Definition

- Data structure is representation of the logical relationship existing between individual elements of data.
- In other words, a data structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.

Introduction

- Data structure affects the design of both structural & functional aspects of a program.

  Program=algorithm + Data Structure

- You know that a algorithm is a step by step procedure to solve a particular function.

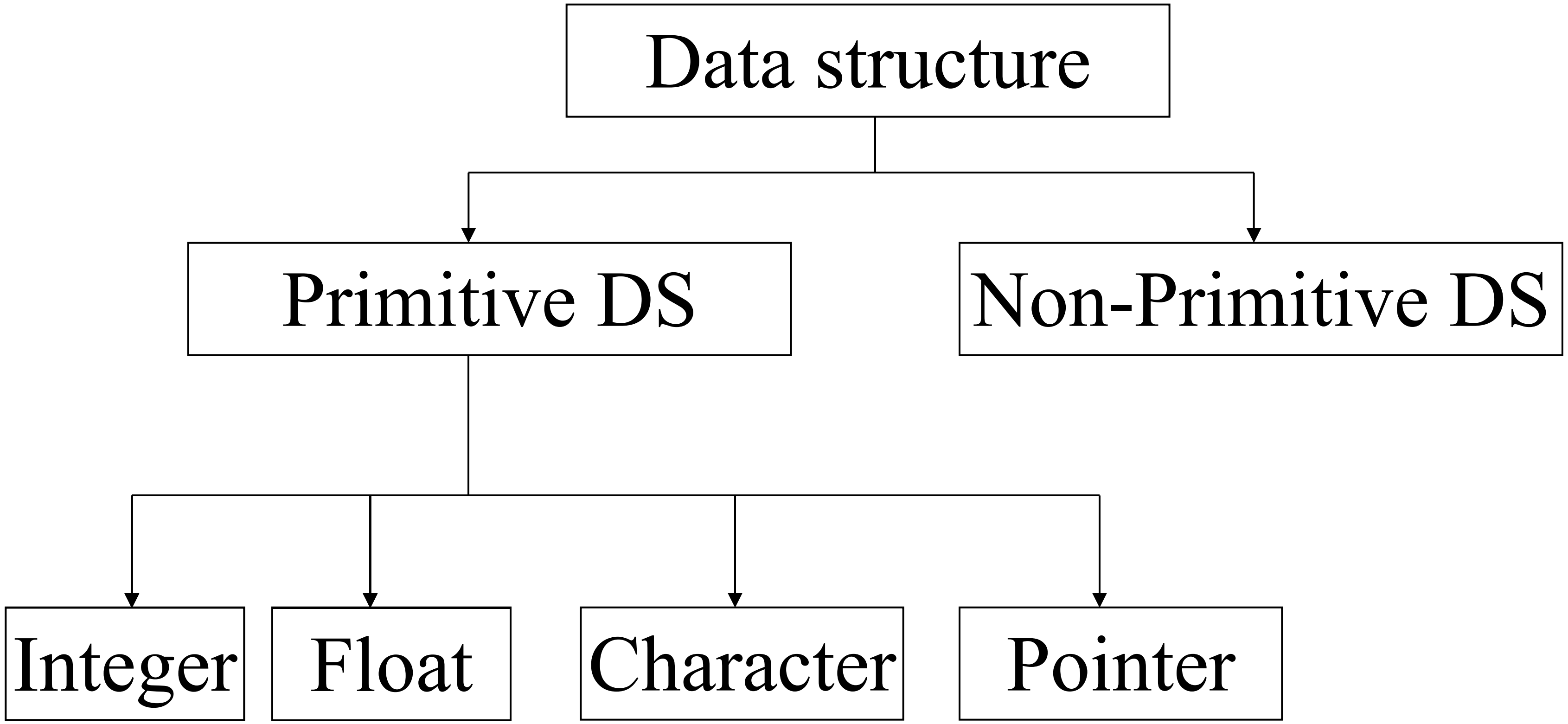
Introduction

- That means, algorithm is a set of instruction written to carry out certain tasks & the data structure is the way of organizing the data with their logical relationship retained.
- To develop a program of an algorithm, we should select an appropriate data structure for that algorithm.
- Therefore algorithm and its associated data structures from a program.

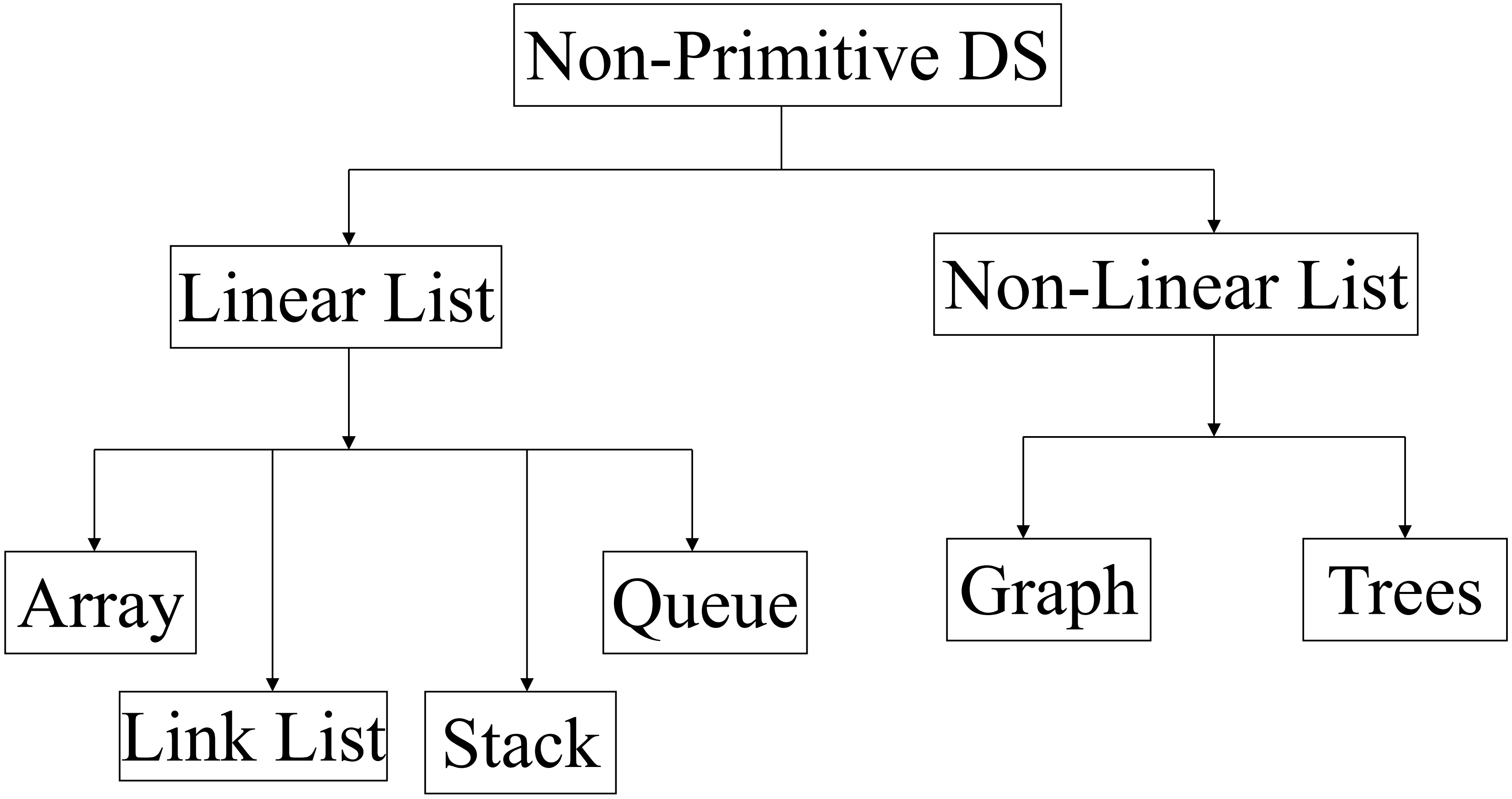
Classification of Data Structure

- Data structure are normally divided into two broad categories:
  - Primitive Data Structure
  - Non-Primitive Data Structure

Classification of Data Structure



Classification of Data Structure



Primitive Data Structure

- There are basic structures and directly operated upon by the machine instructions.
- In general, there are different representation on different computers.
- Integer, Floating-point number, Character constants, string constants, pointers etc, fall in this category.

Non-Primitive Data Structure

- There are more sophisticated data structures.
- These are derived from the primitive data structures.
- The non-primitive data structures emphasize on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items.

Non-Primitive Data Structure

- Lists, Stack, Queue, Tree, Graph are example of non-primitive data structures.
- The design of an efficient data structure must take operations to be performed on the data structure.

Non-Primitive Data Structure

- The most commonly used operation on data structure are broadly categorized into following types:
  - Create
  - Selection
  - Updating
  - Searching
  - Sorting
  - Merging
  - Destroy or Delete

Different between them

- A primitive data structure is generally a basic structure that is usually built into the language, such as an integer, a float.
- A non-primitive data structure is built out of primitive data structures linked together in meaningful ways, such as a or a linked-list, binary search tree, AVL Tree, graph etc.

Description of various Data Structures : Arrays

- An array is defined as a set of finite number of homogeneous elements or same data items.
- It means an array can contain one type of data only, either all integer, all float-point number or all character.

Arrays

- Simply, declaration of array is as follows:

  int arr[10]

- Where int specifies the data type or type of elements arrays stores.
- “arr” is the name of array & the number specified inside the square brackets is the number of elements an array can store, this is also called sized or length of array.

Arrays

- Following are some of the concepts to be remembered about arrays:
  - The individual element of an array can be accessed by specifying name of the array, following by index or subscript inside square brackets.
  - The first element of the array has index zero[0]. It means the first element and last element will be specified as:arr[0] & arr[9]

    Respectively.

Arrays

- The elements of array will always be stored in the consecutive (continues) memory location.
- The number of elements that can be stored in an array, that is the size of array or its length is given by the following equation:

  (Upperbound-lowerbound)+1

Arrays

- For the above array it would be
  (9-0)+1=10,where 0 is the lower bound of array and 9 is the upper bound of array.
- Array can always be read or written through loop. If we read a one-dimensional array it require one loop for reading and other for writing the array.

Arrays

- For example: Reading an array

```
For(i=0;i<=9;i++)
    scanf("%d",&arr[i]);
```

- For example: Writing an array

```
For(i=0;i<=9;i++)
    printf("%d",arr[i]);
```

Arrays

- If we are reading or writing two-dimensional array it would require two loops. And similarly the array of a N dimension would required N loops.
- Some common operation performed on array are:
  - Creation of an array
  - Traversing an array

Arrays

- Insertion of new element
- Deletion of required element
- Modification of an element
- Merging of arrays

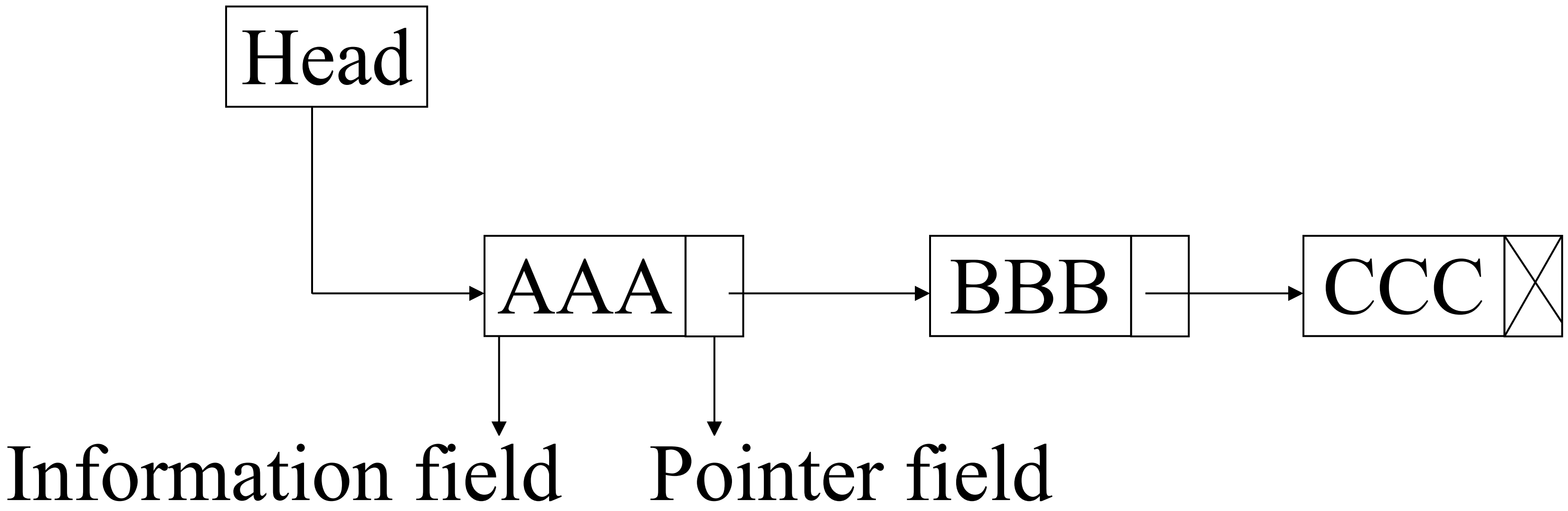
Lists

- A lists (Linear linked list) can be defined as a collection of variable number of data items.
- Lists are the most commonly used non-primitive data structures.
- An element of list must contain at least two fields, one for storing data or information and other for storing address of next element.
- As you know for storing address we have a special data structure of list the address must be pointer type.

Lists

- Technically each such element is referred to as a node, therefore a list can be defined as a collection of nodes as show bellow:

[Linear Liked List]



Lists

- Types of linked lists:
  - Single linked list
  - Doubly linked list
  - Single circular linked list
  - Doubly circular linked list

Stack

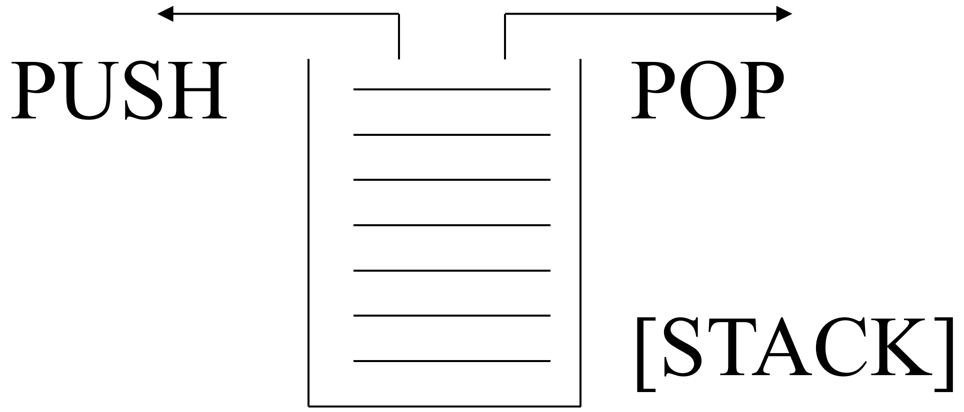
- A stack is also an ordered collection of elements like arrays, but it has a special feature that deletion and insertion of elements can be done only from one end called the top of the stack (TOP)
- Due to this property it is also called as last in first out type of data structure (LIFO).

Stack

- It could be through of just like a stack of plates placed on table in a party, a guest always takes off a fresh plate from the top and the new plates are placed on to the stack at the top.
- It is a non-primitive data structure.
- When an element is inserted into a stack or removed from the stack, its base remains fixed where the top of stack changes.

Stack

- Insertion of element into stack is called PUSH and deletion of element from stack is called POP.
- The bellow show figure how the operations take place on a stack:



Stack

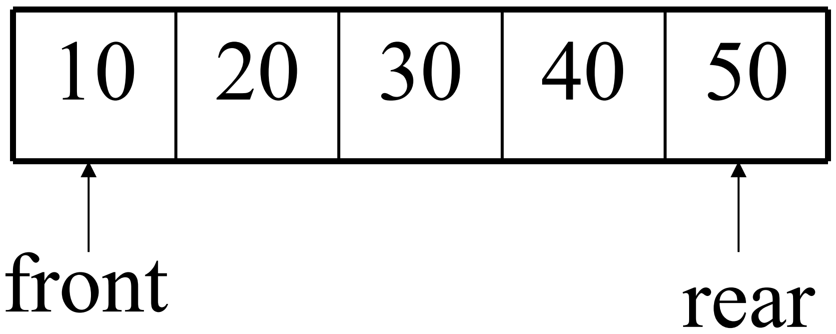
- The stack can be implemented into two ways:
  - Using arrays (Static implementation)
  - Using pointer (Dynamic implementation)

Queue

- Queue are first in first out type of data structure (i.e. FIFO)
- In a queue new elements are added to the queue from one end called REAR end and the element are always removed from other end called the FRONT end.
- The people standing in a railway reservation row are an example of queue.

Queue

- Each new person comes and stands at the end of the row and person getting their reservation confirmed get out of the row from the front end.
- The bellow show figure how the operations take place on a stack:



Queue

- The queue can be implemented into two ways:
  - Using arrays (Static implementation)
  - Using pointer (Dynamic implementation)

Trees

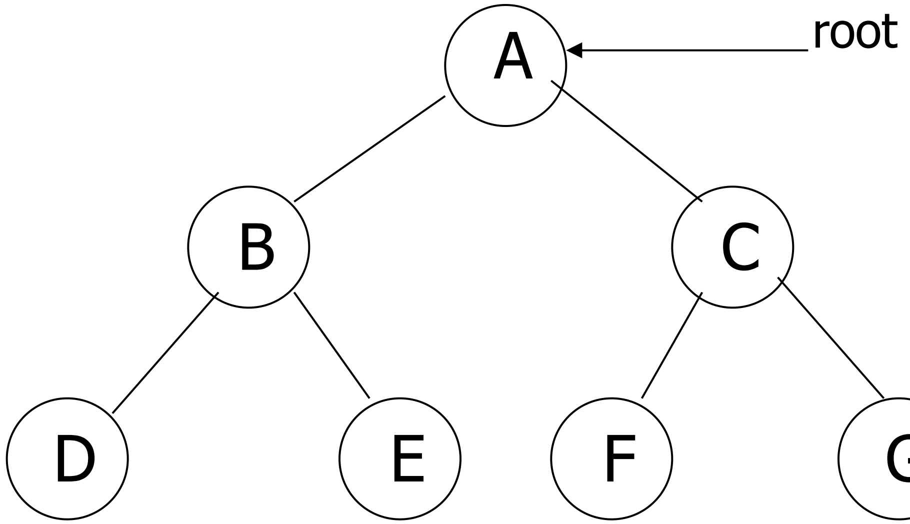
- A tree can be defined as finite set of data items (nodes).
- Tree is non-linear type of data structure in which data items are arranged or stored in a sorted sequence.
- Tree represent the hierarchical relationship between various elements.

Trees

- In trees:
- There is a special data item at the top of hierarchy called the Root of the tree.
- The remaining data items are partitioned into number of mutually exclusive subset, each of which is itself, a tree which is called the sub tree.
- The tree always grows in length towards bottom in data structures, unlike natural trees which grows upwards.

Trees

- The tree structure organizes the data into branches, which related the information.



