postorder traversal is as follows:

```
void postorder(node *root)
{
        if( root != NULL )
        {
                postorder  (root  ->  lchild);
                postorder (root -> rchild);
                print (root -> data);
        }
}
```

Level order Traversal:

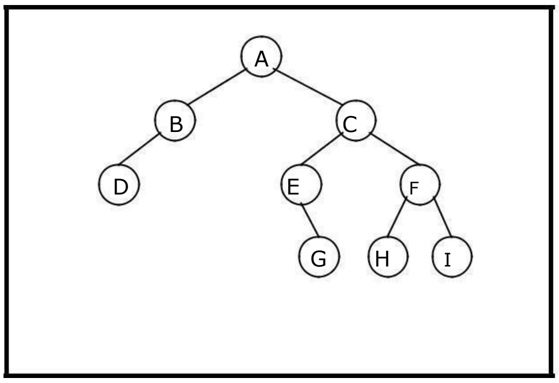
In a level order traversal, the nodes are visited level by level starting from the root, and going from left to right. The level order traversal requires a queue data structure. So, it is not possible to develop a recursive procedure to traverse the binary tree in level order. This is nothing but a breadth first search technique.

The algorithm for level order traversal is as follows:

```
void levelorder()
{
        int j;
        for(j = 0; j < ctr; j++)
        {
                if(tree[j] != NULL)
                        print tree[j] -> data;
        }
}
```

Example 1:

Traverse the following binary tree in pre, post, inorder and level order.



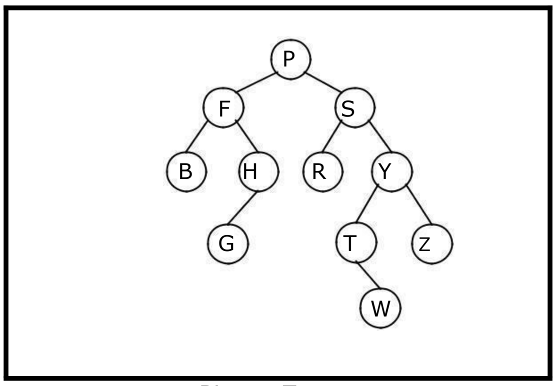
- Preo rder traversal yields: A, B, D, C , E, G , F , H, I
- Posto rder traversal yields: D, B, G , E, H, I, F , C , A
- Ino rder traversal yields: D, B, A, E, G , C , H, F , I
- Level o rder traversal yields: A, B, C , D, E, F , G , H, I

Bina ry T re e

Pre, Post, Inorder and lee l order T rav ers ing

Example 2:

Traverse the following binary tree in pre, post, inorder and level order.



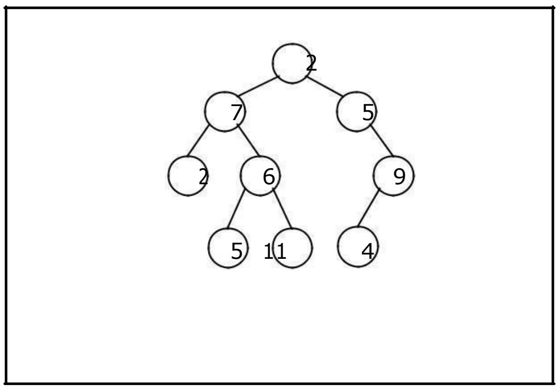
- Preo rder traversal yields: P, F , B, H, G , S, R, Y, T, W, Z
- Posto rder traversal yields: B, G , H, F , R, W, T, Z, Y, S, P
- Ino rder traversal yields: B, F , G , H, P, R, S, T, W, Y, Z
- Level o rder traversal yields: P, F , S, B, H, R, Y, G , T, Z, W

Bina ry T re e

Pre, P o st, Inorder a nd lev e l order T rav ers ing

Example 3:

Traverse the following binary tree in pre, post, inorder and level order.



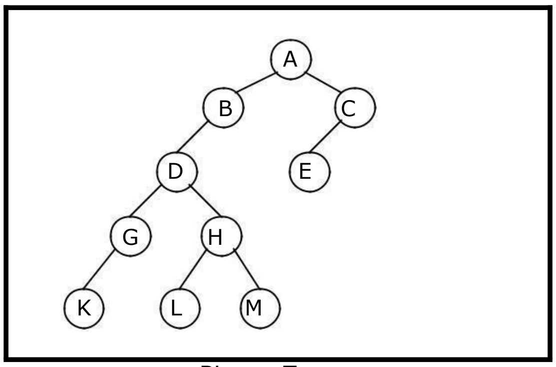
Bina ry T re e

- Preo rder traversal yields: 2, 7 , 2 , 6 , 5 , 11 , 5 , 9 , 4
- Posto rder travarsal yields: 2, 5 , 11 , 6 , 7 , 4 , 9 , 5 , 2
- Ino rder travarsal yields: 2, 7 , 5 , 6 , 11 , 2 , 5 , 4 , 9
- Level o rder traversal yields: 2, 7 , 5 , 2 , 6 , 9 , 5 , 11 , 4

Pre, P o st, Inorder a nd lev e l order T rav ers ing

Example 4:

Traverse the following binary tree in pre, post, inorder and level order.



Bina ry T re e

- Preo rder traversal yields: A, B, D, G , K, H, L, M , C , E
- Posto rder travarsal yields: K, G , L, M , H, D, B, E, C , A
- Ino rder travarsal yields: K, G , D, L, H, M , B, A, E, C
- Level o rder traversal yields: A, B, C , D, E, G , H, K, L, M

Pre, P o st, Inorder a nd lev e l order T rav ers ing

Building Binary Tree from Traversal Pairs:

Sometimes it is required to construct a binary tree if its traversals are known. From a single traversal it is not possible to construct unique binary tree. However any of the two traversals are given then the corresponding tree can be drawn uniquely:

- Inorder and preorder
- Inorder and postorder
- Inorder and level order

The basic principle for formulation is as follows:

If the preorder traversal is given, then the first node is the root node. If the postorder traversal is given then the last node is the root node. Once the root node is identified, all the nodes in the left sub-trees and right sub-trees of the root node can be identified using inorder.

Same technique can be applied repeatedly to form sub-trees.

It can be noted that, for the purpose mentioned, two traversal are essential out of which one should be inorder traversal and another preorder or postorder; alternatively, given preorder and postorder traversals, binary tree cannot be obtained uniquely.

Example 1:

Construct a binary tree from a given preorder and inorder sequence:

Preorder: A B D G C E H I F
Inorder: D G B A H E I C F

Solution:

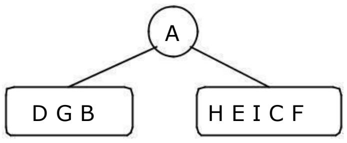
*From Preorder sequence **A** B D G C E H I F, the root is: A*

From Inorder sequence *D G B* ***A*** *H E I C F*, we get the left and right sub trees:

Left sub tree is: D G B

Right sub tree is: H E I C F

The Binary tree upto this point looks like:

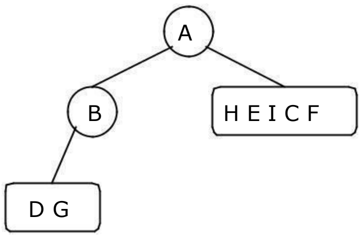


To find the root, left and right sub trees for D G B:

*From the preorder sequence **B** D G, the root of tree is: B*

From the inorder sequence D G **B,** we can find that D and G are to the left of B.

The Binary tree upto this point looks like:

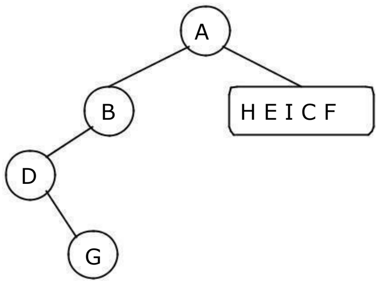


To find the root, left and right sub trees for D G:

From the preorder sequence **D** *G, the root of the tree is: D*

From the inorder sequence ***D*** *G*, we can find that there is no left node to D and G is at the right of D.

The Binary tree upto this point looks like:

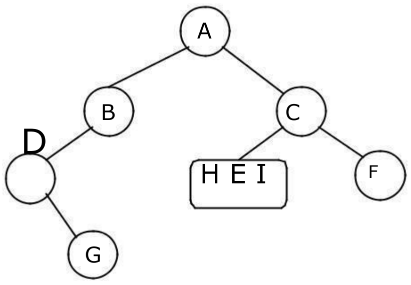


To find the root, left and right sub trees for H E I C F:

From the preorder sequence ***C*** *E H I F, the root of the left sub tree is: C*

From the inorder sequence *H E I* ***C*** *F*, we can find that H E I are at the left of C and F is at the right of C.

The Binary tree upto this point looks like:

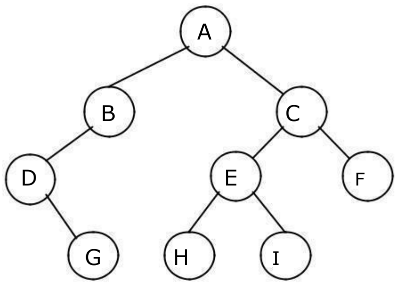


To find the root, left and right sub trees for H E I:

From the preorder sequence ***E*** *H I, the root of the tree is: E*

From the inorder sequence *H* ***E*** *I*, we can find that H is at the left of E and I is at the right of E.

The Binary tree upto this point looks like:



Example 2:

Construct a binary tree from a given postorder and inorder sequence:

Inorder: D G B A H E I C F
Postorder: G D B H I E F C A

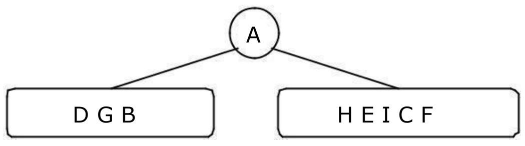
Solution:

From Postorder sequence G D B H I E F C **A**, *the root is: A*

From Inorder sequence _D G B_ **A** _H E I C F_, we get the left and right sub trees:

> *Left sub tree is: D G B*
> *Right sub tree is: H E I C F*

The Binary tree upto this point looks like:

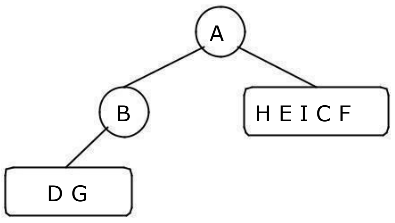


To find the root, left and right sub trees for D G B:

From the postorder sequence G D B, the root of tree is: B

From the inorder sequence _D G_ **B,** we can find that D G are to the left of B and there is no right subtree for B.

The Binary tree upto this point looks like:

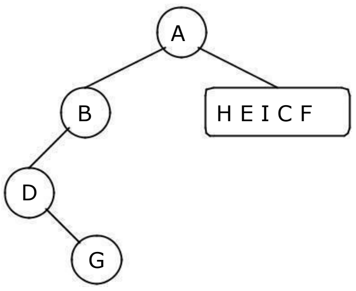


To find the root, left and right sub trees for D G:

From the postorder sequence G ***D****, the root of the tree is: D*

From the inorder sequence **D** _G_, we can find that is no left subtree for D and G is to the right of D.

The Binary tree upto this point looks like:

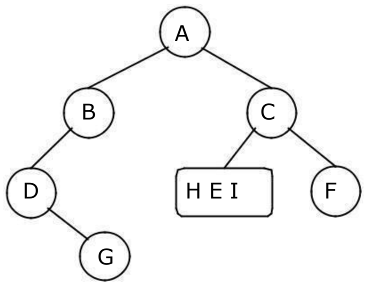


To find the root, left and right sub trees for H E I C F:

From the postorder sequence H I E F ***C****, the root of the left sub tree is: C*

From the inorder sequence _H E I_ ***C*** _F_, we can find that H E I are to the left of C and F is the right subtree for C.

The Binary tree upto this point looks like:

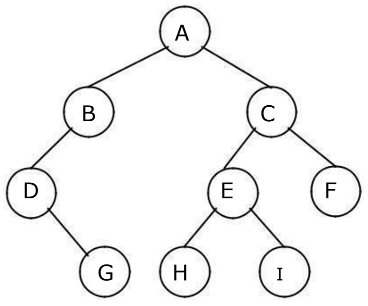


<u>To find the root, left and right sub trees for H E I:</u>

*From the postorder sequence H I **E**, the root of the tree is: E*

From the inorder sequence <u>*H*</u> **E** <u>*I*</u>, we can find that H is left subtree for E and I is to the right of E.

The Binary tree upto this point looks like:



Example 3:

Construct a binary tree from a given preorder and inorder sequence:

Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9
Preorder: n6 n2 n1 n4 n3 n5 n9 n7 n8

Solution:

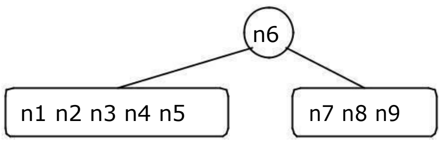
From Preorder sequence **n6** n2 n1 n4 n3 n5 n9 n7 n8, *the root is: n6*

From Inorder sequence <u>*n1 n2 n3 n4 n5*</u> **n6** <u>*n7 n8 n9*</u>, we get the left and right sub trees:

Left sub tree is: n1 n2 n3 n4 n5

Right sub tree is: n7 n8 n9

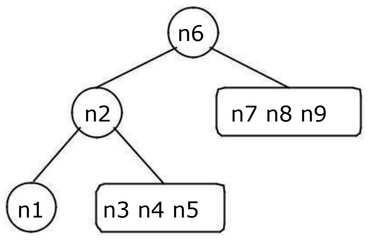
The Binary tree upto this point looks like:



To find the root, left and right sub trees for n1 n2 n3 n4 n5:

*From the preorder sequence **n2** n1 n4 n3 n5, the root of tree is: n2*

From the inorder sequence *n1* **n2** *n3 n4 n5***,** we can find that n1 is to the left of n2 and n3 n4 n5 are to the right of n2. The Binary tree upto this point looks like:

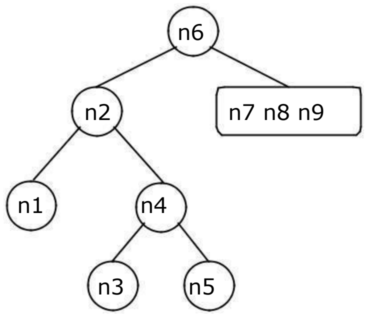


To find the root, left and right sub trees for n3 n4 n5:

*From the preorder sequence **n4** n3 n5, the root of the tree is: n4*

From the inorder sequence *n3* **n4** *n5*, we can find that n3 is to the left of n4 and n5 is at the right of n4.

The Binary tree upto this point looks like:

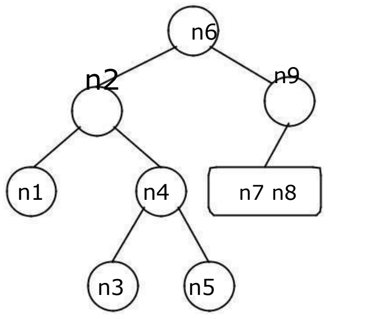


To find the root, left and right sub trees for n7 n8 n9:

*From the preorder sequence **n9** n7 n8, the root of the left sub tree is: n9*

From the inorder sequence *n7 n8* **n9**, we can find that n7 and n8 are at the left of n9 and no right subtree of n9.

The Binary tree upto this point looks like:

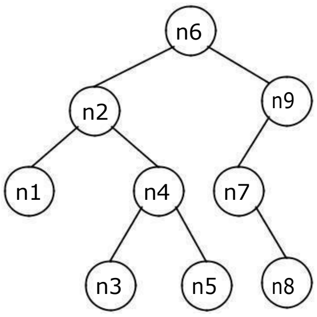


To find the root, left and right sub trees for n7 n8:

*From the preorder sequence **n7** n8, the root of the tree is: n7*

From the inorder sequence **n7** _n8_, we can find that is no left subtree for n7 and n8 is at the right of n7.

The Binary tree upto this point looks like:



Example 4:

Construct a binary tree from a given postorder and inorder sequence:

Inorder: n1 n2 n3 n4 n5 n6 n7 n8 n9
Postorder: n1 n3 n5 n4 n2 n8 n7 n9 n6

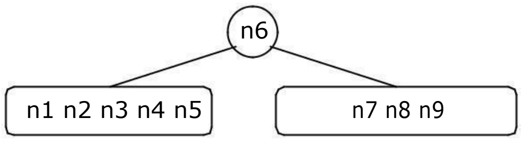
Solution:

From Postorder sequence n1 n3 n5 n4 n2 n8 n7 n9 **n6**, *the root is: n6*

From Inorder sequence _n1 n2 n3 n4 n5_ **n6** _n7 n8 n9_, we get the left and right sub trees:

Left sub tree is: n1 n2 n3 n4 n5
Right sub tree is: n7 n8 n9

The Binary tree upto this point looks like:

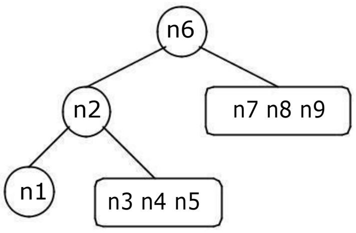


To find the root, left and right sub trees for _n1 n2 n3 n4 n5_:

From the postorder sequence n1 n3 n5 n4 **n2**, *the root of tree is: n2*

From the inorder sequence _n1_ **n2** _n3 n4 n5_**,** we can find that n1 is to the left of n2 and n3 n4 n5 are to the right of n2.

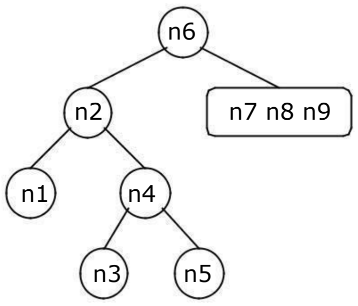
The Binary tree upto this point looks like:



To find the root, left and right sub trees for n3 n4 n5:

From the postorder sequence n3 n5 **n4***, the root of the tree is: n4*

From the inorder sequence *n3* **n4** *n5*, we can find that n3 is to the left of n4 and n5 is to the right of n4. The Binary tree upto this point looks like:

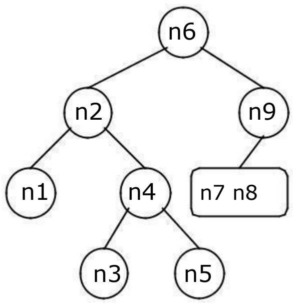


To find the root, left and right sub trees for n7 n8 and n9:

From the postorder sequence n8 n7 **n9***, the root of the left sub tree is: n9*

From the inorder sequence *n7 n8* **n9**, we can find that n7 and n8 are to the left of n9 and no right subtree for n9.

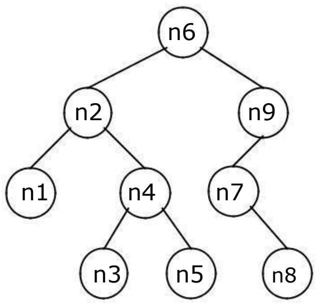
The Binary tree upto this point looks like:



To find the root, left and right sub trees for n7 and n8:

From the postorder sequence n8 **n7***, the root of the tree is: n7*

From the inorder sequence **n7** *n8*, we can find that there is no left subtree for n7 and n8 is to the right of n7. The Binary tree upto this point looks like:



Lecture 19

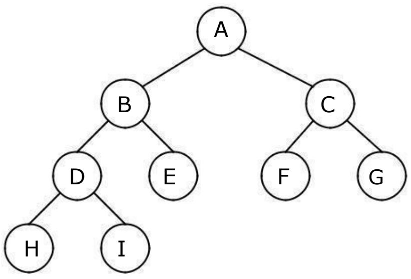
Threaded Binary Tree:

The linked representation of any binary tree has more null links than actual pointers. If there are 2n total links, there are n+1 null links. A clever way to make use of these null links has been devised by A.J. Perlis and C. Thornton.

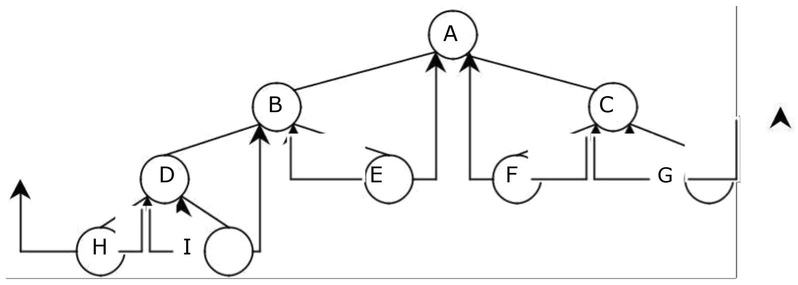
Their idea is to replace the null links by pointers called Threads to other nodes in the tree.

If the RCHILD(p) is normally equal to zero, we will replace it by a pointer to the node which would be printed after P when traversing the tree in inorder.

