INTRODUCTION TO DATA STRUCTURES:

A Data typerefers to a named group of data which share similar properties or characteristics and which have common behavior among them. Three fundamental data types used in C programming are int for integer values, float for floating-point numbers and char for character values.

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.

The identification of the inbuilt data structure is very important in nature. And the structure of input and output data can be use to derive the structure of a program. Data structures affects the design of both structural and functional aspects of a program.

Algorithm + Data structure = Program

We know that an algorithm is a step-by-step procedure to solve a particular function i.e., it is a set of instructions written to carry out certain tasks and the data structure is the way of organizing the data with their logical relationship maintained. To develop a program of an algorithm, we should select an appropriate data structure for that algorithm. Therefore, algorithm and its associated data structures form a program.

CLASSIFICATION OF DATA STRUCTURE

Data structures are normally divided into two broad categories.

(i) Primitive Data Structures(built-in)

(ii) Non-Primitive Data Structures(user defined)

PRIMITIVE DATA STRUCTURES (BUILT-IN) :

These are basic structures and are directly operated upon by the machine instructions. These, in general, have different representations on different computers. Integers, floating-point numbers, character constants, string constants, pointers etc. fall in this category.

NON-PRIMITIVE DATA STRUCTURES (USER-DEFINED):

These are more complicated 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. Arrays, structures, lists and files are examples.
The data appearing in our data structures are processed by means of certain

operations. In fact, the particular data structure that one chooses for a given situation depends largely on the frequency with which specific operations are performed.

OPERATIONS OF DATA STRUCTURES:

The basic operations that are performed on data structures are as follows :

1. Traversing : Accessing each record exactly once so that certain items in the record may be processed. (This accessing and processing is sometimes called "visiting" the record).

2. Searching : Searching operation finds the presence of the desired data item in the list of data item. It may also find the locations of all elements that satisfy certain conditions.

3. Inserting : Inserting means addition of a new data element in a data structure.

4. Deleting : Deleting means removal of a data element from a data structure.
Sometimes, two or more of the operations may be used in a given situation. For e.g, we may want to delete a data element from a data structure, which may mean we first need to search for the location of the record.
The following two operations, which are used in special situations, will also be considered :

(1) Sorting : Sorting is the process of arranging all data items in a data structure in a particular order say for example, either in ascending order or in descending order.

(2) Merging : Combining the records of two different sorted files into a single sorted file.

DESCRIPTION OF VARIOUS DATA STRUCTURES:

1. ARRAYS

An array is defined as a set of finite number of homogeneous elements or data items. It means an array can contain one type of data only, either all integers, all floating□point numbers, or all characters. Declaration of arrays is as follows :
int A[10];
where int specifies the data type or type of elements array stores. "A" is the name of the array, and the number specified inside the square brackets is the number of elements an array can store, this is also called size or length of array.
Some important concepts of arrays are :
(1) The individual element of an array can be accessed by specifying name of the array, followed by index or subscript (which is an integer number specifying the location of element in the array) inside square brackets. For example to access the fifth element of array a, we have to give the following statement :
A[4];
(2) The first element of the array has index zero [0]. It means the first element and last

element of the above array will be specified as :
A[0], and A[9] respectively.
(3) The elements of array will always be stored in consecutive memory locations.
(4) The number of elements that can be stored in an array i.e., the size of array or its
length is given by the following equation :
(upperbound – lowerbound) + 1

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.

LINKED LISTS:

A linked list is a linear collection of data elements, called node pointing to the next nodes by means of pointers. Each node is divided into two parts :

the first part containing the information of the element, and the second part called the link or next pointer containing the address of the next node in the list. Technically, each such element is referred to as a node, therefore a list can be defined.

STACKS
A stack is a non-primitive linear data structure. It is an ordered list in which addition
of new data item and deletion of already existing data item is done from only one end,
known as Top of Stack (TOS). As all the deletion and insertion in a stack is done from top
of the stack, the last added element will be the first to be removed from the stack. Due to
this reason, the stack is also called Last-In-First-Out (LIFO) type of list.

A common model of a stack is plates in a marriage party. Fresh plates are “pushed”
onto the top and “popped” off the top.

Some of you may eat biscuits. If you assume only one side of the cover is torn and
biscuits are taken off one by one. This is called popping and similarly, if you want to
preserve some biscuits for some time later, you will put them back into the pack
through the same torn end called pushing.