Graph

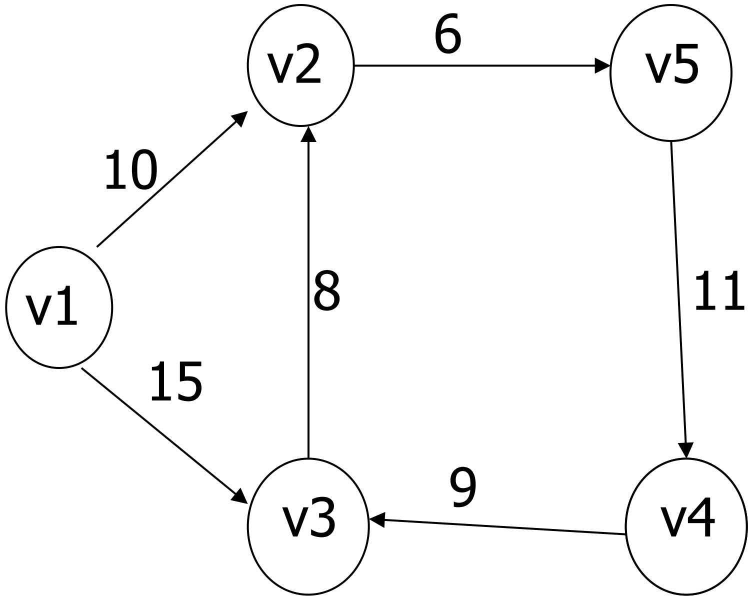
- Graph is a mathematical non-linear data structure capable of representing many kind of physical structures.
- It has found application in Geography, Chemistry and Engineering sciences.
- Definition: A graph G(V,E) is a set of vertices V and a set of edges E.

Graph

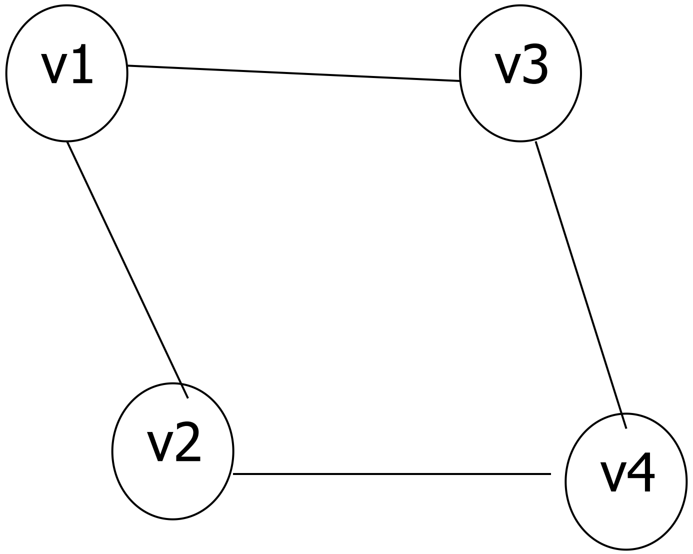
- An edge connects a pair of vertices and many have weight such as length, cost and another measuring instrument for according the graph.
- Vertices on the graph are shown as point or circles and edges are drawn as arcs or line segment.

Graph

- Example of graph:



[a] Directed & Weighted Graph



[b] Undirected Graph

Graph

- Types of Graphs:
  - Directed graph
  - Undirected graph
  - Simple graph
  - Weighted graph
  - Connected graph
  - Non-connected graph

Abstract Data Type and Data Structure

Array:

- Definition:-
  - *Abstract Data Types (ADTs)* stores data and allow various operations on the data to access and change it.
  - A mathematical model, together with various operations defined on the model
  - An ADT is a collection of data and associated operations for manipulating that data

- Data Structures
  - Physical implementation of an ADT
  - data structures used in implementations are provided in a language *(primitive* or *built-in)* or are built from the language constructs *(user-defined)*
  - Each operation associated with the ADT is implemented by one or more subroutines in the implementation

Abstract Data Type

- ADTs support *abstraction*, *encapsulation*, and *information hiding*.
- *Abstraction* is the structuring of a problem into well-defined entities by defining their data and operations.
- The principle of hiding the used data structure and to only provide a well-defined interface is known as *encapsulation.*

The Core Operations of ADT

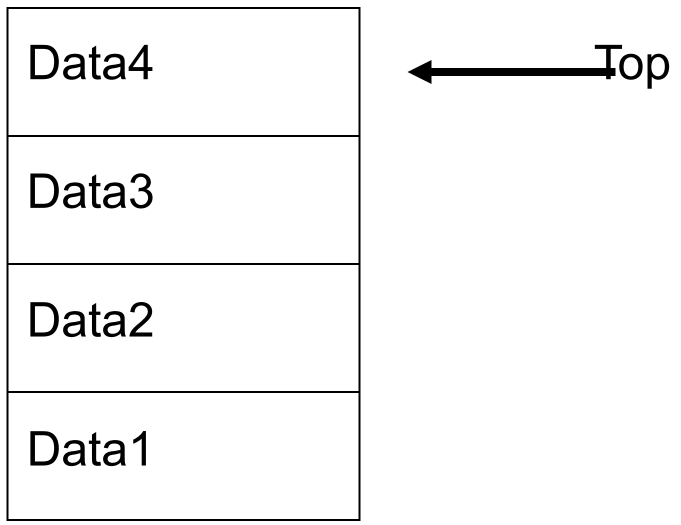
- Every Collection ADT should provide a way to:
  - add an item
  - remove an item
  - find, retrieve, or access an item

- Many, many more possibilities
  - is the collection empty
  - make the collection empty
  - give me a sub set of the collection

- No single data structure works well for all purposes, and so it is important to know the strengths and limitations of several of them

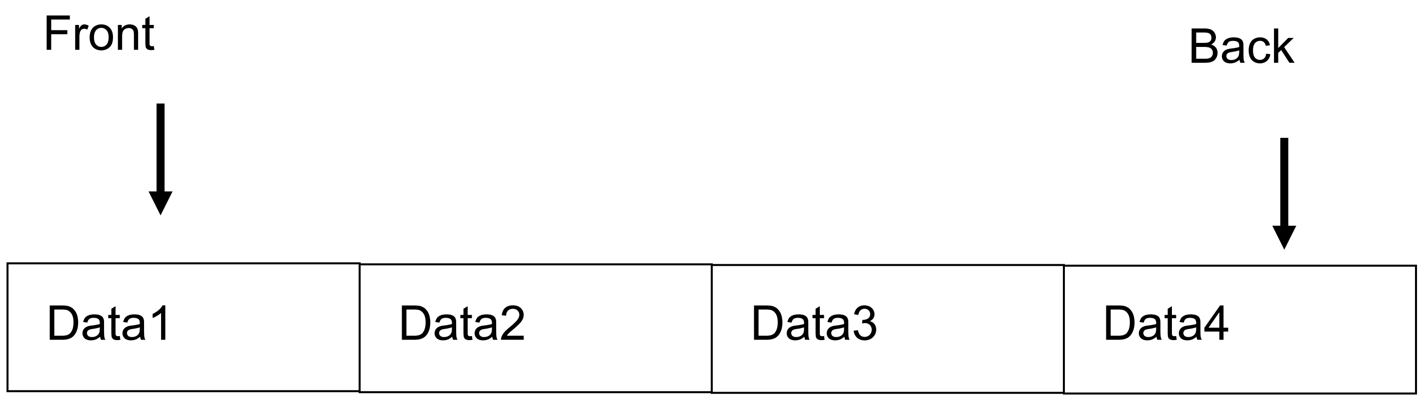
Stacks

- Collection with access only to the last element inserted
- Last in first out
- insert/push
- remove/pop
- top
- make empty



Queues

- Collection with access only to the item that has been present the longest
- Last in last out or first in first out
- enqueue, dequeue, front
- priority queues and dequeue



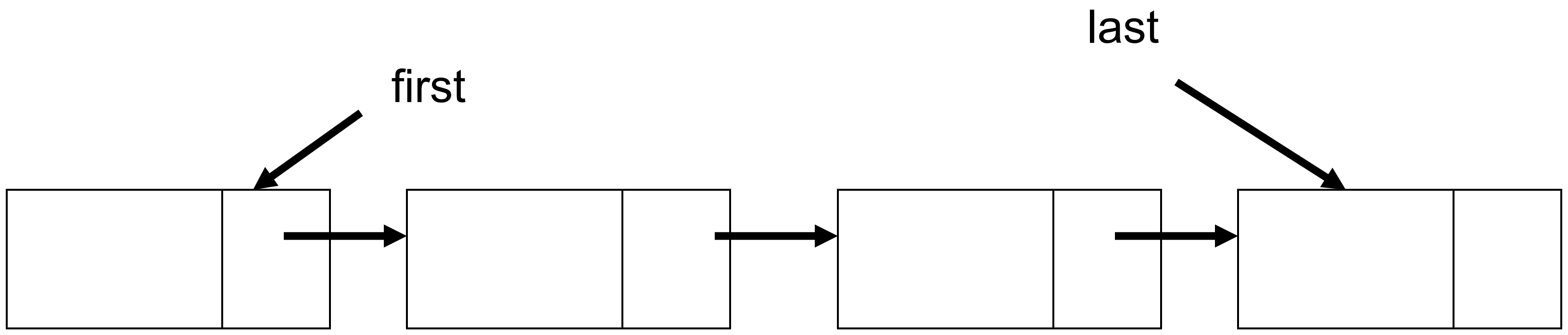
List

- A ***Flexible*** structure, because can grow and shrink on demand.

Elements can be:

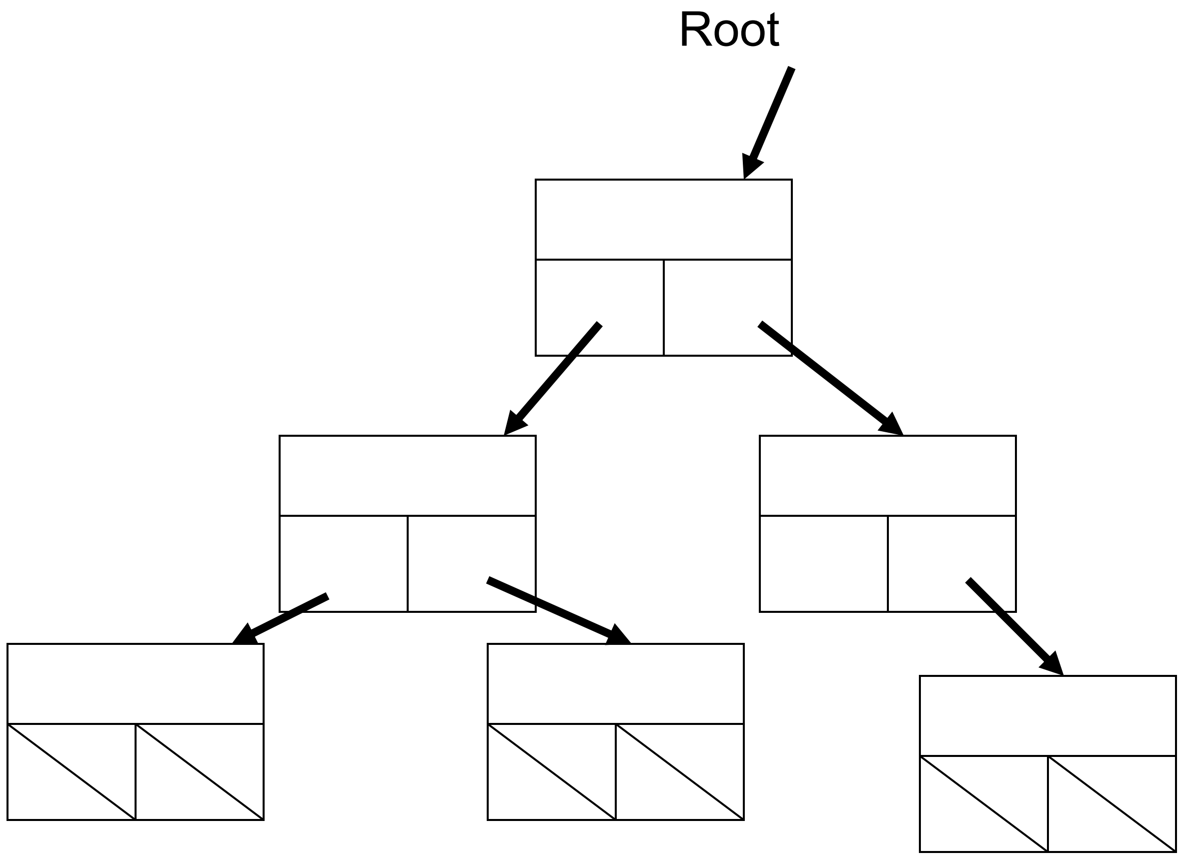
- Inserted
- Accessed
- Deleted

At ***any*** position



Tree

- A ***Tree*** is a collection of elements called ***nodes***.
- One of the node is distinguished as a ***root***, along with a relation (“parenthood”) that places a hierarchical structure on the nodes.



Linked List:

```
DispList(struct node *S)
{ while(S!=NULL)
  { pf("%d->", S->data);
    S=S->next;
  }pf
}
```

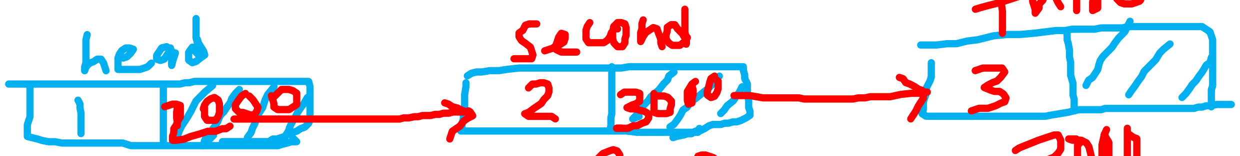
1→2→3→~~NULL~~

RAM
20
int
block

```
struct node
{ int data;
  struct node *next;
};

struct node *S = &a;
S->next         S++
```

```
void main()
{
  struct node *head=NULL;
  struct node *second=NULL;
  struct node *third =NULL;
head=(struct node *)malloc(sizeof(
                    struct node));
second=
third=
```




```
head->data=1;
head->next=second;
second->data=2;
second->next=third;
Third->data=3;
Third->next=NULL;
DispList(head); }
```

Find node with x

```
FindData(struct node *s, int x)
{
  if(s==NULL)  return 0;
  while(s->data != x && s->next != NULL)
  { s=s->next; }
  if (s->data == x)  return s;
  else { pf("Not Found");
  return 0; }
}
```

10	20	30	40
1000	2000	3000	4000

s = 1000 2000 3000 4000

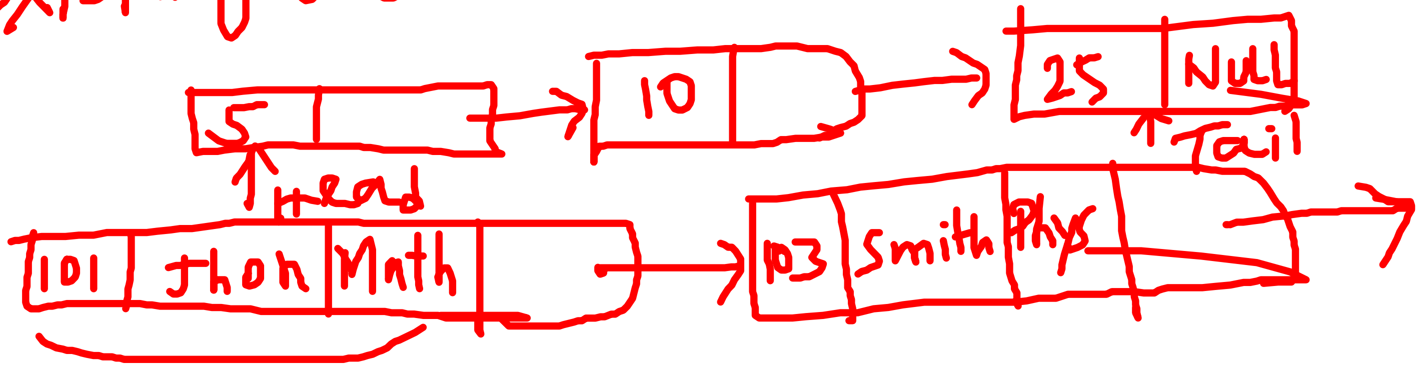
P = 1000 1000 2000 3000

Second last node

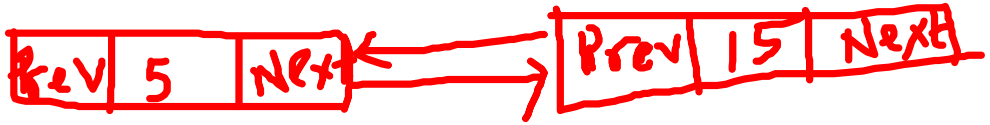
```
struct node *Secondlast
    (struct node *s)
{
  if(s==NULL) return 0;
  struct node *p;
     p=s;
  while(s->next!=NULL)
  {    p=s;
       s=s->next;
  }
  return *p;
}
```

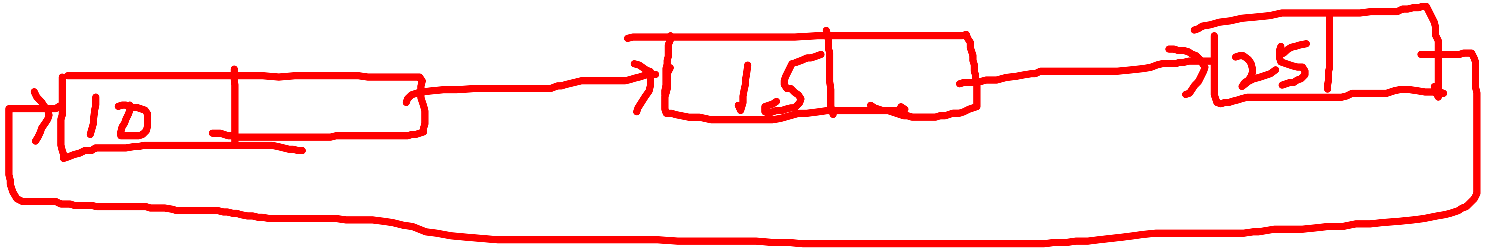
Operation

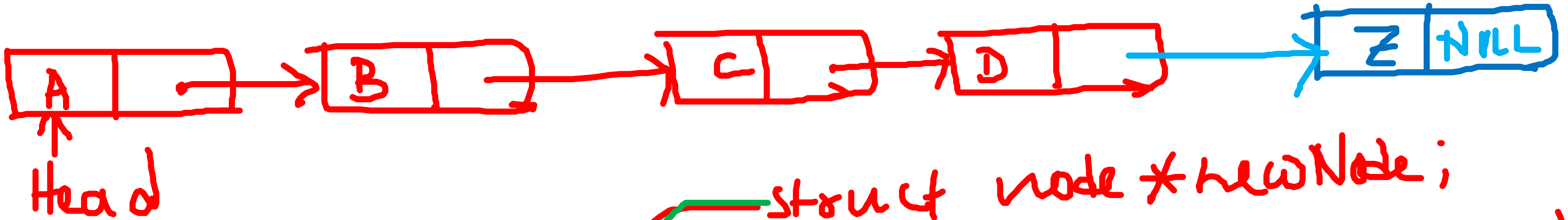
1. inserting new data element → front, middle (before, after), end
2. Deleting the existing data →
3. print
4. Traverse
5. Sorting
6. serching