A null LCHILD link at node P is replaced by a pointer to the node which immediately precedes node P in inorder. For example, Let us consider the tree:



The Threaded Tree corresponding to the above tree is:

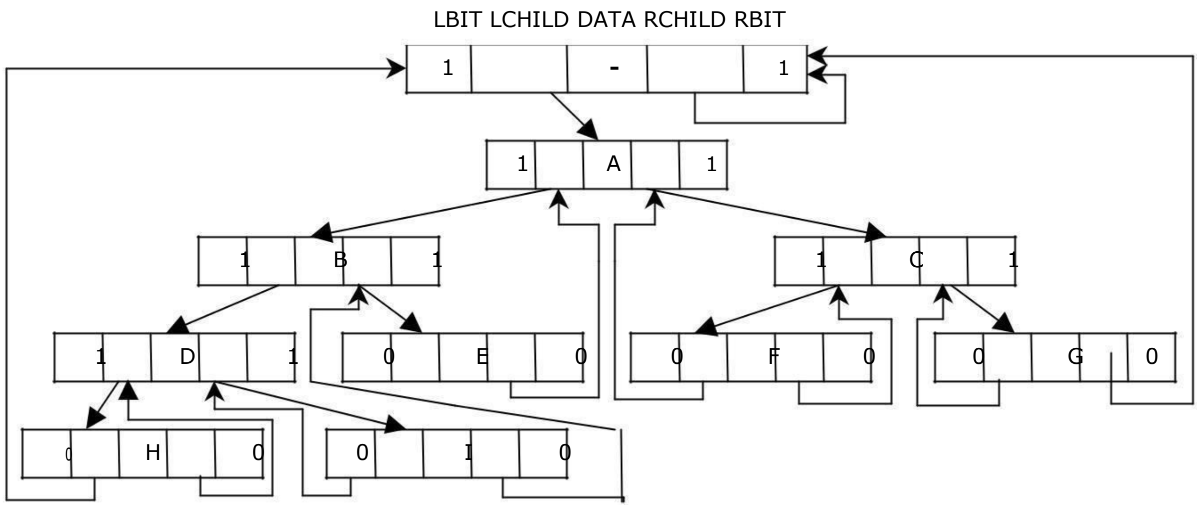


The tree has 9 nodes and 10 null links which have been replaced by Threads. If we traverse T in inorder the nodes will be visited in the order H D I B E A F C G.

For example, node „E" has a predecessor Thread which points to „B" and a successor Thread which points to „A". In memory representation Threads and normal pointers are distinguished between as by adding two extra one bit fields LBIT and RBIT.

LBIT(P) = 1	if LCHILD(P) is a normal pointer
LBIT(P) = 0	if LCHILD(P) is a Thread
RBIT(P) = 1	if RCHILD(P) is a normal pointer
RBIT(P) = 0	if RCHILD(P) is a Thread

In the above figure two threads have been left dangling in LCHILD(H) and RCHILD(G). In order to have no loose Threads we will assume a head node for all threaded binary trees. The Complete memory representation for the tree is as follows. The tree T is the left sub-tree of the head node.



Lecture 20

Binary Search Tree:

A binary search tree is a binary tree. It may be empty. If it is not empty then it satisfies the following properties:

1. Every element has a key and no two elements have the same key.
2. The keys in the left subtree are smaller than the key in the root.
3. The keys in the right subtree are larger than the key in the root.
4. The left and right subtrees are also binary search trees.

Figure 5.2.5(a) is a binary search tree, whereas figure 5.2.5(b) is not a binary search tree.

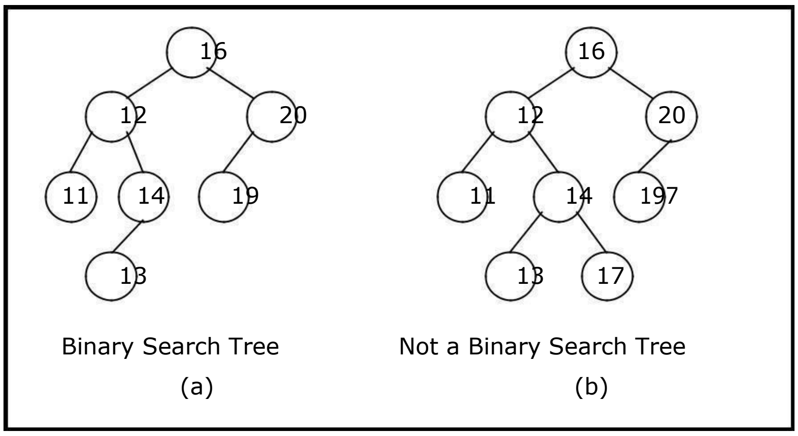


Figure 5.2.5. Examples of binary search trees

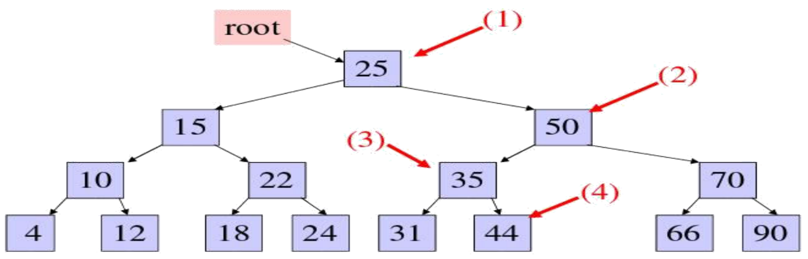
Searching a key

To search a given key in Bianry Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root"s key, we recur for right subtree of root node. Otherwise we recur for left subtree.

<u>Example: search for 45 in the tree</u>

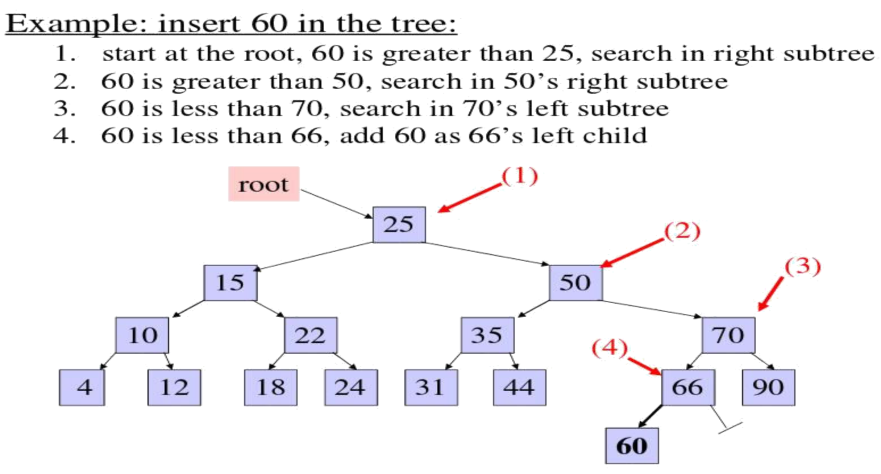
(key fields are show in node rather than in separate obj ref to by data field):

1. start at the root, 45 is greater than 25, search in right subtree
2. 45 is less than 50, search in 50's left subtree
3. 45 is greater than 35, search in 35's right subtree
4. 45 is greater than 44, but 44 has no right subtree so 45 is not in the BST



Insertion of a key

A new key is always inserted at leaf. We start searching a key from root till we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.



Example: insert 60 in the tree:

1. start at the root, 60 is greater than 25, search in right subtree
2. 60 is greater than 50, search in 50's right subtree
3. 60 is less than 70, search in 70's left subtree
4. 60 is less than 66, add 60 as 66's left child

Deletion of a key

When we delete a node, there possibilities arise.

1) **Node to be deleted is leaf:** Simply remove from the tree.

```
              50                            50

           /     \         delete(20)      /   \

          30       70      --------->     30      70

        /   \    /   \                     \    /   \

       20   40   60     80                 40   60    80
```

2) **Node to be deleted has only one child:** Copy the child to the node and delete the child.

```
              50                            50

           /     \         delete(30)      /   \

          30       70      --------->     40      70

           \     /   \                          /   \

            40   60     80                      60    80
```

3) **Node to be deleted has two children:** Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

```
              50                            60
           /     \         delete(50)      /   \
          40      70       --------->     40    70
                 /   \                            \
                60    80                           80
```

The important thing to note is, inorder successor is needed only when right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in right child of the node.

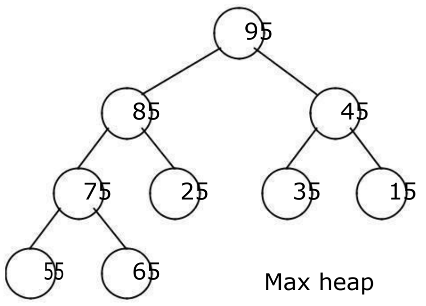
Lecture 21

Heap and Heap Max and Min Heap:

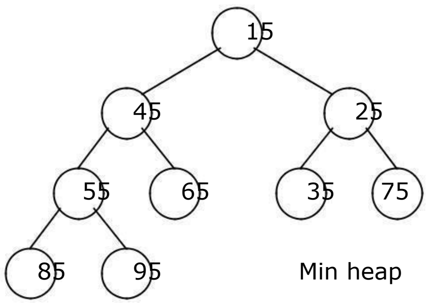
Heap is a data structure, which permits one to insert elements into a set and also to find the largest element efficiently. A data structure, which provides these two operations, is called a priority queue.

Max and Min Heap data structures:

A max heap is an almost complete binary tree such that the value of each node is greater than or equal to those in its children.



Max heap



Min heap

A min heap is an almost complete binary tree such that the value of each node is less than or equal to those in its children.

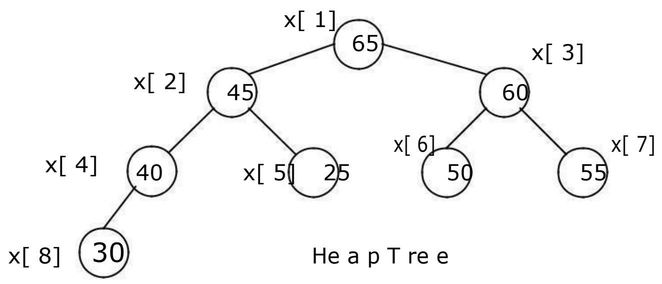
Representation of Heap Tree:

Since heap is a complete binary tree, a heap tree can be efficiently represented using one dimensional array. This provides a very convenient way of figuring out where children belong to.

- The root of the tree is in location 1.
- The left child of an element stored at location *i* can be found in location 2**i*.
- The right child of an element stored at location *i* can be found in location 2**i*+1.
- The parent of an element stored at location *i* can be found at location floor(*i*/2).

The elements of the array can be thought of as lying in a tree structure. A heap tree represented using a single array looks as follows:

X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]	X[8]
65	45	60	40	25	50	55	30



He a p T re e

Priority queue implementation using heap tree:

Priority queue can be implemented using circular array, linked list etc. Another simplified implementation is possible using heap tree; the heap, however, can be represented using an array. This implementation is therefore free from the complexities of circular array and linked list but getting the advantages of simplicities of array.

As heap trees allow the duplicity of data in it. Elements associated with their priority values are to be stored in from of heap tree, which can be formed based on their priority values. The top priority element that has to be processed first is at the root; so it can be deleted and heap can be rebuilt to get the next element to be processed, and so on. As an illustration, consider the following processes with their priorities:

Process	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}
Priority	5	4	3	4	5	5	3	2	1	5

These processes enter the system in the order as listed above at time 0, say. Assume that a process having higher priority value will be serviced first. The heap tree can be formed considering the process priority values. The order of servicing the process is successive deletion of roots from the heap.

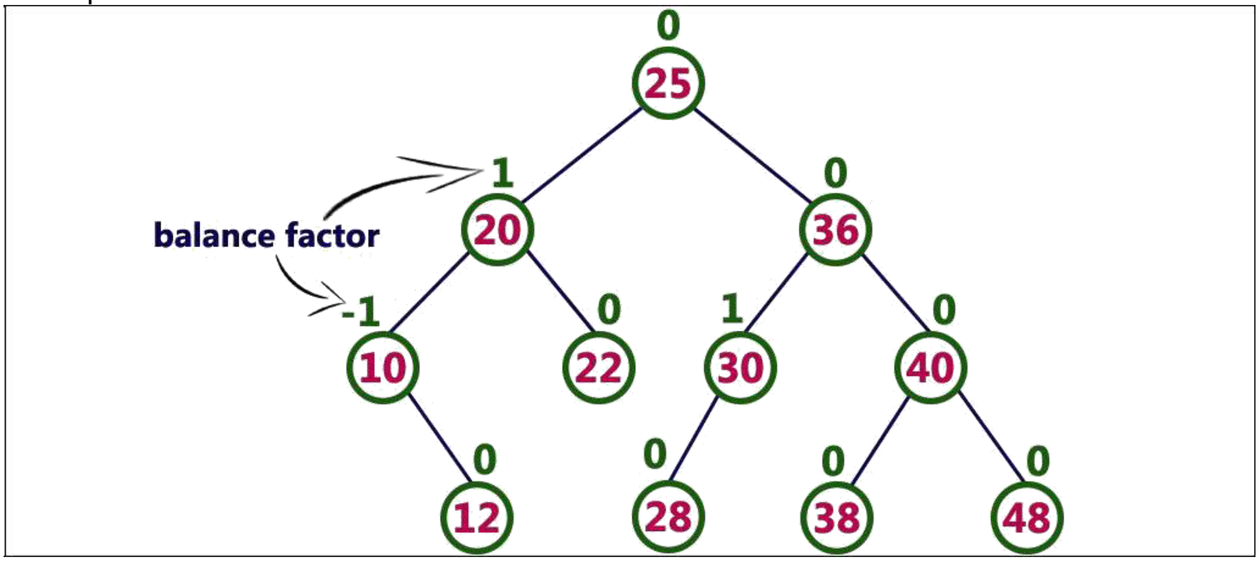
Lecture 22

AVL TREE

AVL tree is a self balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced, if the difference between the heights of left and right subtrees of every node in the tree is either-1, 0 or +1. In other words, a binary tree is said to be balanced if for every node, height of its children differ by at most one. In an AVL tree, every node maintains a extra information known as balance factor. The AVL tree was introduced in the year of 1962 by G.M A Adelson-Velsky and E.M.Landis.

An AVL tree is defined as follows...
An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either-1, 0 or +1.
Balance factor of a node is the difference between the heights of left and right subtrees of that node. The balance factor of a node is calculated either height of left subtree - height of right subtree (OR) height of right subtree - height of left subtree. In the following explanation, we are calculating as follows...
Balance factor = height Of Left Subtree – height Of Right Subtree

Example

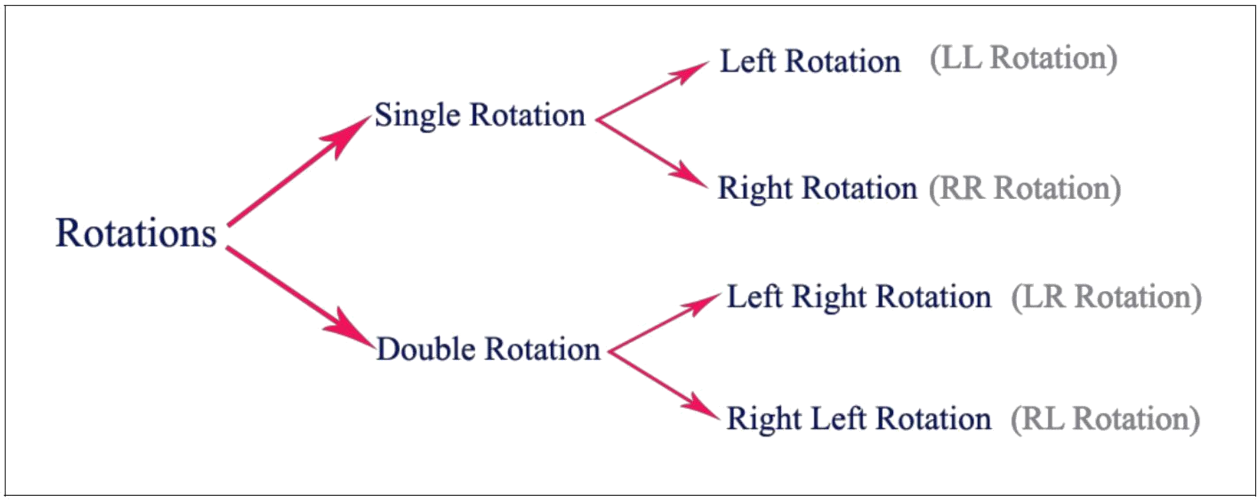


The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree. Rotation operations are used to make a tree balanced Every AVL Tree is a binary search tree but all the Binary Search Trees need not to be AVL trees.

AVL Tree Rotations

In AVL tree, after performing every operation like insertion and deletion we need to check the balance factor of every node in the tree. If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. We use rotation operations to make the tree balanced whenever the tree is becoming imbalanced due to any operation.

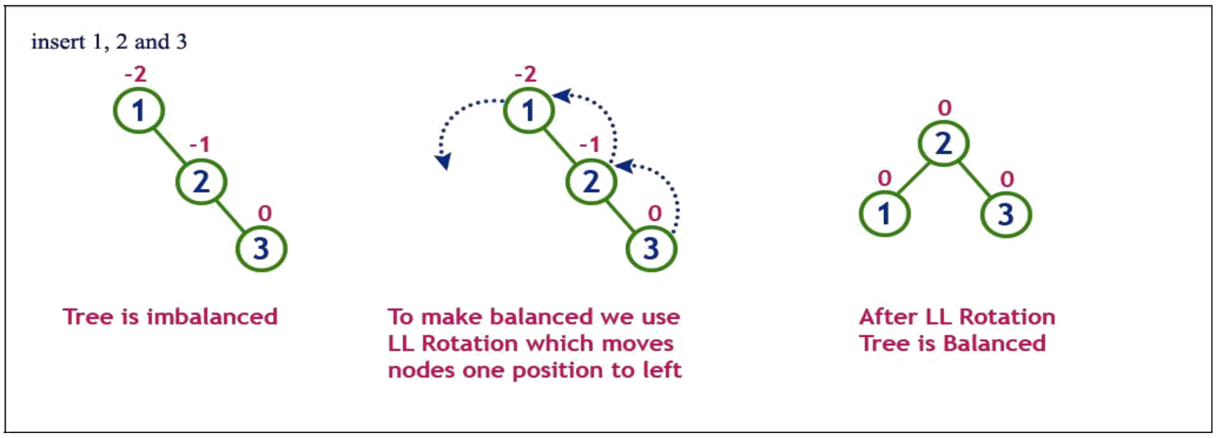
Rotation is the process of moving the nodes to either left or right to make tree balanced.



Rotations
Single Rotation
Double Rotation
Left Rotation (LL Rotation)
Right Rotation (RR Rotation)
Left Right Rotation (LR Rotation)
Right Left Rotation (RL Rotation)

Single Left Rotation (LL Rotation)

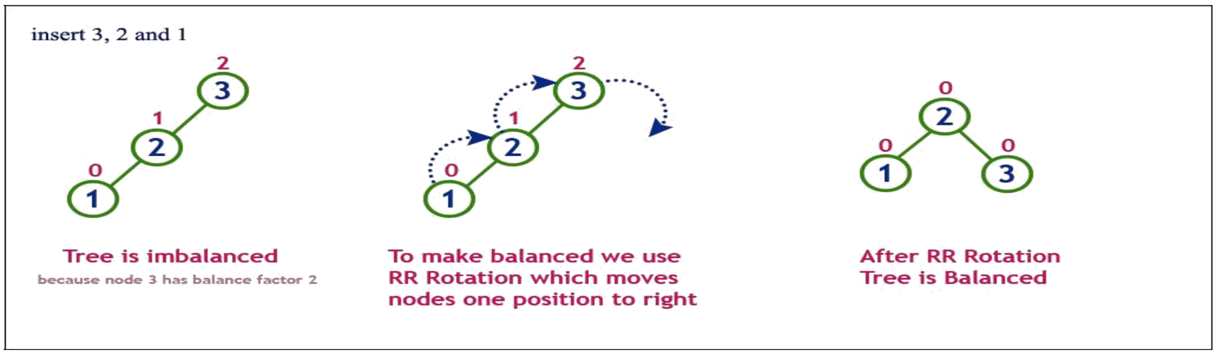
In LL Rotation every node moves one position to left from the current position. To understand LL Rotation, let us consider following insertion operations into an AVL Tree…



insert 1, 2 and 3
Tree is imbalanced
To make balanced we use LL Rotation which moves nodes one position to left
After LL Rotation Tree is Balanced

Single Right Rotation (RR Rotation)

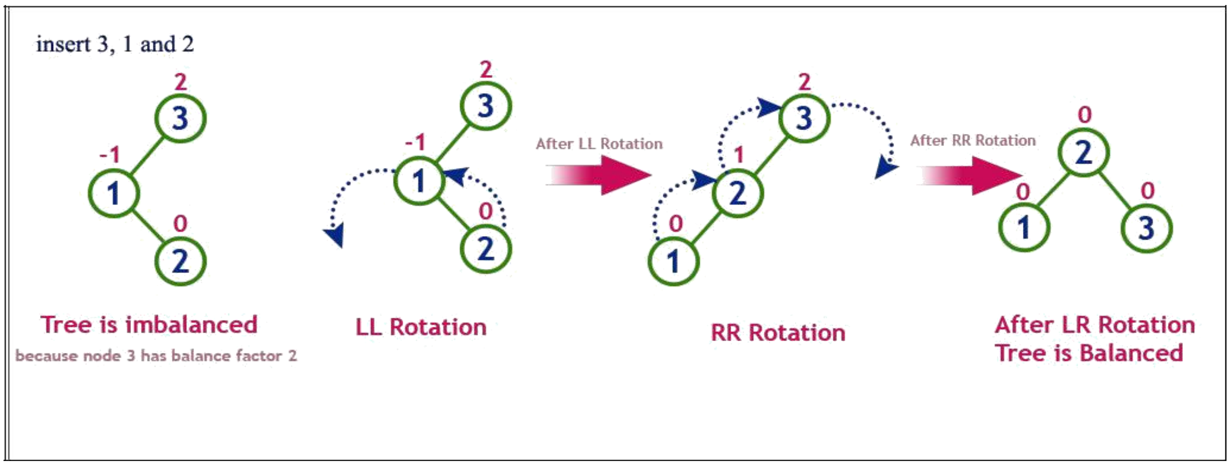
In RR Rotation every node moves one position to right from the current position. To understand RR Rotation, let us consider following insertion operations into an AVL Tree...



insert 3, 2 and 1
Tree is imbalanced
because node 3 has balance factor 2
To make balanced we use RR Rotation which moves nodes one position to right
After RR Rotation Tree is Balanced

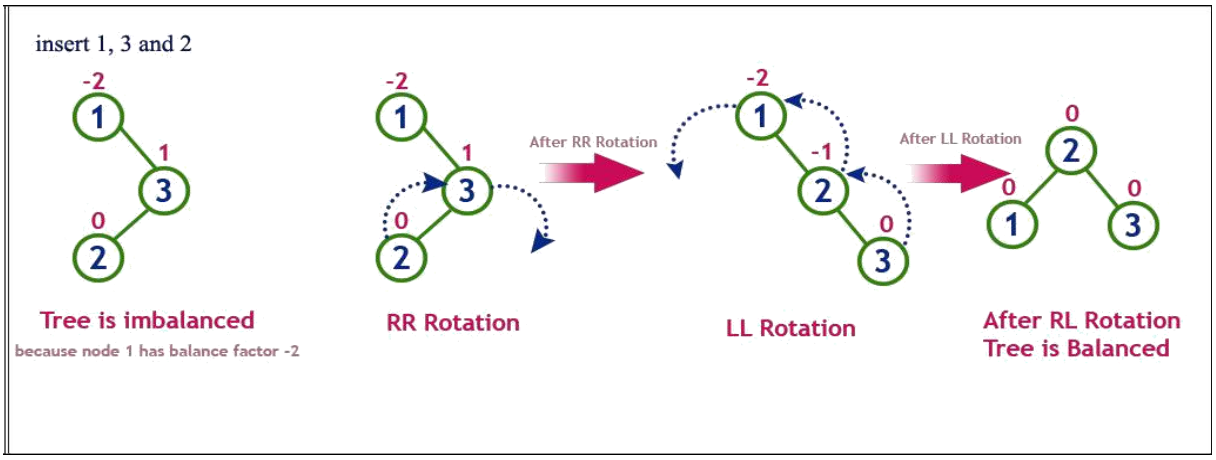
Left Right Rotation (LR Rotation)

The LR Rotation is combination of single left rotation followed by single right rotation. In LR Roration, first every node moves one position to left then one position to right from the current position. To understand LR Rotation, let us consider following insertion operations into an AVL Tree...