QUEUES:

Queue is a linear data structure that permits insertion of an element at one end and
deletion of an element at the other end. The end at which the deletion of an element take place is
called front, and the end at which insertion of a new element can take place is called rear. The
deletion or insertion of elements can take place only at the front and rear end of the list
respectively.
The first element that gets added into the queue is the first one to get removed from the

list. Hence, Queue is also referred to as First-In-First-Out (FIFO) list. The name ‘Queue’ comes
from the everyday use of the term. Consider a railway reservation booth, at which we have to get
into the reservation queue. New customers got into the queue from the rear end, whereas the
customers who get their seats reserved leave the queue from the front end. It means the
customers are serviced in the order in which they arrive the service center (i.e. first come first
serve type of service). The same characteristics apply to our Queue. Fig. (3) shows the pictorial
representation of a Queue.

TREES:

A Tree can be defined as a finite set of data items (nodes). Tree is a non-linear type of data structure in which data items are arranged of stored in a sorted sequence. Trees represent the hierarchical relationship between various elements. In trees : 1.There is a special data item at the top of hierarchy called the Root of the Tree.2.The remaining data items are partitioned into number of mutually exclusive (i.e. disjoint)subsets, each of which is itself, a tree, which is called the subtree.3.The tree always grows in length towards bottom in data structures, unlike natural trees whichgrows upwards.

GRAPHS:

Data sometimes contain a relationship between pairs of elements which is not necessarily hierarchical in nature. Geometrical arrangement are very important while working with real life projects.

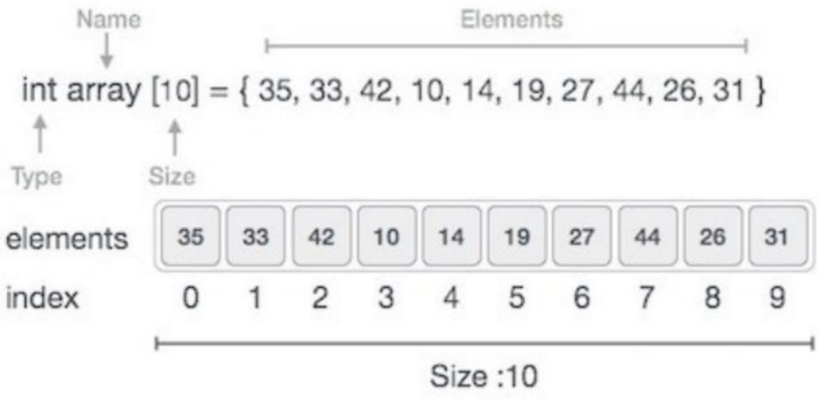
We have represented vertices by dots (which are called vertex or node) and bridges by lines which are called edges. This type of drawing is called graph. Hence a graph can be defined as a ordered set (V,E), where V(G) represents the set of all elements called vertices and E(G) represents the edges between these vertices.

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

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

Array Representation

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



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

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.

Basic Operations

Following are the basic operations supported by an array.

- **Traverse** – print all the array elements one by one.

- **Insertion** − Adds an element at the given index.
- **Deletion** − Deletes an element at the given index.
- **Search** − Searches an element using the given index or by the value.
- **Update** − Updates an element at the given index.

Insertion Operation

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

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

Algorithm

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

Example

Result

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

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

Deletion Operation

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

Algorithm

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

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

Search Operation

You can perform a search for an array element based on its value or its index.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to find an element with a value of ITEM using sequential search.

```
1. Start
2. Set J = 0
3. Repeat steps 4 and 5 while J < N
4. IF LA[J] is equal ITEM THEN GOTO STEP 6
5. Set J = J +1
6. PRINT J, ITEM
7. Stop
```

Update Operation

Update operation refers to updating an existing element from the array at a given index.

Algorithm

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

```
1. Start
2. Set LA[K-1] = ITEM
3. Stop
```

A **data structure** is classified into two categories: **Linear** and Non-**Linear data structures**. A **data structure** is said to be **linear** if the elements form a sequence, for example Array, Linked list, queue etc. Elements in a nonlinear **data structure** do not form a sequence, for example Tree, Hash tree, Binary tree, etc.