1. Singly (SLL)
2. Doubly (DLL)
3. Circular (CLL)



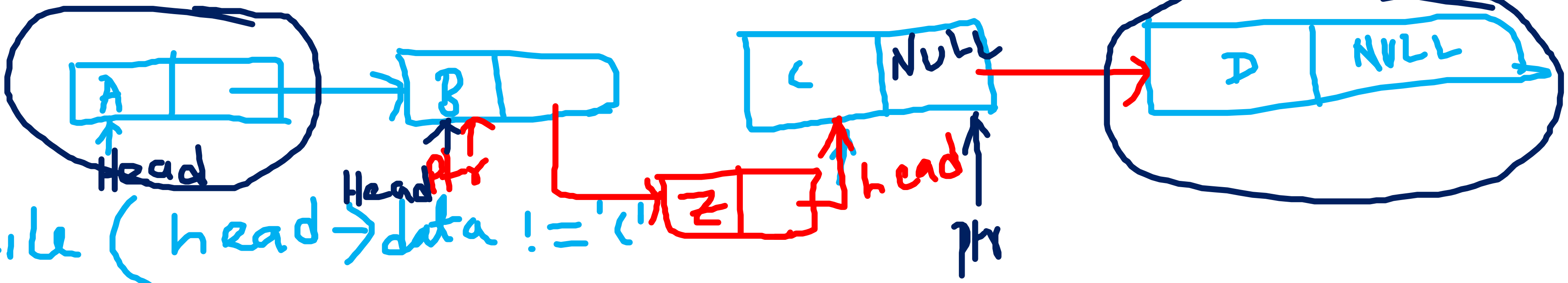





```
void insertAtEnd(char a[])
{ struct node *
  temp = head;
  while(temp->next!=NULL)
  {
    temp = temp->next;
  }
  struct node *newNode;
  strcpy(newNode->data, a);
  temp->next = newNode;
  newNode->next = NULL;
```

```
struct node *newNode;
void insertAtFront(char a[])
{
  strcpy(newNode->data, a)
  newNode->next = head;
  head = newNode;
}
```

'Z'




```
while ( head->data != 'c'
        && head->Next != NULL)
{
    head = head->next;
}
ptr = head;
head->next = newNode;
NewNode->next = ptr->next;
```

```
while ( ~~head->data != 'c'~~ &&
        head->next != NULL)
{ ptr = head;
  head = head->next;
}
ptr->next = newNode;
newNode->next = head;
```

```
{ Ptr = head;
  head = ptr->next;
  free(Ptr);
```

```
ptr->next = NULL;
free (head)
```

Data structure: Storing & Organizing

Linked List

- Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.
- The last node of the list contains pointer to the null.
- Uses of Linked List
  - The list is not required to be contiguously present in the memory. The node can reside any where in the memory and linked together to make a list. This achieves optimized utilization of space.
  - list size is limited to the memory size and doesn't need to be declared in advance.
  - Empty node can not be present in the linked list.
  - We can store values of primitive types or objects in the singly linked list.

Why use linked list over array?

- Till now, we were using array data structure to organize the group of elements that are to be stored individually in the memory. However, Array has several advantages and disadvantages which must be known in order to decide the data structure which will be used throughout the program.
- Array contains following limitations:
- The size of array must be known in advance before using it in the program.
- Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
- All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all its predecessors.
- Linked list is the data structure which can overcome all the limitations of an array. Using linked list is useful because,
  - It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
  - Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

Singly linked list or One way chain

- Singly linked list can be defined as the collection of ordered set of elements. The number of elements may vary according to need of the program. A node in the singly linked list consist of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.
- One way chain or singly linked list can be traversed only in one direction. In other words, we can say that each node contains only next pointer, therefore we can not traverse the list in the reverse direction.
- Consider an example where the marks obtained by the student in three subjects are stored in a linked list as shown in the figure.

- In the above figure, the arrow represents the links. The data part of every node contains the marks obtained by the student in the different subject. The last node in the list is identified by the null pointer which is present in the address part of the last node. We can have as many elements we require, in the data part of the list.

Complexity

- **Data Structure** **Time Complexity**

  - » **Average** **Worst**

- Access Search Insertion Deletion Access Search Insertion Deletion
- Singly Linked List
- θ(n) θ(n) θ(1) θ(1) O(n) O(n) O(1) O(1)
- **Space Compleity Worst**
- O(n)

Operation

- The insertion into a singly linked list can be performed at different positions.
- Insertion at beginning
- It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list.
- Insertion at end of the list
- It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.
- Insertion after specified node
- It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted.
  - struct node
  - {
  - **int** data;
  - struct node *next;
  - };
  - struct node *head, *ptr;
  - ptr = (struct node *)malloc(sizeof(struct node *));

Deletion and Traversing

- The Deletion of a node from a singly linked list can be performed at different positions.
- Deletion at beginning
- It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers.
- Deletion at the end of the list
- It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.
- Deletion after specified node
- It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.
- Traversing
- In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.
- Searching
- In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned.

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
};
struct node *head;

void beginsert ();
void lastinsert ();
void randominsert();
void begin_delete();
void last_delete();
void random_delete();
void display();
void search();
void main ()
{
   int choice =0;
   while(choice != 9)
   {
      printf("\n\n*********Main Menu*********\n");
      printf("\nChoose one option from the following list ...\n
");
      printf("\n===============================================
========\n");
      printf("\n1.Insert in begining\n2.Insert at last\n3.Insert a
t any random location\n4.Delete from Beginning\n
      5.Delete from last\n6.Delete node after specified locatio
n\n7.Search for an element\n8.Show\n9.Exit\n");
      printf("\nEnter your choice?\n");
      scanf("\n%d",&choice);
```

```
     switch(choice)
     {
       case 1:
       beginsert();
       break;
       case 2:
       lastinsert();
       break;
       case 3:
       randominsert();
       break;
       case 4:
       begin_delete();
       break;
       case 5:
       last_delete();
       break;
       case 6:
       random_delete();
       break;
       case 7:
       search();
       break;
       case 8:
       display();
       break;
       case 9:
       exit(0);
       break;
       default:
       printf("Please enter valid choice..");
     }
   }
}
```

```c
void beginsert()
{
   struct node *ptr;
   int item;
   ptr = (struct node *) malloc(sizeof(struct node *));
   if(ptr == NULL)
   {
      printf("\nOVERFLOW");
   }
   else
   {
      printf("\nEnter value\n");
      scanf("%d",&item);
      ptr->data = item;
      ptr->next = head;
      head = ptr;
      printf("\nNode inserted");
   }

}
```

```c
void lastinsert()
{
   struct node *ptr,*temp;
   int item;
   ptr = (struct node*)malloc(sizeof(struct node));
   if(ptr == NULL)
   {
      printf("\nOVERFLOW");
   }
   else
   {
      printf("\nEnter value?\n");
      scanf("%d",&item);
      ptr->data = item;
      if(head == NULL)
      {
         ptr -> next = NULL;
         head = ptr;
         printf("\nNode inserted");
      }
      else
      {
         temp = head;
         while (temp -> next != NULL)
         {
            temp = temp -> next;
         }
         temp->next = ptr;
         ptr->next = NULL;
         printf("\nNode inserted");

      }
   }
}
```

```
void randominsert()
{
   int i,loc,item;
   struct node *ptr, *temp;
   ptr = (struct node *) malloc (sizeof(struct node));
   if(ptr == NULL)
   {
      printf("\nOVERFLOW");
   }
   else
   {
      printf("\nEnter element value");
      scanf("%d",&item);
      ptr->data = item;
      printf("\nEnter the location after which you want to inser
t ");
      scanf("\n%d",&loc);
      temp=head;
      for(i=0;i<loc;i++)
      {
         temp = temp->next;
         if(temp == NULL)
         {
            printf("\ncan't insert\n");
            return;
         }

      }
      ptr ->next = temp ->next;
      temp ->next = ptr;
      printf("\nNode inserted");
   }
}
```

```
void begin_delete()
{
   struct node *ptr;
   if(head == NULL)
   {
      printf("\nList is empty\n");
   }
   else
   {
      ptr = head;
      head = ptr->next;
      free(ptr);
      printf("\nNode deleted from the begining ...\n");
   }
}
void last_delete()
{
   struct node *ptr,*ptr1;
   if(head == NULL)
   {
      printf("\nlist is empty");
   }
   else if(head -> next == NULL)
   {
      head = NULL;
      free(head);
      printf("\nOnly node of the list deleted ...\n");
   }
   else
   {
      ptr = head;
      while(ptr->next != NULL)
      {
         ptr1 = ptr;
         ptr = ptr ->next;
      }
      ptr1->next = NULL;
      free(ptr);
      printf("\nDeleted Node from the last ...\n");
   }
}
```

```c
void random_delete()
{
   struct node *ptr,*ptr1;
   int loc,i;
   printf("\n Enter the location of the node af
ter which you want to perform deletion \n
");
   scanf("%d",&loc);
   ptr=head;
   for(i=0;i<loc;i++)
   {
      ptr1 = ptr;
      ptr = ptr->next;

      if(ptr == NULL)
      {
         printf("\nCan't delete");
         return;
      }
   }
   ptr1 ->next = ptr ->next;
   free(ptr);
   printf("\nDeleted node %d ",loc+1);
}
```

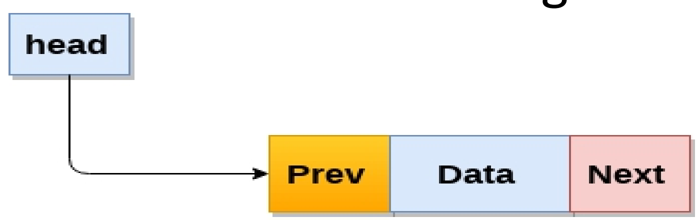
```c
void search()
{
   struct node *ptr;
   int item,i=0,flag;
   ptr = head;
   if(ptr == NULL)
   {
      printf("\nEmpty List\n");
   }
   else
   {
      printf("\nEnter item which you want to search?\n");
      scanf("%d",&item);
      while (ptr!=NULL)
      {
         if(ptr->data == item)
         {
            printf("item found at location %d ",i+1);
            flag=0;
         }
         else
         {
            flag=1;
         }
         i++;
         ptr = ptr -> next;
      }
      if(flag==1)
      {
         printf("Item not found\n");
      }
   }

}
```

```c
void display()
{
    struct node *ptr;
    ptr = head;
    if(ptr == NULL)
    {
        printf("Nothing to print");
    }
    else
    {
        printf("\nprinting values . . . . .\n");
        while (ptr!=NULL)
        {
            printf("\n%d",ptr->data);
            ptr = ptr -> next;
        }
    }
}
```

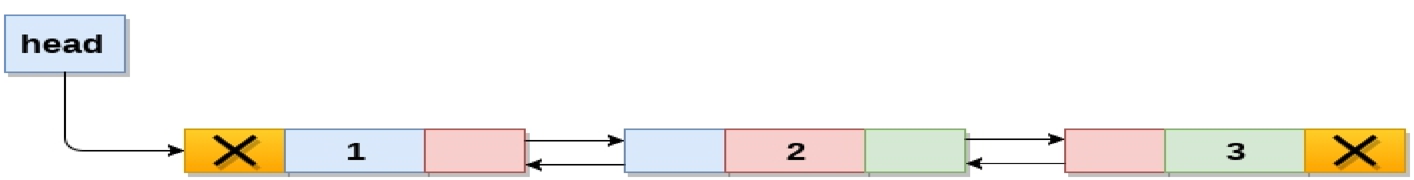
Doubly linked list

- Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.



Node

- A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.

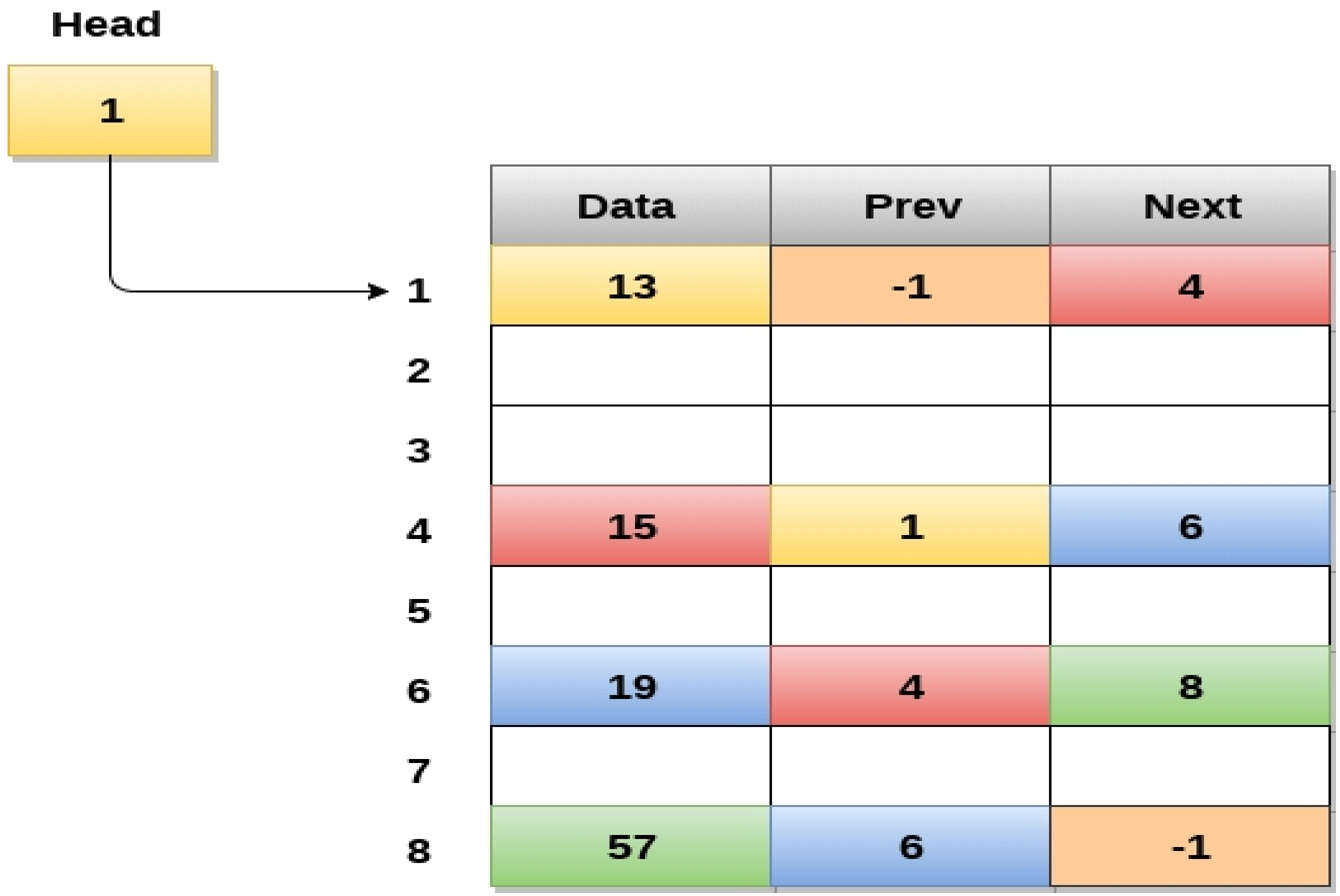


Doubly Linked List

- In C, structure of a node in doubly linked list can be given as :
- struct node
- {
-    struct node *prev;
-    **int** data;
-    struct node *next;
- }
- The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.
- In a singly linked list, we could traverse only in one direction, because each node contains address of the next node and it doesn't have any record of its previous nodes. However, doubly linked list overcome this limitation of singly linked list. Due to the fact that, each node of the list contains the address of its previous node, we can find all the details about the previous node as well by using the previous address stored inside the previous part of each node.

Memory Representation of a doubly linked list

- Memory Representation of a doubly linked list is shown in the following image. Generally, doubly linked list consumes more space for every node and therefore, causes more expansive basic operations such as insertion and deletion. However, we can easily manipulate the elements of the list since the list maintains pointers in both the directions (forward and backward).
- In the following image, the first element of the list that is i.e. 13 stored at address 1. The head pointer points to the starting address 1. Since this is the first element being added to the list therefore the **prev** of the list **contains** null. The next node of the list resides at address 4 therefore the first node contains 4 in its next pointer.
- We can traverse the list in this way until we find any node containing null or -1 in its next part.



Memory Representation of a Doubly linked list

Operations on doubly linked list

- **Node Creation**
- struct node
- {
-     struct node *prev;
-     **int** data;
-     struct node *next;
- };
- struct node *head;

- **Operation**
- Insertion at beginning
- Adding the node into the linked list at beginning.
- Insertion at end
- Adding the node into the linked list to the end.
- Insertion after specified node
- Adding the node into the linked list after the specified node.
- Deletion at beginning
- Removing the node from beginning of the list
- Deletion at the end
- Removing the node from end of the list.
- Deletion of the node having given data
- Removing the node which is present just after the node containing the given data.
- Searching
- Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
- Traversing
- Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

```c
#include<stdio.h>
#include<stdlib.h>
struct node
{
   struct node *prev;
   struct node *next;
   int data;
};
struct node *head;
void insertion_beginning();
void insertion_last();
void insertion_specified();
void deletion_beginning();
void deletion_last();
void deletion_specified();
void display();
void search();
void main ()
{
int choice =0;
   while(choice != 9)
   {
      printf("\n*********Main Menu*********\n");
      printf("\nChoose one option from the following list ...\n");
      printf("\n===============================================
======\n");
      printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at a
ny random location\n4.Delete from Beginning\n
      5.Delete from last\n6.Delete the node after the given data\
n7.Search\n8.Show\n9.Exit\n");
      printf("\nEnter your choice?\n");
      scanf("\n%d",&choice);
    switch(choice)
   {
     case 1:
     insertion_beginning();
     break;
     case 2:
           insertion_last();
     break;
     case 3:
     insertion_specified();
     break;
     case 4:
     deletion_beginning();
     break;
     case 5:
     deletion_last();
     break;
     case 6:
     deletion_specified();
     break;
     case 7:
     search();
     break;
     case 8:
     display();
     break;
     case 9:
     exit(0);
     break;
     default:
     printf("Please enter valid choice..");
   }
  }
}
```

```c
void insertion_beginning()
{
  struct node *ptr;
  int item;
  ptr = (struct node *)malloc(sizeof(struct node));
  if(ptr == NULL)
  {
     printf("\nOVERFLOW");
  }
  else
  {
   printf("\nEnter Item value");
   scanf("%d",&item);

  if(head==NULL)
  {
     ptr->next = NULL;
     ptr->prev=NULL;
     ptr->data=item;
     head=ptr;
  }
  else
  {
     ptr->data=item;
     ptr->prev=NULL;
     ptr->next = head;
     head->prev=ptr;
     head=ptr;
  }
  printf("\nNode inserted\n");
}

