|  |  |
| --- | --- |
| |  | | --- | | **CMR ENGINEERING COLLEGE** | |
| **(**Approved by AICTE, Affiliated to JNTU, Hyderabad**)** |
| KANDLAKOYA (V), MEDCHAL ROAD, HYDERABAD-501401. |
| Ph: 08418 200037, 92470 22662, Fax: 08418 200240, www.cmrec.org. |
| **Department of Information Technology** |

**Step Material**

**Subject : Data Structures through c++**

**SHORT ANSWER QUESTIONS**

**UNIT I**

**1. Difference between Class and structure?**

Class is the ADT where as structure is udt. Class needs access specifier such as private, public & private where as structure members can be accessed by public by default & don’t need any accessfiers. Class is oops where structure is borrowed from traditional structured [pop] concept.

**2. What is abstract Class?**

An abstract class is a class that is designed to be specifically used as a base class. An abstract class contains at least one pure virtual function. You declare a pure virtual function by using a pure specifier [= 0] in the declaration of a virtual member function in the class declaration.

**3. List out the advantages of new operator over malloc[]?**

It automatically computes the size of the data object. It automatically returns the correct pointer type It is possible to initialize the objects while creating\_ the memory space. It can be overloaded.

**4. What is the difference between local variable and data member?**

A data member belongs to an object of a class whereas local variable belongs to its current scope. A local variable is declared within the body of a function and can be used only from the point at which it is declared to the immediately following closing brace. A data member is declared in a class definition, but not in the body of any of the class member functions. Data members are accessible to all member function of the class.

**5. What is the function parameter? Difference between parameter and Argument?**

function parameter is a variable declared in the prototype or declaration of a function: void foo[int x]; // prototype -- x is a parameter void foo[int x] // declaration -- x is a parameter { } An argument is the value that is passed to the function in place of a parameter

**6. Define Polymorphism?**

Polymorphism is another important oops concept. Polymorphism means the ability to take more than one form. For example, an operation may exhibit different behavior in different instances. Behavior depends upon the types of data used in the operation

**7. What is function Prototype?**

A function prototype or function interface in C, Perl, PHP or C++ is a declaration of a function that omits the function body but does specify the function's return type, name and argument types. While a function definition specifies what a function does, a function prototype can be thought of as specifying its interface.

**8. What is overloading?**

Overloading refers to the use of the same thing for different purposes. There are 2 types of overloading:

• Function overloading

• Operator overloading

**9. What are objects? How are they created?**

Objects are basic run-time entities in an object-oriented programming system. The class variables are known as objects.

Objects are created by using the syntax:

classname obj1,obj2,…,objn; (or)

during definition of the class:

class classname

{

-------

}obj1,obj2,…,objn;

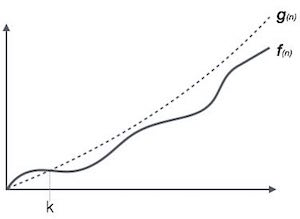
**10. What do you mean by friend functions?**

C++ allows some common functions to be made friendly with any number of classes, thereby allowing the function to have access to the private data of thse classes. Such a function need not be a member of any of these classes. Such common functions are called friend functions.

**11. Explain about Big Oh Notation?**

Big Oh Notation, Ο

The notation Ο(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.



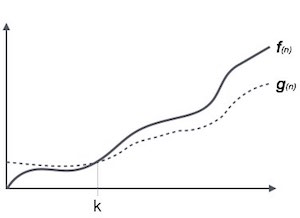
For example, for a function ***f*(n)**

Ο(*f*(n)) = { *g*(n) : there exists c > 0 and n0 such that *f*(n) ≤ c.*g*(n) for all n > n0. }

**12. Explain about Omega notation?**

Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.



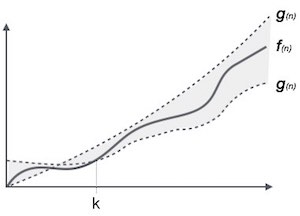
For example, for a function ***f*(n)**

Ω(*f*(n)) ≥ { *g*(n) : there exists c > 0 and n0 such that *g*(n) ≤ c.*f*(n) for all n > n0. }

**13. Explain about Theta notation?**

Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows −



θ(*f*(n)) = { *g*(n) if and only if *g*(n) = Ο(*f*(n)) and *g*(n) = Ω(*f*(n)) for all n > n0. }

## ****14. Explain about**** Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the following characteristics −

* **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
* **Input** − An algorithm should have 0 or more well-defined inputs.
* **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
* **Finiteness** − Algorithms must terminate after a finite number of steps.
* **Feasibility** − Should be feasible with the available resources.
* **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code.

**15 .Explain about space complexity?**

## Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components −

* A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
* A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I. Following is a simple example that tries to explain the concept −

Algorithm: SUM(A, B)

Step 1 - START

Step 2 - C ← A + B + 10

Step 3 - Stop

Here we have three variables A, B, and C and one constant. Hence S(P) = 1 + 3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

## 16. Explain about Time Complexity?

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n-bit integers takes **n** steps. Consequently, the total computational time is T(n) = c ∗ n, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.

**17. Explain about Array data structure?**

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.

**18. Explain about representation of array?**

## 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.

**19. Explain about array 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.

**20. Describe the importance of destructor?**

A destructor destroys the objects that have been created by a constructor upon exit from the program or block to release memory space for future use. It is a member function whose name is the same as the class name but is preceded by a tilde. Syntax: ~classname(){ }

**UNIT II**

**1) What is data structure?**

Data structures refers to the way data is organized and manipulated. It seeks to find ways to make data access more efficient. When dealing with data structure, we not only focus on one piece of data, but rather different set of data and how they can relate to one another in an organized manner.

**2) What is LIFO?**

LIFO is short for Last In First Out, and refers to how data is accessed, stored and retrieved. Using this scheme, data that was stored last , should be the one to be extracted first. This also means that in order to gain access to the first data, all the other data that was stored before this first data must first be retrieved and extracted.

**3) What is a queue?**

A queue is a data structures that can simulates a list or stream of data. In this structure, new elements are inserted at one end and existing elements are removed from the other end.

**4) What is a stack?**

A stack is a data structure in which only the top element can be accessed. As data is stored in the stack, each data is pushed downward, leaving the most recently added data on top.

**5) What are multidimensional arrays?**

Multidimensional arrays make use of multiple indexes to store data. It is useful when storing data that cannot be represented using a single dimensional indexing, such as data representation in a board game, tables with data stored in more than one column.

**6) Are linked lists considered linear or non-linear data structures?**

It actually depends on where you intend to apply linked lists. If you based it on storage, a linked list is considered non-linear. On the other hand, if you based it on access strategies, then a linked list is considered linear.

**7) How does dynamic memory allocation help in managing data?**

Aside from being able to store simple structured data types, dynamic memory allocation can combine separately allocated structured blocks to form composite structures that expand and contract as needed.

**8) What is FIFO?**

FIFO is short for First-in, First-out, and is used to represent how data is accessed in a queue. Data has been inserted into the queue list the longest is the one that is removed first.

**9) What is an ordered list?**

An ordered list is a list in which each node’s position in the list is determined by the value of its key component, so that the key values form an increasing sequence, as the list is traversed.

**10) What is merge sort?**

Merge sort takes a divide-and-conquer approach to sorting data. In a sequence of data, adjacent ones are merged and sorted to create bigger sorted lists. These sorted lists are then merged again to form an even bigger sorted list, which continuous until you have one single sorted list.

**11) Differentiate NULL and VOID.**

Null is actually a value, whereas Void is a data type identifier. A variable that is given a Null value simply indicates an empty value. Void is used to identify pointers as having no initial size.

**12) What is the primary advantage of a linked list?**

A linked list is a very ideal data structure because it can be modified easily. This means that modifying a linked list works regardless of how many elements are in the list.

**13) What is the difference between a PUSH and a POP?**

Pushing and popping applies to the way data is stored and retrieved in a stack. A push denotes data being added to it, meaning data is being “pushed” into the stack. On the other hand, a pop denotes data retrieval, and in particular refers to the topmost data being accessed.

**14) What is the minimum number of queues needed when implementing a priority queue?**

The minimum number of queues needed in this case is two. One queue is intended for sorting priorities while the other queue is intended for actual storage of data.

**15) Differentiate linear from non linear data structure.**

Linear data structure is a structure wherein data elements are adjacent to each other. Examples of linear data structure include arrays, linked lists, stacks and queues. On the other hand, non-linear data structure is a structure wherein each data element can connect to more than two adjacent data elements. Examples of non linear data structure include trees and graphs.

**UNIT III**

**1. Define non-linear data structure?**

Data structure which is capable of expressing more complex relationship than that of physical adjacency is called non-linear data structure.

**2. Define tree?**

A tree is a data structure, which represents hierarchical relationship between individual Data items.

**3.Define child and parent of a tree?**

The root of each subtree is said to be a child of ‘r’ and ‘r’ is the parent of each subtree root.

**4. Define leaf?**

In a directed tree any node which has out degree o is called a terminal node or a leaf.

**5. What is a Binary tree?**

A Binary tree is a finite set of elements that is either empty or is partitioned into three disjoint subsets. The first subset contains a single element called the root of the tree. The other two subsets are themselves binary trees called the left and right sub trees.

**6. What are the applications of binary tree?**

Binary tree is used in data processing. a. File index schemes b. Hierarchical database management system

**7. What is meant by traversing?**

Traversing a tree means processing it in such a way, that each node is visited only once.

**8. What are the different types of traversing?**

The different types of traversing are a.Pre-order traversal-yields prefix form of expression. b. In-order traversal-yields infix form of expression. c. Post-order traversal-yields postfix form of expression.

**9. What are the two methods of binary tree implementation?**

Two methods to implement a binary tree are, a. Linear representation. b. Linked representation

**UNIT IV**

**1. List the types of sorting technique?**

Insertion sort

Quick sort

Merge sort

Heap sort

**2. Determine the complexity of linear search?**

Best case complexity requires O (1) time.

