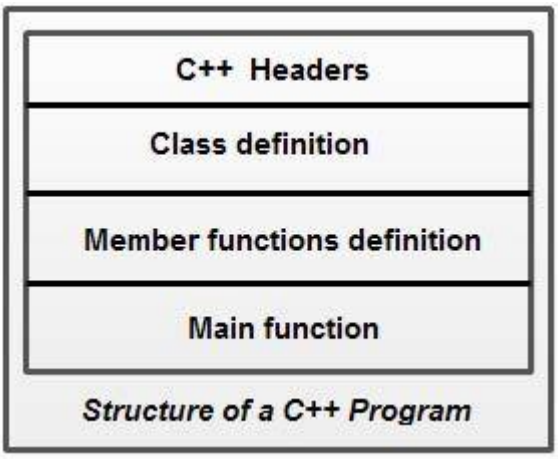
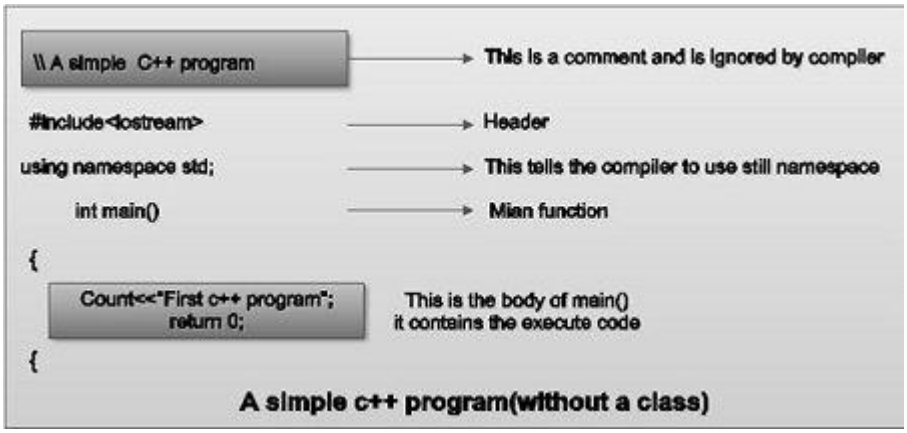
VTU Data structures using c++

Module-1

1a) Structure of C++ program





C++ supports two comment styles: single line comment and multiline comment. Single line comments are used to define line-by-line descriptions. Double slash *(//)* is used to represent single line comments. To understand the concept of single line comment, consider this statement.

b)Inheritance is one of four pillars of Object-Oriented Programming (OOPs). It is a feature that enables a class to acquire properties and characteristics of another class. Inheritance allows you to reuse your code since the derived class or the child class can reuse the members of the base class by inheriting them. Consider a real-life example to clearly understand the concept of inheritance. A child inherits some properties from his/her parents, such as the ability to speak, walk, eat, and so on. But these properties are not especially inherited in his parents only. His parents inherit these properties from another class called mammals. This mammal class again derives these characteristics from the animal class. Inheritance works in the same manner.

During inheritance, the data members of the base class get copied in the derived class and can be accessed depending upon the visibility mode used. The order of the accessibility is always in a decreasing order i.e., from public to protected. There

are mainly five types of Inheritance in C++ that you will explore in this article. They are as follows:

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

c)
```
#include<iostream>
using namespace std;
int main()
{
    int numOne, numTwo, larg;
    cout<<"Enter the Two Numbers: ";
    cin>>numOne>>numTwo;
    if(numOne>numTwo)
        larg = numOne;
    else
        larg = numTwo;
    cout<<"\nLargest = "<<larg;
    cout<<endl;
    return 0;
}
```

Using class

```
include<iostream>
using namespace std;
class CodesCracker
{
    public:
        int findLargest(int, int);
};
int CodesCracker::findLargest(int a, int b)
{
    if(a>b)
        return a;
    else
        return b;
}
int main()
{
    CodesCracker c;
    int numOne, numTwo, larg;
    cout<<"Enter the Two Numbers: ";
    cin>>numOne>>numTwo;
    larg = c.findLargest(numOne, numTwo);
    cout<<"\nLargest = "<<larg;
    cout<<endl;
    return 0;
```

OR

```
2a) If (condition) then:
     [Module A]
[End of If structure]
f (condition A), then:
     [Module A]
Else if (condition B), then:
     [Module B]
        ..
        ..
Else if (condition N), then:
     [Module N]
[End If structure]
Repeat for i = A to N by I:
       [Module]
[End of loop]
```

Module-2

3a) new operator

The new operator denotes a request for memory allocation on the Free Store. If sufficient memory is available, a new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

Syntax to use new operator

```
pointer-variable = new data-type;
```

Here, the pointer variable is the pointer of type data-type. Data type could be any built-in data type including array or any user-defined data type including structure and class.