Right Left Rotation (RL Rotation)

The RL Rotation is combination of single right rotation followed by single left rotation. In RL Roration, first every node moves one position to right then one position to left from the current position. To understand RL Rotation, let us consider following insertion operations into an AVL Tree...



The following operations are performed on an AVL tree...

1. Search
2. Insertion
3. Deletion

Search Operation in AVL Tree

In an AVL tree, the search operation is performed with O(log n) time complexity. The search operation is performed similar to Binary search tree search operation. We use the following steps to search an element in AVL tree...

Step 1: Read the search element from the user

Step 2: Compare, the search element with the value of root node in the tree.

Step 3: If both are matching, then display "Given node found!!!" and terminate the function.

Step 4: If both are not matching, then check whether search element is smaller or larger than that node value.

Step 5: If search element is smaller, then continue the search process in left subtree.

Step 6: If search element is larger, then continue the search process in right subtree.

Step 7: Repeat the same until we found exact element or we completed with a leaf node

Step 8: If we reach to the node with search value, then display "Element is found" and terminate the function.

Step 9: If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with O(log n) time complexity. In AVL Tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

Step 1: Insert the new element into the tree using Binary Search Tree insertion logic.

Step 2: After insertion, check the Balance Factor of every node.

Step 3: If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.

Step 4: If the Balance Factor of any node is other than 0 or 1 or -1 then tree is said to be imbalanced. Then perform the suitable.

Rotation to make it balanced. And go for next operation.

Lecture 23

Deletion Operation in AVL Tree

In an AVL Tree, the deletion operation is similar to deletion operation in BST. But after every deletion operation we need to check with the Balance Factor condition. If the tree is balanced after deletion then go for next operation otherwise perform the suitable rotation to make the tree Balanced.

I) First: find the node x where k is stored

II) Second: delete the contents of node x

Claim: Deleting a node in an AVL tree can be reduced to deleting a leaf

There are three possible cases (just like for BSTs):

1) If x has no children (i.e., is a leaf), delete x.
2) If x has one child, let x' be the child of x.
   Notice: x' cannot have a child, since subtrees of T can differ in height by at most one -replace the contents of x with the contents of x'
   -delete x' (a leaf)
3) If x has two children,
   -find x's successor z (which has no left child)
   -replace x's contents with z's contents, and
   -delete z.
   [since z has at most one child, so we use case (1) or (2) to delete z]

   In all 3 cases, we end up removing a leaf.

III) Third: Go from the deleted leaf towards the

root and at each ancestor of that leaf:

-update the balance factor
-rebalance with rotations if necessary.

Example #1: Single rotation

AVL tree T:

```
                13 (0)
              /        \
        9 (0)           15(+)
       /    \          /    \
    6  (-)  12  (-)   14 (0)  20  (0)
    /       /                 /   \
    2  (0)  11(0)          18(0)   30 (0)
```

Delete(T, 13):

Before rotation, but after updating balance factors:

```
                  14 (0)
               /         \
         9 (0)            15(++)
        /    \            /    \
     6  (-)   12  (-)     X    20 (0)
     /        /                 /  \
     2 (0)   11(0)           18 (0)  30 (0)
```

After rotation:

```
                  14 (0)
               /         \
         9 (0)            20  (-)
        /    \          /      \
     6  (-)   12  (-)   15 (+)   30 (0)
     /        /            \
     2 (0)   11(0)          18  (0)
```

Example #2: Double rotation

```
AVL tree T:
                  9 (+)
                /        \
         6 (-)              15(-)
        /    \            /       \
     2  (-)    7 (0)     13(-)      20  (0)
     /                  /  \        /     \
     1 (0)           12  (-) 14 (0)  18 (0)  30 (0)
                       /
                     11  (0)
```

Delete(T, 2):

Before rotation, but after updating balance factors:

```
                  9 (++)
               /         \
         6 (0)              15(-)
        /    \            /       \
     1  (0)    7 (0)     13(-)      20  (0)
       /                  /  \        /     \
     X                12  (-) 14 (0)  18 (0)  30 (0)
                         /
                      11  (0)
```

After first rotation:

```
                9 (++)
              /        \
         6 (0)           13(+)
        /    \          /     \
     1 (0)    7 (0)   12  (-)    15 (+)
                       /         /      \
                  11 (0)      14  (0) 20  (0)
                                      /      \
                                   18(0)   30 (0)
```

After second rotation:

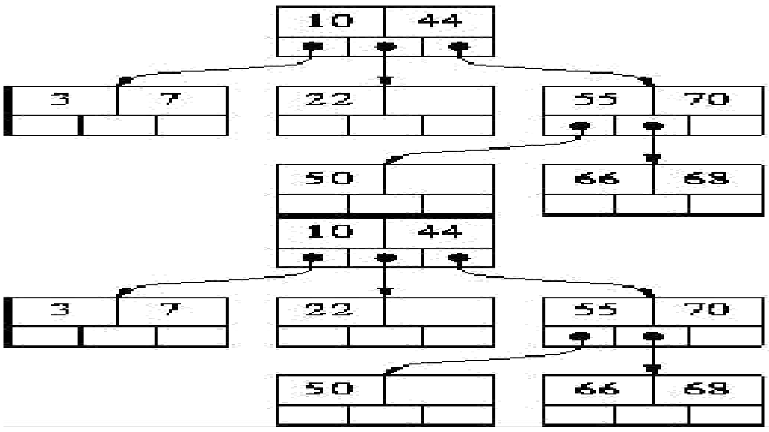
```
                 13 (0)
              /         \
         9(0)              15(+)
        /      \           /    \
      6 (0)     12 (-)   14 (0) 20 (0)
     /  \         /              /     \
  1 (0)   7 (0)  11 (0)         18  (0)  30 (0)
```

Lecture 24

M-way Search Tree

Binary search tree: 1 value and 2 subtrees per node.
M-way search tree: M - 1 value and M subtrees per node. M is the degree of the tree.

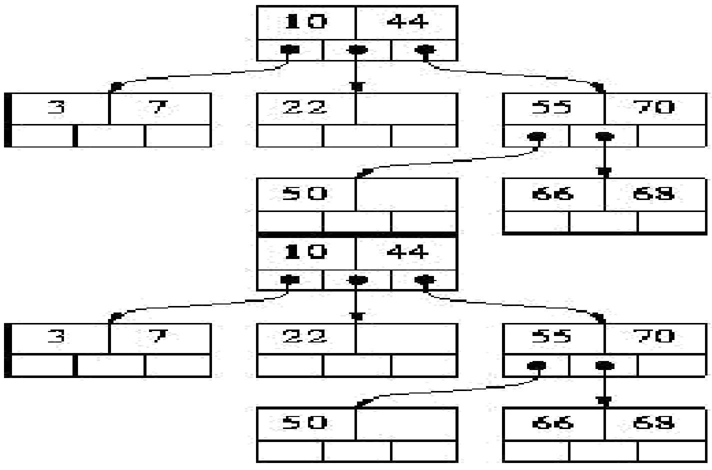
- In fact, each may contain up to M - 1 values. A node with k values must have k + 1 subtrees.
- In a node, values are stored in ascending order:
- V1 < V2 < ... < Vk
- The subtrees are placed between adjacent values: each value has a left and right subtree.
- V(i)'s right subtree = V(i+1)'s left subtree.
- All the values in V(i)'s left subtree are < V(i).
- All the values in V(i)'s right subtree are > V(i).



Search in M-way Trees

Searching for X:
1. If X < V(1), recursively search in V(1)'s left subtree.
2. If X > V(k), recursively search in V(k)'s right subtree.
3. If X = V(i), for some i, X is found!
4. Else, for some i, V(i) < X < V(i+1); recursively search in subtree between V(i)and V(i+1)

Search for 68 in:



B TREE

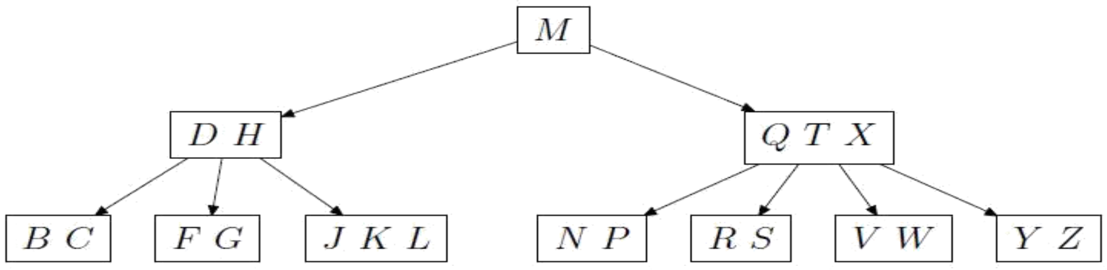
What are B-trees?

B-trees are balanced search trees: height = Olog(n)for the worst case.

They were designed to work well on Direct Access secondary storage devices (magnetic disks). B trees are widely used in database systems.

B-trees try to read as much information as possible in every disk access operation.

The 21 English consonants as keys of a B-tree:



Every internal node x containing n[x] keys has n[x] + 1 children.
All leaves are at the same depth in the tree

B-tree: definition

A B-tree T is a rooted tree (with root [T]) with properties:

• Every node x has four fields:

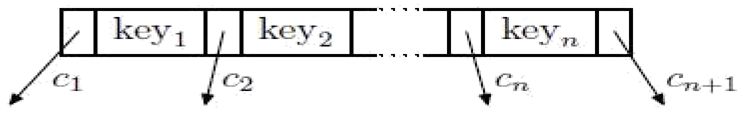
1. The number of keys currently stored in node x, n[x].
2. The n[x] keys themselves, stored in non-decreasing
order: key1[x] _ key2[x] _ · · · _ keyn[x][x] .
3. A boolean value,

$$\mathrm{leaf}[x] = \begin{cases} \text{True} & \text{if } x \text{ is a leaf}, \\ \text{False} & \text{if } x \text{ is an internal node}. \end{cases}$$

4. n[x] + 1 pointers, c1[x], c2[x], . . . , cn[x]+1[x] to its children.
(As leaf nodes have no children their ci are undefined).

Representing pointers and keys in a node:

:



Properties:

• The keys keyi[x] separate the ranges of keys stored in each subtree: if ki is any key stored in the subtree with root ci[x], then:
k1 _ key1[x] _ k2 _ key2[x] _ . . . _ keyn[x] _ kn[x]+1 .
• All leaves have the same height, which is the tree"s height h.
• There are upper on lower bounds on the number of keys on a node.
To specify these bounds we use a fixed integer t _ 2, the minimum degree of the B-tree:

– **lower bound**: every node other than root must have at least $\boxed{t-1}$ keys $\Longrightarrow$ At least $\boxed{t}$ children.

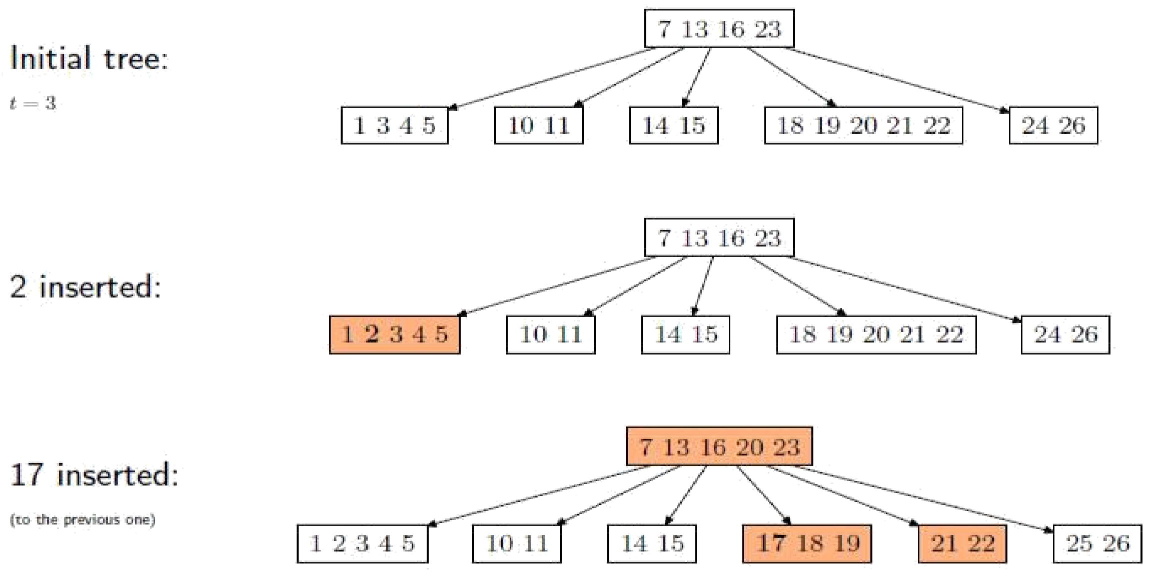
– **upper bound**: every node can contain at most $\boxed{2t-1}$ keys $\Longrightarrow$ every internal node has at most $\boxed{2t}$ children.

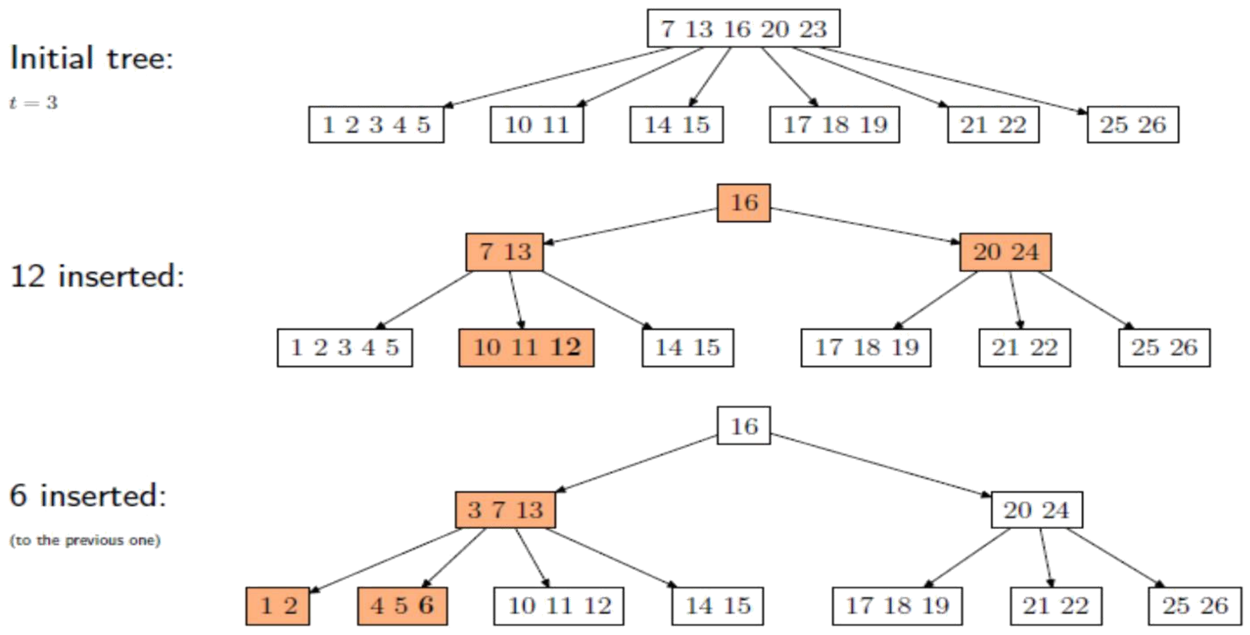
Basic operations on B-trees

Details of the following operations:
• B-Tree-Search
• B-Tree-Create
• B-Tree-Insert
• B-Tree-Delete

Conventions:
• Root of B-tree is always in main memory (Disk-Read on the root is never required)
• Any node pased as parameter must have had a Disk-Read operation performedon them.
Procedures presented are all top down algorithms (no need to back up) starting at the root of the tree.

Inserting a key – Examples





Deleting a Key from a B-tree

Similar to insertion, with the addition of a couple of special cases
• Key can be deleted from any node.
• More complicated procedure, but similar performance figures: O(h) disk accesses, O(th) = O(t logt n) CPU time
• Deleting is done in a single pass down the tree, but needs to return to the node with the deleted key if it is an internal node
• In the latter case, the key is first moved down to a leaf. Final deletion always takes place on a leaf.

Deleting a Key — Cases I

Considering 3 distinct cases for deletion

• Let k be the key to be deleted, x the node containing the key. Then the cases are:

1. If key k is in node x and x is a leaf, simply delete k from x

2. If key k is in node x and x is an internal node, there are three cases to consider:

(a) If the child y that precedes k in node x has at least t keys (more than the minimum), then find the predecessor key k0 in the subtree rooted at y. Recursively delete k0 and replace k with k0 in x.

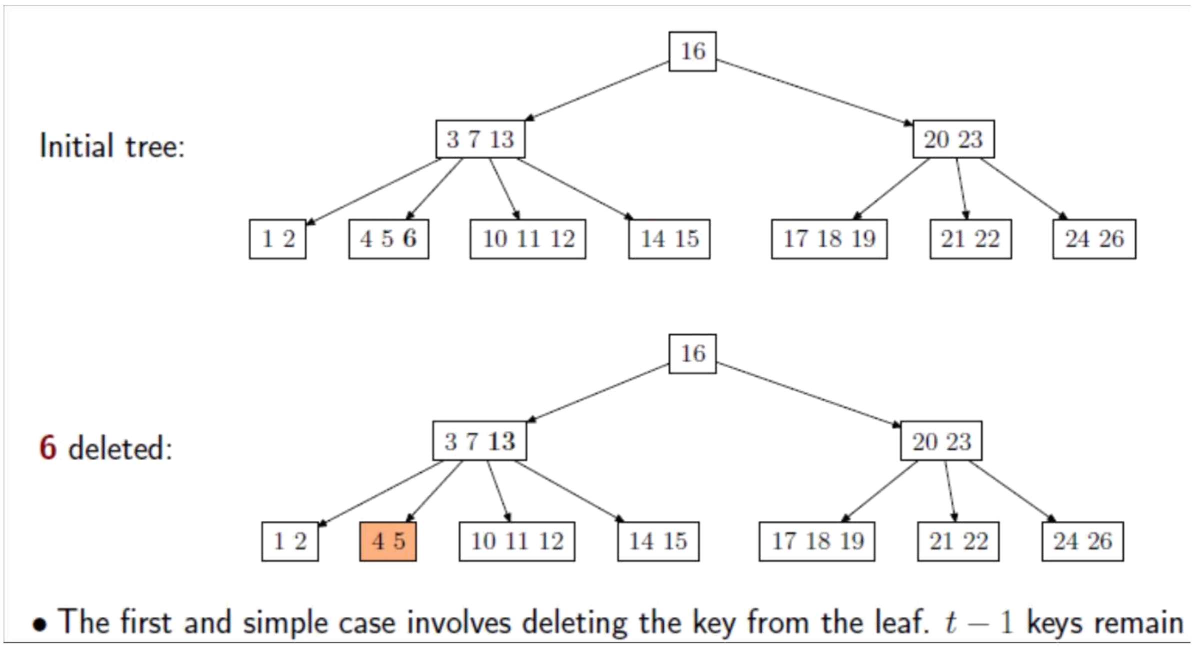
(b) Symmetrically, if the child z that follows k in node x has at least t keys, find the successor k0 and delete and replace as before. Note that finding k0 and deleting it can be performed in a single downward pass.