Worst case complexity requires O (n) time.

Average case complexity requires O (n) time.

**3. Determine the complexity of Binary search?**

Best case complexity requires O (1) time.

Average case complexity requires O (log n) time.

**4. What is the complexity of insertion sort algorithm?**

Best case: Only one comparison is made in each pass.

The time complexity is O(n2 ).

Worst case: The time complexity is O(n2 ).

Average case: The time complexity is O(n2 )

**5. Mention the complexities of merge sort and shell sort?**

Merge sort: The merge sort algorithm passed over the entire list and requires atmost log n passes and merges n elements in each pass. The total number of comparisons required by the merge sort is given by, O ( n log n) Shell sort: Best case : If appropriate sequence of increments is required by the merge sort is given by O ( n log n). Worst case : If the increment sequence is not chosen properly the running time of the shell sort is O (N2 )

**6. Name the type of searching?**

Linear search & Binary search.

**7 . Determine the average running of Quick sort?**

The fastest sorting algorithm is Quick sort.

Best Case: Input array is evenly divided. So the running time complexity is O (n log n). Worst Case: Input array is not evenly divided. So the running time complexity is O (n 2 ). Average Case: The running time complexity is O (n log n).

**8. What is replacement selection?**

We read as many records as possible and sort them. Writing the result to some tapes. This seems like the best approach possible until one realizes that as soon as the first record is written to a output tape the memory it used becomes available for another record. If the next record on the input tape is larger than the record we have just output then it can be included in the item. Using this we can give algorithm. This is called replacement selection.

**9. What is sorting?**

Sorting is the process of arranging the given items in a logical order. Sorting is an example where the analysis can be precisely performed.

**10. What is mergesort?**

The mergesort algorithm is a classic divide and conquer strategy. The problem is divided into two arrays and merged into single array

**11. What is maxheap?**

We want the elements in the more typical increasing sorted order, we can change the ordering property so that the parent has a larger key than the child. it is called max heap.

**12. What is divide and conquer strategy?**

In divide and conquer strategy the given problem is divided into smaller problems and solved recursively. The conquering phase consists of patching together the answers. Divide and conquer is a very powerful use of recursion that we will see many times.

**1 3. Differentiate between merge sort and quick sort?**

1. Divide and conquer strategy,

2. Partition by position

Quicksort

1. Divide and conquer strategy,

2. Partition by value

**14. Mention some methods for choosing the pivot element in quicksort?**

1. Choosing first element

2. Generate random number &

3. Median of three

**15 What is the need of external sorting?**

External sorting is required where the input is too large to fit into memory. So external sorting is necessary where the program is too large. It is a basic external sorting in which there are two inputs and two outputs tapes.

**16. Define multi way merge**

If we have extra tapes then we can expect to reduce the number of passes required to sort.

**17. Define polyphase merge?**

The k-way merging strategy requires the use of 2 k tapes. This could be prohibitive for some applications. It is possible to get

**18. What is replacement selection?**

We read as many records as possible and sort them. Writing the result to some tapes. This seems like the best approach possible until one realizes that as soon as the first record is written to a output tape the memory it used becomes available for another record. If the next record on the input tape is larger than the record we have just output then it can be included in the item. Using this we can give algorithm. This is called replacement selection.

**19. What is sorting?**

Sorting is the process of arranging the given items in a logical order. Sorting is an example where the analysis can be precisely performed.

**20. What is mergesort?**

The mergesort algorithm is a classic divide and conquer strategy. The problem is divided into two arrays and merged into single array

**UNIT V**

**1. Define Graph?**

A graph G consist of a nonempty set V which is a set of nodes of the graph, a set E which is the set of edges of the graph, and a mapping from the set for edge E to a set of pairs of elements of V. It can also be represented as G=(V, E).

**2. Define adjacent nodes?**

Any two nodes which are connected by an edge in a graph are called adjacent nodes. For Example, if and edge xÎE is associated with a pair of nodes (u,v) where u, v Î V, then we say that the edge x connects the nodes u and v.

**3.Name the different ways of representing a graph?**

a. Adjacency matrix

b. Adjacency list

**4. What are the two traversal strategies used in traversing a graph?**

a. Breadth first search

b. Depth first search

**5 . What is an acyclic graph?**

A simple diagram which does not have any cycles is called an acyclic graph.

**6. Give some example of NP complete problems?**

Hamiltonian circuit.

Travelling salesmen problems

**7. What are AVL trees?**

AVL tree is a binary search tree with a balancing condition.For every node in the tree the heigh of the left and right subtrees can differ at most by 1.The height ofan empty tree is defined to be -1.It ensures that the depth of the tree is O(log N)

**8. What is topological sort?**

A topological sort is an ordering of vertices in a directed acyclic graph,such that if there is a path from vi then vj appears after vi in the ordering.

**9. What is single source shortest path problem?**

Given as an input a weighted graph, G=(V,E) and a distinguished vertex,’s’ find the shortest weighted path from ‘s’ to every other vertex in G.

**10.Mention some shortest –path problems?**

Unweighted shortest paths Dijikstra’s algorithm All-pairs shortest paths

11.What are the algorithms used to find the minimum spanning tree?

Prim’s algorithm

Kruskal’s algorithm

**12. Define complete binary tree?**

It is a complete binary tree only if all levels, except possibly the last level have the maximum number of nodes maximum. A complete binary tree of height ‘h’ has between 2h and 2h+1 – 1 node

**13. Define binary search tree**?

Why it is preferred rather than the sorted linear array and linked list?

Binary search tree is a binary tree in which key values of the left sub trees are lesser than the root value and the key values of the right sub tree are always greater than the root value. In linear array or linked list the values are arranged in the form of increasing or decreasing order. If we want to access any element means, we have to traverse the entire list. But if we use BST, the element to be accessed is greater or smaller than the root element means we can traverse either the right or left sub tree and can access the element irrespective of searching the entire tree.

**14. Give various implementations of trees**?

Linear implementation

Linked list implementation

**15. What is the difference between binary tree and binary search tree?**

Binary Tree A tree is said to be a binary tree if it, has at most two children Binary Search Tree A BST is binary tree in which the key values in the left sub tree is lesser than the root and key values of the right sub tree is greater than the root value. It doesn’t have any order. It should be in order.

**16. Show the maximum number of nodes in a binary tree of height H is 2H+1 – 1?**

Consider H = 3 No. of nodes in a full binary tree = 2H+1 – 1 = 23+1 – 1 = 24 – 1 = 15 nodes We know that a full binary tree with height h=3 has maximum 15 nodes. Hence proved.

**17. List out the cases involved in deleting a node from a binary search tree?**

Case 1: Node to be deleted is a Leaf node

Case 2: Node to be deleted has one child

Case:3: Node to be deleted has two children.

**18. Write the pre-order, in-order, post-order traversal for the tree?**

In order traversal: (A \* B) + (C \* D) + E Pre order traversal: + \* A B + \* C D EPost order traversal: A B \* C D \* E +

**19. A + (B-C)\*D+(E\*F), if the above arithmetic expression is represented using a binary tree, Find the number of non-leaf nodes in the tree**?

Expression tree is a binary tree in which non – leaf nodes are operators and the leaf nodes are operands. In the above example, we have 5 operators. Therefore the number of non-leaf nodes in the tree is 5.

**20. What is a BST – binary search tree?**

Binary search tree is a binary tree in which key values of the left sub trees are lesser than the root value and the key values of the right sub tree are always greater than the root value.

**21. Define threaded Binary Tree?**

A binary tree is threaded by making all right child pointers that would normally be null point to the inorder successor of the node, and all left child pointers that would normally be null point to the inorder predecessor of the node.

**22. Write the advantages of threaded binary tree?**

1. By doing threading we avoid the recursive method of traversing a Tree , which makes use of stack and consumes a lot of memory and time.

2. The node can keep record of its root

**LONG ANSWER QUESTIONS**

**UNIT I**

**1. Explain about polymorphism in c++**

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Consider the following example where a base class has been derived by other two classes −

#include <iostream>

using namespace std;

class Shape {

protected:

int width, height;

public:

Shape( int a = 0, int b = 0){

width = a;

height = b;

}

int area() {

cout << "Parent class area :" <<endl;

return 0;

}

};

class Rectangle: public Shape {

public:

Rectangle( int a = 0, int b = 0):Shape(a, b) { }

int area () {

cout << "Rectangle class area :" <<endl;

return (width \* height);

}

};

class Triangle: public Shape {

public:

Triangle( int a = 0, int b = 0):Shape(a, b) { }

int area () {

cout << "Triangle class area :" <<endl;

return (width \* height / 2);

}

};

// Main function for the program

int main() {

Shape \*shape;

Rectangle rec(10,7);

Triangle tri(10,5);

// store the address of Rectangle

shape = &rec;

// call rectangle area.

shape->area();

// store the address of Triangle

shape = &tri;

// call triangle area.

shape->area();

return 0;

}

When the above code is compiled and executed, it produces the following result −

Parent class area :

Parent class area :

The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called**static resolution** of the function call, or **static linkage** - the function call is fixed before the program is executed. This is also sometimes called **early binding** because the area() function is set during the compilation of the program.

But now, let's make a slight modification in our program and precede the declaration of area() in the Shape class with the keyword **virtual** so that it looks like this −

class Shape {

protected:

int width, height;

public:

Shape( int a = 0, int b = 0) {

width = a;

height = b;

}

virtual int area() {

cout << "Parent class area :" <<endl;

return 0;

}

};

After this slight modification, when the previous example code is compiled and executed, it produces the following result −

Rectangle class area

Triangle class area

This time, the compiler looks at the contents of the pointer instead of it's type. Hence, since addresses of objects of tri and rec classes are stored in \*shape the respective area() function is called.