Linear data structures

Linear data structures organize their data elements in a linear fashion, where data elements are attached one after the other. Data elements in a liner data structure are traversed one after the other and only one element can be directly reached while traversing. Linear data structures are very easy to implement, since the memory of the computer is also organized in a linear fashion. Some commonly used linear data structures are arrays, linked lists, stacks and queues. An arrays is a collection of data elements where each element could be identified using an index. A linked list is a sequence of nodes, where each node is made up of a data element and a reference to the next node in the sequence. A stack is actually a list where data elements can only be added or removed from the top of the list. A queue is also a list, where data elements can be added from one end of the list and removed from the other end of the list.

Nonlinear data structures

In nonlinear data structures, data elements are not organized in a sequential fashion. A data item in a nonlinear data structure could be attached to several other data elements to reflect a special relationship among them and all the data items cannot be traversed in a single run. Data structures like multidimensional arrays, trees and graphs are some examples of widely used nonlinear data structures. A multidimensional array is simply a collection of one-dimensional arrays. A tree is a data structure that is made up of a set of linked nodes, which can be used to represent a hierarchical relationship among data elements. A graph is a data structure that is made up of a finite set of edges and vertices. Edges represent connections or relationships among vertices that stores data elements.

Difference between Linear and Nonlinear Data Structures

Main difference between linear and nonlinear data structures lie in the way they organize data elements. In linear data structures, data elements are organized sequentially and therefore they are easy to implement in the computer's memory. In nonlinear data structures, a data element can be attached to several other data elements to represent specific relationships that exist among them. Due to this nonlinear structure, they might be difficult to be implemented in computer's linear memory compared to implementing linear data structures. Selecting one data structure type over the other should be done carefully by considering the relationship among the data elements that needs to be stored.

Linked List

A linked list is a data structure that consists of a sequence of data records such that in each record there is a field that contains a reference (i.e., a link) to the next record in the sequence.

A linked list whose nodes contain two fields: an integer value and a link to the next node

Types of linked list:

Singly – link list
Doubly – link list
Circular linked list
Circular doubly linked list

TRAVERSING A LINKED LIST:

Algorithm:

```
Set PTR := START. [Initializes pointer PTR.]
Repeat Steps 3 and 4 while PTR ≠ NULL.
        Apply PROCESS to INFO[PTR].
        Set PTR := LINK[PTR]. [ PTR now points to                                    the
next node.]
    [End of Step 2 loop.]
Exit.
```

SEARCHING A LINKED LIST(List is unsorted:

Algorithm: SEARCH(INFO, LINK, START, ITEM, LOC)

```
Set PTR := START.
Repeat Step 3 while PTR ≠ NULL.
     If ITEM = INFO[PTR], then:
                    Set LOC := PTR.
               Else:
                    Set PTR := LINK[PTR].
               [End of If structure.]
        [End of Step 2 loop.]
4. [Search is unsuccessful.] Set LOC := NULL.
5. Exit.
```

Algorithm: SRCHSL(INFO, LINK, START, ITEM, LOC)

```
Set PTR := START.
Repeat Step 3 while PTR ≠ NULL.
```

```
        If ITEM < INFO[PTR], then:
                Set PTR := LINK[PTR].
            Else if ITEM = INFO[PTR]
                Set LOC := PTR and Exit.
            Else:
                Set LOC := NULL and Exit.
            [End of If structure.]
    [End of Step 2 loop.]
4. [Search is unsuccessful.] Set LOC := NULL.
5. Exit.
```

Stack: Linked List Implementation

Push and pop at the head of the list
New nodes should be inserted at the front of the list, so that they become the top of the stack
Nodes are removed from the front (top) of the list.

MEMORY ALLOCATION; GARBAGE COLLECTION
FREE POOL
It is a list which consist of unused memory cells, this list has its own pointer called AVAIL.
OVERFLOW
Overflow will occur when AVAIL=NULL and there is an insertion.
UNDERFLOW
It refers to the situation where one wants to delete data from a data structure that is empty. i.e. START = NULL.