(c) Otherwise, if both y and z have only t−1 (minimum number) keys, merge k and all of z into y, so that both k and the pointer to z are removed from x. y now contains 2t-1 keys, and subsequently k is deleted.

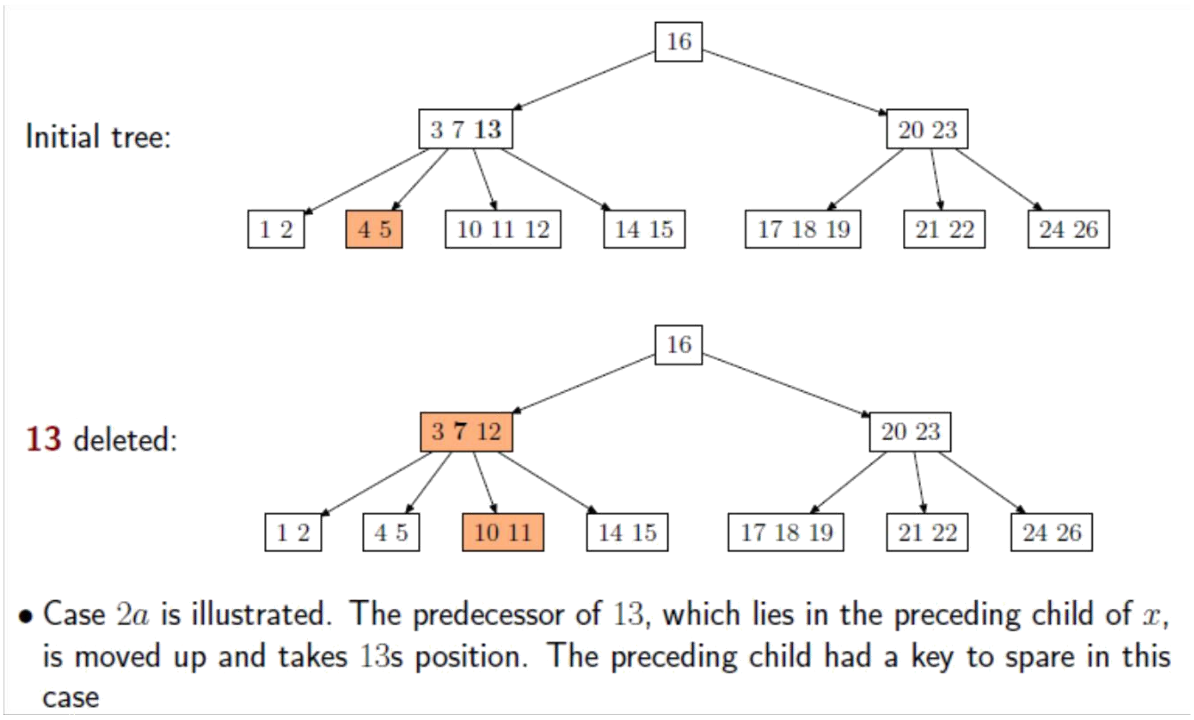
3. If key k is not present in an internal node x, determine the root of the appropriate subtree that must contain k. If the root has only t − 1 keys, execute either of the following two cases to ensure that we descend to a node containing at least t keys. Finally, recurs to the appropriate child of x.

(a) If the root has only t−1 keys but has a sibling with t keys, give the root an extra key by moving a key from x to the root, moving a key from the roots immediate left or right sibling up into x, and moving the appropriate child from the sibling to x.

(b) If the root and all of its siblings have t−1 keys, merge the root with one sibling. This involves moving a key down from x into the new merged node to become the median key for that node.

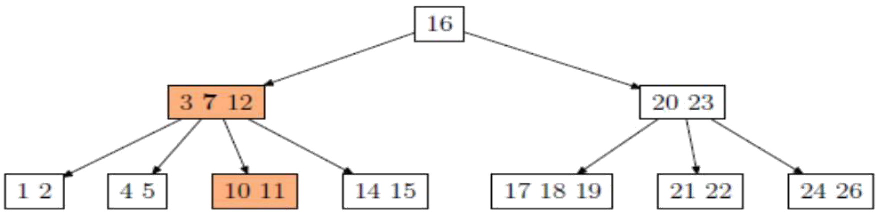


• The first and simple case involves deleting the key from the leaf. $t-1$ keys remain

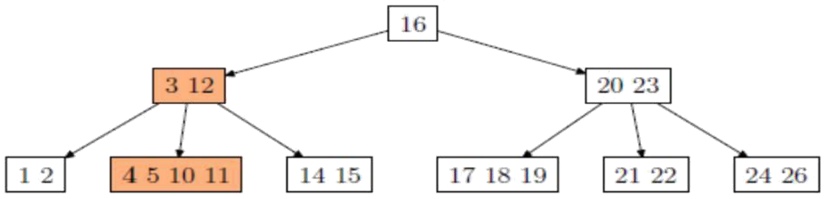


• Case $2a$ is illustrated. The predecessor of 13, which lies in the preceding child of x, is moved up and takes 13s position. The preceding child had a key to spare in this case

Initial tree:

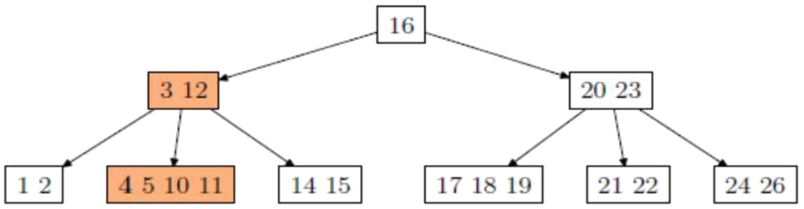


7 deleted:



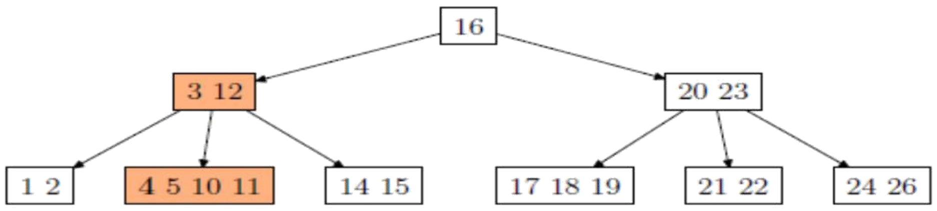
- Here, both the preceding and successor children have $t-1$ keys, the minimum allowed. 7 is initially pushed down and between the children nodes to form one leaf, and is subsequently removed from that leaf

Initial tree: Key **4** to be deleted

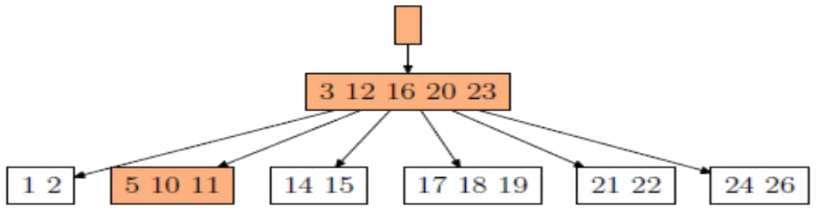


- The catchy part. Recursion cannot descend to node $3, 12$ because it has $t-1$ keys. In case the two leaves to the left and right had more than $t-1$, $3, 12$ could take one and 3 would be moved down.
- Also, the sibling of $3, 12$ has also $t-1$ keys, so it is not possible to move the root to the left and take the leftmost key from the sibling to be the new root
- Therefore the root has to be pushed down merging its two children, so that 4 can be safely deleted from the leaf

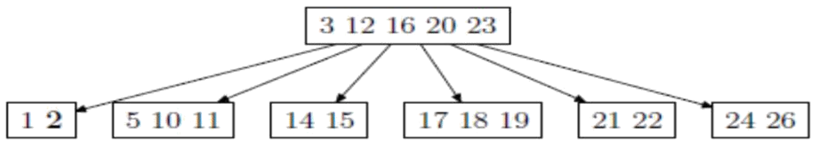
Initial tree:



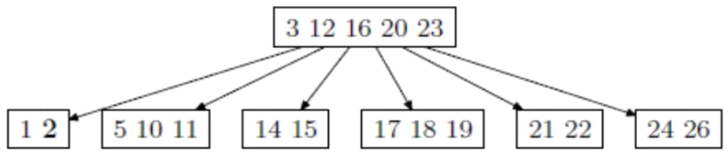
4 deleted:



Outcome:

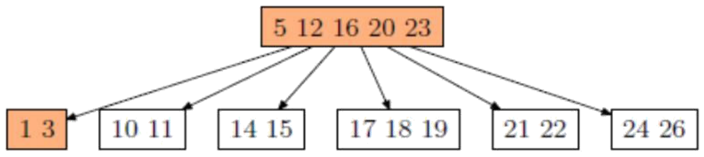


Initial tree:



2 deleted:

(to the previous one)

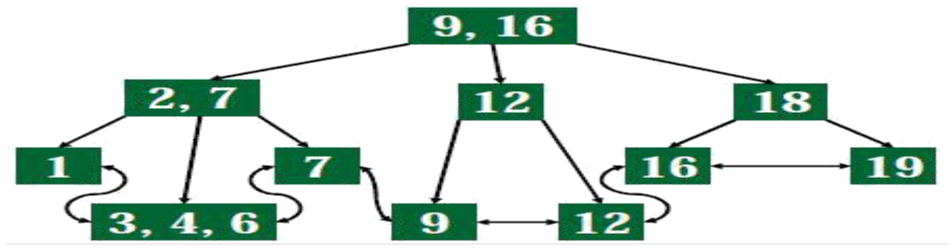


- In this case, $1, 2$ has $t - 1$ keys, but the sibling to the right has t. Recursion moves 5 to fill 3s position, 5 is moved to the appropriate leaf, and deleted from there

B+ TREE

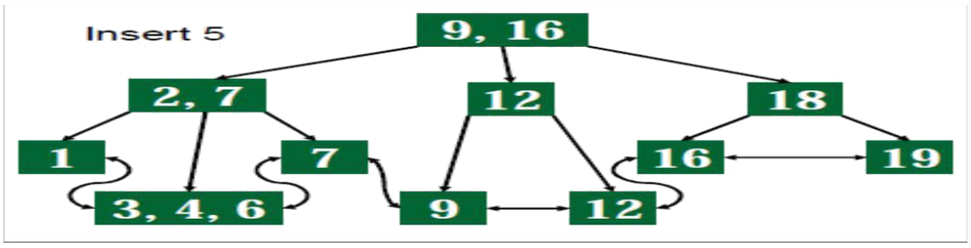
Similar to B trees, with a few slight differences All data is stored at the leaf nodes (leaf pages); all other nodes (index pages) only store keys Leaf pages are linked to each other Keys may be duplicated; every key to the right of a particular key is >= to that key.

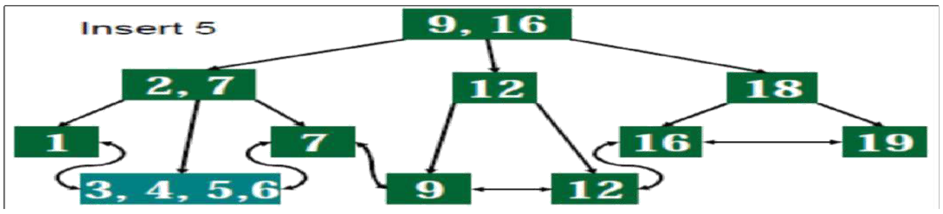
Example

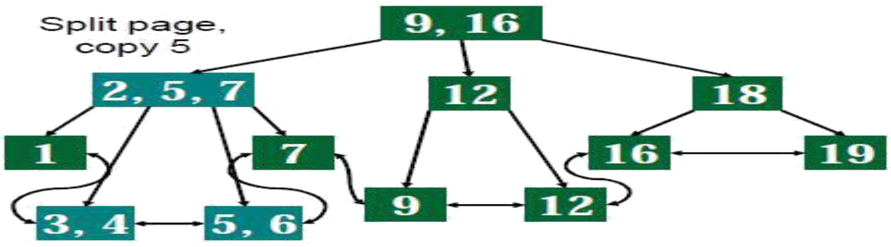


B$^+$ TREE INSERTION

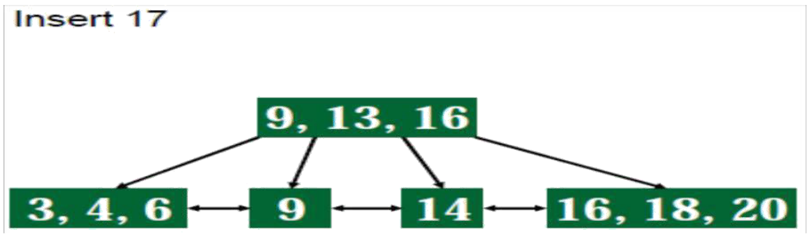
Insert at bottom level
If leaf page overflows, split page and copy middle element to next index page.
If index page overflows, split page and move middle element to next index page.

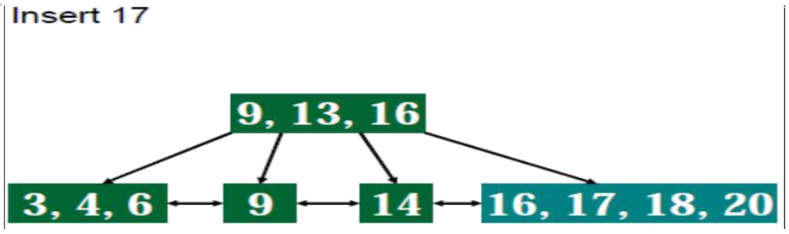


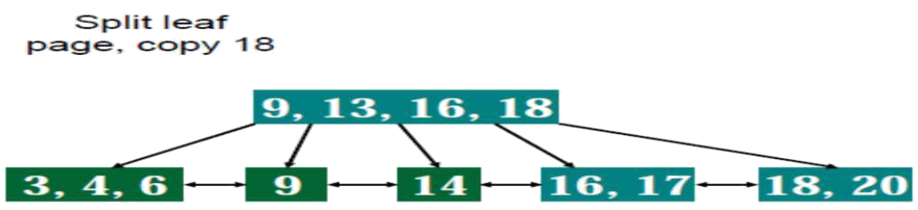


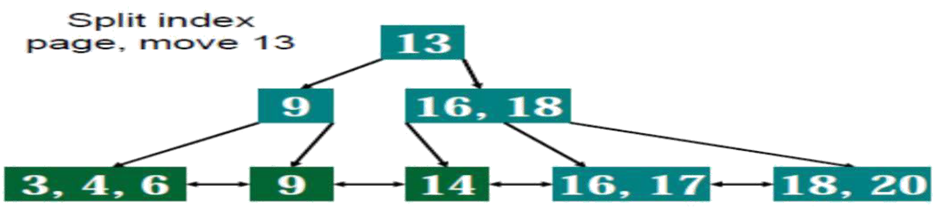


EXAMPLE 2



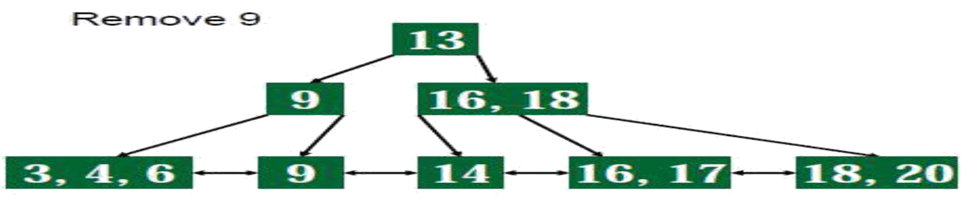


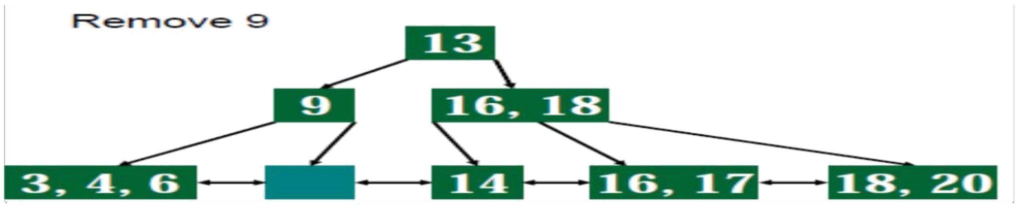


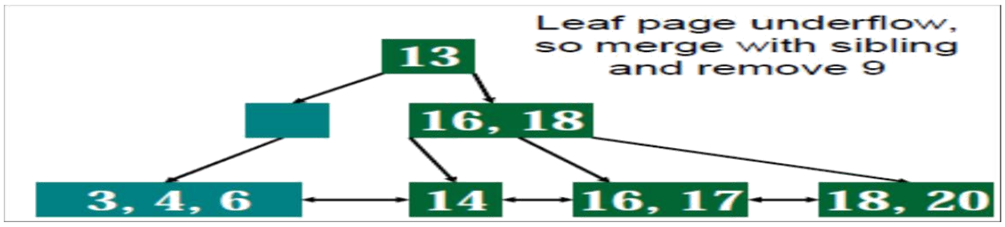


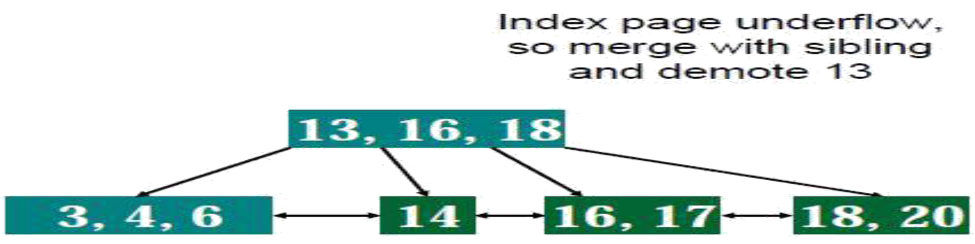
B⁺ TREE DELETION

Delete key and data from leaf page
If leaf page is underflow, merge it with sibling and delete the key in between them.
If index page underflows, merge with sibling and move down key in between them.









Lecture 25

Graphs

Introduction to Graphs:

Graph G is a pair (V, E), where V is a finite set of vertices and E is a finite set of edges. We will often denote n = |V|, e = |E|.

A graph is generally displayed as figure 6.5.1, in which the vertices are represented by circles and the edges by lines.

An edge with an orientation (i.e., arrow head) is a directed edge, while an edge with no orientation is our undirected edge.

If all the edges in a graph are undirected, then the graph is an undirected graph. The graph in figure 6.5.1(a) is an undirected graph. If all the edges are directed; then the graph is a directed graph. The graph of figure 6.5.1(b) is a directed graph. A directed graph is also called as digraph. A graph G is connected if and only if there is a simple path between any two nodes in G.

A graph G is said to be complete if every node a in G is adjacent to every other node v in G. A complete graph with n nodes will have n(n-1)/2 edges. For example, Figure 6.5.1.(a) and figure 6.5.1.(d) are complete graphs.

A directed graph G is said to be connected, or strongly connected, if for each pair (u, v) for nodes in G there is a path from u to v and also a path from v to u. On the other hand, G is said to be unilaterally connected if for any pair (u, v) of nodes in G there is a path from u to v or a path from v to u. For example, the digraph shown in figure 6.5.1 (e) is strongly connected.

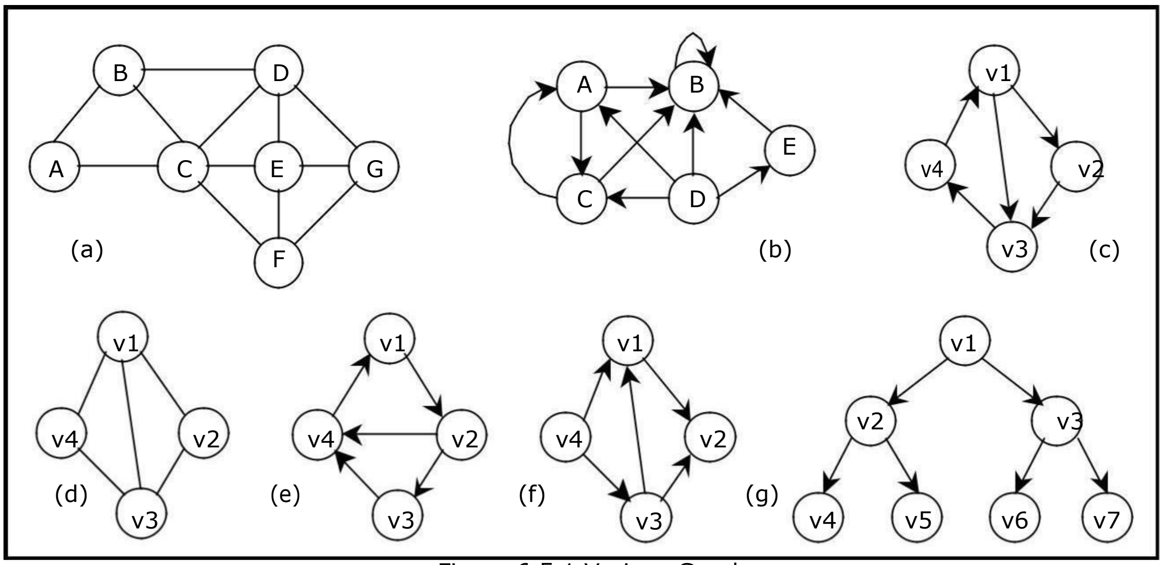


Figure 6.5.1 Various Graphs

We can assign weight function to the edges: $w_G(e)$ is a weight of edge $e \in E$. The graph which has such function assigned is called weighted graph.