As you can see, each of the child classes has a separate implementation for the function area(). This is how **polymorphism** is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

## Virtual Function

A **virtual** function is a function in a base class that is declared using the keyword **virtual**. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as **dynamic linkage**, or **late binding**.

## Pure Virtual Functions

It is possible that you want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

We can change the virtual function area() in the base class to the following −

class Shape {

protected:

int width, height;

public:

Shape(int a = 0, int b = 0) {

width = a;

height = b;

}

// pure virtual function

virtual int area() = 0;

};

The = 0 tells the compiler that the function has no body and above virtual function will be called **pure virtual function**.

**2. Explain about classes and objects in c++?**

The main purpose of C++ programming is to add object orientation to the C programming language and classes are the central feature of C++ that supports object-oriented programming and are often called user-defined types.

A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called members of the class.

## C++ Class Definitions

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Box data type using the keyword **class** as follows:

class Box {

public:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

The keyword **public** determines the access attributes of the members of the class that follow it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as **private** or **protected** which we will discuss in a sub-section.

## Define C++ Objects

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box:

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

Both of the objects Box1 and Box2 will have their own copy of data members.

## Accessing the Data Members

The public data members of objects of a class can be accessed using the direct member access operator (.). Let us try the following example to make the things clear:

#include <iostream>

using namespace std;

class Box {

public:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

int main( ) {

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

double volume = 0.0; // Store the volume of a box here

// box 1 specification

Box1.height = 5.0;

Box1.length = 6.0;

Box1.breadth = 7.0;

// box 2 specification

Box2.height = 10.0;

Box2.length = 12.0;

Box2.breadth = 13.0;

// volume of box 1

volume = Box1.height \* Box1.length \* Box1.breadth;

cout << "Volume of Box1 : " << volume <<endl;

// volume of box 2

volume = Box2.height \* Box2.length \* Box2.breadth;

cout << "Volume of Box2 : " << volume <<endl;

return 0;

}

When the above code is compiled and executed, it produces the following result:

Volume of Box1 : 210

Volume of Box2 : 1560

It is important to note that private and protected members can not be accessed directly using direct member access operator (.). We will learn how private and protected members can be accessed.

**3. Explain about function overloading in c++?**

## Function overloading in C++

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You can not overload function declarations that differ only by return type.

Following is the example where same function **print()** is being used to print different data types:

#include <iostream>

using namespace std;

class printData {

public:

void print(int i) {

cout << "Printing int: " << i << endl;

}

void print(double f) {

cout << "Printing float: " << f << endl;

}

void print(char\* c) {

cout << "Printing character: " << c << endl;

}

};

int main(void) {

printData pd;

// Call print to print integer

pd.print(5);

// Call print to print float

pd.print(500.263);

// Call print to print character

pd.print("Hello C++");

return 0;

}

When the above code is compiled and executed, it produces the following result:

Printing int: 5

Printing float: 500.263

Printing character: Hello C++

**4. Explain about operator overloading in c++?**

## Operators overloading in C++

You can redefine or overload most of the built-in operators available in C++. Thus a programmer can use operators with user-defined types as well.

Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

Box operator+(const Box&);

declares the addition operator that can be used to **add** two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions. In case we define above function as non-member function of a class then we would have to pass two arguments for each operand as follows:

Box operator+(const Box&, const Box&);

Following is the example to show the concept of operator over loading using a member function. Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using **this** operator as explained below:

#include <iostream>

using namespace std;

class Box {

public:

double getVolume(void) {

return length \* breadth \* height;

}

void setLength( double len ) {

length = len;

}

void setBreadth( double bre ) {

breadth = bre;

}

void setHeight( double hei ) {

height = hei;

}

// Overload + operator to add two Box objects.

Box operator+(const Box& b) {

Box box;

box.length = this->length + b.length;

box.breadth = this->breadth + b.breadth;

box.height = this->height + b.height;

return box;

}

private:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

// Main function for the program

int main( ) {

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

Box Box3; // Declare Box3 of type Box

double volume = 0.0; // Store the volume of a box here

// box 1 specification

Box1.setLength(6.0);

Box1.setBreadth(7.0);

Box1.setHeight(5.0);

// box 2 specification

Box2.setLength(12.0);

Box2.setBreadth(13.0);

Box2.setHeight(10.0);

// volume of box 1

volume = Box1.getVolume();

cout << "Volume of Box1 : " << volume <<endl;

// volume of box 2

volume = Box2.getVolume();

cout << "Volume of Box2 : " << volume <<endl;

// Add two object as follows:

Box3 = Box1 + Box2;

// volume of box 3

volume = Box3.getVolume();

cout << "Volume of Box3 : " << volume <<endl;

return 0;

}

When the above code is compiled and executed, it produces the following result:

Volume of Box1 : 210

Volume of Box2 : 1560

Volume of Box3 : 5400

**6. Explain about exception handling in c++ ?**

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,** and**throw**.

* **throw** − A program throws an exception when a problem shows up. This is done using a **throw** keyword.
* **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The**catch** keyword indicates the catching of an exception.
* **try** − A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch as follows −

try {

// protected code

} catch( ExceptionName e1 ) {

// catch block

} catch( ExceptionName e2 ) {

// catch block

} catch( ExceptionName eN ) {

// catch block

}

You can list down multiple **catch** statements to catch different type of exceptions in case your **try** block raises more than one exception in different situations.

## Throwing Exceptions

Exceptions can be thrown anywhere within a code block using **throw**statement. The operand of the throw statement determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs −

double division(int a, int b) {

if( b == 0 ) {

throw "Division by zero condition!";

}

return (a/b);

}

## Catching Exceptions

The **catch** block following the **try** block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

try {

// protected code

} catch( ExceptionName e ) {

// code to handle ExceptionName exception

}

Above code will catch an exception of **ExceptionName** type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows −

try {

// protected code

} catch(...) {

// code to handle any exception

}

The following is an example, which throws a division by zero exception and we catch it in catch block.

#include <iostream>

using namespace std;

double division(int a, int b) {

if( b == 0 ) {

throw "Division by zero condition!";

}

return (a/b);

}

int main () {

int x = 50;

int y = 0;

double z = 0;

try {

z = division(x, y);

cout << z << endl;

} catch (const char\* msg) {

cerr << msg << endl;

}

return 0;

}

Because we are raising an exception of type **const char\***, so while catching this exception, we have to use const char\* in catch block. If we compile and run above code, this would produce the following result −

Division by zero condition!

**7. Explain about function templates in c++?**

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**.

You can use templates to define functions as well as classes, let us see how do they work:

## Function Template

The general form of a template function definition is shown here:

template <class type> ret-type func-name(parameter list) {

// body of function

}

Here, type is a placeholder name for a data type used by the function. This name can be used within the function definition.

The following is the example of a function template that returns the maximum of two values:

#include <iostream>

#include <string>

using namespace std;

template <typename T>

inline T const& Max (T const& a, T const& b) {

return a < b ? b:a;

}

int main () {

int i = 39;

int j = 20;

cout << "Max(i, j): " << Max(i, j) << endl;

double f1 = 13.5;

double f2 = 20.7;

cout << "Max(f1, f2): " << Max(f1, f2) << endl;

string s1 = "Hello";

string s2 = "World";

cout << "Max(s1, s2): " << Max(s1, s2) << endl;

return 0;

}

If we compile and run above code, this would produce the following result:

Max(i, j): 39

Max(f1, f2): 20.7

Max(s1, s2): World

**8. Explain about class template in c++?**

## Class Template

Just as we can define function templates, we can also define class templates. The general form of a generic class declaration is shown here:

template <class type> class class-name {

.

.

.

}

Here, **type** is the placeholder type name, which will be specified when a class is instantiated. You can define more than one generic data type by using a comma-separated list.

Following is the example to define class Stack<> and implement generic methods to push and pop the elements from the stack:

#include <iostream>

#include <vector>

#include <cstdlib>

#include <string>

#include <stdexcept>

using namespace std;

template <class T>

class Stack

{

private:

vector<T> elements;

public:

void push(T const &);

void pop();

T top();

bool empty();

};

template <class T>

void Stack<T>::push(T const &elem) {

elements.push\_back(elem);

}

template <class T>

void Stack<T>::pop() {

if (elements.empty()) {

throw out\_of\_range("Stack<>::pop(): empty stack");

} else {

elements.pop\_back();

}

}

template <class T>

T Stack<T>::top() {

if (empty()) {

throw out\_of\_range("Stack<>::top(): empty stack");

}

return elements.back();

}

template <class T>

bool Stack<T>::empty() {

return elements.empty();

}

int main() {

try {

Stack<int> intStack; // stack of ints

Stack<string> stringStack; // stack of strings

// manipulate integer stack

intStack.push(7);

cout << intStack.top() << endl;

// manipulate string stack

stringStack.push("hello");

cout << stringStack.top() << std::endl;

stringStack.pop();

stringStack.pop();

}

catch (exception const &ex) {

cerr << "Exception: " << ex.what() << endl;

return -1;

}

}

If we compile and run above code, this would produce the following result:

7

hello

Exception: Stack<>::pop(): empty stack

**9. Explain about dynamic memory allocation in c++?**

A good understanding of how dynamic memory really works in C++ is essential to becoming a good C++ programmer. Memory in your C++ program is divided into two parts −

* **The stack** − All variables declared inside the function will take up memory from the stack.
* **The heap** − This is unused memory of the program and can be used to allocate the memory dynamically when program runs.

Many times, you are not aware in advance how much memory you will need to store particular information in a defined variable and the size of required memory can be determined at run time.

You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called **new** operator.

If you are not in need of dynamically allocated memory anymore, you can use**delete** operator, which de-allocates memory that was previously allocated by new operator.