}
```

```c
void insertion_last()
{
  struct node *ptr,*temp;
  int item;
  ptr = (struct node *) malloc(sizeof(struct node));
  if(ptr == NULL)
  {
     printf("\nOVERFLOW");
  }
  else
  {
     printf("\nEnter value");
     scanf("%d",&item);
      ptr->data=item;
     if(head == NULL)
     {
        ptr->next = NULL;
        ptr->prev = NULL;
        head = ptr;
     }
     else
     {
       temp = head;
       while(temp->next!=NULL)
       {
          temp = temp->next;
       }
       temp->next = ptr;
       ptr ->prev=temp;
       ptr->next = NULL;
       }
            }
   printf("\nnode inserted\n");
  }
```

```c
void insertion_specified()
{
  struct node *ptr,*temp;
  int item,loc,i;
  ptr = (struct node *)malloc(sizeof(struct node));
  if(ptr == NULL)
  {
     printf("\n OVERFLOW");
  }
  else
  {
     temp=head;
     printf("Enter the location");
     scanf("%d",&loc);
     for(i=0;i<loc;i++)
     {
        temp = temp->next;
        if(temp == NULL)
        {
           printf("\n There are less than %d elements", loc);
           return;
        }
     }
     printf("Enter value");
     scanf("%d",&item);
     ptr->data = item;
     ptr->next = temp->next;
     ptr -> prev = temp;
     temp->next = ptr;
     temp->next->prev=ptr;
     printf("\nnode inserted\n");
  }
}
```

```c
void deletion_beginning()
{
   struct node *ptr;
   if(head == NULL)
   {
      printf("\n UNDERFLOW");
   }
   else if(head->next == NULL)
   {
      head = NULL;
      free(head);
      printf("\nnode deleted\n");
   }
   else
   {
      ptr = head;
      head = head -> next;
      head -> prev = NULL;
      free(ptr);
      printf("\nnode deleted\n");
   }

}
```

```
void deletion_last()
{
   struct node *ptr;
   if(head == NULL)
   {
      printf("\n UNDERFLOW");
   }
   else if(head->next == NULL)
   {
      head = NULL;
      free(head);
      printf("\nnode deleted\n");
   }
   else
   {
      ptr = head;
      if(ptr->next != NULL)
      {
         ptr = ptr -> next;
      }
      ptr -> prev -> next = NULL;
      free(ptr);
      printf("\nnode deleted\n");
   }
}
```

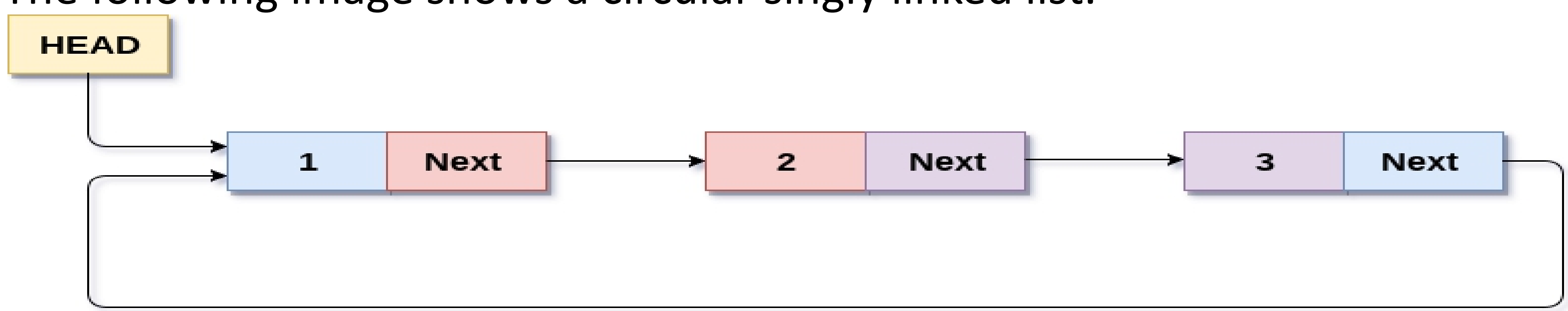
```
void deletion_specified()
{
   struct node *ptr, *temp;
   int val;
   printf("\n Enter the data after which the node is to be del
eted : ");
   scanf("%d", &val);
   ptr = head;
   while(ptr -> data != val)
   ptr = ptr -> next;
   if(ptr -> next == NULL)
   {
      printf("\nCan't delete\n");
   }
   else if(ptr -> next -> next == NULL)
   {
      ptr ->next = NULL;
   }
   else
   {
      temp = ptr -> next;
      ptr -> next = temp -> next;
      temp -> next -> prev = ptr;
      free(temp);
      printf("\nnode deleted\n");
   }
}
```

```c
void display()
{
    struct node *ptr;
    printf("\n printing values...\n");
    ptr = head;
    while(ptr != NULL)
    {
        printf("%d\n",ptr->data);
        ptr=ptr->next;
    }
}
```

```c
void search()
{
    struct node *ptr;
    int item,i=0,flag;
    ptr = head;
    if(ptr == NULL)
    {
        printf("\nEmpty List\n");
    }
    else
    {
        printf("\nEnter item which you want to search?\n");
        scanf("%d",&item);
        while (ptr!=NULL)
        {
            if(ptr->data == item)
            {
                printf("\nitem found at location %d ",i+1);
                flag=0;
                break;
            }
            else
            {
                flag=1;
            }
            i++;
            ptr = ptr -> next;
        }
        if(flag==1)
        {
            printf("\nItem not found\n");
        }
    }
}
```

Circular Singly Linked List

- In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.
- We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.
- The following image shows a circular singly linked list.

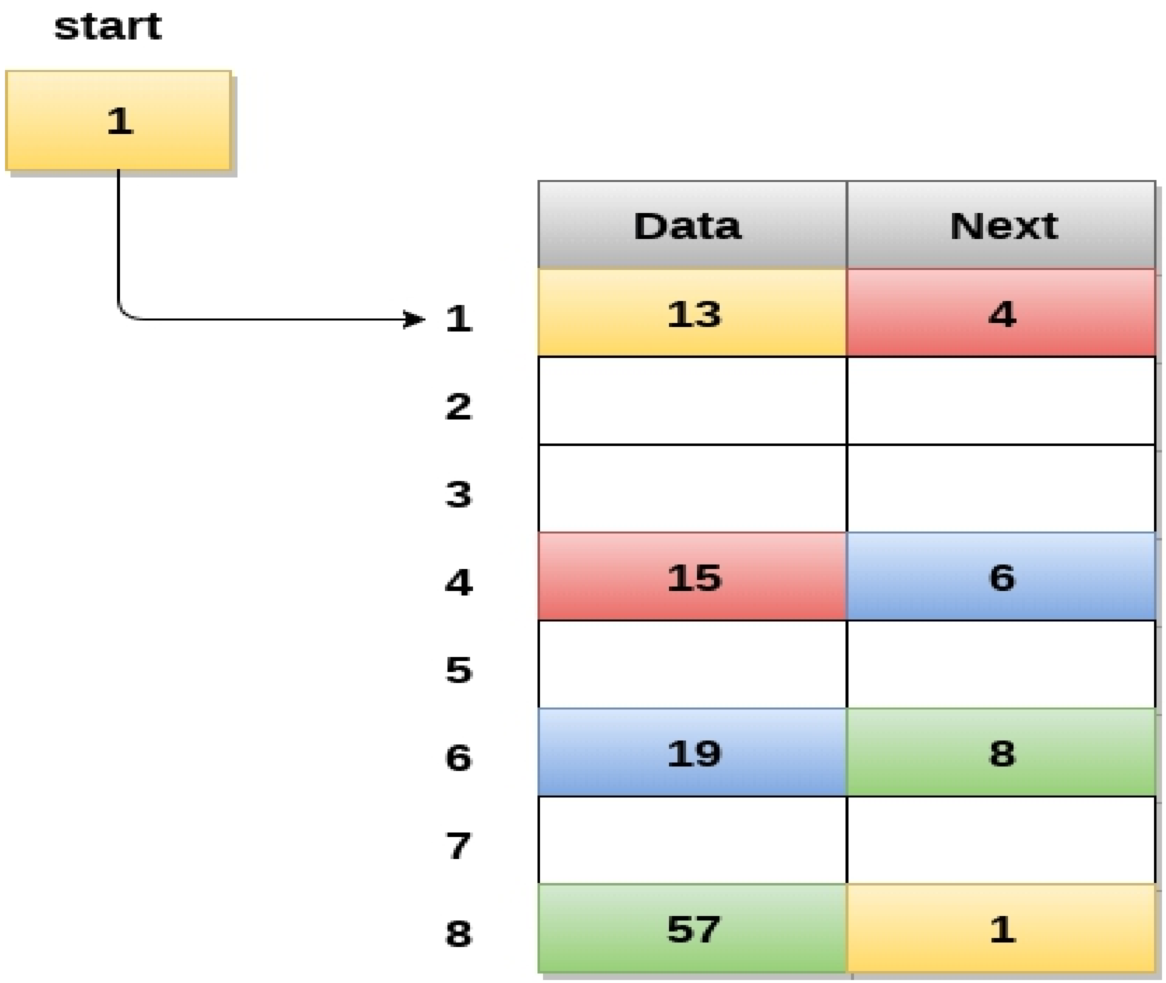


Circular Singly Linked List

- Circular linked list are mostly used in task maintenance in operating systems. There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.

Memory Representation of circular linked list:

- In the following image, memory representation of a circular linked list containing marks of a student in 4 subjects. However, the image shows a glimpse of how the circular list is being stored in the memory. The start or head of the list is pointing to the element with the index 1 and containing 13 marks in the data part and 4 in the next part. Which means that it is linked with the node that is being stored at 4th index of the list.
- However, due to the fact that we are considering circular linked list in the memory therefore the last node of the list contains the address of the first node of the list.
- We can also have more than one number of linked list in the memory with the different start pointers pointing to the different start nodes in the list. The last node is identified by its next part which contains the address of the start node of the list. We must be able to identify the last node of any linked list so that we can find out the number of iterations which need to be performed while traversing the list.



Memory Representation of a circular linked list

Operation

- Insertion at beginning
- Adding a node into circular singly linked list at the beginning.
- Insertion at the end
- Adding a node into circular singly linked list at the end.
- Deletion at beginning
- Removing the node from circular singly linked list at the beginning.
- Deletion at the end
- Removing the node from circular singly linked list at the end.
- Searching
- Compare each element of the node with the given item and return the location at which the item is present in the list otherwise return null.
- Traversing
- Visiting each element of the list at least once in order to perform some specific operation.

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
};
struct node *head;

void beginsert ();
void lastinsert ();
void randominsert();
void begin_delete();
void last_delete();
void random_delete();
void display();
void search();
void main ()
{
   int choice =0;
   while(choice != 7)
   {
      printf("\n*********Main Menu*********\n");
      printf("\nChoose one option from the following list ...\n
");
      printf("\n=========================================
=========\n");
      printf("\n1.Insert in begining\n2.Insert at last\n3.Delet
e from Beginning\n4.Delete from last\n5.Search for an elem
ent\n6.Show\n7.Exit\n");
      printf("\nEnter your choice?\n");
      scanf("\n%d",&choice);
 switch(choice)
 {
   case 1:
   beginsert();
   break;
   case 2:
   lastinsert();
   break;
   case 3:
   begin_delete();
   break;
   case 4:
   last_delete();
   break;
   case 5:
   search();
   break;
   case 6:
   display();
   break;
   case 7:
   exit(0);
   break;
   default:
   printf("Please enter valid choice..");
 }
 }
}
```

```c
void beginsert()
{
   struct node *ptr,*temp;
   int item;
   ptr = (struct node *)malloc(sizeof(struct node));
   if(ptr == NULL)
   {
      printf("\nOVERFLOW");
   }
   else
   {
      printf("\nEnter the node data?");
      scanf("%d",&item);
      ptr -> data = item;
      if(head == NULL)
      {
         head = ptr;
         ptr -> next = head;
      }
      else
      {
         temp = head;
         while(temp->next != head)
            temp = temp->next;
         ptr->next = head;
         temp -> next = ptr;
         head = ptr;
      }
      printf("\nnode inserted\n");
   }

}
```

```c
void lastinsert()
{
   struct node *ptr,*temp;
   int item;
   ptr = (struct node *)malloc(sizeof(struct node));
   if(ptr == NULL)
   {
      printf("\nOVERFLOW\n");
   }
   else
   {
      printf("\nEnter Data?");
      scanf("%d",&item);
      ptr->data = item;
      if(head == NULL)
      {
         head = ptr;
         ptr -> next = head;
      }
      else
      {
         temp = head;
         while(temp -> next != head)
         {
            temp = temp -> next;
         }
         temp -> next = ptr;
         ptr -> next = head;
      }

      printf("\nnode inserted\n");
   }
}
```

```c
void begin_delete()
{
    struct node *ptr;
    if(head == NULL)
    {
        printf("\nUNDERFLOW");
    }
    else if(head->next == head)
    {
        head = NULL;
        free(head);
        printf("\nnode deleted\n");
    }

    else
    {   ptr = head;
        while(ptr -> next != head)
            ptr = ptr -> next;
        ptr->next = head->next;
        free(head);
        head = ptr->next;
        printf("\nnode deleted\n");

    }
}
```

```c
void last_delete()
{
   struct node *ptr, *preptr;
   if(head==NULL)
   {
      printf("\nUNDERFLOW");
   }
   else if (head ->next == head)
   {
      head = NULL;
      free(head);
      printf("\nnode deleted\n");

   }
   else
   {
      ptr = head;
      while(ptr ->next != head)
      {
         preptr=ptr;
         ptr = ptr->next;
      }
      preptr->next = ptr -> next;
      free(ptr);
      printf("\nnode deleted\n");

   }
}
```

```
void search()
{
   struct node *ptr;
   int item,i=0,flag=1;
   ptr = head;
   if(ptr == NULL)
   {
      printf("\nEmpty List\n");
   }
   else
   {
      printf("\nEnter item which you want to search?\n");
      scanf("%d",&item);
      if(head ->data == item)
      {
      printf("item found at location %d",i+1);
      flag=0;
      }
      else
      {
      while (ptr->next != head)
      {
         if(ptr->data == item)
         {
            printf("item found at location %d ",i+1);
            flag=0;
            break;
         }
         else
         {
            flag=1;
         }
         i++;
         ptr = ptr -> next;
      }
      }
      if(flag != 0)
      {
         printf("Item not found\n");
      }
   }

}
```

```
void display()
{
   struct node *ptr;
   ptr=head;
   if(head == NULL)
   {
      printf("\nnothing to print");
   }
   else
   {
      printf("\n printing values ... \n");

      while(ptr -> next != head)
      {


         printf("%d\n", ptr -> data);
         ptr = ptr -> next;
      }
      printf("%d\n", ptr -> data);
   }


}
```

Algorithm

- First Start scanning the expression from left to right
- If the scanned character is an operand, output it, i.e. print it
- Else
  - If the precedence of the scanned operator is higher than the precedence of the operator in the stack(or stack is empty or has'(‘), then push operator in the stack
  - Else, Pop all the operators, that have greater or equal precedence than the scanned operator. Once you pop them push this scanned operator. (If we see a parenthesis while popping then stop and push scanned operator in the stack)
- If the scanned character is an ‘(‘, push it to the stack.
- If the scanned character is an ‘)’, pop the stack and output it until a ‘(‘ is encountered, and discard both the parenthesis.
- Now, we should repeat the steps 2 – 6 until the whole infix i.e. whole characters are scanned.
- Print output
- Do the pop and output (print) until stack is not empty

Infix to Postfix Conversion

Expression = A + B * C / D - F + A ^ E

Scanned Symbol	Stack	Output	Reason	
A		A	Step 2	
+	+	A	Step 3.1	
B	+	AB	Step 2	
*	+*	AB	Step 3.1	
C	+*	ABC	Step 2	
/	+/	ABC*	Step 3.2	/ prec. is equal to * so not higher, so going to step 3.2
D	+/	ABC*D	Step 2	
-	-	ABC*D/+	Step 3.2	/ will be popped, added to o/p & then + popped & o/p. - will be pushed
F	-	ABC*D/+F	Step 2	
+	+	ABC*D/+F-	Step 3.2	- will be popped, added to o/p and then + to stack
A	+	ABC*D/+F-A	Step 2	
^	+^	ABC*D/+F-A	Step 2	
E	+^	ABC*D/+F-AE	Step 2	
(empty)		ABC*D/+F-AE^+	Step 8	

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.	

- Given Infix - ((a/b)+c)-(d+(e*f))
- **Step 1:** Reverse the infix string. Note that while reversing the string you must interchange left and right parentheses.
- **Step 2:** Obtain the postfix expression of the expression obtained from Step 1.
- **Step 3:** Reverse the postfix expression to get the prefix expression
- This is how you convert manually for theory question in the exam
  - **String after reversal –**))f*e(+d(-)c+)b/a((
  - **String after interchanging right and left parenthesis** – ((f*e)+d)-(c+(b/a))
  - **Apply postfix –**
  - **Reverse Postfix Expression**

Sr. no.	Expression	Stack	Prefix
0	(	(	
1	(	((	
2	f	((	f
3	*	((*	f
4	e	((*	fe
5	)	(	fe*
6	+	(+	fe*
7	d	(+	fe*d
8	)		fe*d+
9	−	−	fe*d+
10	(	-(	fe*d+
11	c	-(	fe*d+c
12	+	-(+	fe*d+c
13	(	-(+(	fe*d+c
14	b	-(+(	fe*d+cb
15	/	-(+(/	fe*d+cb
16	a	-(+(/	fe*d+cba
17	)	-(+	fe*d+cba/
18	)	−	fe*d+cba/+
19			fe*d+cba/+-

```
Final prefix: -+/abc+d*ef
```

- int checkIfOperand(char ch)
- { return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z'); }
- // Fucntion to compare precedence // If we return larger value means higher precedence
- int precedence(char ch) {
- switch (ch) { case '+': case '-': return 1;
- case '*': case '/': return 2;
- case '^': return 3; } return -1; }
- // The driver function for infix to postfix conversion
- int covertInfixToPostfix(char* expression) { int i, j; // Stack size should be equal to expression size for safety
- struct Stack* stack = create(strlen(expression));
- if(!stack) // just checking is stack was created or not
- return -1 ;
- for (i = 0, j = -1;expression[i]; ++i) { // Here we are checking is the character we scanned is operand or not // and this adding to to output.
- if (checkIfOperand(expression[i]))
- expression[++j] = expression[i]; // Here, if we scan character '(', we need push it to the stack.
- else if (expression[i] == '(')
- push(stack, expression[i]); // Here, if we scan character is an ')', we need to pop and print from the stack // do this until an '(' is encountered in the stack.
- else if (expression[i] == ')') { while (!isEmpty(stack) && peek(stack) != '(') expression[++j] = pop(stack);
- if (!isEmpty(stack) && peek(stack) != '(') return -1; // invalid expression
- else pop(stack); }
- else // if an opertor
- { while (!isEmpty(stack) && precedence(expression[i]) <= precedence(peek(stack))) expression[++j] = pop(stack); push(stack, expression[i]); } } // Once all inital expression characters are traversed // adding all left elements from stack to exp
- while (!isEmpty(stack)) expression[++j] = pop(stack); expression[++j] = '\0'; printf("%s", expression); }

```
void InfixtoPrefix(char *exp){

int size = strlen(exp);

// reverse string
reverse(exp);
//change brackets
brackets(exp);
//get postfix
getPostfix(exp);
// reverse string again
reverse(exp);
}
```