Example:

```
// Pointer initialized with NULL
// Then request memory for the variable
int *p = NULL;
p = new int;
```

```
2b) #include <iostream>
```

```
using namespace std;
int fact(int n) {
  if ((n==0)||(n==1))
  return 1;
  else
  return n*fact(n-1);
}
int main() {
  int n = 4;
  cout<<"Factorial of "<<n<<" is "<<fact(n);
  return 0;
}
```

c) Call by value in C++

In call by value, **original value is not modified.**

In call by value, value being passed to the function is locally stored by the function parameter in stack memory location. If you change the value of function parameter, it is changed for the current function only. It will not change the value of variable inside the caller method such as main().

Let's try to understand the concept of call by value in C++ language by the example given below:

```
1. #include <iostream>
2. using namespace std;
3. void change(int data);
4. int main()
5. {
6. int data = 3;
7. change(data);
8. cout << "Value of the data is: " << data<< endl;
9. return 0;
10. }
11. void change(int data)
12. {
13. data = 5;
14. }
```

OR

```
// Combine declaration of pointer
// and their assignment
int *p = new int;
```

Initialize memory: We can also initialize the memory for built-in data types using a new operator. For custom data types, a constructor is required (with the data type as input) for initializing the value. Here's an example of the initialization of both data types :

```
pointer-variable = new data-type(value);
```

```
int* p = new int(25);
float* q = new float(75.25);

// Custom data type
struct cust
{
        int p;
        cust(int q) : p(q) {}
        cust() = default;
        //cust& operator=(const cust& that) = default;
};

int main()
{
        // Works fine, doesn't require constructor
        cust* var1 = new cust;

        //OR

        // Works fine, doesn't require constructor
```

```
        var1 = new cust();

        // Notice error if you comment this line
        cust* var = new cust(25);
        return 0;
}3b) int* p = new int(25);
float* q = new float(75.25);

// Custom data type
struct cust
{
        int p;
        cust(int q) : p(q) {}
        cust() = default;
        //cust& operator=(const cust& that) = default;
};

int main()
{
        // Works fine, doesn't require constructor
        cust* var1 = new cust;

        //OR

        // Works fine, doesn't require constructor
        var1 = new cust();

        // Notice error if you comment this line
        cust* var = new cust(25);
        return 0;
}
```

OR

4a)

Linear Linked list is the default linked list and a linear data structure in which data is not stored in contiguous memory locations but each data node is connected to the next data node via a pointer, hence forming a chain.

The element in such a linked list can be inserted in 2 ways:

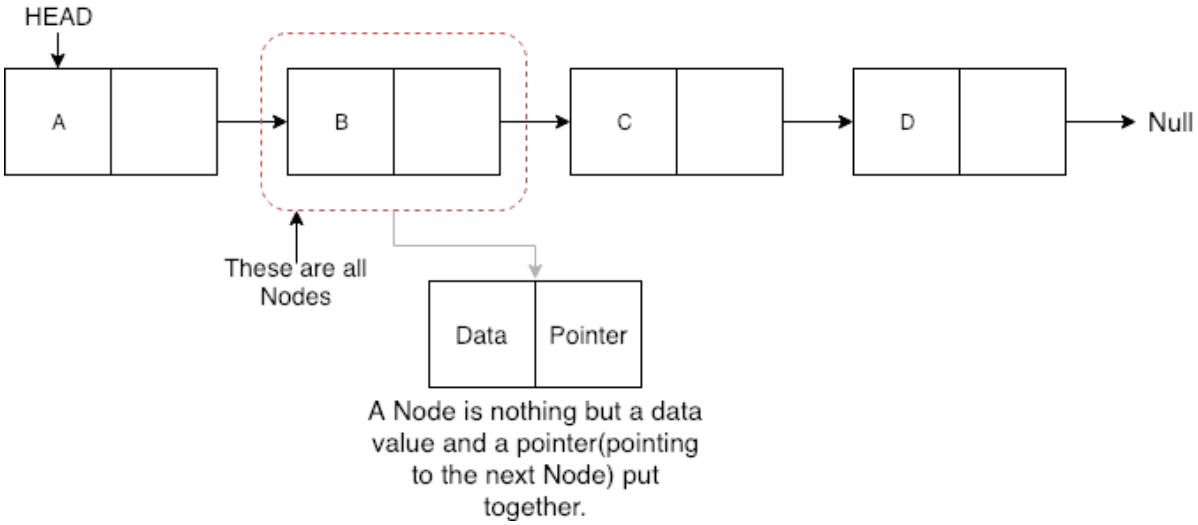
- Insertion at beginning of the list.
- Insertion at the end of the list.