## new and delete Operators

There is following generic syntax to use **new** operator to allocate memory dynamically for any data-type.

new data-type;

Here, **data-type** could be any built-in data type including an array or any user defined data types include class or structure. Let us start with built-in data types. For example we can define a pointer to type double and then request that the memory be allocated at execution time. We can do this using the **new**operator with the following statements −

double\* pvalue = NULL; // Pointer initialized with null

pvalue = new double; // Request memory for the variable

The memory may not have been allocated successfully, if the free store had been used up. So it is good practice to check if new operator is returning NULL pointer and take appropriate action as below −

double\* pvalue = NULL;

if( !(pvalue = new double )) {

cout << "Error: out of memory." <<endl;

exit(1);

}

The **malloc()** function from C, still exists in C++, but it is recommended to avoid using malloc() function. The main advantage of new over malloc() is that new doesn't just allocate memory, it constructs objects which is prime purpose of C++.

At any point, when you feel a variable that has been dynamically allocated is not anymore required, you can free up the memory that it occupies in the free store with the ‘delete’ operator as follows −

delete pvalue; // Release memory pointed to by pvalue

Let us put above concepts and form the following example to show how ‘new’ and ‘delete’ work −

#include <iostream>

using namespace std;

int main () {

double\* pvalue = NULL; // Pointer initialized with null

pvalue = new double; // Request memory for the variable

\*pvalue = 29494.99; // Store value at allocated address

cout << "Value of pvalue : " << \*pvalue << endl;

delete pvalue; // free up the memory.

return 0;

}

If we compile and run above code, this would produce the following result −

Value of pvalue : 29495

## Dynamic Memory Allocation for Arrays

Consider you want to allocate memory for an array of characters, i.e., string of 20 characters. Using the same syntax what we have used above we can allocate memory dynamically as shown below.

char\* pvalue = NULL; // Pointer initialized with null

pvalue = new char[20]; // Request memory for the variable

To remove the array that we have just created the statement would look like this −

delete [] pvalue; // Delete array pointed to by pvalue

Following the similar generic syntax of new operator, you can allocate for a multi-dimensional array as follows −

double\*\* pvalue = NULL; // Pointer initialized with null

pvalue = new double [3][4]; // Allocate memory for a 3x4 array

However, the syntax to release the memory for multi-dimensional array will still remain same as above −

delete [] pvalue; // Delete array pointed to by pvalue

## Dynamic Memory Allocation for Objects

Objects are no different from simple data types. For example, consider the following code where we are going to use an array of objects to clarify the concept −

#include <iostream>

using namespace std;

class Box {

public:

Box() {

cout << "Constructor called!" <<endl;

}

~Box() {

cout << "Destructor called!" <<endl;

}

};

int main() {

Box\* myBoxArray = new Box[4];

delete [] myBoxArray; // Delete array

return 0;

}

If you were to allocate an array of four Box objects, the Simple constructor would be called four times and similarly while deleting these objects, destructor will also be called same number of times.

If we compile and run above code, this would produce the following result −

Constructor called!

Constructor called!

Constructor called!

Constructor called!

Destructor called!

Destructor called!

Destructor called!

Destructor called!

**10. Explain about asymptotic notations?**

Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as *f*(n) and may be for another operation it is computed as *g*(n2). This means the first operation running time will increase linearly with the increase in **n** and the running time of the second operation will increase exponentially when **n** increases. Similarly, the running time of both operations will be nearly the same if **n** is significantly small.

Usually, the time required by an algorithm falls under three types −

* **Best Case** − Minimum time required for program execution.
* **Average Case** − Average time required for program execution.
* **Worst Case** − Maximum time required for program execution.

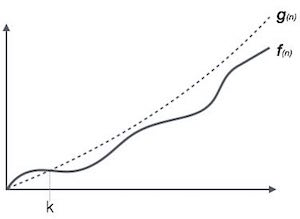
## Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

* Ο Notation
* Ω Notation
* θ Notation

### Big Oh Notation, Ο

The notation Ο(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

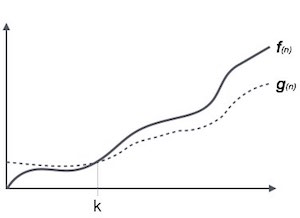


For example, for a function ***f*(n)**

Ο(*f*(n)) = { *g*(n) : there exists c > 0 and n0 such that *f*(n) ≤ c.*g*(n) for all n > n0. }

### Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

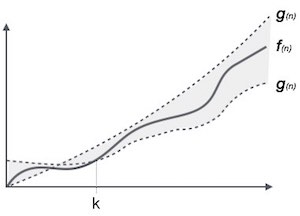


For example, for a function ***f*(n)**

Ω(*f*(n)) ≥ { *g*(n) : there exists c > 0 and n0 such that *g*(n) ≤ c.*f*(n) for all n > n0. }

### Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows −



θ(*f*(n)) = { *g*(n) if and only if *g*(n) = Ο(*f*(n)) and *g*(n) = Ω(*f*(n)) for all n > n0. }

## Common Asymptotic Notations

Following is a list of some common asymptotic notations −

|  |  |  |
| --- | --- | --- |
| constant | − | Ο(1) |
| logarithmic | − | Ο(log n) |
| linear | − | Ο(n) |
| n log n | − | Ο(n log n) |
| quadratic | − | Ο(n2) |
| cubic | − | Ο(n3) |
| polynomial | − | nΟ(1) |
| exponential | − | 2Ο(n) |

**UNIT II**

1. **Explain stack and its basic operations?**

Ans:

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

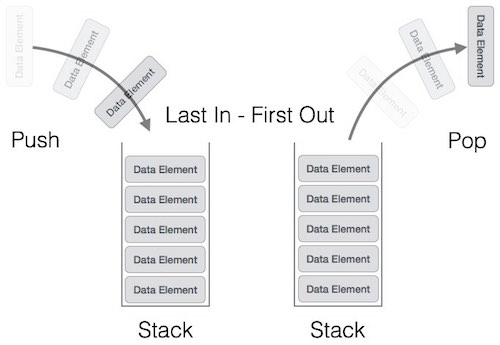


A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

## Stack Representation

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

## 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.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top**pointer provides top value of the stack without actually removing it.

First we should learn about procedures to support stack functions −

### peek()

Algorithm of peek() function −

begin procedure peek

return stack[top]

end procedure

Implementation of peek() function in C programming language −

**Example**

int peek() {

return stack[top];

}

### isfull()

Algorithm of isfull() function −

begin procedure isfull

if top equals to MAXSIZE

return true

else

return false

endif

end procedure

Implementation of isfull() function in C programming language −

**Example**

bool isfull() {

if(top == MAXSIZE)

return true;

else

return false;

}

### isempty()

Algorithm of isempty() function −

begin procedure isempty

if top less than 1

return true

else

return false

endif

end procedure

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code −

**Example**

bool isempty() {

if(top == -1)

return true;

else

return false;

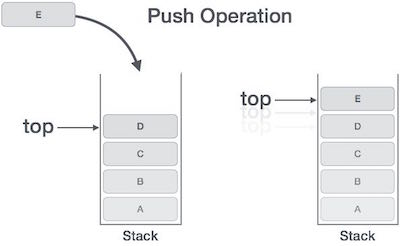
}

## Briefly explain about push and pop operations of stack?

## Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

* **Step 1** − Checks if the stack is full.
* **Step 2** − If the stack is full, produces an error and exit.
* **Step 3** − If the stack is not full, increments **top** to point next empty space.
* **Step 4** − Adds data element to the stack location, where top is pointing.
* **Step 5** − Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

### 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

Implementation of this algorithm in C, is very easy. See the following code −

**Example**

void push(int data) {

if(!isFull()) {

top = top + 1;

stack[top] = data;

} else {

printf("Could not insert data, Stack is full.\n");

}

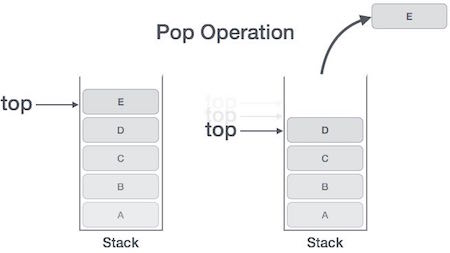
}

## 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.



### 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

Implementation of this algorithm in C, is as follows −

**Example**

int pop(int data) {

if(!isempty()) {

data = stack[top];

top = top - 1;

return data;

} else {

printf("Could not retrieve data, Stack is empty.\n");

}

}

1. **Explain about expression parsing?**

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 **prefix**ed 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 **postfix**ed to the operands i.e., the operator is written after the operands. For example, **ab+**. This is equivalent to its infix notation **a + b**.

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

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

## Parsing Expressions

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

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

### Precedence

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

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

### Associativity

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

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

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

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

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

## 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 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.

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.

1. **Explain about queue and its operations?**

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 −



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.

Here's the C programming code −

**Example**

bool isempty() {

if(front < 0 || front > rear)

return true;

else

return false;

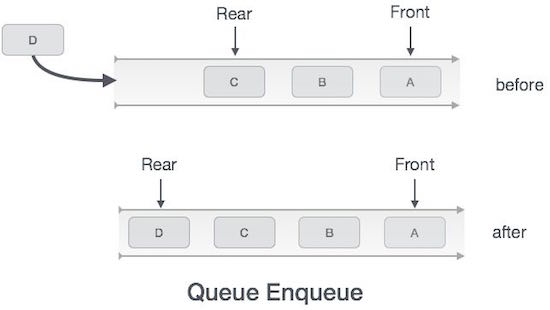
}

## 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.



Sometimes, we also check to see if a queue is initialized or not, to handle any unforeseen situations.