The number of incoming edges to a vertex v is called in–degree of the vertex (denote indeg(v)). The number of outgoing edges from a vertex is called out-degree (denote outdeg(v)). For example, let us consider the digraph shown in figure 6.5.1(f),

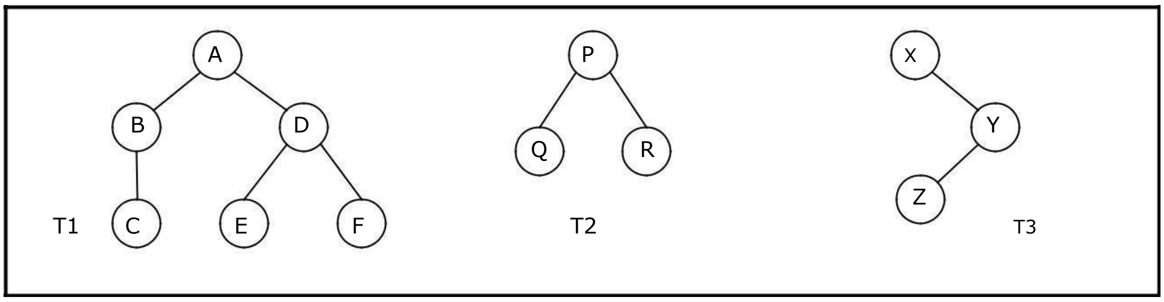
indegree(v_1) = 2	outdegree(v_1) = 1
indegree(v_2) = 2	outdegree(v_2) = 0

A path is a sequence of vertices (v_1, v_2, ... , v_k), where for all i, (v_i, v_{i+1}) ε E. A path is simple if all vertices in the path are distinct. If there is a path containing one or more edges which starts from a vertex V_i and terminates into the same vertex then the path is known as a cycle. For example, there is a cycle in figure 6.5.1(a), figure 6.5.1(c) and figure 6.5.1(d).

If a graph (digraph) does not have any cycle then it is called **acyclic graph**. For example, the graphs of figure 6.5.1 (f) and figure 6.5.1 (g) are acyclic graphs.

A graph G" = (V", E") is a sub-graph of graph G = (V, E) iff V" ⊆ V and E" ⊆ E.

A **Forest** is a set of disjoint trees. If we remove the root node of a given tree then it becomes forest. The following figure shows a forest F that consists of three trees T1, T2 and T3.



A Forest F

A graph that has either self loop or parallel edges or both is called **multi-graph**.

Tree is a connected acyclic graph (there aren"t any sequences of edges that go around in a loop). A spanning tree of a graph G = (V, E) is a tree that contains all vertices of V and is a subgraph of G. A single graph can have multiple spanning trees.

Let T be a spanning tree of a graph G. Then

1. *Any two vertices in T are connected by a unique simple path.*
2. *If any edge is removed from T, then T becomes disconnected.*
3. *If we add any edge into T, then the new graph will contain a cycle.*
4. *Number of edges in T is n-1.*

Representation of Graphs:

There are two ways of representing digraphs. They are:

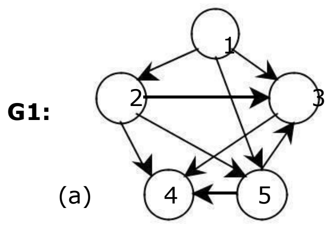
- Adjacency matrix.
- Adjacency List.
- Incidence matrix.

Adjacency matrix:

In this representation, the adjacency matrix of a graph G is a two dimensional n x n matrix, say A = ($a_{i,j}$), where

$$a_{i,j} = \begin{cases} 1 & \text{if there is an edge from } v_i \text{ to } v_j \\ 0 & \text{otherwise} \end{cases}$$

The matrix is symmetric in case of undirected graph, while it may be asymmetric if the graph is directed. This matrix is also called as Boolean matrix or bit matrix.



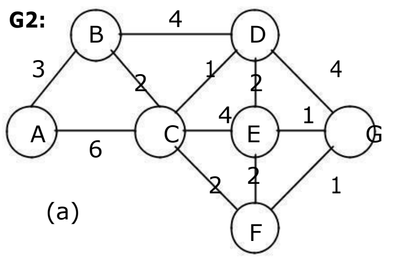
(b)

	1	2	3	4	5
1	0	1	1	0	1
2	0	0	1	1	1
3	0	0	0	1	0
4	0	0	0	0	0
5	0	0	1	1	0

Figure 6.5.2. A graph and its Adjacency matrix

Figure 6.5.2(b) shows the adjacency matrix representation of the graph G1 shown in figure 6.5.2(a). The adjacency matrix is also useful to store multigraph as well as weighted graph. In case of multigraph representation, instead of entry 0 or 1, the entry will be between number of edges between two vertices.

In case of weighted graph, the entries are weights of the edges between the vertices. The adjacency matrix for a weighted graph is called as cost adjacency matrix. Figure 6.5.3(b) shows the cost adjacency matrix representation of the graph G2 shown in figure 6.5.3(a).



(b)

	A	B	C	D	E	F	G
A	0	3	6	∝	∝	∝	∝
B	3	0	2	4	∝	∝	∝
C	6	2	0	1	4	2	∝
D	∝	4	1	0	2	∝	4
E	∝	∝	4	2	0	2	1
F	∝	∝	2	∝	2	0	1
G	∝	∝	∝	4	1	1	0

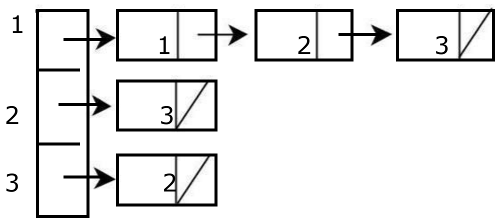
Figure 6.5.3 Weighted graph and its Cost adjacency matrix

Adjacency List:

In this representation, the n rows of the adjacency matrix are represented as n linked lists. An array Adj[1, 2, ... n] of pointers where for $1 \leq v \leq n$, Adj[v] points to a linked list containing the vertices which are adjacent to v (i.e. the vertices that can be reached from v by a single edge). If the edges have weights then these weights may also be stored in the linked list elements. For the graph G in figure 6.5.4(a), the adjacency list in shown in figure 6.5.4 (b).

$$\begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 1 & 1 & 1 \\ 2 & 0 & 0 & 1 \\ 3 & 0 & 1 & 0 \end{array}$$

(a) Adjacency Matrix



(b) Adjacency List

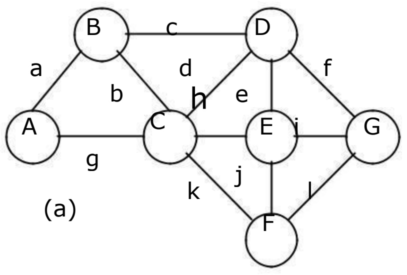
Figure 6.5.4 Adjacency matrix and adjacency list

Incidence Matrix:

In this representation, if G is a graph with n vertices, e edges and no self loops, then incidence matrix A is defined as an n by e matrix, say $A = (a_{i,j})$, where

$$a_{i,j} = \begin{cases} 1 & \text{if there is an edge } j \text{ incident to } v_i \\ 0 & \text{otherwise} \end{cases}$$

Here, n rows correspond to n vertices and e columns correspond to e edges. Such a matrix is called as vertex-edge incidence matrix or simply incidence matrix.



(a)

(b)

	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**
A	1	0	0	0	0	0	1	0	0	0	0	0
B	1	1	1	0	0	0	0	0	0	0	0	0
C	0	1	0	1	0	0	1	1	0	0	1	0
D	0	0	1	1	1	1	0	0	0	0	0	0
E	0	0	0	0	1	0	0	1	1	1	0	0
F	0	0	0	0	0	0	0	0	0	1	1	1
G	0	0	0	0	0	1	0	0	1	0	0	1

Figure 6.5.4 Graph and its incidence matrix

Figure 6.5.4(b) shows the incidence matrix representation of the graph G1 shown in figure 6.5.4(a).

Lecture 26

Traversing a Graph

Many graph algorithms require one to systematically examine the nodes and edges of a graph G. There are two standard ways to do this. They are:

- Breadth first traversal (BFT)
- Depth first traversal (DFT)

The BFT will use a queue as an auxiliary structure to hold nodes for future processing and the DFT will use a STACK.

During the execution of these algorithms, each node N of G will be in one of three states, called the *status* of N, as follows:

1. STATUS = 1 (Ready state): The initial state of the node N.

2. STATUS = 2 (Waiting state): The node N is on the QUEUE or STACK, waiting to be processed.

3. STATUS = 3 (Processed state): The node N has been processed.

Both BFS and DFS impose a tree (the BFS/DFS tree) on the structure of graph. So, we can compute a spanning tree in a graph. The computed spanning tree is not a minimum spanning tree. The spanning trees obtained using depth first search are called depth first spanning trees. The spanning trees obtained using breadth first search are called Breadth first spanning trees.

Breadth first search and traversal:

The general idea behind a breadth first traversal beginning at a starting node A is as follows. First we examine the starting node A. Then we examine all the neighbors of A. Then we examine all the neighbors of neighbors of A. And so on. We need to keep track of the neighbors of a node, and we need to guarantee that no node is processed more than once. This is accomplished by using a QUEUE to hold nodes that are waiting to be processed, and by using a field STATUS that tells us the current status of any node. The spanning trees obtained using BFS are called Breadth first spanning trees.

Breadth first traversal algorithm on graph G is as follows:

This algorithm executes a BFT on graph G beginning at a starting node A.

```
Initialize all nodes to the ready state (STATUS = 1).

1.    Put the starting node A in QUEUE and change its status to the waiting
      state (STATUS = 2).

2.    Repeat the following steps until QUEUE is empty:

      a.    Remove the front node N of QUEUE. Process N and change the
            status of N to the processed state (STATUS = 3).

      b.    Add to the rear of QUEUE all the neighbors of N that are in the
            ready state (STATUS = 1), and change their status to the waiting
            state (STATUS = 2).

3.    Exit.
```

Depth first search and traversal:

Depth first search of undirected graph proceeds as follows: First we examine the starting node V. Next an unvisited vertex 'W' adjacent to 'V' is selected and a depth first search from 'W' is initiated. When a vertex 'U' is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited, which has an unvisited vertex 'W' adjacent to it and initiate a depth first search from W. The search terminates when no unvisited vertex can be reached from any of the visited ones.

This algorithm is similar to the inorder traversal of binary tree. DFT algorithm is similar to BFT except now use a STACK instead of the QUEUE. Again field STATUS is used to tell us the current status of a node.

The algorithm for depth first traversal on a graph G is as follows.

This algorithm executes a DFT on graph G beginning at a starting node A.

1. Initialize all nodes to the ready state (STATUS = 1).

2. Push the starting node A into STACK and change its status to the waiting state (STATUS = 2).

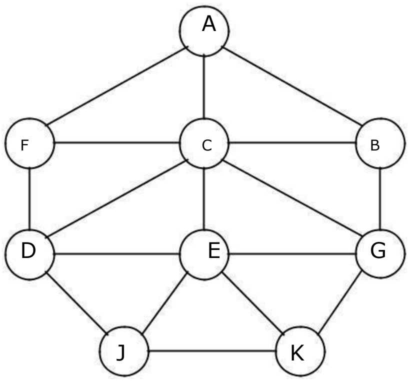
3. Repeat the following steps until STACK is empty:

    a. Pop the top node N from STACK. Process N and change the status of N to the processed state (STATUS = 3).

    b. Push all the neighbors of N that are in the ready state (STATUS = 1), and change their status to the waiting state (STATUS = 2).
4. Exit.

Example 1:

Consider the graph shown below. Traverse the graph shown below in breadth first order and depth first order.



A Gra ph G

Node	***Adjacency List***
A	F, C, B
B	A, C, G
C	A, B, D, E, F, G
D	C, F, E, J
E	C, D, G, J, K
F	A, C, D
G	B, C, E, K
J	D, E, K
K	E, G, J

Adjacency list for graph G

Lecture 27

Breadth-first search and traversal:

The steps involved in breadth first traversal are as follows:

Current Node	QUEUE	Processed Nodes	Status								
			A	B	C	D	E	F	G	J	K
			1	1	1	1	1	1	1	1	1
	A		2	1	1	1	1	1	1	1	1
A	F C B	A	3	2	2	1	1	2	1	1	1
F	C B D	A F	3	2	2	2	1	3	1	1	1
C	B D E G	A F C	3	2	3	2	2	3	2	1	1
B	D E G	A F C B	3	3	3	2	2	3	2	1	1
D	E G J	A F C B D	3	3	3	3	2	3	2	2	1
E	G J K	A F C B D E	3	3	3	3	3	3	2	2	2
G	J K	A F C B D E G	3	3	3	3	3	3	3	2	2
J	K	A F C B D E G J	3	3	3	3	3	3	3	3	2
K	EMPTY	A F C B D E G J K	3	3	3	3	3	3	3	3	3

For the above graph the breadth first traversal sequence is: ***A F C B D E G J K***.

Depth-first search and traversal:

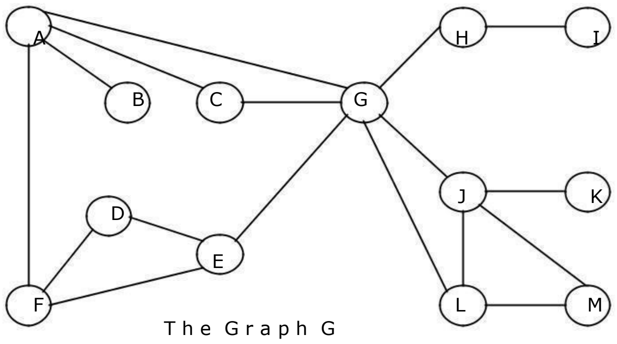
The steps involved in depth first traversal are as follows:

Current Node	Stack	Processed Nodes	Status								
			A	B	C	D	E	F	G	J	K
			1	1	1	1	1	1	1	1	1
	A		2	1	1	1	1	1	1	1	1
A	B C F	A	3	2	2	1	1	2	1	1	1
F	B C D	A F	3	2	2	2	1	3	1	1	1
D	B C E J	A F D	3	2	2	3	2	3	1	2	1
J	B C E K	A F D J	3	2	2	3	2	3	1	3	2
K	B C E G	A F D J K	3	2	2	3	2	3	2	3	3
G	B C E	A F D J K G	3	2	2	3	2	3	3	3	3
E	B C	A F D J K G E	3	2	2	3	3	3	3	3	3
C	B	A F D J K G E C	3	2	3	3	3	3	3	3	3
B	EMPTY	A F D J K G E C B	3	3	3	3	3	3	3	3	3

For the above graph the depth first traversal sequence is: ***A F D J K G E C B***.

Example 2:

Traverse the graph shown below in breadth first order, depth first order and construct the breadth first and depth first spanning trees.

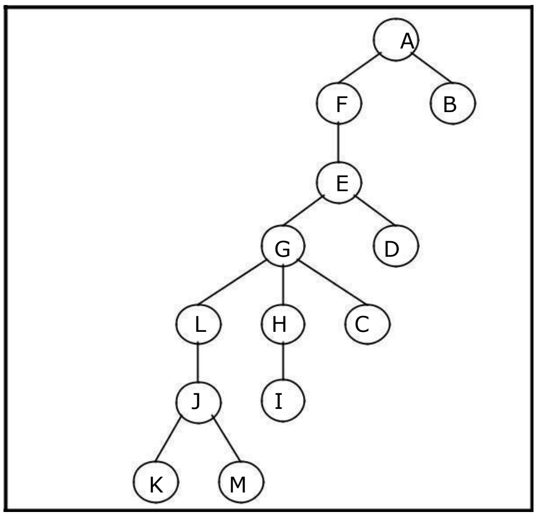


The Graph G

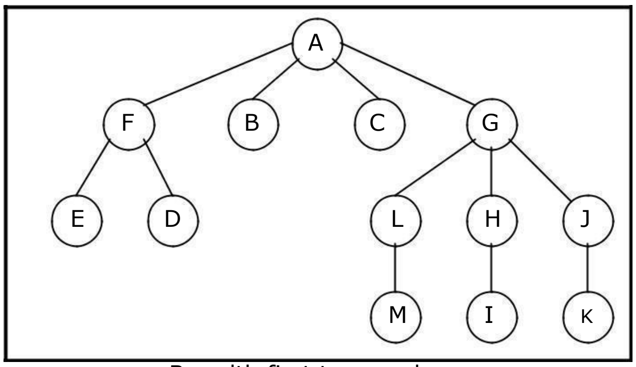
Node	*Adjacency List*
A	F, B, C, G
B	A
C	A, G
D	E, F
E	G, D, F
F	A, E, D
G	A, L, E, H, J, C
H	G, I
I	H
J	G, L, K, M
K	J
L	G, J, M
M	L, J

The adjacency list for the graph G

If the depth first traversal is initiated from vertex A, then the vertices of graph G are visited in the order: ***A F E G L J K M H I C D B***. The depth first spanning tree is shown in the figure given below:

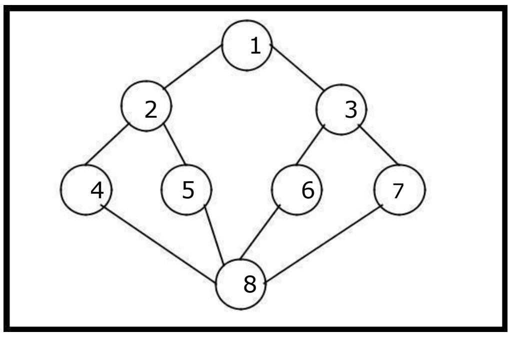

Depth first Traversal

If the breadth first traversal is initiated from vertex A, then the vertices of graph G are visited in the order: ***A F B C G E D L H J M I K***. The breadth first spanning tree is shown in the figure given below:


Breadth first traversal

Example 3:

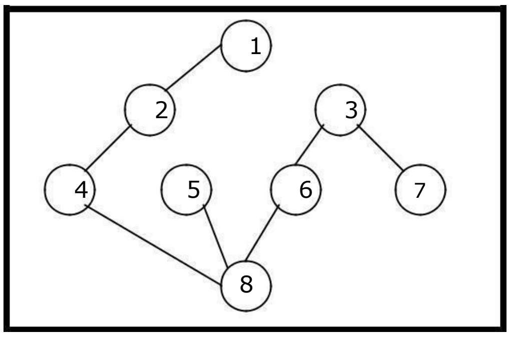
Traverse the graph shown below in breadth first order, depth first order and construct the breadth first and depth first spanning trees.



Graph G

Depth first search and traversal:

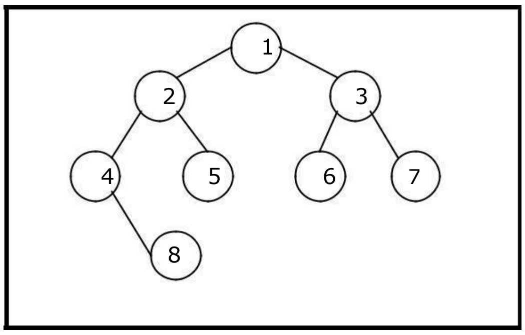
If the depth first is initiated from vertex 1, then the vertices of graph G are visited in the order: 1, 2, 4, 8, 5, 6, 3, 7. The depth first spanning tree is as follows:



Depth First Spanning Tree

Breadth first search and traversal:

If the breadth first search is initiated from vertex 1, then the vertices of G are visited in the order: 1, 2, 3, 4, 5, 6, 7, 8. The breadth first spanning tree is as follows:



Breadth First Spanning Tree

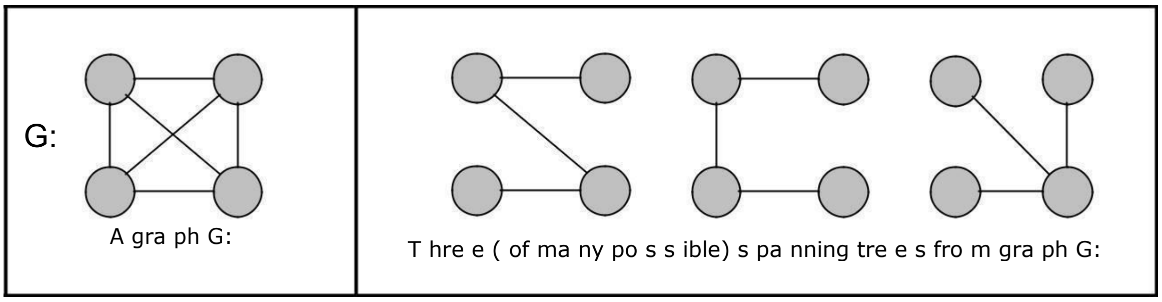
Lecture 28

Minimum Spanning Tree (MST):

A spanning tree for a connected graph is a tree whose vertex set is the same as the vertex set of the given graph, and whose edge set is a subset of the edge set of the given graph. i.e., any connected graph will have a spanning tree.