Hence while writing the code for Linked List we will include methods to insert or add new data elements to the linked list, both, at the beginning of the list and at the end of the list.

We will also be adding some other useful methods like:

- Checking whether Linked List is empty or not.
- Searching any data element in the Linked List
- Deleting a particular Node(data element) from the List

Before learning how we insert data and create a linked list, we must understand the components forming a linked list, and the main component is the **Node**.




```
class Node
{
   public:
   // our linked list will only hold int data
   int data;
   //pointer to the next node
   node* next;

   // default constructor
   Node()
   {
      data = 0;
      next = NULL;
   }

   // parameterised constructor
   Node(int x)
   {
      data = x;
      next = NULL;
   }
}
```

4b)
```
#include <iostream>
using namespace std;

int main() {

 int numbers[5] = {7, 5, 6, 12, 35};

 cout << "The numbers are: ";

 //  Printing array elements
 // using range based for loop
 for (const int &n : numbers) {
  cout << n << "  ";
 }

 cout << "\nThe numbers are: ";

 //  Printing array elements
 // using traditional for loop
 for (int i = 0; i < 5; ++i) {
  cout << numbers[i] << "  ";
 }

 return 0;
}
```
4c)

A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have **0 value**, then it is called a sparse matrix.

Why to use Sparse Matrix instead of simple matrix ?

- **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.

- **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements..

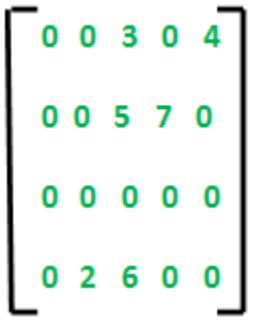
Sparse Matrix Representations can be done in many ways following are two common representations:

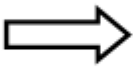
1. Array representation
2. Linked list representation

Method 1: Using Arrays:

2D array is used to represent a sparse matrix in which there are three rows named as

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)





Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

Module-3

5a)

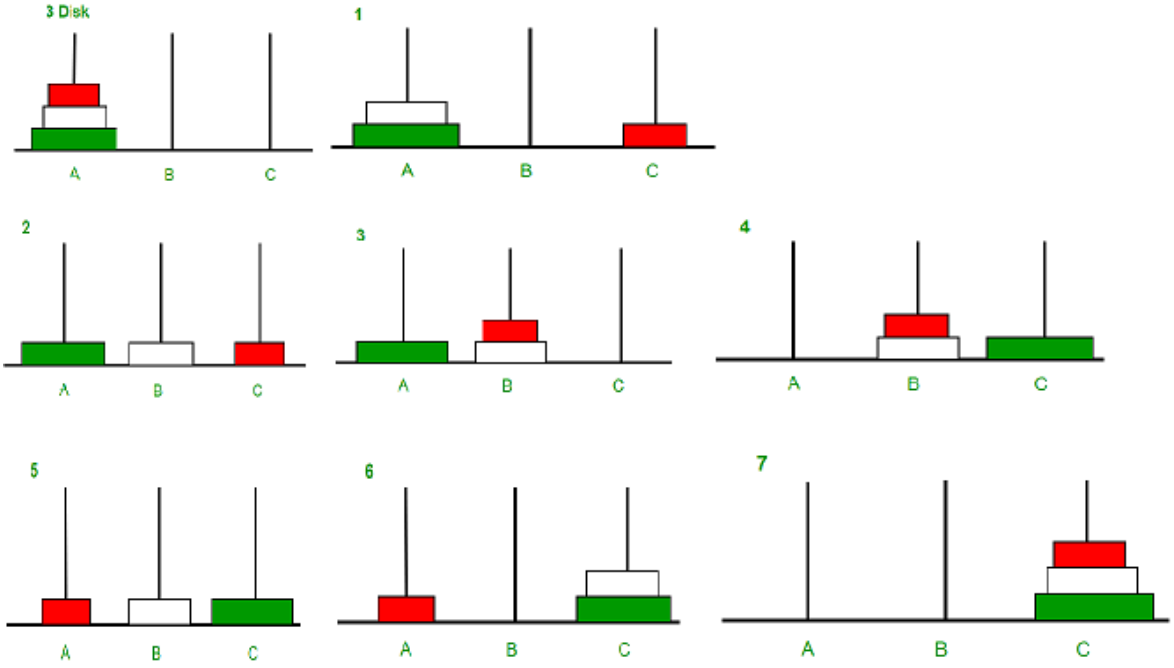
```
#include <iostream>
#include <stack>
using namespace std;
int main() {
        stack<int> stack;
        stack.push(21);
        stack.push(22);
        stack.push(24);
        stack.push(25);

                stack.pop();
        stack.pop();

        while (!stack.empty()) {
                cout << stack.top() <<" ";
```

```
            stack.pop();
        }
}
```