**Algorithm for enqueue operation**

procedure enqueue(data)

if queue is full

return overflow

endif

rear ← rear + 1

queue[rear] ← data

return true

end procedure

Implementation of enqueue() in C programming language −

**Example**

int enqueue(int data)

if(isfull())

return 0;

rear = rear + 1;

queue[rear] = data;

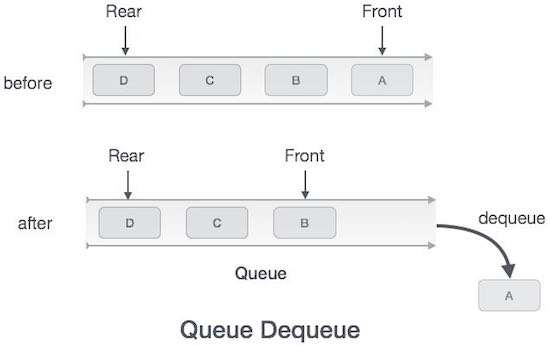
return 1;

end procedure

## 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.



**Algorithm for dequeue operation**

procedure dequeue

if queue is empty

return underflow

end if

data = queue[front]

front ← front + 1

return true

end procedure

Implementation of dequeue() in C programming language −

**Example**

int dequeue() {

if(isempty())

return 0;

int data = queue[front];

front = front + 1;

return data;

}

1. **Explain about Single linked list and its operations?**

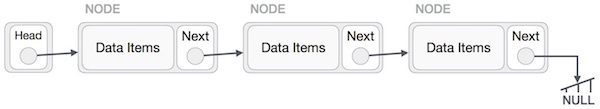
A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

* **Link** − Each link of a linked list can store a data called an element.
* **Next** − Each link of a linked list contains a link to the next link called Next.
* **LinkedList** − A Linked List contains the connection link to the first link called First.

## Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.



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

* Linked List contains a link element called first.
* Each link carries a data field(s) and a link field called next.
* Each link is linked with its next link using its next link.
* Last link carries a link as null to mark the end of the list.

## Types of Linked List

Following are the various types of linked list.

* **Simple Linked List** − Item navigation is forward only.
* **Doubly Linked List** − Items can be navigated forward and backward.
* **Circular Linked List** − Last item contains link of the first element as next and the first element has a link to the last element as previous.

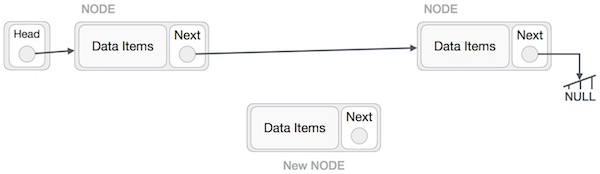
## Basic Operations

Following are the basic operations supported by a list.

* **Insertion** − Adds an element at the beginning of the list.
* **Deletion** − Deletes an element at the beginning of the list.
* **Display** − Displays the complete list.
* **Search** − Searches an element using the given key.
* **Delete** − Deletes an element using the given key.

## Insertion Operation

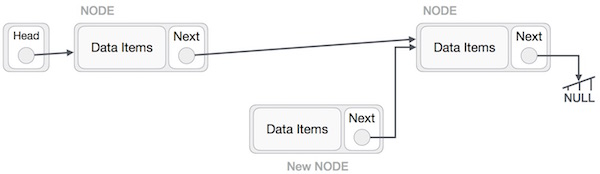
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode) and**C** (RightNode). Then point B.next to C −

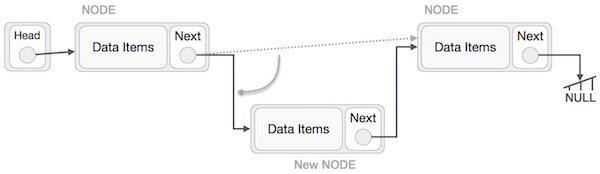
NewNode.next −> RightNode;

It should look like this −

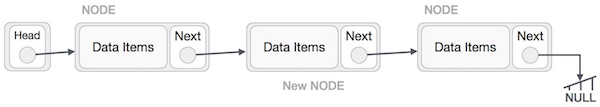


Now, the next node at the left should point to the new node.

LeftNode.next −> NewNode;



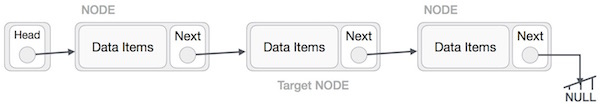
This will put the new node in the middle of the two. The new list should look like this −



Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

## Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.



The left (previous) node of the target node now should point to the next node of the target node −

LeftNode.next −> TargetNode.next;

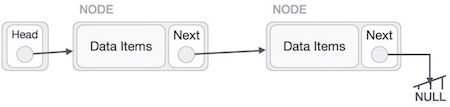


This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

TargetNode.next −> NULL;

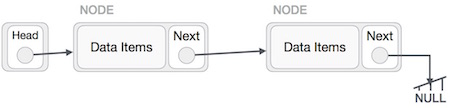


We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

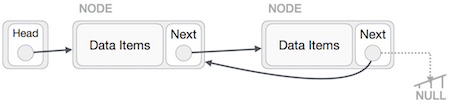


## Reverse Operation

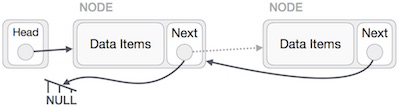
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



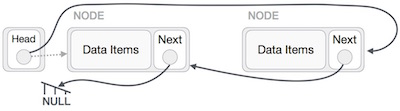
First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node −



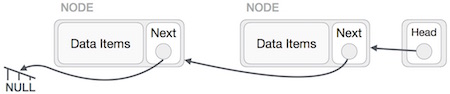
We have to make sure that the last node is not the lost node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.



We'll make the head node point to the new first node by using the temp node.



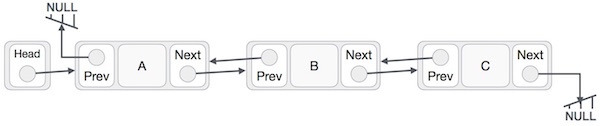
The linked list is now reversed.

1. **Briefly explain about doubly linked list?**

Doubly Linked List is a variation of Linked list in which navigation is possible in both ways, either forward and backward easily as compared to Single Linked List. Following are the important terms to understand the concept of doubly linked list.

* **Link** − Each link of a linked list can store a data called an element.
* **Next** − Each link of a linked list contains a link to the next link called Next.
* **Prev** − Each link of a linked list contains a link to the previous link called Prev.
* **LinkedList** − A Linked List contains the connection link to the first link called First and to the last link called Last.

## Doubly Linked List Representation



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

* Doubly Linked List contains a link element called first and last.
* Each link carries a data field(s) and two link fields called next and prev.
* Each link is linked with its next link using its next link.
* Each link is linked with its previous link using its previous link.
* The last link carries a link as null to mark the end of the list.

## Basic Operations

Following are the basic operations supported by a list.

* **Insertion** − Adds an element at the beginning of the list.
* **Deletion** − Deletes an element at the beginning of the list.
* **Insert Last** − Adds an element at the end of the list.
* **Delete Last** − Deletes an element from the end of the list.
* **Insert After** − Adds an element after an item of the list.
* **Delete** − Deletes an element from the list using the key.
* **Display forward** − Displays the complete list in a forward manner.
* **Display backward** − Displays the complete list in a backward manner.

## Insertion Operation

Following code demonstrates the insertion operation at the beginning of a doubly linked list.

### Example

//insert link at the first location

void insertFirst(int key, int data) {

//create a link

struct node \*link = (struct node\*) malloc(sizeof(struct node));

link->key = key;

link->data = data;

if(isEmpty()) {

//make it the last link

last = link;

} else {

//update first prev link

head->prev = link;

}

//point it to old first link

link->next = head;

//point first to new first link

head = link;

}

## Deletion Operation

Following code demonstrates the deletion operation at the beginning of a doubly linked list.

### Example

//delete first item

struct node\* deleteFirst() {

//save reference to first link

struct node \*tempLink = head;

//if only one link

if(head->next == NULL) {

last = NULL;

} else {

head->next->prev = NULL;

}

head = head->next;

//return the deleted link

return tempLink;

}

## Insertion at the End of an Operation

Following code demonstrates the insertion operation at the last position of a doubly linked list.

### Example

//insert link at the last location

void insertLast(int key, int data) {

//create a link

struct node \*link = (struct node\*) malloc(sizeof(struct node));

link->key = key;

link->data = data;

if(isEmpty()) {

//make it the last link

last = link;

} else {

//make link a new last link

last->next = link;

//mark old last node as prev of new link

link->prev = last;

}

//point last to new last node

last = link;

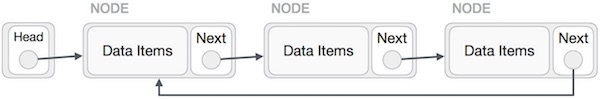
**}**

1. **Briefly explain about circular linked list?**

Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element. Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

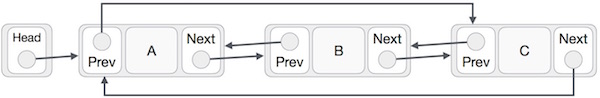
## Singly Linked List as Circular

In singly linked list, the next pointer of the last node points to the first node.



## Doubly Linked List as Circular

In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.



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

* The last link's next points to the first link of the list in both cases of singly as well as doubly linked list.
* The first link's previous points to the last of the list in case of doubly linked list.

## Basic Operations

Following are the important operations supported by a circular list.

* **insert** − Inserts an element at the start of the list.
* **delete** − Deletes an element from the start of the list.
* **display** − Displays the list.

## Insertion Operation

Following code demonstrates the insertion operation in a circular linked list based on single linked list.

### Example

//insert link at the first location

void insertFirst(int key, int data) {

//create a link

struct node \*link = (struct node\*) malloc(sizeof(struct node));

link->key = key;

link->data= data;

if (isEmpty()) {

head = link;

head->next = head;

} else {

//point it to old first node

link->next = head;

//point first to new first node

head = link;

}

}

## Deletion Operation

Following code demonstrates the deletion operation in a circular linked list based on single linked list.

//delete first item

struct node \* deleteFirst() {

//save reference to first link

struct node \*tempLink = head;

if(head->next == head) {

head = NULL;

return tempLink;

}

//mark next to first link as first

head = head->next;

//return the deleted link

return tempLink;

}

## Display List Operation

Following code demonstrates the display list operation in a circular linked list.