Insertion in a linked list:

1.Checking to see if space is available in the AVAIL list. If not that is AVAIL=NULL then the algorithm will print the message OVERFLOW.
2.Removing the first node from the AVAIL list. Using the variable NEW to keep the track of location of new node, this step can be implemented by the pair of assignments :

```
  NEW:=AVAIL
  AVAIL=LINK[AVAIL]
```

3. Copying information into the new node.

```
 INFO[NEW]:=ITEM
```

Insertion at the beginning:

```
INSFIRST(INFO,LINK,START,AVAIL,ITEM)
[OVERFLOW?.]
  If AVAIL=NULL, then Write: OVERFLOW and exit.
[Remove the first node from the AVAIL list.]
  Set NEW:=AVAIL and AVAIL:=LINK[AVAIL].
Set INFO[NEW]:=ITEM. [Copies new data into new node.]
 Set LINK[NEW]:=START. [New node now points to original first node.]
Set START:=NEW. [Changes START so that it points to the new node.]
Exit.
```

Insertion after a given node:

INSLOC (INFO, LINK, START, AVAIL, LOC, ITEM)

```
1. [OVERFLOW?.]
   If AVAIL=NULL, then : Write: OVERFLOW and exit.
2. [Remove the first node from the AVAIL list.]
   Set NEW:=AVAIL
   AVAIL:=LINK[AVAIL].
3. Set INFO[NEW]:=ITEM . [Copies new data into new node.]
4. If LOC=NULL, then: [Insert as first node]
   Set LINK[NEW]:=START and START:=NEW.
   Else: [Insert after node with location LOC]
        Set LINK[NEW]:=LINK[LOC] and LINK[LOC]:=NEW
   [End of If structure.]
5. Exit
```

Insertion into a sorted linked list:

FINDA (INFO, LINK, START, ITEM, LOC)

```
[List empty?] If START = NULL, then: Set LOC := NULL, and Return.
If ITEM < INFO[START], then: Set LOC := NULL, and Return.
Set SAVE := START and PTR := LINK[START].
Repeat Steps 5 and 6 while PTR ≠ NULL.
    If ITEM < INFO[PTR], then:
                        Set  LOC := SAVE and Return.
            [End Of If Structure.]
6.          Set SAVE := PTR and PTR := LINK[PTR].
        [End Of Step 4 Loop.]
7.      Set LOC := SAVE.
8.      Return.
```

Insertion into a sorted linked list:

INSSRT (INFO, LINK, START, AVAIL, ITEM)
Call FINDA(INFO, LINK, START, ITEM, LOC).

```
Call INSLOC(INFO, LINK, START, AVAIL, LOC, ITEM).
Exit.
```

```
1. . [OVERFLOW?.]
   If AVAIL=NULL, then Write: OVERFLOW and exit.
2. [Remove the first node from the AVAIL list.]
   Set NEW:=AVAIL and AVAIL:=LINK[AVAIL].
3. Set INFO[NEW]:=ITEM . [Copies new data into new node.]
   LINK[NEW]:=NULL
4. [Is the list empty?]
  If START:=NULL, then Return(NEW)
5. [Initiate search for the last node]
  SAVE:=START
6. [Search for end of list]
  Repeat while LINK[SAVE]!=NULL
  SAVE:=LINK[SAVE]
8. [Set LINK field of last node to NEW]
   LINK[SAVE]:=NEW
9. [Return first node pointer]
  Return(START)
```

STACK:

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

Basic Operations:

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

push() − Pushing (storing) an element on the stack.

pop() − Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

peek() − get the top data element of the stack, without removing it.

isFull() − check if stack is full.

isEmpty() − check if stack is empty.

Push: Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.

Pop: Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.

Peek or Top: Returns top element of stack.
isEmpty: Returns true if stack is full.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows −

```
begin procedure push: stack, data

  if stack is full
    return null
  endif

  top ← top + 1
  stack[top] ← data
```

```
end procedure
```

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead top is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps −

Step 1 − Checks if the stack is empty.

Step 2 − If the stack is empty, produces an error and exit.

Step 3 − If the stack is not empty, accesses the data element at which top is pointing.

Step 4 − Decreases the value of top by 1.

Step 5 − Returns success.

Stack Pop Operation

Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows −

```
begin procedure pop: stack

  if stack is empty
    return null
  endif

  data ← stack[top]
  top ← top - 1
  return data

end procedure
```

Arithmetic Expression:

The way to write arithmetic expression is known as a notation. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are −

Infix Notation
Prefix (Polish) Notation
Postfix (Reverse-Polish) Notation

These notations are named as how they use operator in expression. We shall learn the same here in this chapter.