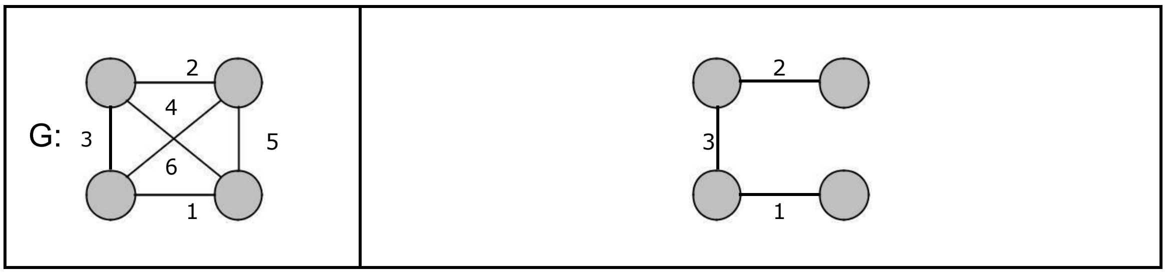
Weight of a spanning tree w(T) is the sum of weights of all edges in T. Minimum spanning tree (MST) is a spanning tree with the smallest possible weight.

Example:



A gra ph G:

T hre e (of ma ny po s s ible) s pa nning tre e s fro m gra ph G:



A w e ight e d gra ph G:

T he mini ma l s pa nning tre e fro m w e ight e d gra ph G:

Let's consider a couple of real-world examples on minimum spanning tree:

- One practical application of a MST would be in the design of a network. For instance, a group of individuals, who are separated by varying distances, wish to be connected together in a telephone network. Although MST cannot do anything about the distance from one connection to another, it can be used to determine the least cost paths with no cycles in this network, thereby connecting everyone at a minimum cost.

- Another useful application of MST would be finding airline routes. The vertices of the graph would represent cities, and the edges would represent routes between the cities. MST can be applied to optimize airline routes by finding the least costly paths with no cycles.

Minimum spanning tree, can be constructed using any of the following two algorithms:

1. Kruskal"s algorithm and
2. Prim"s algorithm.

Both algorithms differ in their methodology, but both eventually end up with the MST. *Kruskal's algorithm uses edges, and Prim's algorithm uses vertex connections* in determining the MST. In *Prim's algorithm at any instance of output it represents tree* whereas in *Kruskal's algorithm at any instance of output it may represent tree or not*.

Kruskal's Algorithm

This is a greedy algorithm. A greedy algorithm chooses some local optimum (i.e. picking an edge with the least weight in a MST).

Kruskal's algorithm works as follows: Take a graph with 'n' vertices, keep on adding the shortest (least cost) edge, while avoiding the creation of cycles, until (n - 1) edges have been added. Sometimes two or more edges may have the same cost.

The order in which the edges are chosen, in this case, does not matter. Different MST"s may result, but they will all have the same total cost, which will always be the minimum cost.

Kruskal"s Algorithm for minimal spanning tree is as follows:

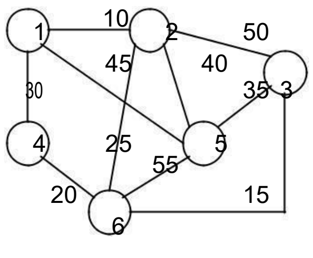
1. Make the tree T empty.
2. Repeat the steps 3, 4 and 5 as long as T contains less than n - 1 edges and E is not empty otherwise, proceed to step 6.
3. Choose an edge (v, w) from E of lowest cost.
4. Delete (v, w) from E.
5. If (v, w) does not create a cycle in T

   then Add (v, w) to T

   else discard (v, w)
6. If T contains fewer than n - 1 edges then print no spanning tree.

Example 1:

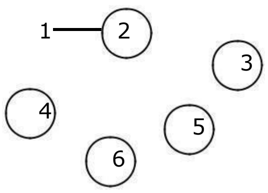
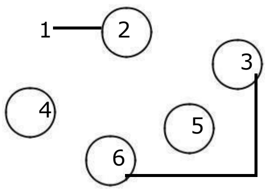
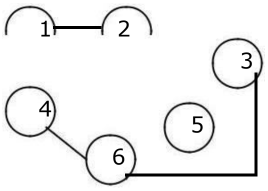
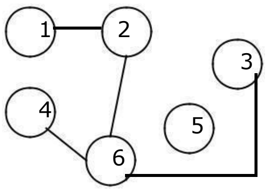
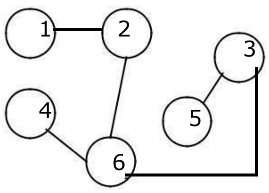
Construct the minimal spanning tree for the graph shown below:



Arrange all the edges in the increasing order of their costs:

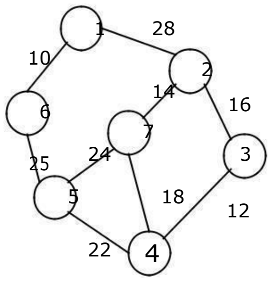
Cost	***10***	***15***	***20***	***25***	***30***	***35***	***40***	***45***	***50***	***55***
Edge	(1, 2)	(3, 6)	(4, 6)	(2, 6)	(1, 4)	(3, 5)	(2, 5)	(1, 5)	(2, 3)	(5, 6)

The stages in Kruskal"s algorithm for minimal spanning tree is as follows:

EDGE	COST	STAGES IN KRUSKAL'S ALGORITHM	REMARKS
(1, 2)	10		The edge between vertices 1 and 2 is the first edge selected. It is included in the spanning tree.
(3, 6)	15		Next, the edge between vertices 3 and 6 is selected and included in the tree.
(4, 6)	20		The edge between vertices 4 and 6 is next included in the tree.
(2, 6)	25		The edge between vertices 2 and 6 is considered next and included in the tree.
(1, 4)	30	Reject	The edge between the vertices 1 and 4 is discarded as its inclusion creates a cycle.
(3, 5)	35		Finally, the edge between vertices 3 and 5 is considered and included in the tree built. This completes the tree. The cost of the minimal spanning tree is 105.

Example 2:

Construct the minimal spanning tree for the graph shown below:

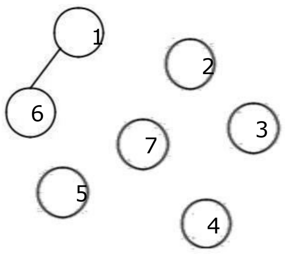
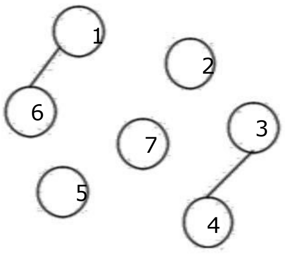
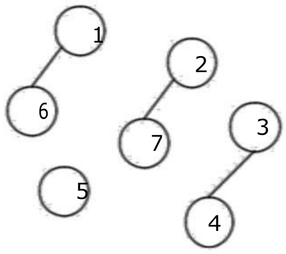


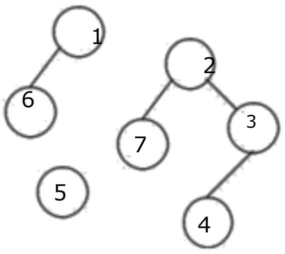
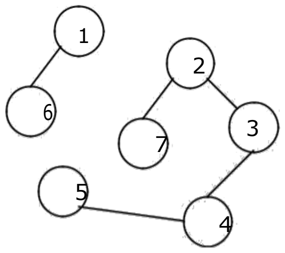
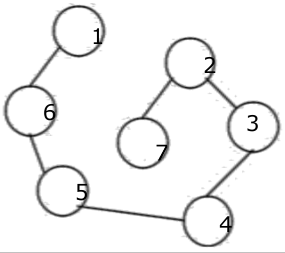
Solution:

Arrange all the edges in the increasing order of their costs:

Cost	***10***	***12***	***14***	***16***	***18***	***22***	***24***	***25***	***28***
Edge	(1, 6)	(3, 4)	(2, 7)	(2, 3)	(4, 7)	(4, 5)	(5, 7)	(5, 6)	(1, 2)

The stages in Kruskal"s algorithm for minimal spanning tree is as follows:

EDGE	**COST**	**STAGES IN KRUSKAL'S ALGORITHM**	**REMARKS**
(1, 6)	10		The edge between vertices 1 and 6 is the first edge selected. It is included in the spanning tree.
(3, 4)	12		Next, the edge between vertices 3 and 4 is selected and included in the tree.
(2, 7)	14		The edge between vertices 2 and 7 is next included in the tree.

(2, 3)	16		The edge between vertices 2 and 3 is next included in the tree.
(4, 7)	18	Reject is discarded as its inclusion creates a	The edge between the vertices 4 and 7 cycle.
(4, 5)	22		The edge between vertices 4 and 7 is considered next and included in the tree.
(5, 7)	24	Reject	The edge between the vertices 5 and 7 is discarded as its inclusion creates a cycle.
(5, 6)	25		Finally, the edge between vertices 5 and 6 is considered and included in the tree built. This completes the tree. The cost of the minimal spanning tree is 99.

MINIMUM-COST SPANNING TREES: PRIM'S ALGORITHM

A given graph can have many spanning trees. From these many spanning trees, we have to select a cheapest one. This tree is called as minimal cost spanning tree.

Minimal cost spanning tree is a connected undirected graph G in which each edge is labeled with a number (edge labels may signify lengths, weights other than costs). Minimal cost spanning tree is a spanning tree for which the sum of the edge labels is as small as possible

The slight modification of the spanning tree algorithm yields a very simple algorithm for finding an MST. In the spanning tree algorithm, any vertex not in the tree but connected to it by an edge can be added. To find a Minimal cost spanning tree, we must be selective - we must always add a new vertex for which the cost of the new edge is as small as possible.

This simple modified algorithm of spanning tree is called prim's algorithm for finding an Minimal cost spanning tree. Prim's algorithm is an example of a greedy algorithm.

Prim's Algorithm:

E is the set of edges in G. cost [1:n, 1:n] is the cost adjacency matrix of an n vertex graph such that cost [i, j] is either a positive real number or ∝ if no edge (i, j) exists. A minimum spanning tree is computed and stored as a set of edges in the array t [1:n-1, 1:2]. (t [i, 1], t [i, 2]) is an edge in the minimum-cost spanning tree. The final cost is returned.

Algorithm Prim (E, cost, n, t)

```
{
        Let (k, l) be an edge of minimum cost in E;
        mincost := cost [k, l];
        t [1, 1] := k; t [1, 2] := l;
        for  i :=1 to n do                          // Initialize near
                if (cost [i, l] < cost [i, k]) then near [i] := l;
                else near [i] := k;
        near [k] :=near [l] := 0;
        for  i:=2 to n - 1 do                       // Find n - 2 additional edges for t.
        {
                Let j be an index such that near [j] ≠ 0
                and cost [j, near [j]] is minimum;
                t [i, 1] := j; t [i, 2] := near [j];
                mincost := mincost + cost [j, near
                [j]]; near [j] := 0
                        for k:= 1 to n do               // Update near[]. if
                           ((near [k] ≠ 0) and (cost [k, near [k]] > cost [k,
                                 j])) then near [k] := j;
        }
        return mincost;
}
```

Module IV

Searching and Sorting

Lecture 29

Sorting

Bubble Sort:

The bubble sort is easy to understand and program. The basic idea of bubble sort is to pass through the file sequentially several times. In each pass, we compare each element in the file with its successor i.e., X[i] with X[i+1] and interchange two element when they are not in proper order. We will illustrate this sorting technique by taking a specific example. Bubble sort is also called as exchange sort.

Example:

Consider the array x[n] which is stored in memory as shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]
33	44	22	11	66	55

Suppose we want our array to be stored in ascending order. Then we pass through the array 5 times as described below:

Pass 1: (first element is compared with all other elements).

We compare X[i] and X[i+1] for i = 0, 1, 2, 3, and 4, and interchange X[i] and X[i+1] if X[i] > X[i+1]. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]	Remarks
33	44	22	11	66	55	
	22	44				
		11	44			
			44	66		
				55	66	
33	22	11	44	55	66	

The biggest number 66 is moved to (bubbled up) the right most position in the array.

Pass 2: (second element is compared).

We repeat the same process, but this time we don"t include X[5] into our comparisons. i.e., we compare X[i] with X[i+1] for i=0, 1, 2, and 3 and interchange X[i] and X[i+1] if X[i] > X[i+1]. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	Remarks
33	22	11	44	55	
22	33				
	11	33			
		33	44		
			44	55	
22	11	33	44	55	

The second biggest number 55 is moved now to X[4].

Pass 3: (third element is compared).

We repeat the same process, but this time we leave both X[4] and X[5]. By doing this, we move the third biggest number 44 to X[3].

X[0]	X[1]	X[2]	X[3]	Remarks
22	11	33	44	
11	22			
	22	33		
		33	44	
11	22	33	44	

Pass 4: (fourth element is compared).

We repeat the process leaving X[3], X[4], and X[5]. By doing this, we move the fourth biggest number 33 to X[2].

X[0]	X[1]	X[2]	Remarks
11	22	33	
11	22		
	22	33	

Pass 5: (fifth element is compared).

We repeat the process leaving X[2], X[3], X[4], and X[5]. By doing this, we move the fifth biggest number 22 to X[1]. At this time, we will have the smallest number 11 in X[0]. Thus, we see that we can sort the array of size 6 in 5 passes.

For an array of size n, we required (n-1) passes.

Program for Bubble Sort:

```
#include <stdio.h>
#include <conio.h>
void bubblesort(int x[], int n)
{
        int i, j, temp;
        for (i = 0; i < n; i++)
        {
                for (j = 0; j < n–i-1 ; j++)
                {
                        if (x[j] > x[j+1])
                        {
                                temp = x[j];
                                x[j] = x[j+1];
                                x[j+1] = temp;
                        }
                }
        }
}

main()
{
        int i, n, x[25];
        clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter Data:");
        for(i = 0; i < n ; i++)
                scanf("%d", &x[i]);
        bubblesort(x, n);
        printf ("\n Array Elements after sorting: ");
        for (i = 0; i < n; i++)
                printf ("%5d", x[i]);
}
```

Time Complexity:

The bubble sort method of sorting an array of size n requires (n-1) passes and (n-1) comparisons on each pass. Thus the total number of comparisons is (n-1) * (n-1) = $n^2 - 2n + 1$, which is $O(n^2)$. Therefore bubble sort is very inefficient when there are more elements to sorting.

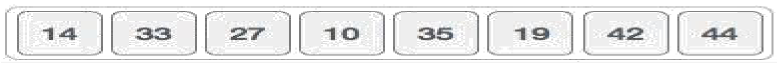
Insertion Sort:

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be inserted in this sorted sub-list has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.

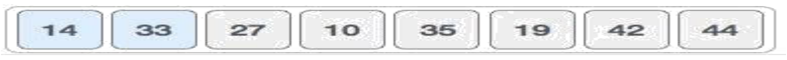
The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n^2), where **n** is the number of items.

Example

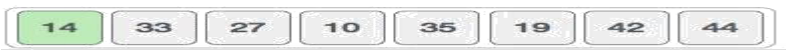
Let we take an unsorted array for our example.



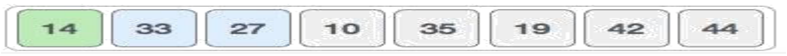
Insertion sort compares the first two elements.



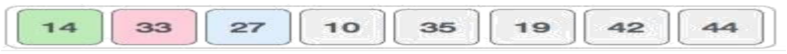
It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



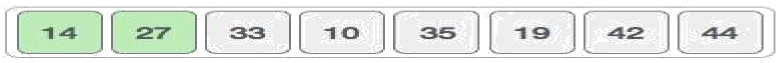
Insertion sort moves ahead and compares 33 with 27.



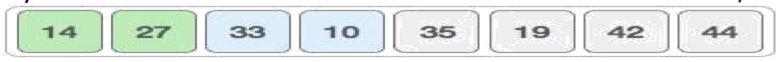
And finds that 33 is not in the correct position.



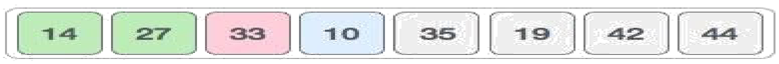
It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping



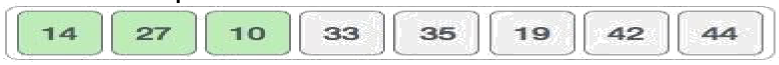
By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.



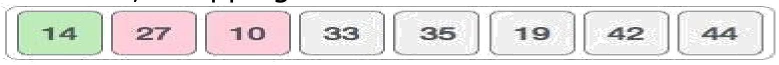
These values are not in a sorted order.



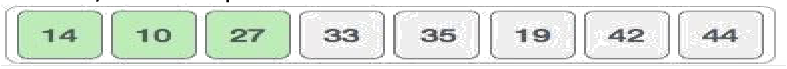
So we swap them.



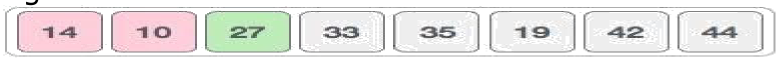
However, swapping makes 27 and 10 unsorted.



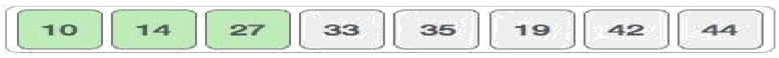
Hence, we swap them too.



Again we find 14 and 10 in an unsorted order.



We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.



This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

1 If it is the first element, it is already sorted. return 1
2 Pick next element
3 Compare with all elements in the sorted sub-list
4 Shift all the elements in the sorted sub-list that is greater than the value to be sorted
5 Insert the value
6 Repeat until list is sorted

Program for selection sort:

```
#include <stdio.h>
#include <math.h>

/* Function to sort an array using insertion sort*/
void insertionSort(int arr[], int n) {

   int i, key, j;
   for (i = 1; i < n; i++)
   {
      key = arr[i];
      j = i-1;

      /* Move elements of arr[0..i-1], that are
         greater than key, to one position ahead
         of their current position */
      while (j >= 0 && arr[j] > key)
      {
          arr[j+1] = arr[j];
          j = j-1;
      }
      arr[j+1] = key;
   }
}

// A utility function ot print an array of size n
void printArray(int arr[], int n)
{
   int i;
   for (i=0; i < n; i++)
      printf("%d ", arr[i]);
   printf("\n");
}
 /* Driver program to test insertion sort */
int main()
{
    int arr[] = {12, 11, 13, 5, 6};
    int n = sizeof(arr)/sizeof(arr[0]);
    insertionSort(arr, n);
    printArray(arr, n);

    return 0;
}
```

Complexity

If we take a closer look at the insertion sort code, we can notice that every iteration of while loop reduces one inversion. The while loop executes only if i > j and arr[i] < arr[j]. Therefore total number of while loop iterations (For all values of i) is same as number of inversions. Therefore overall time complexity of the insertion sort is O(n + f(n)) where f(n) is inversion count. If the inversion count is O(n), then the time complexity of insertion sort is O(n). In worst case, there can be n*(n-1)/2 inversions. The worst case occurs when the array is sorted in reverse order. So the worst case time complexity of insertion sort is O(n^2).

Selection Sort:

Selection sort will not require no more than n-1 interchanges. Suppose x is an array of size n stored in memory. The selection sort algorithm first selects the smallest element in the array x and place it at array position 0; then it selects the next smallest element in the array x and place it at array position 1. It simply continues this procedure until it places the biggest element in the last position of the array.
The array is passed through (n-1) times and the smallest element is placed in its respective position in the array as detailed below:

Pass 1: Find the location j of the smallest element in the array x [0], x[1], ... x[n-1], and then interchange x[j] with x[0]. Then x[0] is sorted.

Pass 2: Leave the first element and find the location j of the smallest element in the sub-array x[1], x[2], ... x[n-1], and then interchange x[1] with x[j]. Then x[0], x[1] are sorted.

Pass 3: Leave the first two elements and find the location j of the smallest element in the sub-array x[2], x[3], ... x[n-1], and then interchange x[2] with x[j]. Then x[0], x[1], x[2] are sorted.

Pass (n-1): Find the location j of the smaller of the elements x[n-2] and x[n-1], and then interchange x[j] and x[n-2]. Then x[0], x[1], ... x[n-2] are sorted. Of course, during this pass x[n-1] will be the biggest element and so the entire array is sorted.

Time Complexity:

In general we prefer selection sort in case where the insertion sort or the bubble sort requires exclusive swapping. In spite of superiority of the selection sort over bubble sort and the insertion sort (there is significant decrease in run time), its efficiency is also $O(n^2)$ for n data items.

Example:

Let us consider the following example with 9 elements to analyze selection Sort:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	***Remarks***
65	70	75	80	50	60	55	85	45	find the first smallest element
i								j	swap a[i] & a[j]
45	70	75	80	50	60	55	85	65	find the second smallest element
	i			j					swap a[i] and a[j]
45	**50**	75	80	70	60	55	85	65	Find the third smallest element
		i				j			swap a[i] and a[j]
45	**50**	**55**	80	70	60	75	85	65	Find the fourth smallest element
			i		j				swap a[i] and a[j]
45	**50**	**55**	**60**	70	80	75	85	65	Find the fifth smallest element
				i				j	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	80	75	85	70	Find the sixth smallest element
					i			j	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	75	85	80	Find the seventh smallest element
						i j			swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	**75**	85	80	Find the eighth smallest element
							i	J	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	**75**	**80**	**85**	The outer loop ends.