5c) Tower of Hanoi using Recursion:

The idea is to use the helper node to reach the destination using recursion. Below is the pattern for this problem:

- *Shift 'N-1' disks from 'A' to 'B', using C.*
- *Shift last disk from 'A' to 'C'.*
- *Shift 'N-1' disks from 'B' to 'C', using A.*




```
// C++ recursive function to
// solve tower of hanoi puzzle
#include <bits/stdc++.h>
using namespace std;

void towerOfHanoi(int n, char from_rod, char to_rod,
                                    char aux_rod)
{
        if (n == 0) {
                return;
        }
```

```
        towerOfHanoi(n - 1, from_rod, aux_rod, to_rod);
        cout << "Move disk " << n << " from rod " << from_rod
                << " to rod " << to_rod << endl;
        towerOfHanoi(n - 1, aux_rod, to_rod, from_rod);
}

// Driver code
int main()
{
        int N = 3;

        // A, B and C are names of rods
        towerOfHanoi(N, 'A', 'C', 'B');
        return 0;
}

// This is code is contributed by rathbhupendra
```

6a) o implement a stack using the singly linked list concept, all the singly linked list operations should be performed based on Stack operations LIFO(last in first out) and with the help of that knowledge, we are going to implement a stack using a singly linked list.
So we need to follow a simple rule in the implementation of a stack which is **last in first out** and all the operations can be performed with the help of a top variable. Let us learn how to perform **Pop, Push, Peek, and Display** operations in the following article:
Stack Operations:

- **push():** Insert a new element into the stack i.e just insert a new element at the beginning of the linked list.
- **pop():** Return the top element of the Stack i.e simply delete the first element from the linked list.
- **peek():** Return the top element.
- **display():** Print all elements in Stack.

Push Operation:

- *Initialise a node*
- *Update the value of that node by data i.e.* ***node->data = data***
- *Now link this node to the top of the linked list*

- *And update top pointer to the current node*

Pop Operation:

- *First Check whether there is any node present in the linked list or not, if not then return*
- *Otherwise make pointer let say **temp** to the top node and move forward the top node by 1 step*
- *Now free this temp node*

Peek Operation:

- *Check if there is any node present or not, if not then return.*
- *Otherwise return the value of top node of the linked list*

Display Operation:

- *Take a **temp** node and initialize it with top pointer*
- *Now start traversing temp till it encounters NULL*
- *Simultaneously print the value of the temp node*

6c) // CPP program to check for balanced brackets.

```
#include <bits/stdc++.h>

using namespace std;


// function to check if brackets are balanced

bool areBracketsBalanced(string expr)

{

	stack<char> s;

	char x;


	// Traversing the Expression

	for (int i = 0; i < expr.length(); i++)

	{

		if (expr[i] == '(' || expr[i] == '['

			|| expr[i] == '{')

		{

			// Push the element in the stack

			s.push(expr[i]);

			continue;

		}
```

```
// IF current current character is not opening
// bracket, then it must be closing. So stack
// cannot be empty at this point.
if (s.empty())
        return false;

switch (expr[i]) {
case ')':

        // Store the top element in a
        x = s.top();
        s.pop();
        if (x == '{' || x == '[')
                return false;
        break;

case '}':

        // Store the top element in b
        x = s.top();
        s.pop();
        if (x == '(' || x == '[')
                return false;
        break;

case ']':

        // Store the top element in c
        x = s.top();
        s.pop();
```

```
                        if (x == '(' || x == '{')
                                return false;
                        break;
                }
        }

        // Check Empty Stack
        return (s.empty());
}

// Driver code
int main()
{
        string expr = "{()}[]";

        // Function call
        if (areBracketsBalanced(expr))
                cout << "Balanced";
        else
                cout << "Not Balanced";
        return 0;
}
```