Infix Notation

We write expression in infix notation, e.g. a - b + c, where operators are used in-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

Prefix Notation

In this notation, operator is prefixed to operands, i.e. operator is written ahead of operands. For example, +ab. This is equivalent to its infix notation a + b. Prefix notation is also known as Polish Notation.

Postfix Notation

This notation style is known as Reversed Polish Notation. In this notation style, the operator is postfixed to the operands i.e., the operator is written after the operands. For example, ab+. This is equivalent to its infix notation a + b.

Postfix Evaluation Algorithm:

We shall now look at the algorithm on how to evaluate postfix notation −

Step 1 − scan the expression from left to right
Step 2 − if it is an operand push it to stack
Step 3 − if it is an operator pull operand from stack and perform operation
Step 4 − store the output of step 3, back to stack
Step 5 − scan the expression until all operands are consumed
Step 6 − pop the stack and perform operation

Queue:

Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

Queue Example:

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Queue Representation:

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −

Queue Example:

As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations:

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

enqueue() − add (store) an item to the queue.

dequeue() − remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are −

peek() − Gets the element at the front of the queue without removing it.

isfull() − Checks if the queue is full.

isempty() − Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by front pointer and while enqueing (or storing) data in the queue we take help of rear pointer.

Let's first learn about supportive functions of a queue −

peek():
This function helps to see the data at the front of the queue. The algorithm of peek() function is as follows −

Algorithm:

```
begin procedure peek
  return queue[front]
end procedure
```

Implementation of peek() function in C programming language −

Example:

```
int peek() {
  return queue[front];
}
```

isfull():

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function −

Algorithm:

```
begin procedure isfull

  if rear equals to MAXSIZE
    return true
  else
    return false
  endif

end procedure
```

Implementation of isfull() function in C programming language −

Example:

```
bool isfull() {
  if(rear == MAXSIZE - 1)
    return true;
  else
    return false;
}
```

isempty()

Algorithm of isempty() function −

Algorithm

```
begin procedure isempty

  if front is less than MIN  OR front is greater than rear
    return true
  else
    return false
  endif

end procedure
```

If the value of front is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Enqueue Operation

Queues maintain two data pointers, front and rear. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

Step 1 − Check if the queue is full.

Step 2 − If the queue is full, produce overflow error and exit.

Step 3 − If the queue is not full, increment rear pointer to point the next empty space.

Step 4 − Add data element to the queue location, where the rear is pointing.

Step 5 − return success.

Dequeue Operation:

Accessing data from the queue is a process of two tasks − access the data where front is pointing and remove the data after access. The following steps are taken to perform dequeue operation −

Step 1 − Check if the queue is empty.

Step 2 − If the queue is empty, produce underflow error and exit.

Step 3 − If the queue is not empty, access the data where front is pointing.

Step 4 − Increment front pointer to point to the next available data element.

Step 5 − Return success.

Trees:

Tree represents the nodes connected by edges. We will discuss binary tree or binary search tree specifically.

Binary Tree is a special datastructure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.

Binary Tree:

Important Terms:

Following are the important terms with respect to tree.

Path − Path refers to the sequence of nodes along the edges of a tree.

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.

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.

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.

Binary Search Tree Representation:

Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.

Binary Search Tree:

We're going to implement tree using node object and connecting them through references.

Tree Node:

The code to write a tree node would be similar to what is given below. It has a data part and references to its left and right child nodes.

In a tree, all nodes share common construct.

BST Basic Operations:

The basic operations that can be performed on a binary search tree data structure, are the following −

Insert − Inserts an element in a tree/create a tree.

Search − Searches an element in a tree.

Preorder Traversal − Traverses a tree in a pre-order manner.

Inorder Traversal − Traverses a tree in an in-order manner.

Postorder Traversal − Traverses a tree in a post-order manner.

We shall learn creating (inserting into) a tree structure and searching a data item in a tree in this chapter. We shall learn about tree traversing methods in the coming chapter.