//display the list

void printList() {

struct node \*ptr = head;

printf("\n[ ");

//start from the beginning

if(head != NULL) {

while(ptr->next != ptr) {

printf("(%d,%d) ",ptr->key,ptr->data);

ptr = ptr->next;

}

}

printf(" ]");

}

1. Explain about array representation and its operations ?

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.

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

|  |  |
| --- | --- |
| **Data Type** | **Default Value** |
| bool | false |
| char | 0 |
| int | 0 |
| float | 0.0 |
| double | 0.0f |
| void |  |
| wchar\_t | 0 |

## Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on 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

### Example

Following is the implementation of the above algorithm −

#include <stdio.h>

main() {

int LA[] = {1,3,5,7,8};

int item = 10, k = 3, n = 5;

int i = 0, j = n;

printf("The original array elements are :\n");

for(i = 0; i<n; i++) {

printf("LA[%d] = %d \n", i, LA[i]);

}

n = n + 1;

while( j >= k) {

LA[j+1] = LA[j];

j = j - 1;

}

LA[k] = item;

printf("The array elements after insertion :\n");

for(i = 0; i<n; i++) {

printf("LA[%d] = %d \n", i, LA[i]);

}

}

When we compile and execute the above program, it produces the following result −

### Output

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

The array elements after insertion :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 10

LA[4] = 7

LA[5] = 8

For other variations of array insertion operation [click here](https://www.tutorialspoint.com/data_structures_algorithms/array_insertion_algorithm.htm)

## 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

### Example

Following is the implementation of the above algorithm −

#include <stdio.h>

main() {

int LA[] = {1,3,5,7,8};

int k = 3, n = 5;

int i, j;

printf("The original array elements are :\n");

for(i = 0; i<n; i++) {

printf("LA[%d] = %d \n", i, LA[i]);

}

j = k;

while( j < n) {

LA[j-1] = LA[j];

j = j + 1;

}

n = n -1;

printf("The array elements after deletion :\n");

for(i = 0; i<n; i++) {

printf("LA[%d] = %d \n", i, LA[i]);

}

}

When we compile and execute the above program, it produces the following result −

### Output

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

The array elements after deletion :

LA[0] = 1

LA[1] = 3

LA[2] = 7

LA[3] = 8

## 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

### Example

Following is the implementation of the above algorithm −

#include <stdio.h>

main() {

int LA[] = {1,3,5,7,8};

int item = 5, n = 5;

int i = 0, j = 0;

printf("The original array elements are :\n");

for(i = 0; i<n; i++) {

printf("LA[%d] = %d \n", i, LA[i]);

}

while( j < n){

if( LA[j] == item ) {

break;

}

j = j + 1;

}

printf("Found element %d at position %d\n", item, j+1);

}

When we compile and execute the above program, it produces the following result −

### Output

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

Found element 5 at position 3

## 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 Kth position of LA.

1. Start

2. Set LA[K-1] = ITEM

3. Stop

### Example

Following is the implementation of the above algorithm −

#include <stdio.h>

main() {

int LA[] = {1,3,5,7,8};

int k = 3, n = 5, item = 10;

int i, j;

printf("The original array elements are :\n");

for(i = 0; i<n; i++) {

printf("LA[%d] = %d \n", i, LA[i]);

}

LA[k-1] = item;

printf("The array elements after updation :\n");

for(i = 0; i<n; i++) {

printf("LA[%d] = %d \n", i, LA[i]);

}

}

When we compile and execute the above program, it produces the following result −

### Output

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

The array elements after updation :

LA[0] = 1

LA[1] = 3

LA[2] = 10

LA[3] = 7

LA[4] = 8

**UNIT IV**

**1. Explain about selection sort in c++?**

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of Ο(n2), where **n** is the number of items.

## How Selection Sort Works?

Consider the following depicted array as an example.

Unsorted Array

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

Selection Sort

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

Selection Sort

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

Selection Sort

We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

Selection Sort

After two iterations, two least values are positioned at the beginning in a sorted manner.

Selection Sort

The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process −



Now, let us learn some programming aspects of selection sort.

### Algorithm

**Step 1** − Set MIN to location 0

**Step 2** − Search the minimum element in the list

**Step 3** − Swap with value at location MIN

**Step 4** − Increment MIN to point to next element

**Step 5** − Repeat until list is sorted

### Pseudocode

procedure selection sort

list : array of items

n : size of list

for i = 1 to n - 1

/\* set current element as minimum\*/

min = i

/\* check the element to be minimum \*/

for j = i+1 to n

if list[j] < list[min] then

min = j;

end if

end for

/\* swap the minimum element with the current element\*/

if indexMin != i then

swap list[min] and list[i]

end if

end for

end procedure

2. Explain about merge sort ?

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being Ο(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.

## How Merge Sort Works?

To understand merge sort, we take an unsorted array as the following −

Unsorted Array

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

Merge Sort Division

This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

Merge Sort Division

We further divide these arrays and we achieve atomic value which can no more be divided.

Merge Sort Division

Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.

Merge Sort Combine

In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

Merge Sort Combine

After the final merging, the list should look like this −

Merge Sort

Now we should learn some programming aspects of merge sorting.

### Algorithm

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

**Step 1** − if it is only one element in the list it is already sorted, return.

**Step 2** − divide the list recursively into two halves until it can no more be divided.

**Step 3** − merge the smaller lists into new list in sorted order.

### Pseudocode

We shall now see the pseudocodes for merge sort functions. As our algorithms point out two main functions − divide & merge.

Merge sort works with recursion and we shall see our implementation in the same way.

procedure mergesort( var a as array )

if ( n == 1 ) return a

var l1 as array = a[0] ... a[n/2]

var l2 as array = a[n/2+1] ... a[n]

l1 = mergesort( l1 )

l2 = mergesort( l2 )

return merge( l1, l2 )

end procedure

procedure merge( var a as array, var b as array )

var c as array

while ( a and b have elements )

if ( a[0] > b[0] )

add b[0] to the end of c

remove b[0] from b

else

add a[0] to the end of c

remove a[0] from a

end if

end while

while ( a has elements )

add a[0] to the end of c

remove a[0] from a

end while

while ( b has elements )

add b[0] to the end of c

remove b[0] from b

end while

return c

end procedure

**3. Explain about shell sort in c++?**

Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.

This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as**interval**. This interval is calculated based on Knuth's formula as −

### Knuth's Formula

h = h \* 3 + 1

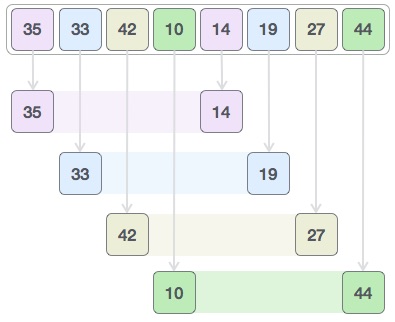
where −

h is interval with initial value 1

This algorithm is quite efficient for medium-sized data sets as its average and worst case complexity are of Ο(n), where **n** is the number of items.

## How Shell Sort Works?

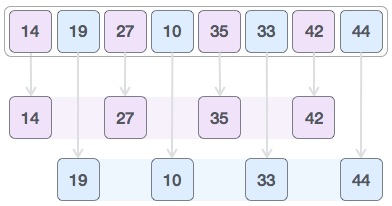
Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous examples. For our example and ease of understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 44}



We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this −

Shell Sort

Then, we take interval of 2 and this gap generates two sub-lists - {14, 27, 35, 42}, {19, 10, 33, 44}



We compare and swap the values, if required, in the original array. After this step, the array should look like this −

Shell Sort

Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array.

Following is the step-by-step depiction −



We see that it required only four swaps to sort the rest of the array.

### Algorithm

Following is the algorithm for shell sort.

**Step 1** − Initialize the value of *h*

**Step 2** − Divide the list into smaller sub-list of equal interval *h*

**Step 3** − Sort these sub-lists using **insertion sort**

**Step 3** − Repeat until complete list is sorted

## Pseudocode

Following is the pseudocode for shell sort.

procedure shellSort()

A : array of items

/\* calculate interval\*/

while interval < A.length /3 do:

interval = interval \* 3 + 1

end while

while interval > 0 do:

for outer = interval; outer < A.length; outer ++ do:

/\* select value to be inserted \*/

valueToInsert = A[outer]

inner = outer;

/\*shift element towards right\*/

while inner > interval -1 && A[inner - interval] >= valueToInsert do:

A[inner] = A[inner - interval]

inner = inner - interval

end while

/\* insert the number at hole position \*/

A[inner] = valueToInsert

end for

/\* calculate interval\*/

interval = (interval -1) /3;

end while

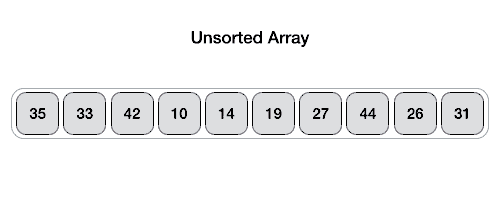
end procedure

**4. Explain abot quick sort ?**

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quick sort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst case complexity are of Ο(n2), where **n** is the number of items.

## Partition in Quick Sort

Following animated representation explains how to find the pivot value in an array.

The pivot value divides the list into two parts. And recursively, we find the pivot for each sub-lists until all lists contains only one element.

## Quick Sort Pivot Algorithm

Based on our understanding of partitioning in quick sort, we will now try to write an algorithm for it, which is as follows.