- void reverse(char *exp){

```
int size = strlen(exp);
int j = size, i=0;
char temp[size];

temp[j--]='\0';
while(exp[i]!='\0')
{
temp[j] = exp[i];
j--;
i++;
}
strcpy(exp,temp);
}
```

- ```
  void brackets(char* exp){
  int i = 0;
  while(exp[i]!='\0')
  {
  if(exp[i]=='(')
  exp[i]=')';
  else if(exp[i]==')')
  exp[i]='(';
  i++;
  }
  }
  ```
  ```

```
int getPostfix(char* expression)
{
int i, j;

// Stack size should be equal to expression size
for safety
struct Stack* stack = create(strlen(expression));
if(!stack) // just checking is stack was created
or not
return -1 ;

for (i = 0, j = -1; expression[i]; ++i)
{
// Here we are checking is the character we
scanned is operand or not
// and this adding to to output.
if (checkIfOperand(expression[i]))
expression[++j] = expression[i];

// Here, if we scan character '(', we need push
it to the stack.
else if (expression[i] == '(')
push(stack, expression[i]);

// Here, if we scan character is an ')', we need to pop
and print from the stack
// do this until an '(' is encountered in the stack.
else if (expression[i] == ')')
{
while (!isEmpty(stack) && peek(stack) != '(')
expression[++j] = pop(stack);
if (!isEmpty(stack) && peek(stack) != '(')
return -1; // invalid expression
else
pop(stack);
}
else // if an opertor
{
while (!isEmpty(stack) && precedence(expression[i])
<= precedence(peek(stack)))
expression[++j] = pop(stack);
push(stack, expression[i]);
}

}

// Once all inital expression characters are traversed
// adding all left elements from stack to exp
while (!isEmpty(stack))
expression[++j] = pop(stack);

expression[++j] = '\0';

}
```

- Iterate the given expression from left to right, one character at a time
- If a character is operand, push it to stack.
- If a character is an operator,
  - pop operand from the stack, say it’s s1.
  - pop operand from the stack, say it’s s2.
  - perform **(s2 operator s1)** and push it to stack.
- Once the expression iteration is completed, initialize the result string and pop out from the stack and add it to the result.
- Return the result.

Postfix Expression : ABC/-AK/L-*			
Token	**Action**	**Stack**	**Notes**
A	Push **A** to stack	[A]	
B	Push **B** to stack	[A, B]	
C	Push **C** to stack	[A, B, C]	
/	Pop **C** from stack	[A, B]	Pop two operands from stack, C and B. Perform B/C and push (B/C) to stack
	Pop **B** from stack	[A]	
	Push **(B/C)** to stack	[A, (B/C)]	
-	Pop **(B/C)** from stack	[A]	Pop two operands from stack, (B/C) and A. Perform A-(B/C) and push (A-(B/C)) to stack
	Pop **A** from stack	[]	
	Push **(A-(B/C))** to stack	[((A-(B/C))]	
A	Push **A** to stack	[(A-(B/C)), A]	
K	Push **K** to stack	[(A-(B/C)), A, K]	
/	Pop **K** from stack	[(A-(B/C)), A]	Pop two operands from stack, K and A. Perform A/K and push (A/K) to stack
	Pop **A** from stack	[((A-(B/C))]	
	Push **(A/K)** to stack	[(A-(B/C)), (A/K)]	
L	Push **L** to stack	[(A-(B/C)), (A/K), L]	
-	Pop **L** from stack	[(A-(B/C)), (A/K)]	Pop two operands from stack, L and (A/K). Perform (A/K)-L and push ((A/K)-L) to stack
	Pop **(A/K)** from stack	[((A-(B/C))]	
	Push **((A/K)-L)** to stack	[(A-(B/C)), ((A/K)-L)]	
*	Pop **((A/K)-L)** from stack	[((A-(B/C))]	Pop two operands from stack, (A/K)-L and A-(B/C). Perform (A-(B/C))*((A/K)-L) and push ((A-(B/C))*((A/K)-L)) to stack
	Pop **((A-(B/C))** from stack	[]	
	Push **((A-(B/C))*((A/K)-L))** to stack	[((A-(B/C))*((A/K)-L))]	
Infix Expression: ((A-(B/C))*((A/K)-L))			

Convert the following postfix expression A B C * D E F ^ / G * – H * + into its equivalent infix expression.

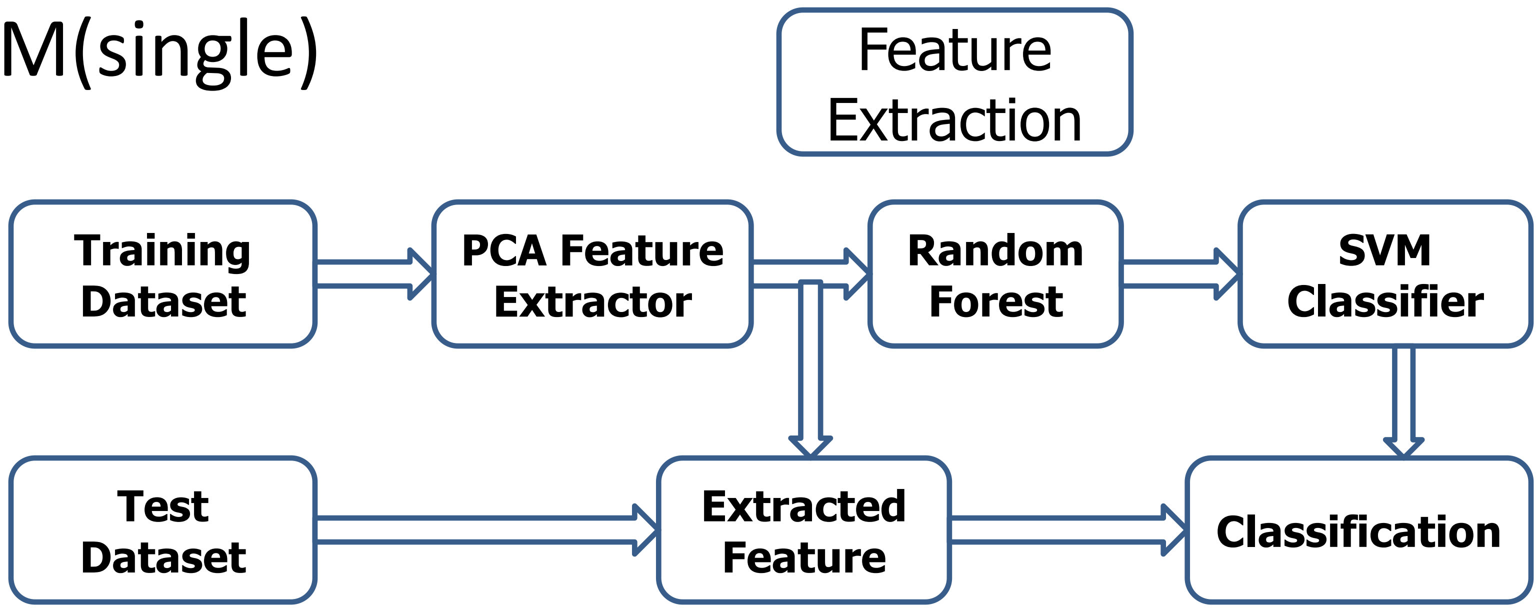
Symbol	Stack	Remarks
A	A	Push A
B	A B	Push B
C	A B C	Push C
*	A (B*C)	Pop two operands and place the operator in between the operands and push the string.
D	A (B*C) D	Push D
E	A (B*C) D E	Push E
F	A (B*C) D E F	Push F
^	A (B*C) D (E^F)	Pop two operands and place the operator in between the operands and push the string.
/	A (B*C) (D/(E^F))	Pop two operands and place the operator in between the operands and push the string.
G	A (B*C) (D/(E^F)) G	Push G
*	A (B*C) ((D/(E^F))*G)	Pop two operands and place the operator in between the operands and push the string.
-	A ((B*C) – ((D/(E^F))*G))	Pop two operands and place the operator in between the operands and push the string.
H	A ((B*C) – ((D/(E^F))*G)) H	Push H
*	A (((B*C) – ((D/(E^F))*G)) * H)	Pop two operands and place the operator in between the operands and push the string.
+	(A + (((B*C) – ((D/(E^F))*G)) * H))	
End of string	The input is now empty. The string formed is infix.	

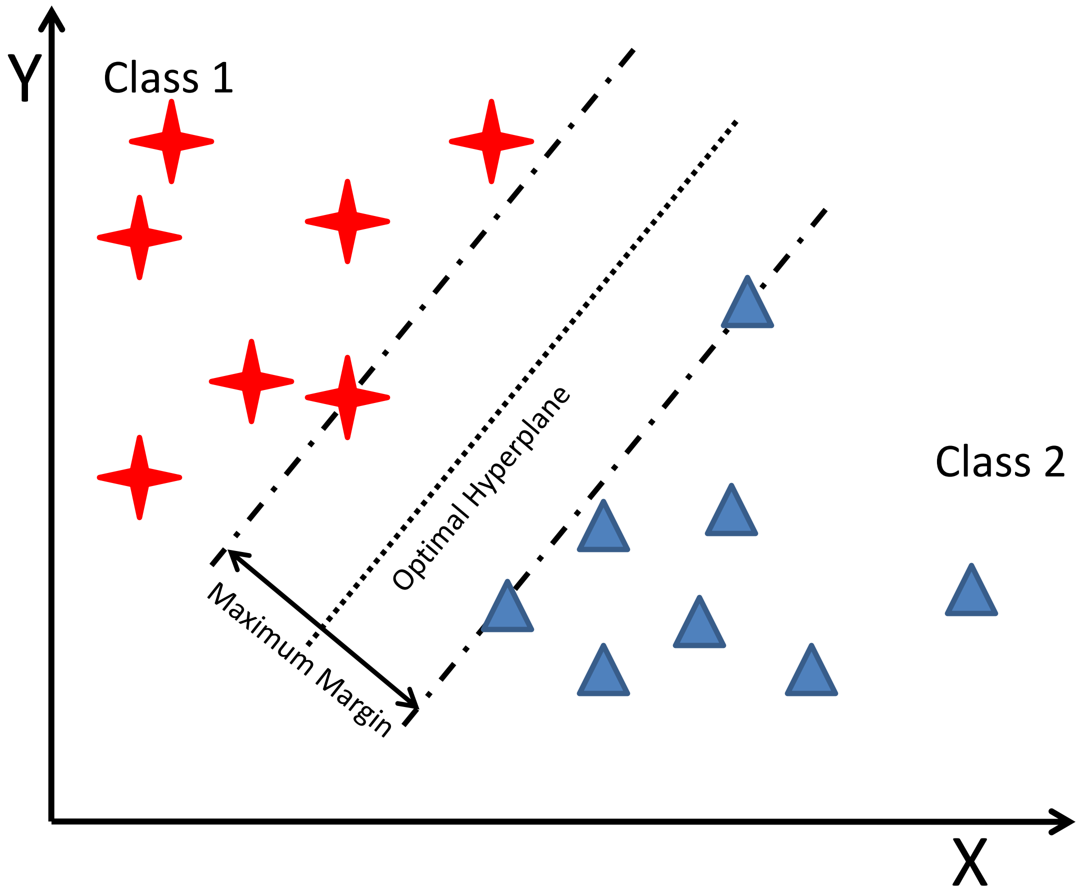
Convert the following postfix expression A B C * D E F ^ / G * – H * + into its equivalent prefix expression.

Symbol	Stack	Remarks
A	A	Push A
B	A, B	Push B
C	A, B, C	Push C
*	A, *BC	Pop two operands and place the operator in front the operands and push the string.
D	A, *BC, D	Push D
E	A, *BC, D, E	Push E
F	A, *BC, D, E, F	Push F
^	A, *BC, D, ^EF	Pop two operands and place the operator in front the operands and push the string.
/	A, *BC, /D^EF	Pop two operands and place the operator in front the operands and push the string.
G	A, *BC, /D^EF, G	Push G
*	A, *BC, */D^EFG	Pop two operands and place the operator in front the operands and push the string.
-	A, - *BC*/D^EFG	Pop two operands and place the operator in front the operands and push the string.
H	A, - *BC*/D^EFG, H	Push H
*	A, *- *BC*/D^EFGH	Pop two operands and place the operator in front the operands and push the string.
+	+A*-*BC*/D^EFGH	
End of string	The input is now empty. The string formed is prefix.	

Queue

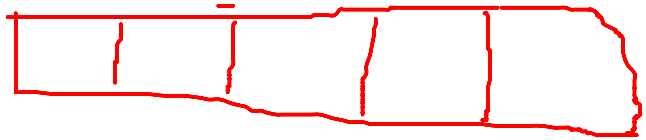
- training data --> PCA --> Random Forest --> SVM (loop)
- test data --> PCA --> Random Forest --> SVM(single)



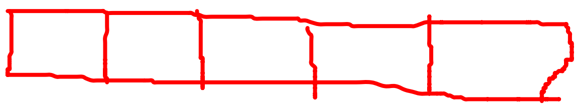

Y
Class 1
Optimal Hyperplane
Maximum Margin
Class 2
X

q_1 q_2 S

1 5 3 · 2 4

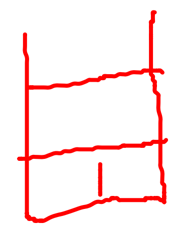
q_1 

Push (S, x)
enqueue (q_2, x)
dequeue (q_1) &
enqueue (q_2,)

q_2 

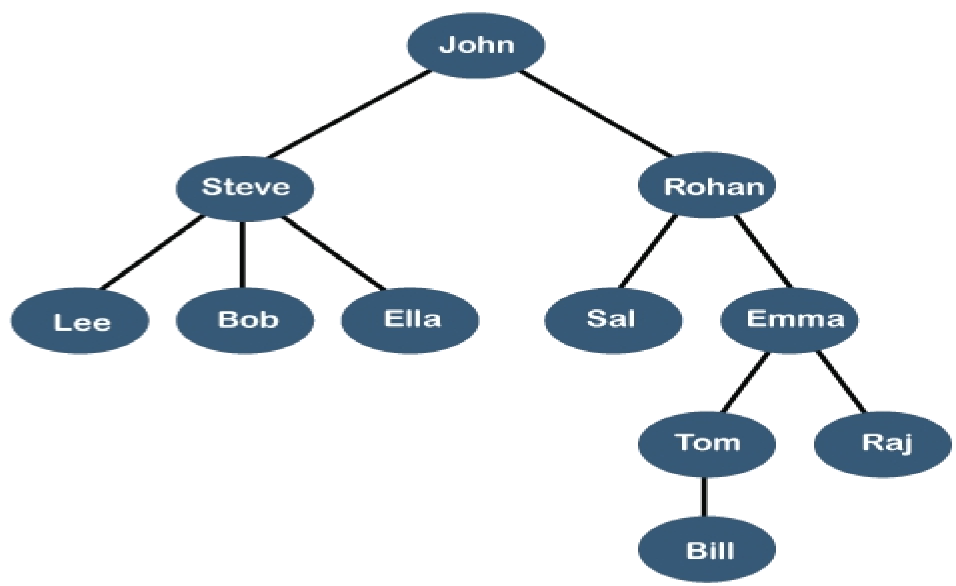
swap q_1, q_2

POP (S)



Tree

A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.

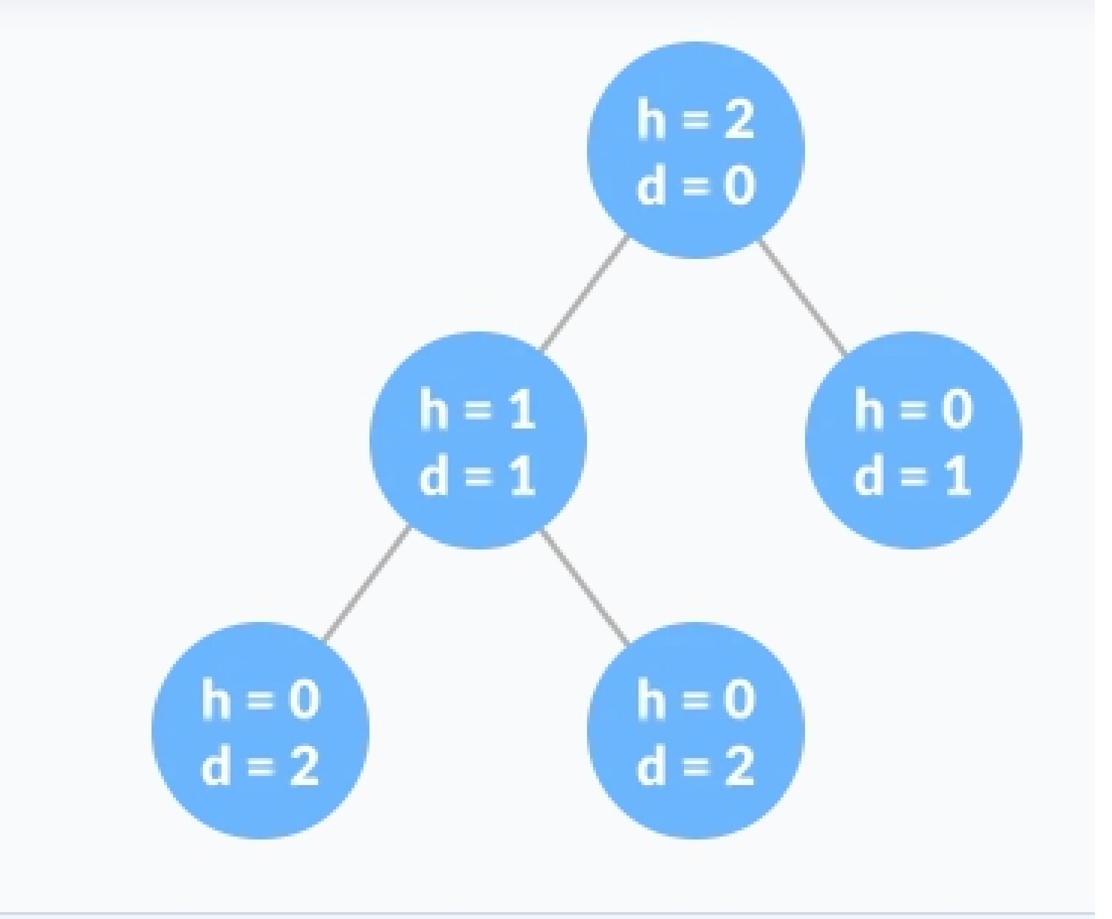


Tree Terminologies

- **Node**
  - A node is an entity that contains a key or value and pointers to its child nodes.
  - The last nodes of each path are called **leaf nodes or external nodes** that do not contain a link/pointer to child nodes.
  - The node having at least a child node is called an **internal node**.
- **Edge**
  - It is the link between any two nodes.
- **Root**

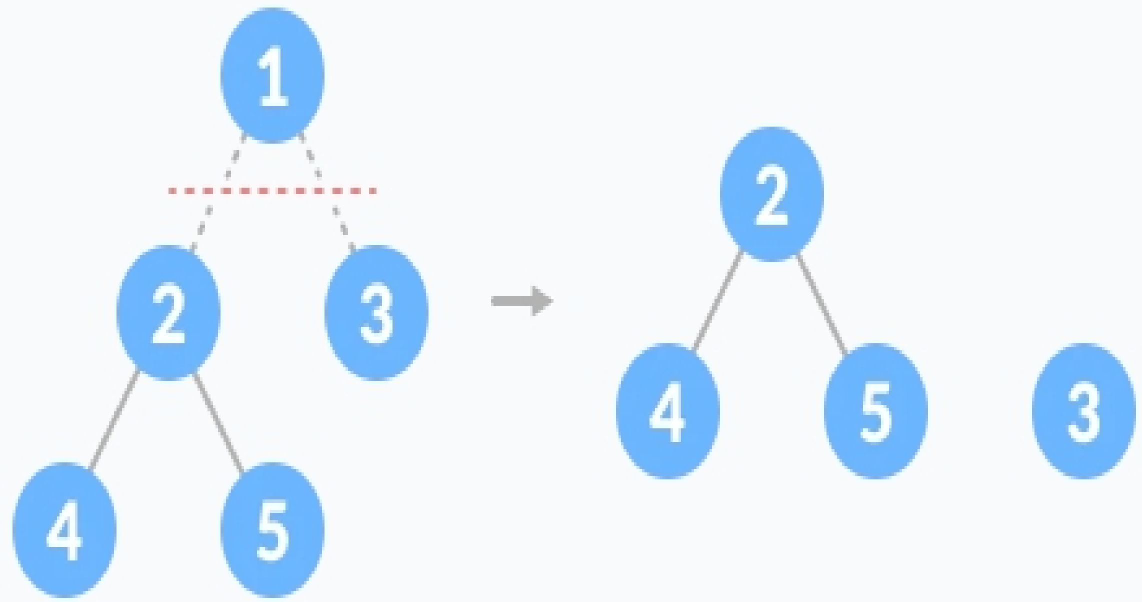
− The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.

- **Height of a Node**
  - The height of a node is the number of edges from the node to the deepest leaf (ie. the longest path from the node to a leaf node).
- **Depth of a Node**
  - The depth of a node is the number of edges from the root to the node.
- **Height of a Tree**
  - The height of a Tree is the height of the root node or the depth of the deepest node.
-



Height and depth of each node in a tree

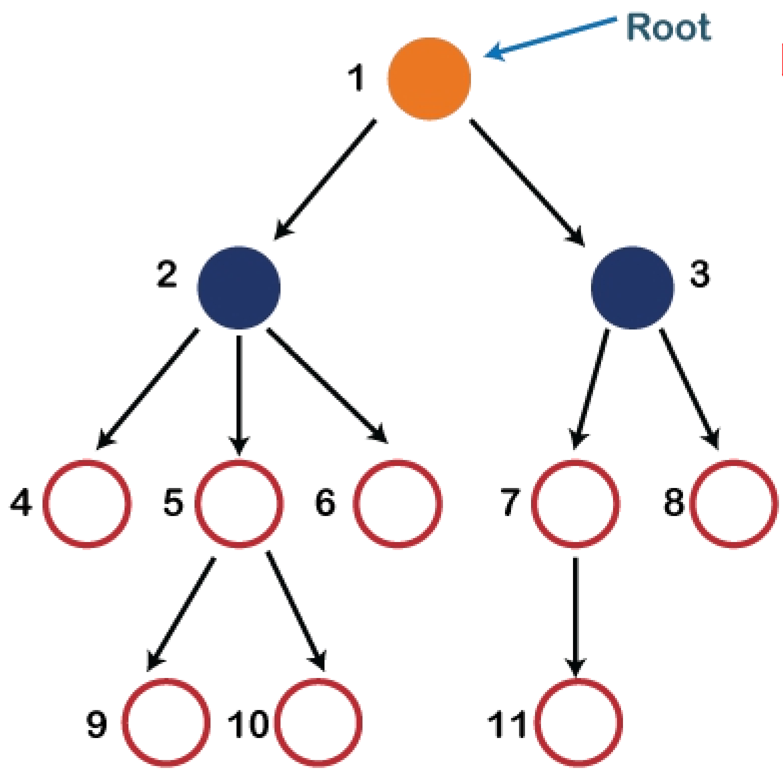
- **Degree of a Node**
- The degree of a node is the total number of branches of that node.
- **Forest**
- A collection of disjoint trees is called a forest.
- You can create a forest by cutting the root of a tree.



Creating forest from a tree

- **Path** − Path refers to the sequence of nodes along the edges of a tree.
- **Parent** − Any node except the root node has one edge upward to a node called parent.
- **Child** − The node below a given node connected by its edge downward is called its child node.
- **Leaf** − The node which does not have any child node is called the leaf node.
- **Sibling:** The nodes that have the same parent are known as siblings.
- **Subtree** − Subtree represents the descendants of a node.
- **Visiting** − Visiting refers to checking the value of a node when control is on the node.
- **Traversing** − Traversing means passing through nodes in a specific order.
- **Levels** − Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **keys** − Key represents a value of a node based on which a search operation is to be carried out for a node.

Introduction to Trees



Level 0

Level 1

Level 2

Level 3

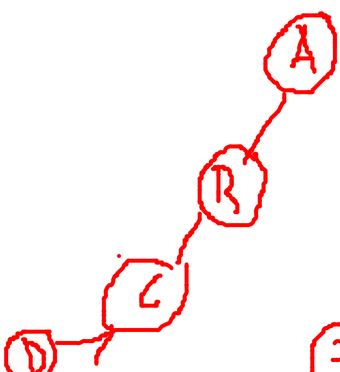
- **Internal nodes:** A node has atleast one child node known as an ***internal***
- **Ancestor node:-** An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.
- **Descendant:** The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.

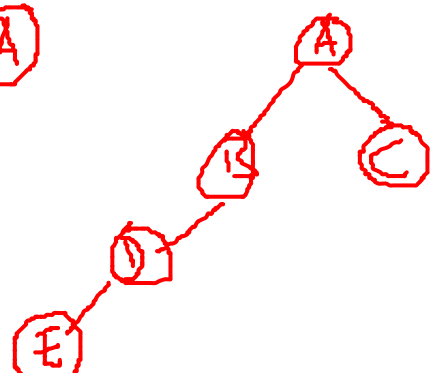
Types of Tree

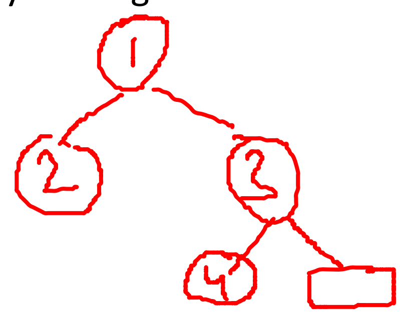
- Binary Tree
- Binary Search Tree
- AVL Tree
- B-Tree

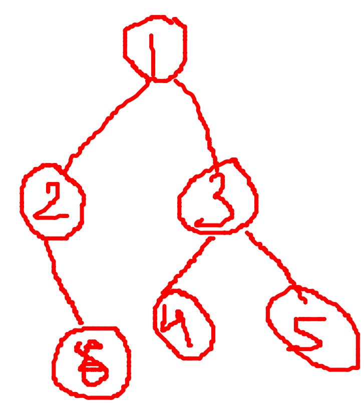
Binary Tree

- A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.
- **Full Binary Tree** A Binary Tree is a full binary tree if every node has 0 or 2 children. The following are the examples of a full binary tree. We can also say a full binary tree is a binary tree in which all nodes except leaf nodes have two children.
- **Complete Binary Tree:** A Binary Tree is a Complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible
- **Perfect Binary Tree** A Binary tree is a Perfect Binary Tree in which all the internal nodes have two children and all leaf nodes are at the same level.
- **Extended Binary Tree**
- A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required.
- **Balanced Binary Tree**
  A binary tree is balanced if the height of the tree is O(Log n) where n is the number of nodes. For Example, the AVL tree maintains O(Log n) height by making sure that the difference between the heights of the left and right subtrees is at most 1.





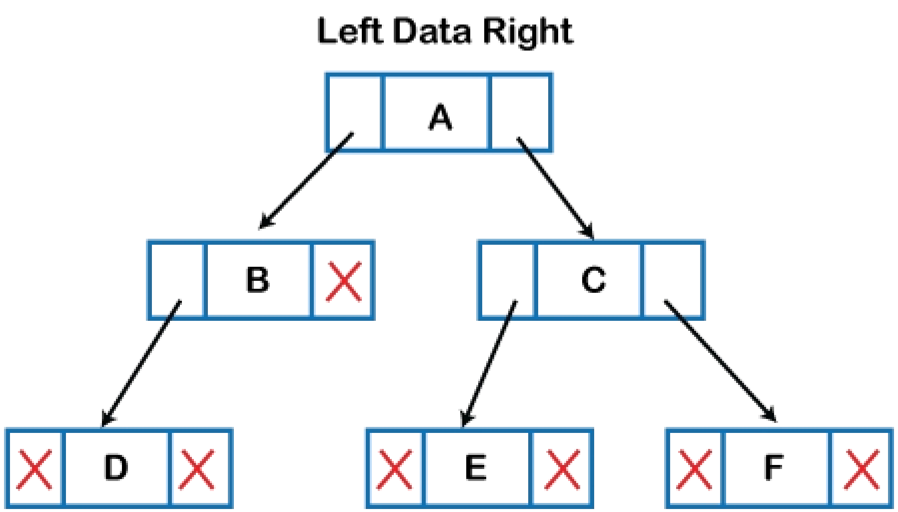




Tree Applications

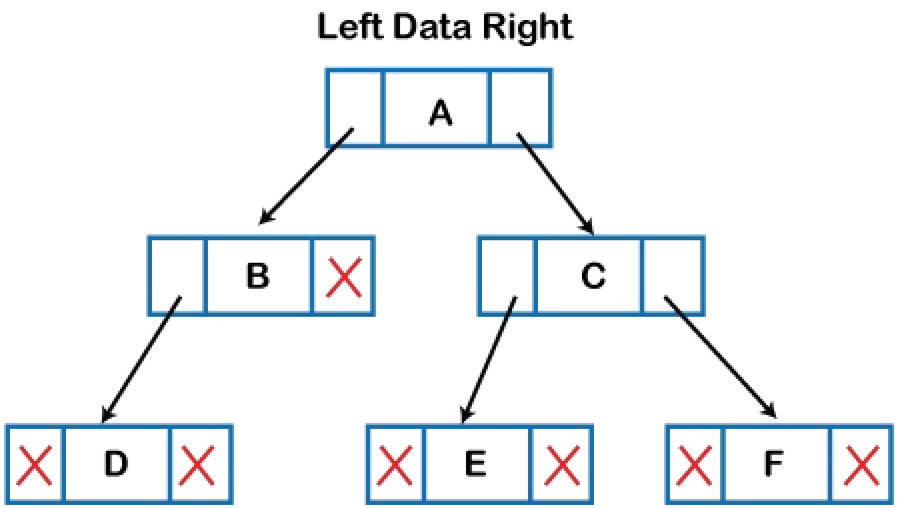
- Binary Search Trees(BSTs) are used to quickly check whether an element is present in a set or not.
- Heap is a kind of tree that is used for heap sort.
- A modified version of a tree called Tries is used in modern routers to store routing information.
- Most popular databases use B-Trees and T-Trees, which are variants of the tree structure we learned above to store their data
- Compilers use a syntax tree to validate the syntax of every program you write.

- A binary tree data structure is represented using two methods. Those methods are as follows...
- **Array Representation**
- **Linked List Representation**


Left Data Right
A
B
C
D
E
F

Implementation of Tree

- The tree data structure can be created by creating the nodes dynamically with the help of the pointers. The tree in the memory can be represented as shown below:



- The above figure shows the representation of the tree data structure in the memory. In the above structure, the node contains three fields. The second field stores the data; the first field stores the address of the left child, and the third field stores the address of the right child.

Root("A");
~~setleft("B", 0)~~.
setright("C", 1);

mirror(node->left),
mirror(node->right),
temp = node->left,
node->left = node->right,
node->right = temp,

- struct node
  { int data;
  struct node *leftChild;
  struct node *rightChild; };
  Struct node *createnode(int data){
  Struct node *ptr=(struct node*)malloc(sizeof(struct node));
  ptr->data=data; ptr->leftChild=NULL;
  ptr->rightChild=NULL; return(ptr);}

  Struct node *root=createnode(1);
  Root->leftChild=createnode(2);
  Root->rightChild=createnode(3);
  Root->leftChild->leftChild=createnode(4);

{ str[0] = key,
{ ~~t = (root * 2) + 1;~~
[str[t] = key.
t = (root * 2) + 2;
str[t] = key,]

Tree Traversal

- In order to perform any operation on a tree, you need to reach to the specific node. The tree traversal algorithm helps in visiting a required node in the tree.

Inorder Traversal

- Algorithm Inorder(tree)
  - 1. Traverse the left subtree, i.e., call Inorder(left-subtree)
  - 2. Visit the root.
  - 3. Traverse the right subtree, i.e., call Inorder(right-subtree)

- void printInorder(struct Node* node)
- {
-     if (node == NULL)
-         return;
-
-     /* first recur on left child */
-     printInorder(node->left);
-
-     /* then print the data of node */
-    printf(“%d”,node->data);
-
-     /* now recur on right child */
-     printInorder(node->right);
- }

Preorder Traversal

- Algorithm Preorder(tree)
  - 1. Visit the root.
  - 2. Traverse the left subtree, i.e., call Preorder(left-subtree)
  - 3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Postorder Traversal

- Algorithm Postorder(tree)
  - 1. Traverse the left subtree, i.e., call Postorder(left-subtree)
  - 2. Traverse the right subtree, i.e., call Postorder(right-subtree)
  - 3. Visit the root.

InOrder(root) visits nodes in the following order:

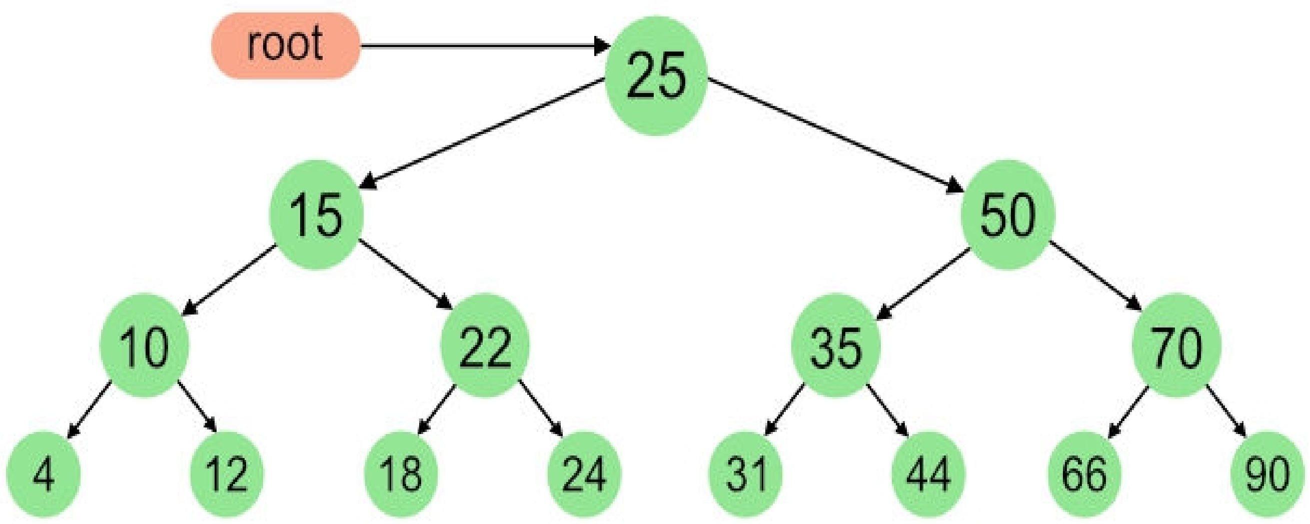
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:

4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



Binary Search Tree

- **Binary Search Tree** is a node-based binary tree data structure which has the following properties:
- The left subtree of a node contains only nodes with keys lesser than the node’s key.
- The right subtree of a node contains only nodes with keys greater than the node’s key.
- The left and right subtree each must also be a binary search tree.