Insert Operation:

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm:

```
If root is NULL
  then create root node
return

If root exists then
  compare the data with node.data

  while until insertion position is located

    If data is greater than node.data
      goto right subtree
    else
```

```
      goto left subtree

   endwhile

   insert data

end If
```

Implementation

The implementation of insert function should look like this −

```
void insert(int data) {
   struct node *tempNode = (struct node*) malloc(sizeof(struct node));
   struct node *current;
   struct node *parent;

   tempNode->data = data;
   tempNode->leftChild = NULL;
   tempNode->rightChild = NULL;

   //if tree is empty, create root node
   if(root == NULL) {
      root = tempNode;
   } else {
      current = root;
      parent  = NULL;

      while(1) {
         parent = current;

         //go to left of the tree
         if(data < parent->data) {
            current = current->leftChild;

            //insert to the left
            if(current == NULL) {
               parent->leftChild = tempNode;
               return;
            }
         }

         //go to right of the tree
         else {
            current = current->rightChild;

            //insert to the right
            if(current == NULL) {
               parent->rightChild = tempNode;
               return;
            }
         }
      }
   }
}
```

Search Operation:

Whenever an element is to be searched, start searching from the root node, then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm:

```
If root.data is equal to search.data
   return root
else
   while data not found

      If data is greater than node.data
         goto right subtree
      else
         goto left subtree

      If data found
         return node
   endwhile

   return data not found

end if
```

Traversal:

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

1.In-order Traversal
2.Pre-order Traversal
3.Post-order Traversal
Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal:

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed in-order, the output will produce sorted key values in an ascending order.

In Order Traversal:

We start from A, and following in-order traversal, we move to its left subtree B. B is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be −

D → B → E → A → F → C → G

Algorithm:

Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
Step 3 − Recursively traverse right subtree.
Pre-order Traversal
In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.

Pre Order Traversal:

We start from A, and following pre-order traversal, we first visit A itself and then move to its left subtree B. B is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

A → B → D → E → C → F → G

Algorithm:

Until all nodes are traversed −
Step 1 − Visit root node.
Step 2 − Recursively traverse left subtree.
Step 3 − Recursively traverse right subtree.
Post-order Traversal
In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.

Post Order Traversal:

We start from A, and following Post-order traversal, we first visit the left subtree B. B is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

D → E → B → F → G → C → A

Algorithm:

Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.

Binary search tree:

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties −

The value of the key of the left sub-tree is less than the value of its parent (root) node's key.

The value of the key of the right sub-tree is greater than or equal to the value of its parent (root) node's key.

Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −

left_subtree (keys) < node (key) ≤ right_subtree (keys)
Representation

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST −

Binary Search Tree
We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

Basic Operations:

Following are the basic operations of a tree

Search − Searches an element in a tree.

Insert − Inserts an element in a tree.

Pre-order Traversal − Traverses a tree in a pre-order manner.

In-order Traversal − Traverses a tree in an in-order manner.

Post-order Traversal − Traverses a tree in a post-order manner.

Node
Define a node having some data, references to its left and right child nodes.

```
struct node {
  int data;
  struct node *leftChild;
  struct node *rightChild;
};
```

Search Operation
Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

Algorithm:

```
struct node* search(int data){
  struct node *current = root;
  printf("Visiting elements: ");

  while(current->data != data){

    if(current != NULL) {
      printf("%d ",current->data);

      //go to left tree
      if(current->data > data){
        current = current->leftChild;
      }  //else go to right tree
      else {
        current = current->rightChild;
      }

      //not found
      if(current == NULL){
        return NULL;
      }
    }
  }

  return current;
}
```

Insert Operation:

Whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Algorithm:

```
void insert(int data) {
  struct node *tempNode = (struct node*) malloc(sizeof(struct node));
  struct node *current;
  struct node *parent;

  tempNode->data = data;
  tempNode->leftChild = NULL;
  tempNode->rightChild = NULL;

  //if tree is empty
  if(root == NULL) {
    root = tempNode;
  } else {
    current = root;
    parent = NULL;

    while(1) {
      parent = current;

      //go to left of the tree
      if(data < parent->data) {
        current = current->leftChild;
        //insert to the left

        if(current == NULL) {
          parent->leftChild = tempNode;
          return;
        }
      }  //go to right of the tree
      else {
        current = current->rightChild;

        //insert to the right
        if(current == NULL) {
          parent->rightChild = tempNode;
          return;
        }
      }
    }
  }
}
```

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}

Graph Data Structure:

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms −

Vertex :
Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.