**Step 1** − Choose the highest index value has pivot

**Step 2** − Take two variables to point left and right of the list excluding pivot

**Step 3** − left points to the low index

**Step 4** − right points to the high

**Step 5** − while value at left is less than pivot move right

**Step 6** − while value at right is greater than pivot move left

**Step 7** − if both step 5 and step 6 does not match swap left and right

**Step 8** − if left ≥ right, the point where they met is new pivot

## Quick Sort Pivot Pseudocode

The pseudocode for the above algorithm can be derived as −

function partitionFunc(left, right, pivot)

leftPointer = left

rightPointer = right - 1

while True do

while A[++leftPointer] < pivot do

//do-nothing

end while

while rightPointer > 0 && A[--rightPointer] > pivot do

//do-nothing

end while

if leftPointer >= rightPointer

break

else

swap leftPointer,rightPointer

end if

end while

swap leftPointer,right

return leftPointer

end function

## Quick Sort Algorithm

Using pivot algorithm recursively, we end up with smaller possible partitions. Each partition is then processed for quick sort. We define recursive algorithm for quicksort as follows −

**Step 1** − Make the right-most index value pivot

**Step 2** − partition the array using pivot value

**Step 3** − quicksort left partition recursively

**Step 4** − quicksort right partition recursively

## Quick Sort Pseudocode

To get more into it, let see the pseudocode for quick sort algorithm −

procedure quickSort(left, right)

if right-left <= 0

return

else

pivot = A[right]

partition = partitionFunc(left, right, pivot)

quickSort(left,partition-1)

quickSort(partition+1,right)

end if

end procedure

5. Explain about bubble sort?

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of Ο(n2) where **n**is the number of items.

## How Bubble Sort Works?

We take an unsorted array for our example. Bubble sort takes Ο(n2) time so we're keeping it short and precise.

Bubble Sort

Bubble sort starts with very first two elements, comparing them to check which one is greater.

Bubble Sort

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

Bubble Sort

We find that 27 is smaller than 33 and these two values must be swapped.

Bubble Sort

The new array should look like this −

Bubble Sort

Next we compare 33 and 35. We find that both are in already sorted positions.

Bubble Sort

Then we move to the next two values, 35 and 10.

Bubble Sort

We know then that 10 is smaller 35. Hence they are not sorted.

Bubble Sort

We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

Bubble Sort

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

Bubble Sort

Notice that after each iteration, at least one value moves at the end.

Bubble Sort

And when there's no swap required, bubble sorts learns that an array is completely sorted.

Bubble Sort

Now we should look into some practical aspects of bubble sort.

## Algorithm

We assume **list** is an array of **n** elements. We further assume that **swap**function swaps the values of the given array elements.

begin BubbleSort(list)

for all elements of list

if list[i] > list[i+1]

swap(list[i], list[i+1])

end if

end for

return list

end BubbleSort

## Pseudocode

We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.

To ease-out the issue, we use one flag variable **swapped** which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.

Pseudocode of BubbleSort algorithm can be written as follows −

procedure bubbleSort( list : array of items )

loop = list.count;

for i = 0 to loop-1 do:

swapped = false

for j = 0 to loop-1 do:

/\* compare the adjacent elements \*/

if list[j] > list[j+1] then

/\* swap them \*/

swap( list[j], list[j+1] )

swapped = true

end if

end for

/\*if no number was swapped that means

array is sorted now, break the loop.\*/

if(not swapped) then

break

end if

end for

end procedure return list

## Implementation

One more issue we did not address in our original algorithm and its improvised pseudocode, is that, after every iteration the highest values settles down at the end of the array. Hence, the next iteration need not include already sorted elements. For this purpose, in our implementation, we restrict the inner loop to avoid already sorted values.

**6. Explain about merge sort?**

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being Ο(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.

## How Merge Sort Works?

To understand merge sort, we take an unsorted array as the following −

Unsorted Array

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

Merge Sort Division

This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

Merge Sort Division

We further divide these arrays and we achieve atomic value which can no more be divided.

Merge Sort Division

Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.

Merge Sort Combine

In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

Merge Sort Combine

After the final merging, the list should look like this −

Merge Sort

Now we should learn some programming aspects of merge sorting.

### Algorithm

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

**Step 1** − if it is only one element in the list it is already sorted, return.

**Step 2** − divide the list recursively into two halves until it can no more be divided.

**Step 3** − merge the smaller lists into new list in sorted order.

### Pseudocode

We shall now see the pseudocodes for merge sort functions. As our algorithms point out two main functions − divide & merge.

Merge sort works with recursion and we shall see our implementation in the same way.

procedure mergesort( var a as array )

if ( n == 1 ) return a

var l1 as array = a[0] ... a[n/2]

var l2 as array = a[n/2+1] ... a[n]

l1 = mergesort( l1 )

l2 = mergesort( l2 )

return merge( l1, l2 )

end procedure

procedure merge( var a as array, var b as array )

var c as array

while ( a and b have elements )

if ( a[0] > b[0] )

add b[0] to the end of c

remove b[0] from b

else

add a[0] to the end of c

remove a[0] from a

end if

end while

while ( a has elements )

add a[0] to the end of c

remove a[0] from a

end while

while ( b has elements )

add b[0] to the end of c

remove b[0] from b

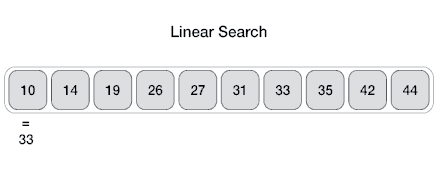
end while

return c

end procedure

**7. Explain about linear search?**

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.



## Algorithm

Linear Search ( Array A, Value x)

Step 1: Set i to 1

Step 2: if i > n then go to step 7

Step 3: if A[i] = x then go to step 6

Step 4: Set i to i + 1

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

## Pseudocode

procedure linear\_search (list, value)

for each item in the list

if match item == value

return the item's location

end if

end for

end procedure

**8. Explain about binary search?**

Binary search is a fast search algorithm with run-time complexity of Ο(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the subarray reduces to zero.

## How Binary Search Works?

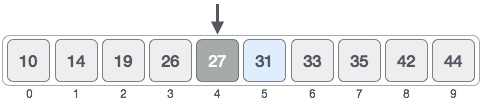
For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



First, we shall determine half of the array by using this formula −

mid = low + (high - low) / 2

Here it is, 0 + (9 - 0 ) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.



We change our low to mid + 1 and find the new mid value again.

low = mid + 1

mid = low + (high - low) / 2

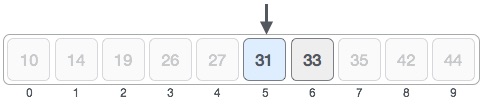
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.



Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

## Pseudocode

Procedure binary\_search

A ← sorted array

n ← size of array

x ← value to be searched

Set lowerBound = 1

Set upperBound = n

while x not found

if upperBound < lowerBound

EXIT: x does not exists.

set midPoint = lowerBound + ( upperBound - lowerBound ) / 2

if A[midPoint] < x

set lowerBound = midPoint + 1

if A[midPoint] > x

set upperBound = midPoint - 1

if A[midPoint] = x

EXIT: x found at location midPoint

end while

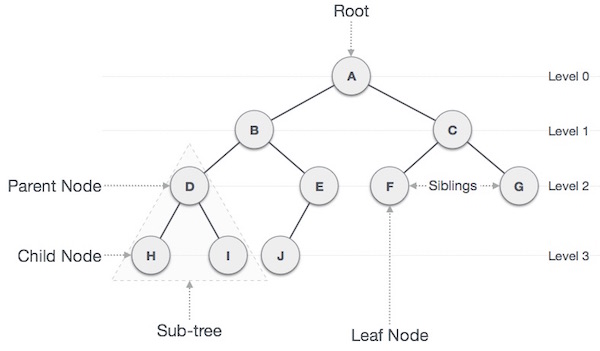
end procedure

**UNIT III**

**1. Explain about binary search tree representation?**

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.



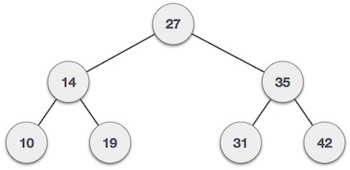
## Important Terms

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.



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.

struct node {

int data;

struct node \*leftChild;

struct node \*rightChild;

};

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

The implementation of this algorithm should look like this.

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;

}

}

**2. Explain about tree traversals?**

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 −

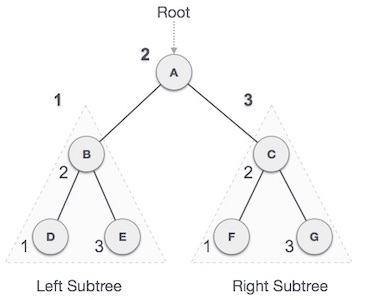
* In-order Traversal
* Pre-order Traversal
* Post-order Traversal

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.



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 −

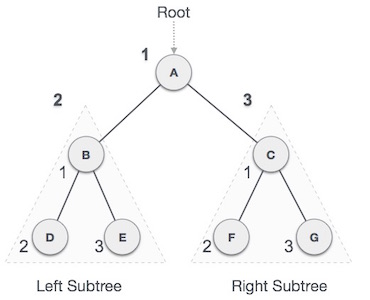
**Step 1** − Recursively traverse left subtree.

**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.



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 −

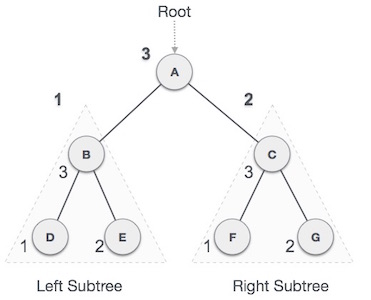
**Step 1** − Visit root node.

**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.



We start from **A**, and following pre-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.

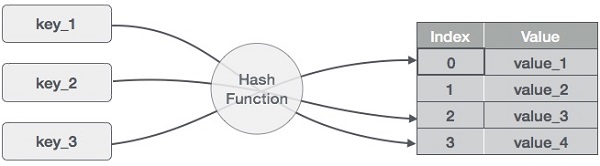
3. Explain about hashing?