```
struct node* search(struct node* root, int key)
{
   // Base Cases: root is null or key is present at root
   if (root == NULL || root->key == key)
      return root;

   // Key is greater than root's key
   if (root->key < key)
      return search(root->right, key);

   // Key is smaller than root's key
   return search(root->left, key);
}
```

Insertion of a key

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

```c
struct node* insert(struct node* node, int key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL)
        return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}
```

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

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

- **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.

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

- Suppose keys are inserted into an initially empty (i) AVL Tree and (ii) B-Tree in the following order: 100, 200, 50, 300, 400, 25, 250, 225, 500, 240, 260. From the above tree, suppose keys are deleted in the following order: 225, 100, 200, 500, 50. Draw the (i) AVL Tree (ii) B-Tree after each deletions are performed

M-way Trees

- Before learning about B-Trees we need to know what M-way trees are, and how B-tree is a special type of M-way tree. An M-way(multi-way) tree is a tree that has the following properties:
- Each node in the tree can have at most **m** children.
- Nodes in the tree have at most **(m-1)** key fields and pointers(references) to the children.

M-way Search Trees

- An M-way search tree is a more constrained m-way tree, and these constrain mainly apply to the key fields and the values in them. The constraints on an **M-way** tree that makes it an M-way search tree are:
- Each node in the tree can associate with **m** children and **m-1** key fields.
- The keys in any node of the tree are arranged in a sorted order(**ascending**).
- The keys in the first **K** children are **less than** the **Kth** key of this node.
- The keys in the last **(m-K**) children are higher than the **Kth** key.

B Trees Data Structure:

- A B tree is an extension of an M-way search tree. Besides having all the properties of an M-way search tree, it has some properties of its own, these mainly are:
- All the leaf nodes in a B tree are at the same level.
- All internal nodes must have **M/2** children.
- If the root node is a non leaf node, then it must have at least two children.
- All nodes except the root node, must have at least **[M/2]-1** keys and at most **M-1** keys.

Heap Tree

- Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly. If α has child node β then −
  - $key(\alpha) \geq key(\beta)$
- As the value of parent is greater than that of child, this property generates Max Heap. Based on this criteria, a heap can be of two types −
- For Input → 35 33 42 10 14 19 27 44 26 31

- Min-Heap − Where the value of the root node is less than or equal to either of its children.
  - Step 1 − Create a new node at the end of heap.
  - Step 2 − Assign new value to the node.
  - Step 3 − Compare the value of this child node with its parent.
  - Step 4 − If value of parent is less than child, then swap them.
  - Step 5 − Repeat step 3 & 4 until Heap property holds.
- Max-Heap − Where the value of the root node is greater than or equal to either of its children.
  - Step 1 − Remove root node.
  - Step 2 − Move the last element of last level to root.
  - Step 3 − Compare the value of this child node with its parent.
  - Step 4 − If value of parent is less than child, then swap them.
  - Step 5 − Repeat step 3 & 4 until Heap property holds.

Sorting

- Bubble Sort: 34, 15, 29, 8

Heap Sort
40, 60, 10,20,50,30

Algorithm for Insertion Sort

(Insertion Sort) INSERTION(A,N)

This algorithm sorts the array A with N elements.

1. Set A[0]:= -∞[Initializes sentinel element.]
2. Repeat Step 3 to 5 for K=2,3,………,N:
3. Set Temp:=A[K] and PTR:=K-1.
4. Repeat while Temp<A[PTR]:
    (a)Set A[PTR+1]:=A[PTR}.[Move element forward.]
    (b)Set PTR:=PTR-1.

   [End of loop.]

5.Set A[PTR+1]:=TEMP.[Insert elements in proper place.]

6.Return.

	0	1	2	3	4	5	6	7	8
A=	31	45	11	89	55	19	21	8	40

11 31 45 89

11 31 45 55 89

11 19 31 45 55 89

11 19 21 31 45 55 89

8 11 19 21 31 45 55 89

8 11 19 21 31 40 45 55 89

temp = A[i]

40 ~~8~~ ~~21~~ ~~19~~ $\boxed{\cancel{45}}$ ~~11~~ ~~89~~ ~~55~~

for (i = 1 to 8)

for (j = i - 1; j >= 0 && temp < A[j]; j--)

{

A[j+1] = A[j];

}

A[j+1] = temp;

$O(n^2)$

$O(n)$

Selection Sort

A[0] A[1] A[7]

77 33 44 11 88 22 66 55

11 33 44 77 88 22 66 55

22 44 77 88 33 66 55

33 77 88 44 66 55

44 88 77 66 55

55 77 66 88

66 77 88

77 88