Non-recursive Program for selection sort:

```
# include<stdio.h>
# include<conio.h>

void selectionSort( int low, int high );

int a[25];

int main()
{
        int num, i= 0;
        clrscr();
        printf( "Enter the number of elements: " );
        scanf("%d", &num);
        printf( "\nEnter the elements:\n" );
        for(i=0; i < num; i++)
                scanf( "%d", &a[i] );
        selectionSort( 0, num - 1 );
        printf( "\nThe elements after sorting are: " );
        for( i=0; i< num; i++ )
                printf( "%d ", a[i] );
        return 0;
}

void selectionSort( int low, int high )
{
        int i=0, j=0, temp=0, minindex;
        for( i=low; i <= high; i++ )
        {
                minindex = i;
                for( j=i+1; j <= high; j++ )
                {
                        if( a[j] < a[minindex] )
                                minindex = j;
                }
                temp = a[i];
                a[i] = a[minindex];
                a[minindex] = temp;
        }
}
```

Recursive Program for selection sort:

```
#include <stdio.h>
#include<conio.h>

int x[6] = {77, 33, 44, 11, 66};
selectionSort(int);

main()
{
        int i, n = 0;
        clrscr();
        printf (" Array Elements before sorting: ");
        for (i=0; i<5; i++)
```

```
                printf ("%d ", x[i]);
        selectionSort(n); /* call selection sort */ printf ("\n Array
        Elements after sorting: "); for (i=0; i<5; i++)

                printf ("%d ", x[i]);
}

selectionSort( int n)
{
        int k, p, temp, min;
        if (n== 4)
                return (-1);
        min = x[n];
        p = n;
        for (k = n+1; k<5; k++)
        {
                if (x[k] <min)
                {
                        min = x[k];
                        p = k;
                }
        }
        temp = x[n]; /* interchange x[n] and x[p] */ x[n] =
        x[p];
        x[p] = temp;
        n++ ;
        selectionSort(n);
}
```

Lecture 30

Quick Sort:

The quick sort was invented by Prof. C. A. R. Hoare in the early 1960"s. It was one of the first most efficient sorting algorithms. It is an example of a class of algorithms that work by "divide and conquer" technique.

The quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosen value. The chosen value is known as the *pivot* element. Once the array has been rearranged in this way with respect to the *pivot*, the same partitioning procedure is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted.

The function partition() makes use of two pointers up and down which are moved toward each other in the following fashion:

1. Repeatedly increase the pointer „up" until a[up] >= pivot.
2. Repeatedly decrease the pointer „down" until a[down] <= pivot.
3. If down > up, interchange a[down] with a[up]
4. Repeat the steps 1, 2 and 3 till the „up" pointer crosses the „down" pointer. If „up" pointer crosses „down" pointer, the position for pivot is found and place pivot element in „down" pointer position.

The program uses a recursive function quicksort(). The algorithm of quick sort function sorts all elements in an array „a" between positions „low" and „high".

1. It terminates when the condition low >= high is satisfied. This condition will be satisfied only when the array is completely sorted.

2. Here we choose the first element as the „pivot". So, pivot = x[low]. Now it calls the partition function to find the proper position j of the element x[low] i.e. pivot. Then we will have two sub-arrays x[low], x[low+1], ... x[j-1] and x[j+1], x[j+2], . . . x[high].

3. It calls itself recursively to sort the left sub-array x[low], x[low+1], ... x[j-1] between positions low and j-1 (where j is returned by the partition function).

4. It calls itself recursively to sort the right sub-array x[j+1], x[j+2], . . x[high] between positions j+1 and high.

The time complexity of quick sort algorithm is of ***O(n log n)***.

Algorithm

Sorts the elements a[p], ... ,a[q] which reside in the global array a[n] into ascending order. The a[n + 1] is considered to be defined and must be greater than all elements in a[n]; a[n + 1] = + ∝

```
quicksort (p, q)
{
      if ( p < q ) then
      {
           call j = PARTITION(a, p, q+1); // j is the position of the partitioning element
           call quicksort(p, j – 1);
           call quicksort(j + 1 , q);
       }
}
```

```
partition(a, m, p)
{
       v = a[m]; up = m; down = p;        // a[m] is the partition element
       do
       {
              repeat
                     up = up + 1;
              until (a[up] ≥ v);

              repeat
                     down = down – 1;
              until (a[down] ≤ v);
              if (up < down) then call interchange(a, up,
        down); } while (up ≥ down);

       a[m] = a[down];
       a[down] = v;
       return (down);
}
```

```
interchange(a, up, down)
{
       p = a[up];
       a[up] = a[down];
       a[down] = p;
}
```

Example:

Select first element as the pivot element. Move „up" pointer from left to right in search of an element larger than pivot. Move the „down" pointer from right to left in search of an element smaller than pivot. If such elements are found, the elements are swapped.

This process continues till the „up" pointer crosses the „down" pointer. If „up" pointer crosses „down" pointer, the position for pivot is found and interchange pivot and element at „down" position.

Let us consider the following example with 13 elements to analyze quick sort:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*Remarks*
38	08	16	06	79	57	24	56	02	58	04	70	45	
pivot				up						down			swap up & down
pivot				04						79			
pivot					up			down					swap up & down
pivot					02			57					
pivot						down	up						swap pivot & down
(24	08	16	06	04	02)	**38**	(56	57	58	79	70	45)	
pivot					down	up							swap pivot & down
(02	08	16	06	04)	**24**								
pivot, down	up												swap pivot & down
02	(08	16	06	04)									
	pivot	up		down									swap up & down
	pivot	04		16									
	pivot		down	Up									
	(06	04)	**08**	(16)									swap pivot & down
	pivot	down	up										
	(04)	**06**											swap pivot & down
	04 pivot, down, up												
				16 pivot, down, up									
(02	**04**	**06**	**08**	**16**	**24)**	38							

							(56	57	58	79	70	45**)**	
							pivot	up				down	swap up & down
							pivot	45				57	
							pivot	down	up				swap pivot & down
							(45)	**56**	(58	79	70	57)	
							45 pivot, down, up						swap pivot & down
									(58 pivot	79 up	70	57) down	swap up & down
										57		79	
										down	up		
									(57)	**58**	(70	79)	swap pivot & down
									57 pivot, down, up				
											(70	79)	
											pivot , down	up	swap pivot & down
											70		
												79 pivot, down , up	
							(45	**56**	**57**	**58**	**70**	**79)**	
02	**04**	**06**	**08**	**16**	**24**	**38**	**45**	**56**	**57**	**58**	**70**	**79**	

Recursive program for Quick Sort:

```
# include<stdio.h>
# include<conio.h>

void quicksort(int, int);
int partition(int, int); void
interchange(int, int); int
array[25];

int main()
{
        int num, i = 0;
        clrscr();
        printf( "Enter the number of elements: " );
        scanf( "%d", &num);
        printf( "Enter the elements: " );
        for(i=0; i < num; i++)
                scanf( "%d", &array[i] );
        quicksort(0, num -1);
        printf( "\nThe elements after sorting are: " );
```

```
        for(i=0; i < num; i++)
                printf("%d ", array[i]);
        return 0;
}

void quicksort(int low, int high)
{
        int pivotpos;
        if( low < high )
        {
                pivotpos = partition(low, high + 1);
                quicksort(low, pivotpos - 1);
                quicksort(pivotpos + 1, high);
        }
}

int partition(int low, int high)
{
        int pivot = array[low];
        int up = low, down = high;

        do
        {
                do
                        up = up + 1;
                while(array[up] < pivot );

                do
                        down = down - 1;
                while(array[down] > pivot);

                if(up < down) interchange(up,
                        down);

        } while(up < down);
        array[low] = array[down];
        array[down] = pivot;
        return down;
}

void interchange(int i, int j)
{
        int temp;
        temp = array[i];
        array[i] = array[j];
        array[j] = temp;
}
```

Lecture 31

Merge Sort

The basic concept of merge sort is divides the list into two smaller sub-lists of approximately
equal size. Recursively repeat this procedure till only one element is left in the sub-list.

After this, various sorted sub-lists are merged to form sorted parent list. This process goes on recursively till the original sorted list arrived.

Algorithm :
Merge sort is based on the divide-and-conquer paradigm. Its worst-case running time has a lower order of growth than insertion sort. Since we are dealing with sub-problems, we state each subproblem as sorting a sub-array A[p .. r]. Initially, p = 1 and r = n, but these values change as we recurse through sub-problems.

To sort A[p .. r]:

1. Divide Step

If a given array A has zero or one element, simply return; it is already sorted. Otherwise, split A[p .. r] into two sub-arrays A[p .. q] and A[q + 1 .. r], each containing about half of the elements of A[p .. r]. That is, q is the halfway point of A[p .. r].

2. Conquer Step

Conquer by recursively sorting the two sub-arrays A[p .. q] and A[q + 1 .. r].

3. Combine Step

Combine the elements back in A[p .. r] by merging the two sorted sub-arrays A[p .. q] and A[q + 1 .. r] into a sorted sequence. To accomplish this step, we will define a procedure MERGE (A, p, q, r).

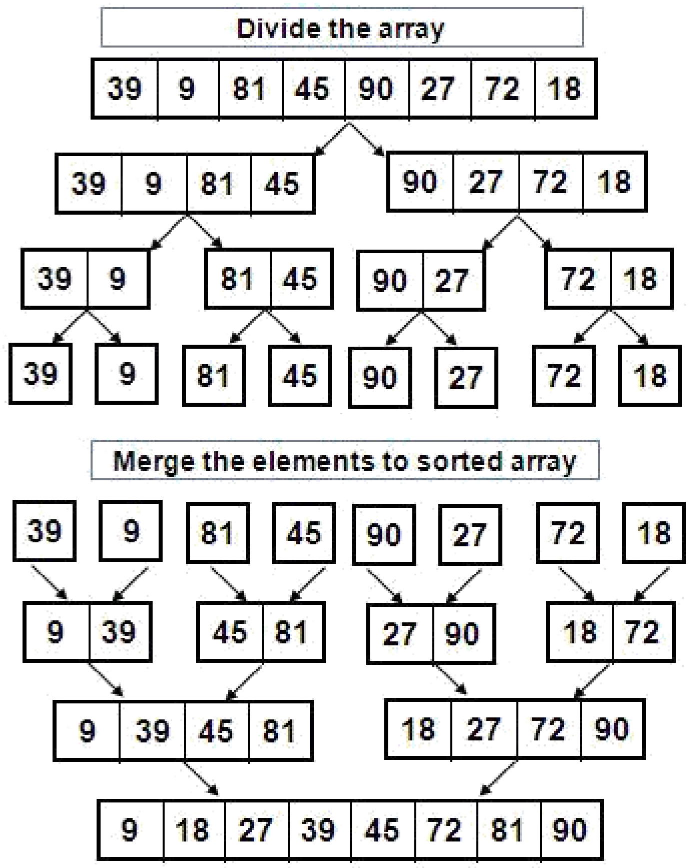
Note that the recursion bottoms out when the sub-array has just one element, so that it is trivially sorted.
To sort the entire sequence A[1 .. n], make the initial call to the procedure MERGE-SORT (A, 1, n).

```
MERGE-SORT (A, p, r)

1. IF p < r // Check for base case
2. THEN q = FLOOR[(p + r)/2] // Divide step
3. MERGE (A, p, q) // Conquer step.
4. MERGE (A, q + 1, r) // Conquer step.
5. MERGE (A, p, q, r) // Conquer step.
```

Example

A list of unsorted elements are: 39 9 81 45 90 27 72 18



Sorted elements are: 9 18 27 39 45 72 81 90

Program for merge sort:

```
#include<stdio.h>
void disp( );
void mergesort(int,int,int);
void msortdiv(int,int);
int a[50],n;
void main( )
{
int i;
clrscr( );
printf("\nEnter the n value:");
scanf("%d",&n);
printf("\nEnter elements for an array:");
for(i=0;i<n;i++)
scanf("%d",&a[i]);
printf("\nBefore Sorting the elements are:");
disp( );
msortdiv(0,n-1);
printf("\nAfter Sorting the elements are:");
disp( );
getch( );
}
```

```
void disp( )
{
int i;
for(i=0;i<n;i++)
printf("%d ",a[i]);
}
void mergesort(int low,int mid,int high)
{
int t[50],i,j,k;
i=low;
j=mid+1;
k=low;
while((i<=mid) && (j<=high))
{
if(a[i]>=a[j])
t[k++]=a[j++];
else
t[k++]=a[i++];
}
while(i<=mid)
t[k++]=a[i++];
while(j<=high)
t[k++]=a[j++];
for(i=low;i<=high;i++)
a[i]=t[i];
}


void msortdiv(int low,int high)
{
int mid;
if(low!=high)
{
mid=((low+high)/2);
msortdiv(low,mid);
msortdiv(mid+1,high);
mergesort(low,mid,high);
}
}
```

Time Complexity of merge sort:
Best case : O (n log n)
Average case : O (n log n)
Worst case : O (n log n)

Lecture 32

Radix Sort

The lower bound for Comparison based sorting algorithm (Merge Sort, Heap Sort, Quick-Sort .. etc) is Ω(nLogn), i.e., they cannot do better than nLogn.
Counting sort is a linear time sorting algorithm that sort in O(n+k) time when elements are in range from 1 to k.

What if the elements are in range from 1 to n^2?

We can't use counting sort because counting sort will take O(n^2) which is worse than comparison based sorting algorithms. Can we sort such an array in linear time?

Radix Sort is the answer. The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit. Radix sort uses counting sort as a subroutine to sort.

Algorithm

1) Do following for each digit i where i varies from least significant digit to the mostsignificantdigit.

2) Sort input array using counting sort (or any stable sort) according to the i"th digit.

Example:
Original, unsorted list:
170, 45, 75, 90, 802, 24, 2, 66

Sorting by least significant digit (1s place) gives: [*Notice that we keep 802 before 2, because 802 occurred before 2 in the original list, and similarly for pairs 170 & 90 and 45 & 75.]

170, 90, 802, 2, 24, 45, 75, 66

Sorting by next digit (10s place) gives: [*Notice that 802 again comes before 2 as 802 comes before 2 in the previous list.]

802, 2, 24, 45, 66, 170, 75, 90

Sorting by most significant digit (100s place) gives:

2, 24, 45, 66, 75, 90, 170, 802

What is the running time of Radix Sort?

Let there be d digits in input integers. Radix Sort takes O(d*(n+b)) time where b is the base for representing numbers, for example, for decimal system, b is 10. What is the value of d? If k is the maximum possible value, then d would be O(logb(k)). So overall time complexity is O((n+b) * logb(k)). Which looks more than the time complexity of comparison based sorting algorithms for a large k. Let us first limit k. Let k <=nc where c is a constant. In that case, the complexity becomes O(nLogb(n)). But it still doesn"t beat comparison based sorting algorithms. What if we make value of b larger? What should be the value of b to make the time complexity linear? If we set b as n, we get the time complexity as O(n). In other words, we can sort an array of integers with range from 1 to nc if the numbers are represented in base n (or every digit takes log2(n) bits).

Is Radix Sort preferable to Comparison based sorting algorithms like Quick-Sort?

If we have $\log_2 n$ bits for every digit, the running time of Radix appears to be better than Quick Sort for a wide range of input numbers. The constant factors hidden in asymptotic notation are higher for Radix Sort and Quick-Sort uses hardware caches more effectively. Also, Radix sort uses counting sort as a subroutine and counting sort takes extra space to sort numbers.

Program for Radix Sort

```
#include<iostream>
using namespace std;

// A utility function to get maximum value in arr[]
int getMax(int arr[], int n)
{
    int mx = arr[0];
    for (int i = 1; i < n;
        i++) if (arr[i] > mx)
            mx = arr[i];
    return mx;
}

// A function to do counting sort of arr[] according to
// the digit represented by exp.
void countSort(int arr[], int n, int exp)
{
    int output[n]; // output array
    int i, count[10] = {0};

    // Store count of occurrences in
    count[] for (i = 0; i < n; i++)
        count[ (arr[i]/exp)%10 ]++;

    // Change count[i] so that count[i] now contains actual
    //  position of this digit in output[]
    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];

    // Build the output array
    for (i = n - 1; i >= 0; i--)
    {
        output[count[ (arr[i]/exp)%10 ] - 1] =
        arr[i]; count[ (arr[i]/exp)%10 ]--;
    }

    // Copy the output array to arr[], so that arr[] now
    // contains sorted numbers according to current
    digit for (i = 0; i < n; i++)
        arr[i] = output[i];
}

// The main function to that sorts arr[] of size n using
// Radix Sort
void radixsort(int arr[], int n)
{
    // Find the maximum number to know number of digits
    int m = getMax(arr, n);

    // Do counting sort for every digit. Note that instead
    // of passing digit number, exp is passed. exp is 10^i
    // where i is current digit number
    for (int exp = 1; m/exp > 0; exp *= 10)
        countSort(arr, n, exp);
}

// A utility function to print an array
void print(int arr[], int n)
{
     for (int i = 0; i < n; i++)
         cout << arr[i] << " ";
}

// Driver program to test above functions
int main()
{
    int arr[] = {170, 45, 75, 90, 802, 24, 2,
    66}; int n = sizeof(arr)/sizeof(arr[0]);
    radixsort(arr, n);
    print(arr,
    n); return 0;
}
```

Lecture 33

Searching

Searching is used to find the location where an element is available. There are two types of search techniques. They are:

1. Linear or sequential search
2. Binary search

Sorting allows an efficient arrangement of elements within a given data structure. It is a way in which the elements are organized systematically for some purpose. For example, a dictionary in which words is arranged in alphabetical order and telephone director in which the subscriber names are listed in alphabetical order. There are many sorting techniques out of which we study the following.

1. Bubble sort
2. Quick sort
3. Selection sort
4. Insertion sort
5. Merge sort
6. Radix sort

There are two types of sorting techniques:

1. Internal sorting
2. External sorting

If all the elements to be sorted are present in the main memory then such sorting is called **internal sorting** on the other hand, if some of the elements to be sorted are kept on the secondary storage, it is called **external sorting**. Here we study only internal sorting techniques.

Linear Search:

This is the simplest of all searching techniques. In this technique, an ordered or unordered list will be searched one by one from the beginning until the desired element is found. If the desired element is found in the list then the search is successful otherwise unsuccessful.

Suppose there are „n**'** elements organized sequentially on a List. The number of comparisons required to retrieve an element from the list, purely depends on where the element is stored in the list. If it is the first element, one comparison will do; if it is second element two comparisons are necessary and so on. On an average you need [(n+1)/2] comparison"s to search an element. If search is not successful, you would need "n**'** comparisons.

The time complexity of linear search is ***O(n)***.

Algorithm:

Let array a[n] stores n elements. Determine whether element „x" is present or not.

```
linsrch(a[n], x)
{
        index = 0;
        flag = 0;
        while (index < n) do
        {
                if (x == a[index])
                {
                        flag = 1;
                        break;
                }
                index ++;
        }
        if(flag == 1)
                printf("Data found at %d position", index);
        else
                printf("data not found");

}
```

Example 1:

Suppose we have the following unsorted list: 45, 39, 8, 54, 77, 38, 24, 16, 4, 7, 9, 20

If we are searching for: 45, we"ll look at 1 element before success
39, we"ll look at 2 elements before success
8, we"ll look at 3 elements before success
54, we"ll look at 4 elements before success
77, we"ll look at 5 elements before success
38 we"ll look at 6 elements before success
24, we"ll look at 7 elements before success
16, we"ll look at 8 elements before success
4, we"ll look at 9 elements before success 7,
we"ll look at 10 elements before success 9,
we"ll look at 11 elements before success 20,
we"ll look at 12 elements before success

For any element not in the list, we"ll look at 12 elements before failure.

Example 2:

Let us illustrate linear search on the following 9 elements:

Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Elements	-15	-6	0	7	9	23	54	82	101

Searching different elements is as follows:

1. Searching for x = 7Search successful, data found at 3rd position.
2. Searching for x = 82Search successful, data found at 7th position.
3. Searching for x = 42Search un-successful, data not found.

A non-recursive program for Linear Search:

```
# include <stdio.h>
# include <conio.h>

main()
{
        int number[25], n, data, i, flag = 0;
        clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter the elements:
        "); for(i = 0; i < n; i++)
                scanf("%d", &number[i]);
        printf("\n Enter the element to be Searched: ");
        scanf("%d", &data);
        for( i = 0; i < n; i++)
        {
                if(number[i] == data)
                {
                        flag = 1;
                        break;
                }
        }
        if(flag == 1)
                printf("\n Data found at location: %d", i+1);
        else
                printf("\n Data not found ");
}
```

A Recursive program for linear search:

```
# include <stdio.h>
# include <conio.h>

void linear_search(int a[], int data, int position, int n)
{
        if(position < n)
        {
                if(a[position] == data)
                        printf("\n Data Found at %d ", position);
                else
                        linear_search(a, data, position + 1, n);
        }
        else
                printf("\n Data not found");
}

void main()
{
        int a[25], i, n, data;
        clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter the elements:");
        for(i = 0; i < n; i++)
        {
                scanf("%d", &a[i]);
        }
        printf("\n Enter the element to be searched:
        "); scanf("%d", &data);
        linear_search(a, data, 0, n);
        getch();
}
```

Lecture 34

BINARY SEARCH

If we have „n" records which have been ordered by keys so that $x_1 < x_2 < \ldots < x_n$. When we are given a element „x", binary search is used to find the corresponding element from the list. In case „x" is present, we have to determine a value „j" such that a[j] = x (successful search). If „x" is not in the list then j is to set to zero (un successful search).

In Binary search we jump into the middle of the file, where we find key a[mid], and compare „x" with a[mid]. If x = a[mid] then the desired record has been found. If x < a[mid] then „x" must be in that portion of the file that precede s a[mid]. Similarly, if a[mid] > x, then further search is only necessary in that part of the file which follows a[mid].

If we use recursive procedure of finding the middle key a[mid] of the un-searched portion of a file, then every un-successful comparison of „x" with a[mid] will eliminate roughly half the un-searched portion from consideration.