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

## Hashing

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.



* (1,20)
* (2,70)
* (42,80)
* (4,25)
* (12,44)
* (14,32)
* (17,11)
* (13,78)
* (37,98)

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. No.** | **Key** | **Hash** | **Array Index** |
| 1 | 1 | 1 % 20 = 1 | 1 |
| 2 | 2 | 2 % 20 = 2 | 2 |
| 3 | 42 | 42 % 20 = 2 | 2 |
| 4 | 4 | 4 % 20 = 4 | 4 |
| 5 | 12 | 12 % 20 = 12 | 12 |
| 6 | 14 | 14 % 20 = 14 | 14 |
| 7 | 17 | 17 % 20 = 17 | 17 |
| 8 | 13 | 13 % 20 = 13 | 13 |
| 9 | 37 | 37 % 20 = 17 | 17 |

**4. Explain about linear probing in hashing?**

## Linear Probing

As we can see, it may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. No.** | **Key** | **Hash** | **Array Index** | **After Linear Probing, Array Index** |
| 1 | 1 | 1 % 20 = 1 | 1 | 1 |
| 2 | 2 | 2 % 20 = 2 | 2 | 2 |
| 3 | 42 | 42 % 20 = 2 | 2 | 3 |
| 4 | 4 | 4 % 20 = 4 | 4 | 4 |
| 5 | 12 | 12 % 20 = 12 | 12 | 12 |
| 6 | 14 | 14 % 20 = 14 | 14 | 14 |
| 7 | 17 | 17 % 20 = 17 | 17 | 17 |
| 8 | 13 | 13 % 20 = 13 | 13 | 13 |
| 9 | 37 | 37 % 20 = 17 | 17 | 18 |

## Basic Operations

Following are the basic primary operations of a hash table.

* **Search** − Searches an element in a hash table.
* **Insert** − inserts an element in a hash table.
* **delete** − Deletes an element from a hash table.

## DataItem

Define a data item having some data and key, based on which the search is to be conducted in a hash table.

struct DataItem {

int data;

int key;

};

## Hash Method

Define a hashing method to compute the hash code of the key of the data item.

int hashCode(int key){

return key % SIZE;

}

## Search Operation

Whenever an element is to be searched, compute the hash code of the key passed and locate the element using that hash code as index in the array. Use linear probing to get the element ahead if the element is not found at the computed hash code.

### Example

struct DataItem \*search(int key) {

//get the hash

int hashIndex = hashCode(key);

//move in array until an empty

while(hashArray[hashIndex] != NULL) {

if(hashArray[hashIndex]->key == key)

return hashArray[hashIndex];

//go to next cell

++hashIndex;

//wrap around the table

hashIndex %= SIZE;

}

return NULL;

}

## Insert Operation

Whenever an element is to be inserted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing for empty location, if an element is found at the computed hash code.

### Example

void insert(int key,int data) {

struct DataItem \*item = (struct DataItem\*) malloc(sizeof(struct DataItem));

item->data = data;

item->key = key;

//get the hash

int hashIndex = hashCode(key);

//move in array until an empty or deleted cell

while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1) {

//go to next cell

++hashIndex;

//wrap around the table

hashIndex %= SIZE;

}

hashArray[hashIndex] = item;

}

## Delete Operation

Whenever an element is to be deleted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing to get the element ahead if an element is not found at the computed hash code. When found, store a dummy item there to keep the performance of the hash table intact.

### Example

struct DataItem\* delete(struct DataItem\* item) {

int key = item->key;

//get the hash

int hashIndex = hashCode(key);

//move in array until an empty

while(hashArray[hashIndex] !=NULL) {

if(hashArray[hashIndex]->key == key) {

struct DataItem\* temp = hashArray[hashIndex];

//assign a dummy item at deleted position

hashArray[hashIndex] = dummyItem;

return temp;

}

//go to next cell

++hashIndex;

//wrap around the table

hashIndex %= SIZE;

}

return NULL;

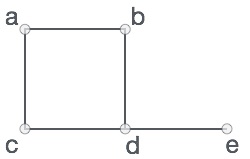
}

**UNIT V**

**1. Explain about graph data structure?**

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 −



In the above graph,

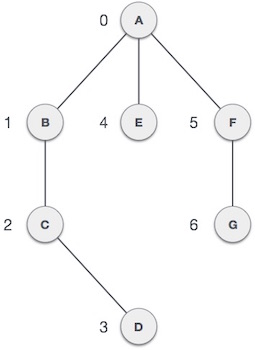
V = {a, b, c, d, e}

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.



## Basic Operations

Following are basic primary operations of a Graph −

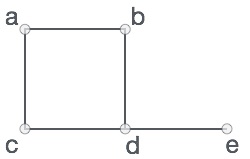
* **Add Vertex** − Adds a vertex to the graph.
* **Add Edge** − Adds an edge between the two vertices of the graph.
* **Display Vertex** − Displays a vertex of the graph.

To know more about Graph, please read [Graph Theory Tutorial](https://www.tutorialspoint.com/graph_theory/index.htm). We shall learn about traversing a graph in the coming chapters.

2. Explain about depth first search traversal?

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 −



In the above graph,

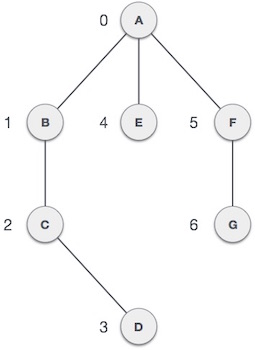
V = {a, b, c, d, e}

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.



## Basic Operations

Following are basic primary operations of a Graph −

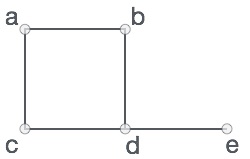
* **Add Vertex** − Adds a vertex to the graph.
* **Add Edge** − Adds an edge between the two vertices of the graph.
* **Display Vertex** − Displays a vertex of the graph.

To know more about Graph, please read [Graph Theory Tutorial](https://www.tutorialspoint.com/graph_theory/index.htm). We shall learn about traversing a graph in the coming chapters.

3. Explain about breadth first search traversal?

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 −



In the above graph,

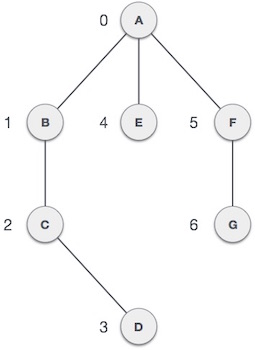
V = {a, b, c, d, e}

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.



## Basic Operations

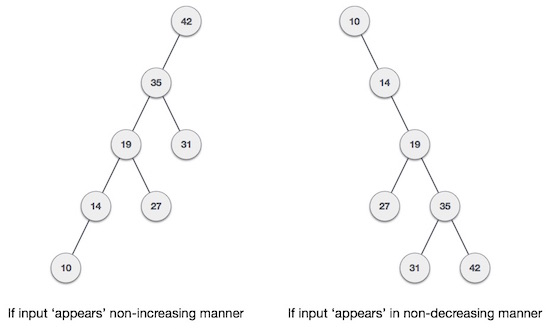
Following are basic primary operations of a Graph −

* **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.

To know more about Graph, please read [Graph Theory Tutorial](https://www.tutorialspoint.com/graph_theory/index.htm). We shall learn about traversing a graph in the coming chapters.

**4. Explain about AVL trees?**

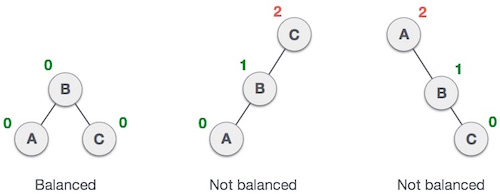
What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this −



It is observed that BST's worst-case performance is closest to linear search algorithms, that is Ο(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor **Adelson**, **Velski** & **Landis**, **AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the **Balance Factor**.

Here we see that the first tree is balanced and the next two trees are not balanced −



In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

***BalanceFactor*** = height(left-sutree) − height(right-sutree)

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

## AVL Rotations

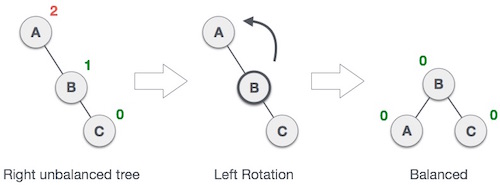
To balance itself, an AVL tree may perform the following four kinds of rotations −

* Left rotation
* Right rotation
* Left-Right rotation
* Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

### Left Rotation

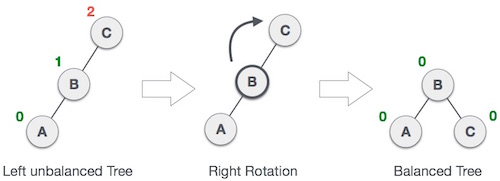
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −



In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.

## Right Rotation

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

### Left-Right Rotation

Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

|  |  |
| --- | --- |
| **State** | **Action** |
| Right Rotation | A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation. |
| Left Rotation | We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**. |
| Left Rotation | Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree. |
| Right Rotation | We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree. |
| Balanced Avl Tree | The tree is now balanced. |

### Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

|  |  |
| --- | --- |
| **State** | **Action** |
| Left Subtree of Right Subtree | A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2. |
| Subtree Right Rotation | First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**. |
| Right Unbalanced Tree | Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation. |
| Left Rotation | A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**. |
| Balanced AVL Tree | The tree is now balanced. |

5. Explain about binary search trees?

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

* The left sub-tree of a node has a key less than or equal to its parent node's key.
* The right sub-tree of a node has a key greater than to its parent node's key.

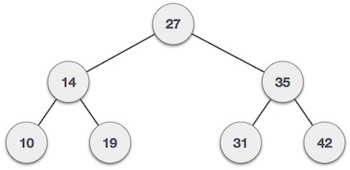
Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −

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 −



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;

}

}

}

}

}