```
for(i=0 to i<n)
{ min=i;
for(j=i+1; j<n)
  if(a[j]<a[min])
     min=j;
if (min!=i)
swap(a[~~min~~],a[j])
```

10 14 27 33 19 35 42 44

Merge sort

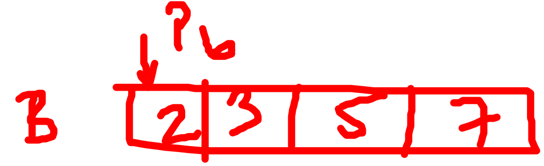
14 33 10 27

10 14 19 27 33 35 42 44

- Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.
- Merge sort first divides the array into equal halves and then combines them in a sorted manner.
- 14, 33, 27, 10, 35, 19, 42, 44

14 33 27 10 35 19 42 44

14 33 27 10 35 19 42 44

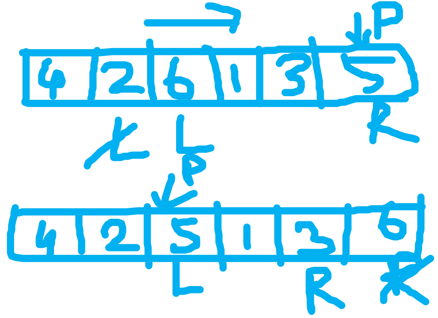


if $A[P_a] < B[P_b]$

$C[P_c] = A[P_a]$

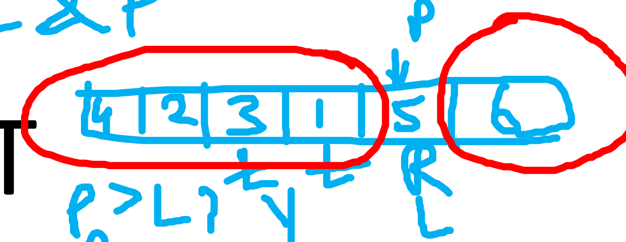
else

$C[P_c] = B[P_b]$



P>L? No. swap L & P

QUICK SORT



P<R? Y
P<R? N

P>L? Y
P>L? Y

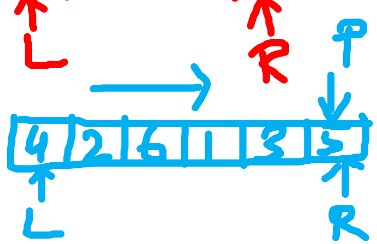
- This sorting algorithm uses the idea of divide and conquer. It finds the element called pivot which divides the array into two halves in such a way that elements in the left half are smaller than pivot and elements in the right half are greater than pivot. QUICK SORT
- Three steps 1. Find pivot that divides the array into two halves. 2. Quick sort the left half. 3. Quick sort the right half. QUICK SORT
- Example Consider an array having 6 elements 5 2 6 1 3 4 Arrange the elements in ascending order using quick sort algorithm
- Array index 0 1 2 3 4 5
- Array element 5 2 6 1 3 4

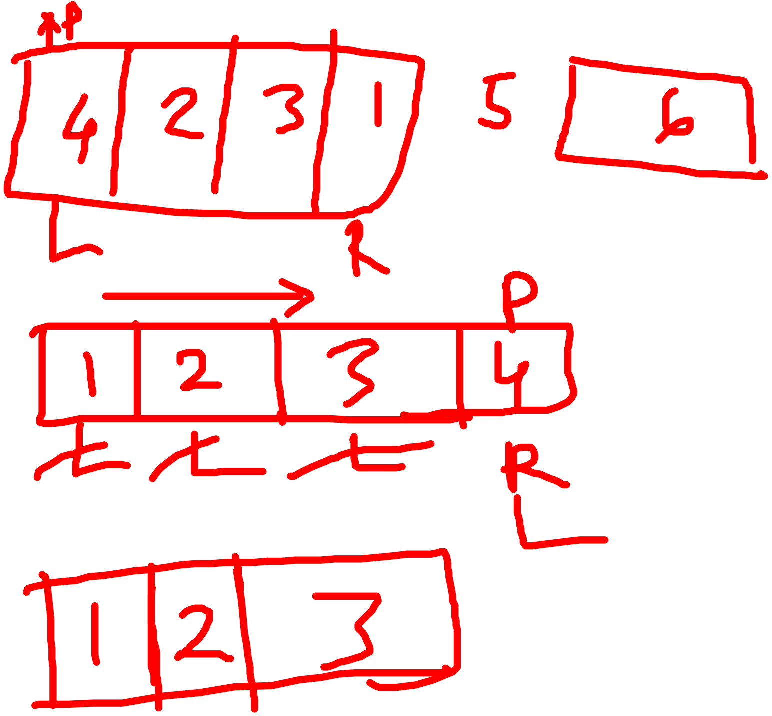
P = 5
R = 4

P<R? No. Swap (L

P>L? Yes, move L one position right

P=5
L=4





P < R? No. Swap & move Pivot

P > L? Y.

P > L? Y

? > L? Y

Hashing

0	1	2	3	4	5	6
5	7	11	8	10	9	2

Key/Value Pair

7189

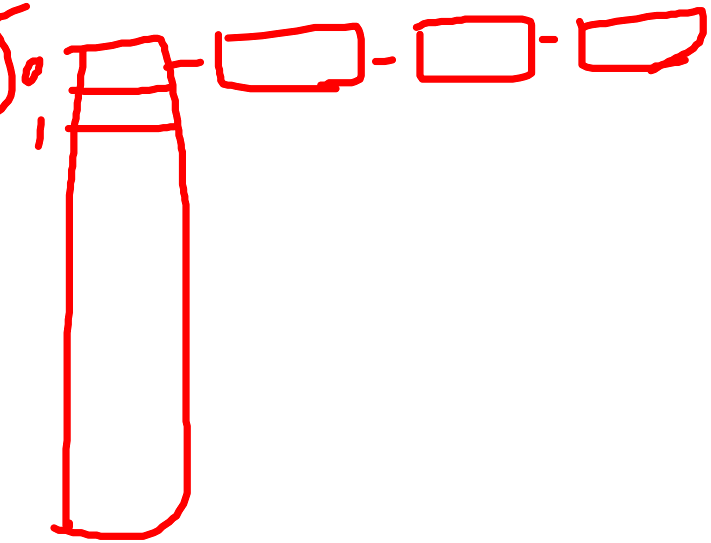
Hash f^n: mapped key → index

CN % 10

CN = 1081, 1523, 3446, 7189, 5524

Hash table:

(2723)

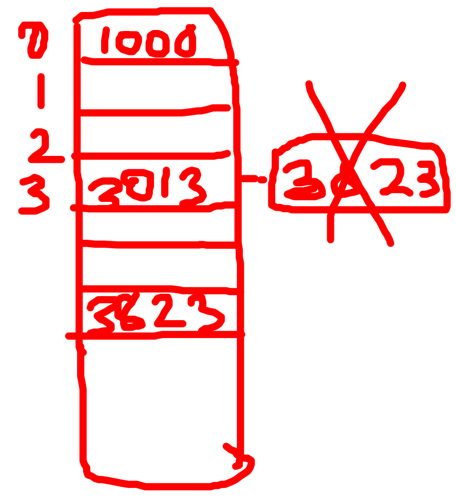


Collision:

Binning: 0 – 99
100 – 199
200 – 299

1. Open Hashing (Chaining):

2. Closed Hashing (Open addressing)



i) Linear Probing

i+1, i+2, i+3

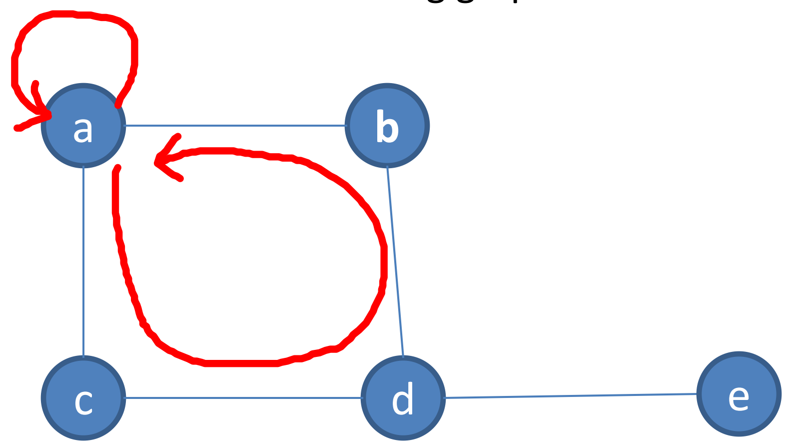
ii) Quadratic Probing: i+1 i+4 i+9 i+16

iii) Double Hashing: i+5 i+2*5 i+3*5

Graph

- A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as vertices, and the links that connect the vertices are called edges.

- Formally, a graph is a pair of sets (V, E), where V is the set of vertices and E is the set of edges, connecting the pairs of vertices. Take a look at the following graph −



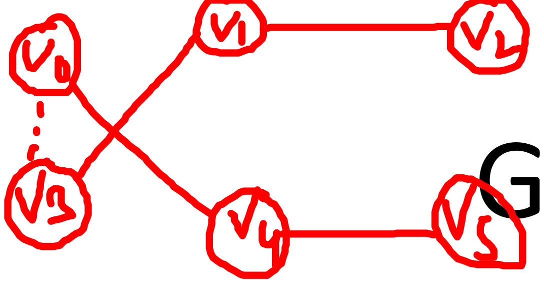
- Graph Basics
- In the above graph,

- V = {a, b, c, d, e}

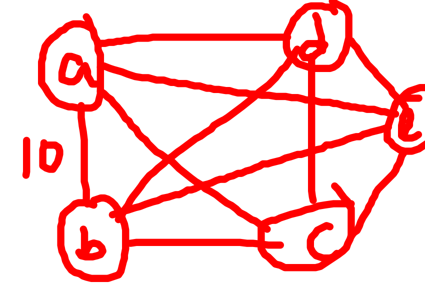
- E = {ab, ac, bd, cd, de}

- Directed and Undirected Graph
- A graph can be directed or undirected. However, in an undirected graph, edges are not associated with the directions with them. An undirected graph is shown in the above figure since its edges are not attached with any of the directions. If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.

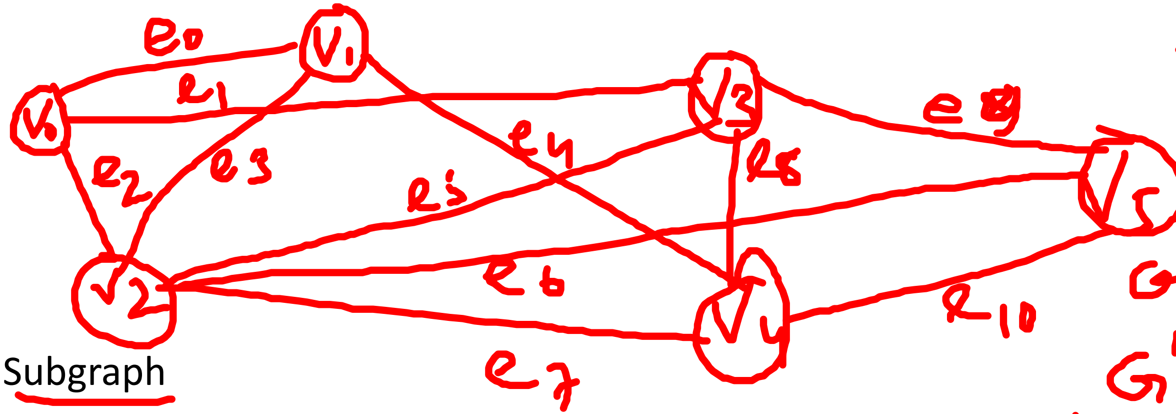
- In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node while node B is called terminal node.



Graph Terminology



- Path: A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.
- Closed Path: A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if V0=VN.
- Simple Path: If all the nodes of the graph are distinct with an exception V0=VN, then such path P is called as closed simple path.
- Cycle: A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.
- Connected Graph: A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph. T without any cycle
- Complete Graph: A complete graph is the one in which every node is connected with all other nodes. A complete graph contain n(n-1)/2 edges where n is the number of nodes in the graph. clique
- Weighted Graph: In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as w(e) which must be a positive (+) value indicating the cost of traversing the edge.
- Digraph: A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.
- Loop: An edge that is associated with the similar end points can be called as Loop.
- Adjacent Nodes:If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.
- Degree of the Node:A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.



$V = \{v_0\ v_1\ v_2\ v_3\ v_4\ v_5\}$

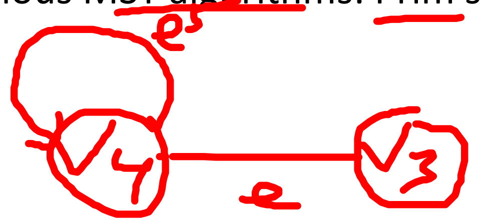
$E = \{e_0\ e_1\ e_2\ \ldots\ e_{10}\}$

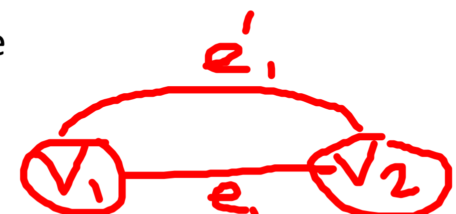
$G = \{V, E\}$

$G' = \{V', E'\}$

$V' \subseteq V \quad E' \subseteq E$

- Subgraph
  - If graph G=(V, E)
  - Then Graph G'=(V',E') is a subgraph of G if V' ⊆ V and E' ⊆ E
- Null Graph: defined as a graph which consists only the isolated vertices.
- Labeled Graphs: A graph G=(V, E) is called a labeled graph if its edges are labeled with some name or data. So, we can write these labels in place of an ordered pair in its edges set.
- Multigraph: If in a graph multiple edges between the same set of vertices are allowed, it is known as Multigraph. In other words, it is a graph having at least one loop or multiple edges.
- Tree: undirected, connected graph with no cycles
- Spanning tree: a spanning tree of G is a connected subgraph of G that is a tree
- Minimum spanning tree (MST): a spanning tree with minimum weight
- Spanning trees and minimum spanning tree are not necessarily unique
- We will look at two famous MST algorithms: Prim's and Kruskal's

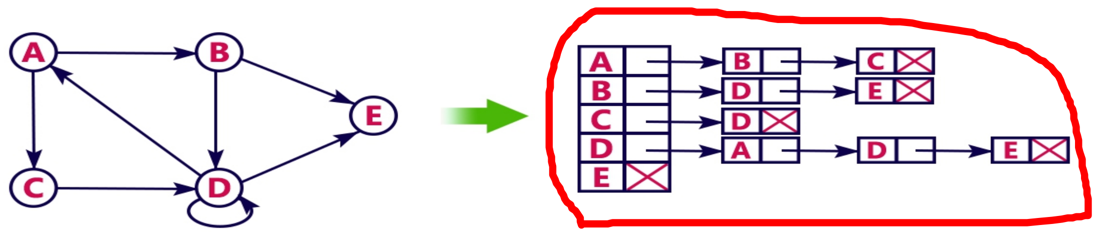




$e_1 = [v_1\ v_2]$

$e'_1 = [v_1\ v_2]$

- Two common data structures for representing graphs:
  1. – Adjacency lists
  2. – Adjacency matrix



- Adjacency List
  - In this representation, every vertex of a graph contains list of its adjacent vertices.
  - For example, consider the following directed graph representation implemented using linked list...