Since the array size is roughly halved after each comparison between „x" and a[mid], and since an array of length „n" can be halved only about $\log_2 n$ times before reaching a trivial length, the worst case complexity of Binary search is about $\log_2 n$.

Algorithm:

Let array a[n] of elements in increasing order, n ≥ 0, determine whether „x" is present, and if so, set j such that x = a[j] else return 0.

```
binsrch(a[], n, x)
{
        low = 1; high = n;
        while (low ≤ high) do
        {
                mid = (low + high)/2 ;
                if (x < a[mid])
                        high = mid – 1;
                else if (x > a[mid])
                        low = mid +1;
                else return mid;
        }
        return 0;
 }
```

low and *high* are integer variables such that each time through the loop either „x" is found or *low* is increased by at least one or *high* is decreased by at least one. Thus we have two sequences of integers approaching each other and eventually *low* will become greater than *high* causing termination in a finite number of steps if „x" is not present.

Example 1:

Let us illustrate binary search on the following 12 elements:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
Elements	4	7	8	9	16	20	24	38	39	45	54	77

If we are searching for x = 4: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 1, high = 2, mid = 3/2 = 1, check 4, ***found***

If we are searching for x = 7: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 1, high = 2, mid = 3/2 = 1, check 4
low = 2, high = 2, mid = 4/2 = 2, check 7, ***found***

If we are searching for x = 8: (This needs 2 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8*, **found***

If we are searching for x = 9: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 4, high = 5, mid = 9/2 = 4, check 9, ***found***

If we are searching for x = 16: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 4, high = 5, mid = 9/2 = 4, check 9
low = 5, high = 5, mid = 10/2 = 5, check 16, ***found***

If we are searching for x = 20: (This needs 1 comparison)
low = 1, high = 12, mid = 13/2 = 6, check 20, ***foun***If we
are searching for x = 24: (This needs 3 comparisons) low = 1,
high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 7, high = 8, mid = 15/2 = 7, check 24, ***found***

If we are searching for x = 38: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 7, high = 8, mid = 15/2 = 7, check 24
low = 8, high = 8, mid = 16/2 = 8, check 38, ***found***

If we are searching for x = 39: (This needs 2 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39, ***found***

If we are searching for x = 45: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39 low
= 10, high = 12, mid = 22/2 = 11, check 54
low = 10, high = 10, mid = 20/2 = 10, check 45, ***found***

If we are searching for x = 54: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 10, high = 12, mid = 22/2 = 11, check 54, ***found***

If we are searching for x = 77: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39 low
= 10, high = 12, mid = 22/2 = 11, check 54
low = 12, high = 12, mid = 24/2 = 12, check 77, ***found***

The number of comparisons necessary by search element:

20 – Require 1 comparison;
8 and 39 – requires 2 comparisons;
4, 9, 24, 54 – requires 3 comparisons and
7, 16, 38, 45, 77 – requires 4 comparisons

Summing the comparisons, needed to find all twelve items and dividing by 12, yielding 37/12 or approximately 3.08 comparisons per successful search on the average.

Example 2:

Let us illustrate binary search on the following 9 elements:

Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Elements	-15	-6	0	7	9	23	54	82	101

Solution:

The number of comparisons required for searching different elements is as follows:

1. If we are searching for x = 101: (Number of comparisons = 4)

low	high	mid
1	9	5
6	9	7
8	9	8
9	9	9
		found

2. Searching for x = 82: (Number of comparisons = 3)

low	high	mid
1	9	5
6	9	7
8	9	8
		found

3. Searching for x = 42: (Number of comparisons = 4)

low	high	mid
1	9	5
6	9	7
6	6	6
7	6	not found

4. Searching for x = -14: (Number of comparisons = 3)

low	high	mid
1	9	5
1	4	2
1	1	1
2	1	not found

Continuing in this manner the number of element comparisons needed to find each of nine elements is:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Elements	-15	-6	0	7	9	23	54	82	101
Comparisons	*3*	*2*	*3*	*4*	*1*	*3*	*2*	*3*	*4*

No element requires more than 4 comparisons to be found. Summing the comparisons needed to find all nine items and dividing by 9, yielding 25/9 or approximately 2.77 comparisons per successful search on the average.

There are ten possible ways that an un-successful search may terminate depending upon the value of x.

If x < a(1), a(1) < x < a(2), a(2) < x < a(3), a(5) < x < a(6), a(6) < x < a(7) or a(7) < x < a(8) the algorithm requires 3 element comparisons to determine that „x" is not present. For all of the remaining possibilities BINSRCH requires 4 element comparisons.

Thus the average number of element comparisons for an unsuccessful search is:

(3 + 3 + 3 + 4 + 4 + 3 + 3 + 3 + 4 + 4) / 10 = 34/10 = 3.4

Time Complexity:

The time complexity of binary search in a successful search is O(log n) and for an unsuccessful search is O(log n).

A non-recursive program for binary search:

```
# include <stdio.h>
# include <conio.h>

main()
{
        int number[25], n, data, i, flag = 0, low, high,
        mid; clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter the elements in ascending order: ");
        for(i = 0; i < n; i++)
                scanf("%d", &number[i]);
        printf("\n Enter the element to be searched: ");
        scanf("%d", &data);
        low = 0; high = n-1;
        while(low <= high)
        {
                mid = (low + high)/2;
                if(number[mid] == data)
                {
                        flag = 1;
                        break;
                }
                else
                {
                        if(data < number[mid])
                                high = mid - 1;
                        else
                                low = mid + 1;
                }
        }
        if(flag == 1)
                printf("\n Data found at location: %d", mid + 1);
        else
                printf("\n Data Not Found ");
}
```

A recursive program for binary search:

```
# include <stdio.h>
# include <conio.h>

void bin_search(int a[], int data, int low, int high)
{
        int mid ;
        if( low <= high)
        {
                mid = (low + high)/2;
                if(a[mid] == data)
                        printf("\n Element found at location: %d ", mid + 1);
                else
                {
                        if(data < a[mid])
                                bin_search(a, data, low, mid-1);
                        else

                                bin_search(a, data, mid+1, high);
                }
        }
        else
                printf("\n Element not found");
}
void main()
{
        int a[25], i, n, data;
        clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter the elements in ascending order: ");
        for(i = 0; i < n; i++)
                scanf("%d", &a[i]);
        printf("\n Enter the element to be searched: ");
        scanf("%d", &data);
        bin_search(a, data, 0, n-1);
        getch();
}
```

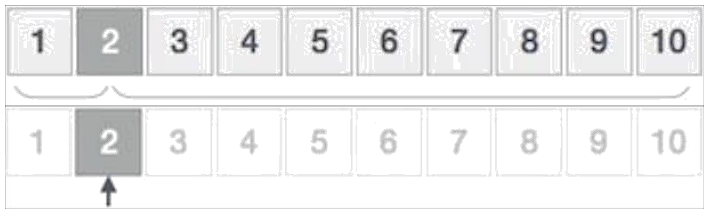
Interpolation search:

Interpolation search is an improved variant of binary search. This search algorithm works on the probing position of the required value. For this algorithm to work properly, the data collection should be in a sorted form and equally distributed.

Binary search has a huge advantage of time complexity over linear search. Linear search has worst-case complexity of O(n) whereas binary search has O(log n).

There are cases where the location of target data may be known in advance. For example, in case of a telephone directory, if we want to search the telephone number of Morphius. Here, linear search and even binary search will seem slow as we can directly jump to memory space where the names start from 'M' are stored**.**

Interpolation search finds a particular item by computing the probe position. Initially, the probe position is the position of the middle most item of the collection.



If a match occurs, then the index of the item is returned. To split the list into two parts, we use the following method −

mid = Lo + ((Hi - Lo) / (A[Hi] - A[Lo])) * (X - A[Lo])

where −
A = list
Lo = Lowest index of the list
Hi = Highest index of the list
A[n] = Value stored at index n in the list

If the middle item is greater than the item, then the probe position is again calculated in the sub-array to the right of the middle item. Otherwise, the item is searched in the subarray to the left of the middle item. This process continues on the sub-array as well until the size of subarray reduces to zero.
Runtime complexity of interpolation search algorithm is **O(log (log n))** as compared to **O(log n)** of BST in favorable situations.

Algorithm

As it is an improvisation of the existing BST algorithm, we are mentioning the steps to search the 'target' data value index, using position probing −

Step 1 − Start searching **data** from middle of the list.
Step 2 − If it is a match, return the index of the item, and exit.
Step 3 − If it is not a match, probe position.
Step 4 − Divide the list using probing formula and find the new middle.
Step 5 − If data is greater than middle, search in higher sub-list.
Step 6 − If data is smaller than middle, search in lower sub-list.
Step 7 − Repeat until match.

program for interpolation search:

```
#include "stdio.h"
 #include "stdlib.h"
 #define MAX 200

int interpolation_search(int a[], int bottom, int top, int item) {

 int mid;
 while (bottom <= top) {
  mid = bottom + (top - bottom) * ((item - a[bottom]) / (a[top] - a[bottom]));
  if (item == a[mid])
   return mid + 1;
  if (item < a[mid])
   top = mid - 1;
  else
   bottom = mid + 1;
 }
 return -1;
}

int main() {
 int arr[MAX];
 int i, num;
 int item, pos;

 printf("\nEnter total elements (num< %d) : ",
 MAX); scanf("%d", &num);

 printf("Enter %d Elements : ", num);
 for (i = 0; i < num; i++)
  scanf("%d", &arr[i]);

 printf("\nELEMENTS ARE\n: ");
 for (i = 0; i < num; i++)
  printf("%d\t", arr[i]);

 printf("\nSearch For : ");
 scanf("%d", &item);
 pos = interpolation_search(&arr[0], 0, num,
 item); if (pos == -1)
  printf("\nElement %d not found\n", item);
 else
  printf("\nElement %d found at position %d\n", item, pos);

 return 0;
}
```

Lecture 35

Hashing

Having an insertion, find and removal of O(log(N)) is good but as the size of the table becomes larger, even this value becomes significant. We would like to be able to use an algorithm for finding of O(1). This is when hashing comes into play!

Hashing using Arrays

When implementing a hash table using arrays, the nodes are not stored consecutively; instead the location of storage is computed using the key and a hash function. The computation of the array index can be visualized as shown below:

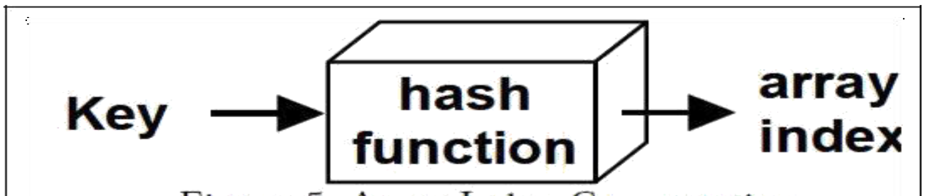


Figure 5. Array Index Computation

The value computed by applying the hash function to the key is often referred to as the hashed key. The entries into the array, are scattered (not necessarily sequential) as can be seen in figure below.

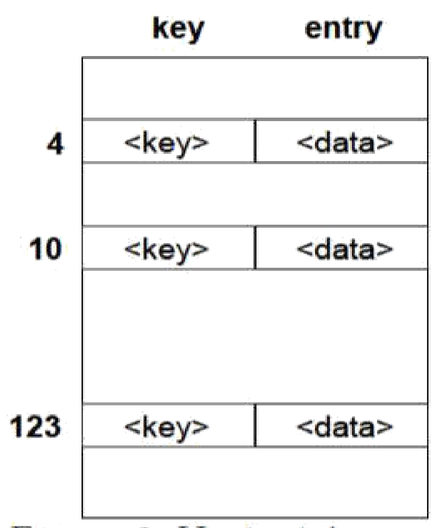


Figure 6. Hashed Array

The cost of the insert, find and delete operations is now only O(1). Can you think of why?
Hash tables are very good if you need to perform a lot of search operations on a relatively stable table (i.e. there are a lot fewer insertion and deletion operations than search operations).

One the other hand, if traversals (covering the entire table), insertions, deletions are a lot more frequent than simple search operations, then ordered binary trees (also called AVL trees) are the preferred implementation choice.

Hashing Performance

There are three factors the influence the performance of hashing:
Hash function

- should distribute the keys and entries evenly throughout the entire table
- should minimize collisions

Collision resolution strategy

- Open Addressing: store the key/entry in a different position
- Separate Chaining: chain several keys/entries in the same position

Table size

- Too large a table, will cause a wastage of memory
- Too small a table will cause increased collisions and eventually force *rehashing* (creating a new hash table of larger size and copying the contents of the current hash table into it)
- The size should be appropriate to the hash function used and should typically be a prime number. Why? (We discussed this in class).

Selecting Hash Functions

The hash function converts the key into the table position. It can be carried out using:

Modular Arithmetic: Compute the index by dividing the key with some value and use the remainder as the index. This forms the basis of the next two techniques. For Example: index:= key MOD table_size

Truncation: Ignoring part of the key and using the rest as the array index. The problem with this approach is that there may not always be an even distribution throughout the table.

For Example: If student id"s are the key 928324312 then select just the last three digits as the index i.e. 312 as the index. => the table size has to be atleast 999. Why?

Folding: Partition the key into several pieces and then combine it in some convenient way.

For Example:

- For an 8 bit integer, compute the index as follows:

Index := (Key/10000 + KeyMOD 10000) MOD Table_Size.

- For character strings, compute the index as follows:

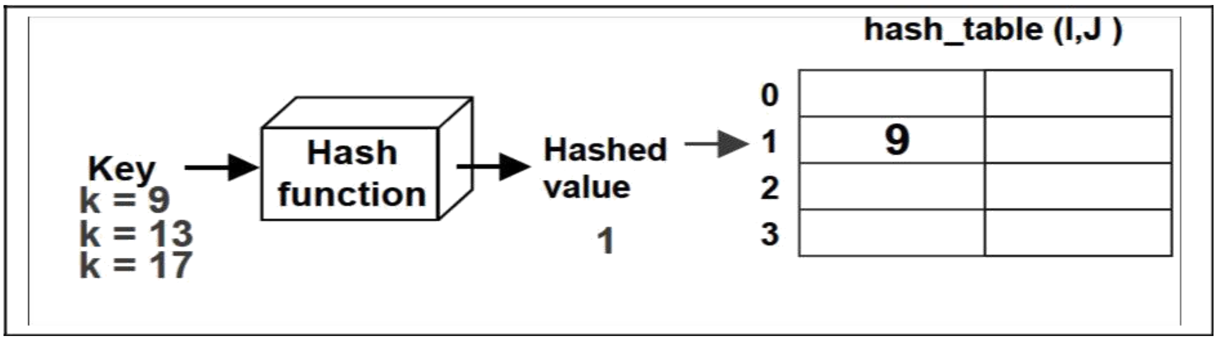
```
Index :=0
For I in 1.. length(string)
Index := Index + ascii_value(String(I))
```

Lecture 36

Collision

Let us consider the case when we have a single array with four records, each with two fields, one for the key and one to hold data (we call this a *single slot bucket*). Let the hashing function be a simple modulus operator i.e. array index is computed by finding the remainder of dividing the key by 4.
Array Index:= key MOD 4

Then key values 9, 13, 17 will all hash to the same index. When two (or more) keys hash to the same value, a **collision** is said to occur.



Collision Resolution

The hash table can be implemented either using

Buckets: An array is used for implementing the hash table. The array has size m*p where m is the number of hash values and p (³ 1) is the number of slots (a slot can hold one entry) as shown in figure below. The *bucket* is said to have p slots.

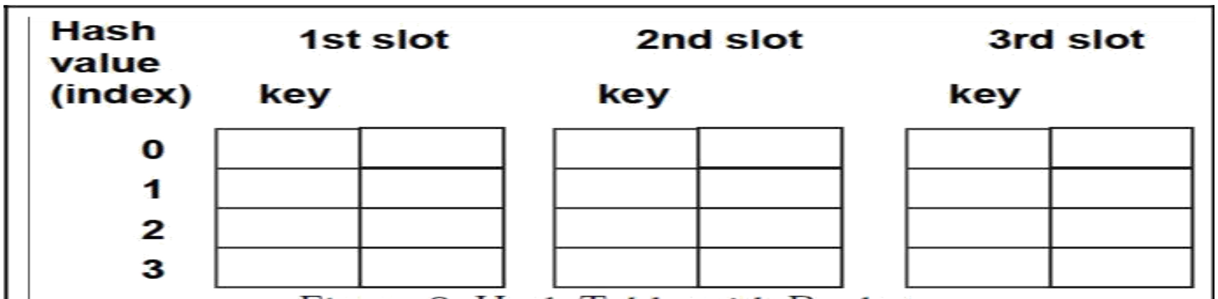
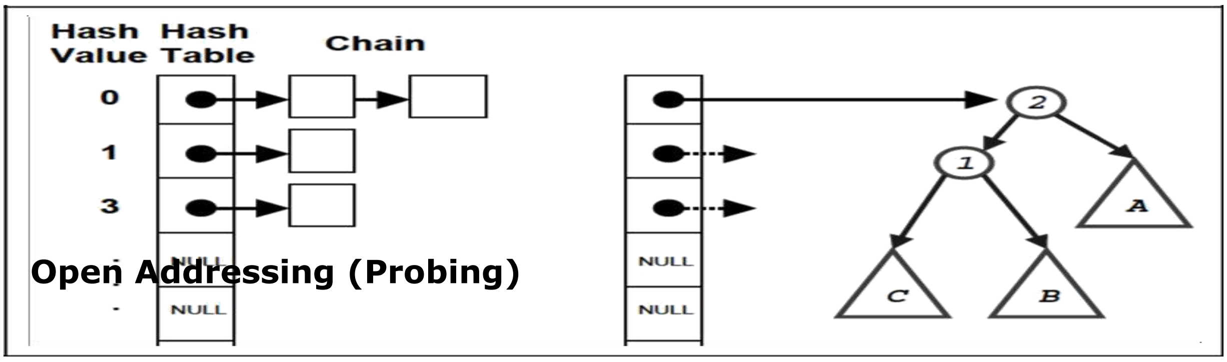


Figure 8. Hash Table with Buckets

Chaining: An array is used to hold the key and a pointer to a liked list (either singly or doubly linked) or a tree. Here the number of nodes is not restricted (unlike with buckets). Each node in the chain is large enough to hold one entry as shown in figure below.



Open Addressing (Probing)

Open addressing / probing is carried out for insertion into fixed size hash tables (hash tables with 1 or more buckets). If the index given by the hash function is occupied, then increment the table position by some number. There are three schemes commonly used for probing:

Linear Probing: The linear probing algorithm is detailed below:

```
Index:= hash(key)
While Table(Index) Is Full do
index := (index + 1) MOD Table_Size
if (index = hash(key))
return table_full
else
Table(Index) := Entry
```

Quadratic Probing: increment the position computed by the hash function in quadratic fashion i.e. increment by 1, 4, 9, 16, ….

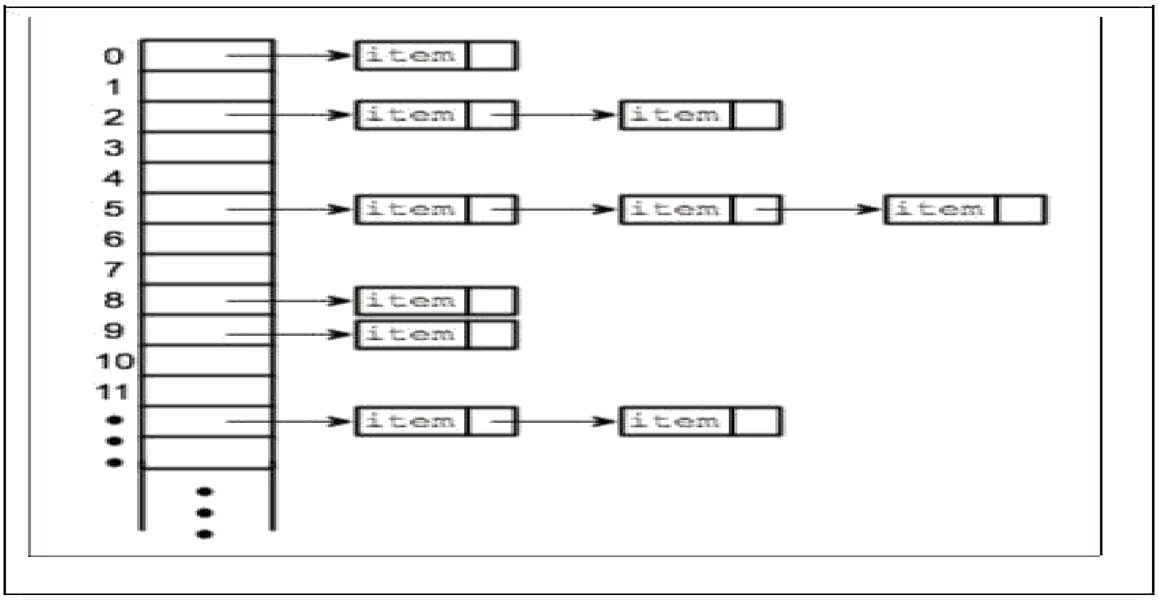
Double Hash: compute the index as a function of two different hash functions.

Chaining

In chaining, the entries are inserted as nodes in a linked list. The hash table itself is an array of head pointers.

The advantages of using chaining are
Insertion can be carried out at the head of the list at the index
The array size is not a limiting factor on the size of the table

The prime disadvantage is the memory overhead incurred if the table size is small.


0
1
2
3
4
5
6
7
8
9
10
11
item