Edge :
Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.

Adjacency :
Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.

Path :
Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.

Graph Basic Operations:

Following are basic primary operations of a Graph −

Add Vertex − Adds a vertex to the graph.

Add Edge − Adds an edge between the two vertices of the graph.

Display Vertex − Displays a vertex of the graph.

Graph Representation:

By Graph representation, we simply mean the technique which is to be used in order to store some graph into the computer's memory.

There are two ways to store Graph into the computer's memory. In this part of this tutorial, we discuss each one of them in detail.

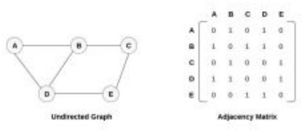
1. Sequential Representation:

In sequential representation, we use adjacency matrix to store the mapping represented by vertices and edges. In adjacency matrix, the rows and columns are represented by the graph vertices. A graph having n vertices, will have a dimension n x n.

An entry Mij in the adjacency matrix representation of an undirected graph G will be 1 if there exists an edge between Vi and Vj.

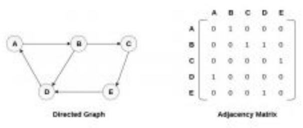
An undirected graph and its adjacency matrix representation is shown in the following figure.

Graph Representation
in the above figure, we can see the mapping among the vertices (A, B, C, D, E) is represented by using the adjacency matrix which is also shown in the figure.



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 1 & 0 \\ B & 1 & 0 & 1 & 1 & 0 \\ C & 0 & 1 & 0 & 0 & 1 \\ D & 1 & 1 & 0 & 0 & 1 \\ E & 0 & 0 & 1 & 1 & 0 \end{array}$$

in the above figure, we can see the mapping among the vertices (A, B, C, D, E) is represented by using the adjacency matrix which is also shown in the figure.



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 0 & 0 & 0 \\ B & 0 & 0 & 1 & 1 & 0 \\ C & 0 & 0 & 0 & 0 & 1 \\ D & 1 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 1 & 0 \end{array}$$

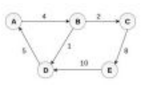
There exists different adjacency matrices for the directed and undirected graph. In directed graph, an entry Aij will be 1 only when there is an edge directed from Vi to Vj.

A directed graph and its adjacency matrix representation is shown in the following figure.

Graph Representation:

Representation of weighted directed graph is different. Instead of filling the entry by 1, the Non- zero entries of the adjacency matrix are represented by the weight of respective edges

The weighted directed graph along with the adjacency matrix representation is shown in the following figure.



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 4 & 0 & 0 & 0 \\ B & 0 & 0 & 2 & 1 & 0 \\ C & 0 & 0 & 0 & 0 & 8 \\ D & 5 & 0 & 0 & 0 & 0 \\ E & 0 & 0 & 0 & 10 & 0 \end{array}$$

Weighted Directed Graph

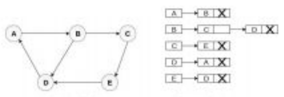
Adjacency Matrix

In the linked representation, an adjacency list is used to store the Graph into the computer's memory.

Consider the undirected graph shown in the following figure and check the adjacency list representation.

Graph Representation

An adjacency list is maintained for each node present in the graph which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed then store the NULL in the pointer field of last node of the list. The sum of the lengths of adjacency lists is equal to the twice of the number of edges present in an undirected graph.



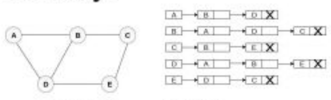
Directed Graph

Adjacency List

Consider the directed graph shown in the following figure and check the adjacency list representation of the graph.

Linked Representation

In the linked representation, an adjacency list is used to store the Graph into the computer's memory.



Undirected Graph

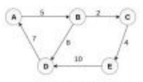
Adjacency List

Consider the undirected graph shown in the following figure and check the adjacency list representation.

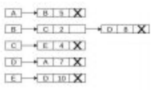
Graph Representation

An adjacency list is maintained for each node present in the graph which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed then store the NULL in the pointer field of last node of the list. The sum of the lengths of adjacency lists is equal to the twice of the number of edges present in an undirected graph.

Consider the directed graph shown in the following figure and check the adjacency list representation of the graph.



Weighted Directed Graph



Adjacency List