```
struct Graph
{ int V, E;
  int ** adj;
};
```

→ struct Graph *G = (struct Graph *) malloc(sizeof(struct Graph));

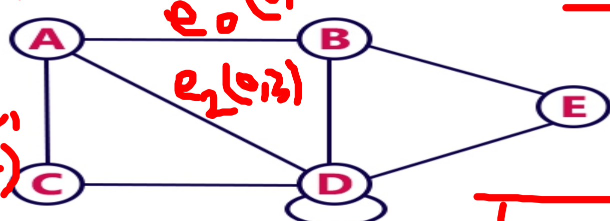
Adjacency Matrix

- In this representation, the graph is represented using a matrix of size total number of vertices by a total number of vertices. That means a graph with 4 vertices is represented using a matrix of size 4X4. In this matrix, both rows and columns represent vertices. This matrix is filled with either 1 or 0. Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.

5 X 5

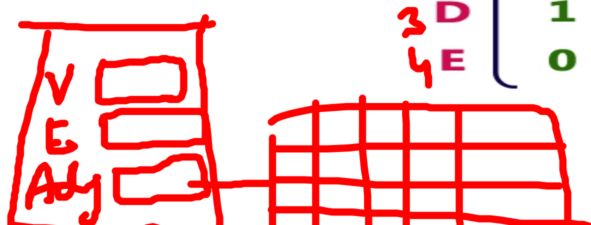
G->adj = malloc(sizeof(int) * (G->V * G->V))

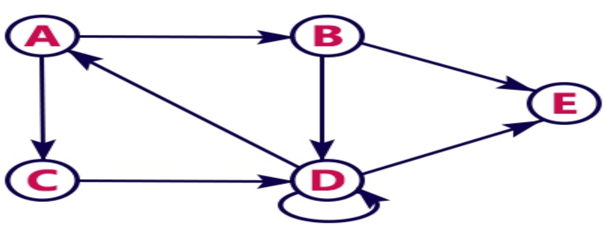
Undirected



	0 A	1 B	2 C	3 D	4 E
0 A	0	1	1	1	0
1 B	1	0	0	1	1
2 C	1	0	0	1	0
3 D	1	1	1	1	1
4 E	0	1	0	1	0

G = {V, E}

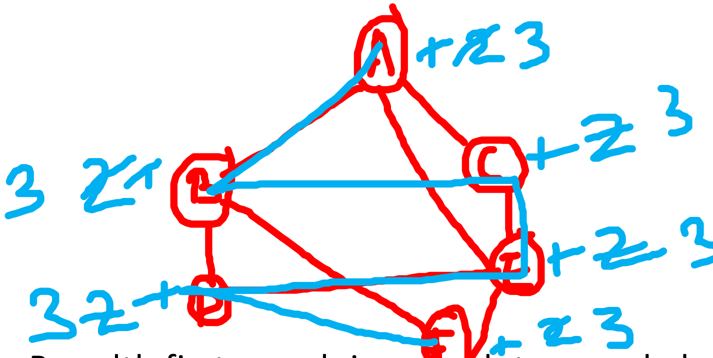




	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	1	1
C	0	0	0	1	0
D	1	0	0	1	1
E	0	0	0	0	0

Graph Traversal Algorithm

- Traversing the graph means examining all the nodes and vertices of the graph. There are two standard methods by using which, we can traverse the graphs.
  - Breadth First Search (BFS) – Queue
  - Depth First Search (DFS) – stack



BFS

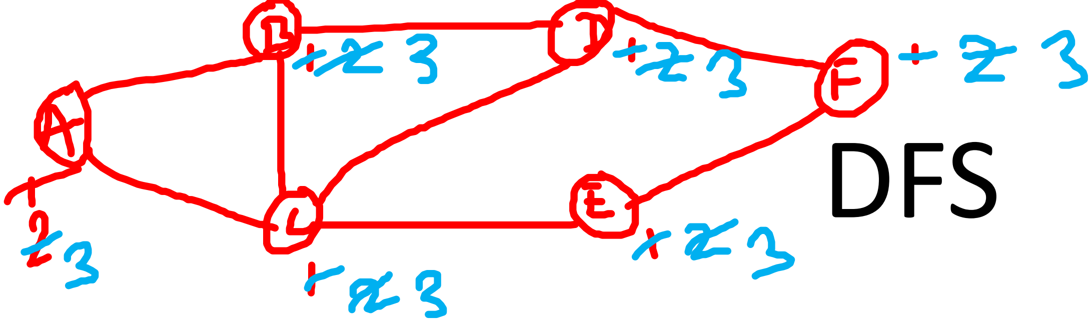
ABCEDF

- Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighbouring nodes. Then, it selects the nearest node and explore all the unexplored nodes. The algorithm follows the same process for each of the nearest node until it finds the goal.
- The algorithm starts with examining the node A and all of its neighbours. In the next step, the neighbours of the nearest node of A are explored and process continues in the further steps. The algorithm explores all neighbours of all the nodes and ensures that each node is visited exactly once and no node is visited twice.

status = 1 2 3

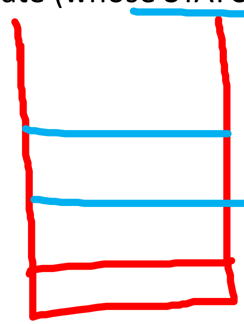
- Algorithm
  - Step 1: SET STATUS = 1 (ready state) for each node in G
  - Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)
  - Step 3: Repeat Steps 4 and 5 until QUEUE is empty
  - Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).
  - Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)
  - [END OF LOOP]
  - Step 6: EXIT

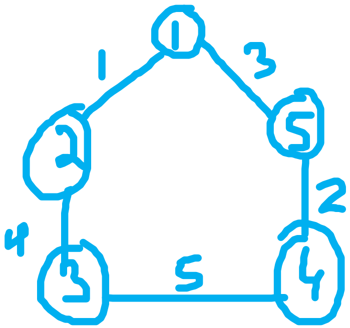
DFS



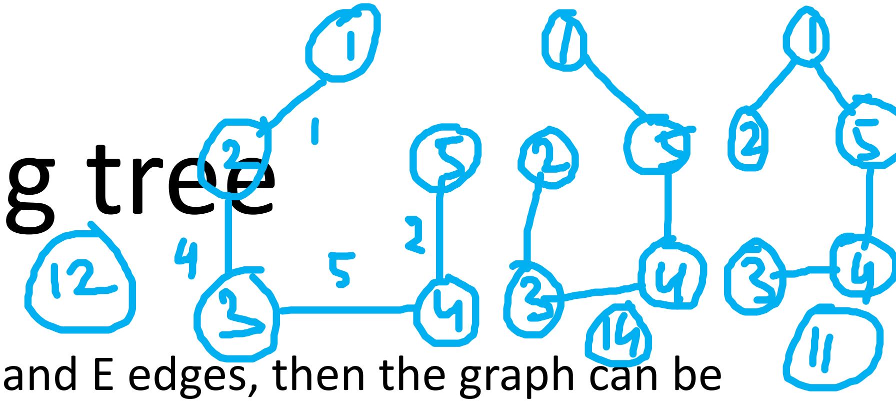
- Depth first search (DFS) algorithm starts with the initial node of the graph G, and then goes to deeper and deeper until we find the goal node or the node which has no children. The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.
- The data structure which is being used in DFS is stack. The process is similar to BFS algorithm. In DFS, the edges that leads to an unvisited node are called discovery edges while the edges that leads to an already visited node are called block edges.
- Algorithm
  - Step 1: SET STATUS = 1 (ready state) for each node in G
  - Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)
  - Step 3: Repeat Steps 4 and 5 until STACK is empty
  - Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)
  - Step 5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their
  - STATUS = 2 (waiting state)
  - [END OF LOOP]
  - Step 6: EXIT

A C E F D B

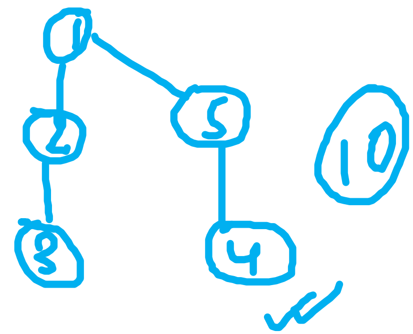




Spanning tree



- If we have a graph containing V vertices and E edges, then the graph can be represented as: G(V, E)
- If we create the spanning tree from the above graph, then the spanning tree would have the same number of vertices as the graph, but the vertices are not equal. The edges in the spanning tree would be equal to the number of edges in the graph minus 1.
- Suppose the spanning tree is represented as:
  - G`(V`, E`)
  - where, V=V` ,      E`ϵ E -1, E`=|V| - 1

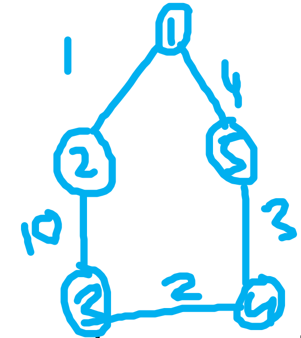


- Minimum Spanning Trees
  - The minimum spanning tree is the tree whose sum of the edge weights is minimum. From the above spanning trees, the total edge weight of the spanning tree 1 is 12, the total edge weight of the spanning tree 2 is 14, and the total edge weight of the spanning tree 3 is 11

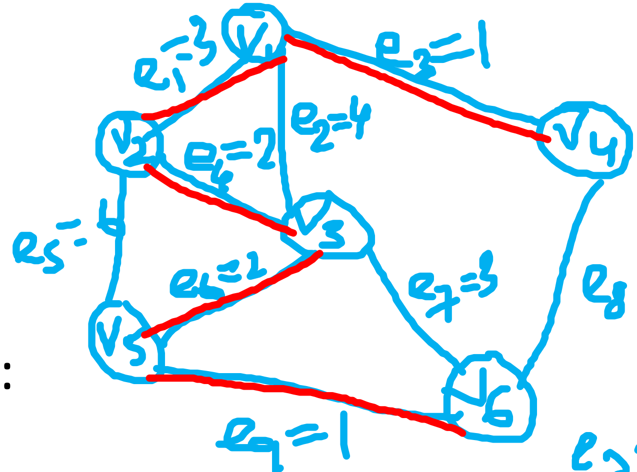
Properties of Spanning tree

- A connected graph can contain more than one spanning tree. The spanning trees which are minimally connected or we can say that the tree which is having a minimum total edge weight would be considered as the minimum spanning tree.
- All the possible spanning trees that can be created from the given graph G would have the same number of vertices, but the number of edges in the spanning tree would be equal to the number of vertices in the given graph minus 1.
- The spanning tree does not contain any cycle.

Kruskal's algorithm



- Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which
  - form a tree that includes every vertex
  - has the minimum sum of weights among all the trees that can be formed from the graph
- We start from the edges with the lowest weight and keep adding edges until we reach our goal. greedy approach
- The steps for implementing Kruskal's algorithm are as follows:
  - Sort all the edges from low weight to high
  - Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.
  - Keep adding edges until we reach all vertices.



$A = \{e_3\ e_9\ e_4\ e_6\ e_1\ \}$

$e_8 = 5$ $E = \{e_3, e_9, e_4, e_6, e_1, e_7, e_2, e_8, e_5\}$

$e_3 = (v_1 \to v_4)$
$e_9 = (v_5 \to v_6)$
$e_4 = (v_2 \to v_3)$
$e_6 = (v_3 \to v_5)$

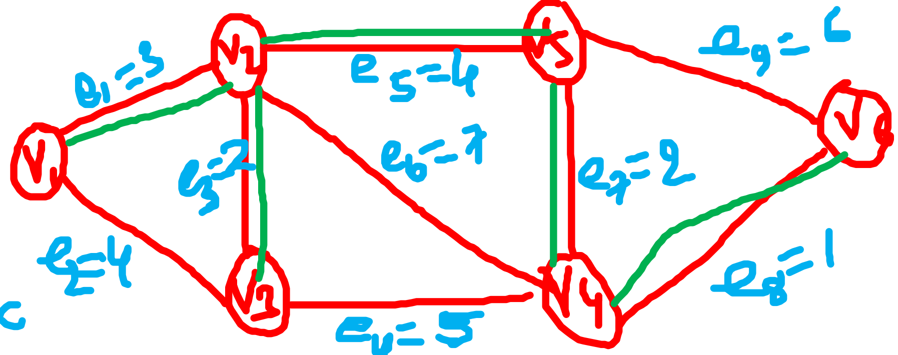
$e_1 = (v_1 \to v_2)$
$e_7 = (v_3 \to v_6)$

- KRUSKAL(G):
- A = ∅
- For each vertex v ∈ G.V: [G]
-     MAKE-SET(v)
- ✓ Sort the edges of E into non-decreasing order by weight w
- For each edge (u, v) ∈ G.E ordered by increasing order by weight(u, v): E[G]
-     if FIND-SET(u) ≠ FIND-SET(v):
-     A = A ∪ {(u, v)}
-     UNION(u, v)
- return A

✓ $S_1 = v_1\ v_4\ v_2\ v_3\ v_5\ v_6$
✓ ~~$S_2 =$~~
~~$S_3 = v_3$~~
✓ ~~$S_4 = v_4$~~
✓ ~~$S_5 = v_5\ v_6$~~
✓ ~~$S_6 = v_6$~~

Prim's algorithm

- Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which
  - form a tree that includes every vertex
  - has the minimum sum of weights among all the trees that can be formed from the graph
- How Prim's algorithm works
- We start from one vertex and keep adding edges with the lowest weight until we reach our goal.
- The steps for implementing Prim's algorithm are as follows:
  - Initialize the minimum spanning tree with a vertex chosen at random.
  - Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
  - Keep repeating step 2 until we get a minimum spanning tree



v
key[v] = ∞
π[v]

- MST-PRIM (G, w, r)
- 1. for each u ∈ V [G]
- 2. do key [u] ← ∞
- 3. π [u] ← NIL
- 4. key [r] ← 0
- 5. Q ← V [G]
- 6. While Q ? ∅
- 7. do u ← EXTRACT - MIN (Q)
- 8. for each v ∈ Adj [u]
- 9. do if v ∈ Q and w (u, v) < key [v]
- 10. then π [v] ← u
- 11. key [v] ← w (u, v)

3

Q	Key	π
~~v1~~	~~∞~~ 0	–
~~v2~~	~~∞~~ 3	– v1
~~v3~~	~~∞~~ ~~4~~ 2	– ~~v1~~ v2
~~v4~~	~~∞~~ ~~7~~ ~~5~~ 2	– ~~v2~~ ~~v3~~ v5
~~v5~~	~~∞~~ 4	– v2
v6	~~∞~~ ~~6~~ 1	– ~~v5~~ v4

$3 < \infty$

$4 < \infty$

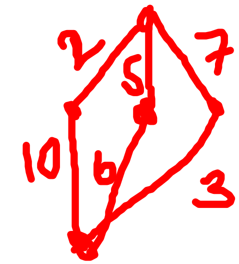
$2 < 4$

$5 < 7$

$2 < 5$

u = v1
V = v2 v3
u = v2
V = v3 v4 v5
u = v3
V = v4
u = v5
V = v4 v6
u = v4
V = v6
u = v6
V = Φ

Dijkstra's Algorithm



- It is a greedy algorithm that solves the single-source shortest path problem for a directed graph G = (V, E) with nonnegative edge weights, i.e., w (u, v) ≥ 0 for each edge (u, v) ∈ E.
- Dijkstra's Algorithm maintains a set S of vertices whose final shortest - path weights from the source s have already been determined. That's for all vertices v ∈ S; we have d [v] = δ (s, v). The algorithm repeatedly selects the vertex u ∈ V - S with the minimum shortest - path estimate, insert u into S and relaxes all edges leaving u.
- Because it always chooses the "lightest" or "closest" vertex in V - S to insert into set S, it is called as the greedy strategy.

$S = \{ \quad \}$

S = { }

Q

Vertex	Dist[V]	π(V)

- Dijkstra's Algorithm (G, w, s)
  - 1. INITIALIZE - SINGLE - SOURCE (G, s)
  - 2. S←∅
  - 3. Q←V [G]
  - 4. while Q ≠ ∅
  - 5. do u ← EXTRACT - MIN (Q)
  - 6. S ← S ∪ {u}
  - 7. for each vertex v ∈ Adj [u]
  - 8. do RELAX (u, v, w)
- RELAX (u, v, w)
  - If d [v] > d [u] + w (u, v)
  - then d [v] ← d [u] + w (u, v)
  - π [v] ← u

$u = C$
$v = B, D, E$
$R(C, B, 3)$ $10 > 5+3$
$R(C, D, 9)$ $\infty > 5+9$
$R(C, E, 2)$ $\infty > 5+2$

$S = \{A\ C\ E\ B\ D\}$
$u = A$
$V = B, C$
$R(A, B, 10)$
$d(v) > d(u) + 10$
$\infty > 0 + 10$
$d(v) = 10$
$\pi(v) = A$
$R(A, C, 5)$
$\infty > 0 + 5$

Step1: Q =[s, t, x, y, z]

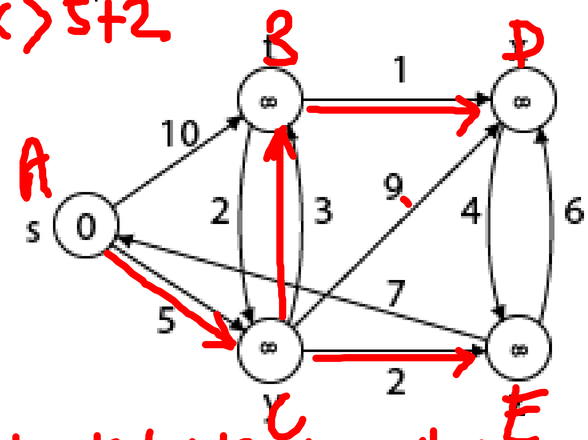
We scanned vertices one by one and find out its adjacent. Calculate the distance of each adjacent to the source vertices.

We make a stack, which contains those vertices which are selected after computation of shortest distance.

Firstly we take's' in stack M (which is a source)

M = [S]   Q = [t, x, y, z]

Step 2: Now find the adjacent of s that are t and y.



Adj [s] → t, y   [Here s is u and t and y are v]

Case - (i) s → t

$d[v] > d[u] + w[u, v]$

$d[t] > d[s] + w[s, t]$

$\infty > 0 + 10$ [false condition]

Then $d[t] \leftarrow 10$

$\pi[t] \leftarrow 5$

Adj [s] ← t, y

$u = E$
$V = A, D$
$R(E, A, 7)$
$0 > 7+7$
$R(E, D, 6)$
$14 > 7+6$

Q	Vertex	D(v)	Prev vertex gave D(v) → π(v)
X	A	0	-
X	B	~~∞~~ ~~10~~ 8	- ~~A~~ C
X	C	~~∞~~ 5	- A
X	D	~~∞~~ ~~14~~ ~~13~~ 9	- ~~C~~ ~~E~~ B
X	E	~~∞~~ 7	- C

Case - (ii) s→ y

$d[v] > d[u] + w[u, v]$

$d[y] > d[s] + w[s, y]$

$\infty > 0 + 5$ [false condition]

$\infty > 5$

Then $d[y] \leftarrow 5$

$\pi[y] \leftarrow 5$

$u = B$
$V = C, D$
$R(B, C, 2)$ $5 > 8+2$ X
$R(B, D, 1)$ $13 > 8+1$

$u = D$
$V = E$
$R(D, E, 4)$
$7 > 9 + 4$

By comparing case (i) and case (ii)

Adj [s] → t = 10, y = 5

y is shortest

y is assigned in 5 = [s, y]

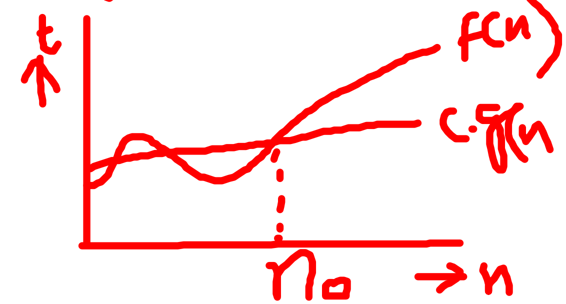
Time Complexity & Space Complexity:-

Asymptotic Notation:

i) Algorithm's efficiency

ii) compare diff. Algo

$f(n) = 10n^2 + 7$



1. Big oh (O) $2n^2 \leq 2n^2+n \leq 3n^2$

f(n) & g(n)

n_0 & $c>0$ $\forall$ integers $n > n_0$

worst case

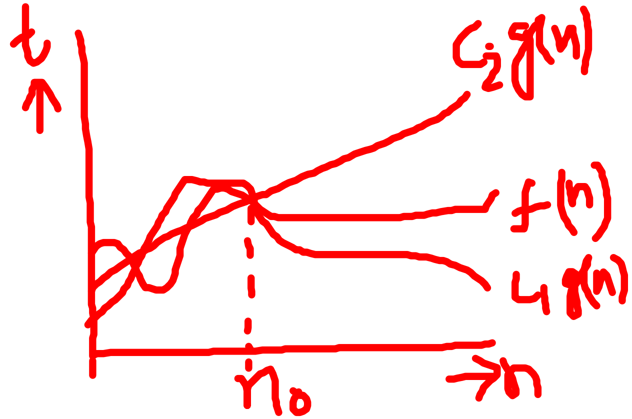
$$f(n) \leq c \cdot g(n)$$

$f(n) \in O\,g(n)$ $f(n) = 2n^2+n$

$f(n) \in \Omega\,g(n)$ $2n^2+n \leq c \cdot g(n^2)$

$2n^2+n \leq 3n^2$

$n \leq n^2$ $1 \leq n$ $n \geq 1$



c_1 & $c_2 > 0$, $n > n_0$

$c_1 g(n) \leq f(n) \leq c_2 g(n)$

$f(n) \in \theta\,g(n)$

$c = 2$

$f(n) \geq c \cdot g(n)$

$2n^2+n \geq 2n^2$