Module-4

7a) // CPP code to illustrate Queue in

// Standard Template Library (STL)

```
#include <iostream>
#include <queue>

using namespace std;
```

```
// Print the queue
void showq(queue<int> gq)
{
	queue<int> g = gq;
	while (!g.empty()) {
		cout << '\t' << g.front();
		g.pop();
	}
	cout << '\n';
}

// Driver Code
int main()
{
	queue<int> gquiz;
	gquiz.push(10);
	gquiz.push(20);
	gquiz.push(30);

	cout << "The queue gquiz is : ";
	showq(gquiz);

	cout << "\ngquiz.size() : " << gquiz.size();
	cout << "\ngquiz.front() : " << gquiz.front();
	cout << "\ngquiz.back() : " << gquiz.back();

	cout << "\ngquiz.pop() : ";
	gquiz.pop();
	showq(gquiz);

	return 0;
```

}

7b) **What is a Hash Function?**

A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in the hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as the index in the hash table.

A good hash function should have the following properties:

1. Efficiently computable.
2. Should uniformly distribute the keys (Each table position equally likely for each key)

For example: For phone numbers, a bad hash function is to take the first three digits. A better function is considered the last three digits. Please note that this may not be the best hash function. There may be better ways.

In practice, we can often employ ***heuristic techniques*** to create a hash function that performs well. Qualitative information about the distribution of the keys may be useful in this design process. In general, a hash function should depend on every single bit of the key, so that two keys that differ in only one bit or one group of bits (regardless of whether the group is at the beginning, end, or middle of the key or present throughout the key) hash into different values. Thus, a hash function that simply extracts a portion of a key is not suitable. Similarly, if two keys are simply digited or character permutations of each other *(such as 139 and 319)*, they should also hash into different values.

7c)
```
// CPP program to implement hashing with chaining

#include<bits/stdc++.h>

using namespace std;


class Hash

{

	int BUCKET; // No. of buckets


	// Pointer to an array containing buckets

	list<int> *table;

public:

	Hash(int V); // Constructor


	// inserts a key into hash table

	void insertItem(int x);
```

```
        // deletes a key from hash table
        void deleteItem(int key);

        // hash function to map values to key
        int hashFunction(int x) {
                return (x % BUCKET);
        }

        void displayHash();
};

Hash::Hash(int b)
{
        this->BUCKET = b;
        table = new list<int>[BUCKET];
}

void Hash::insertItem(int key)
{
        int index = hashFunction(key);
        table[index].push_back(key);
}

void Hash::deleteItem(int key)
{
// get the hash index of key
int index = hashFunction(key);

// find the key in (index)th list
list <int> :: iterator i;
```

```
for (i = table[index].begin();
                        i != table[index].end(); i++) {
            if (*i == key)
            break;
}

// if key is found in hash table, remove it
if (i != table[index].end())
            table[index].erase(i);
}

// function to display hash table
void Hash::displayHash() {
for (int i = 0; i < BUCKET; i++) {
            cout << i;
            for (auto x : table[i])
            cout << " --> " << x;
            cout << endl;
}
}

// Driver program
int main()
{
// array that contains keys to be mapped
int a[] = {15, 11, 27, 8, 12};
int n = sizeof(a)/sizeof(a[0]);

// insert the keys into the hash table
Hash h(7); // 7 is count of buckets in
                                    // hash table
```

```
for (int i = 0; i < n; i++)
        h.insertItem(a[i]);

// delete 12 from hash table
h.deleteItem(12);

// display the Hash table
h.displayHash();

return 0;
}
```

OR

8a) Dictionary ADT

Dictionary (map, association list) is a data structure, which is generally an association of unique keys with some values. One may bind a value to a key, delete a key (and naturally an associated value) and lookup for a value by the key. Values are not required to be unique. Simple usage example is an explanatory dictionary. In the example, words are keys and explanations are values.

Dictionary ADT

Operations

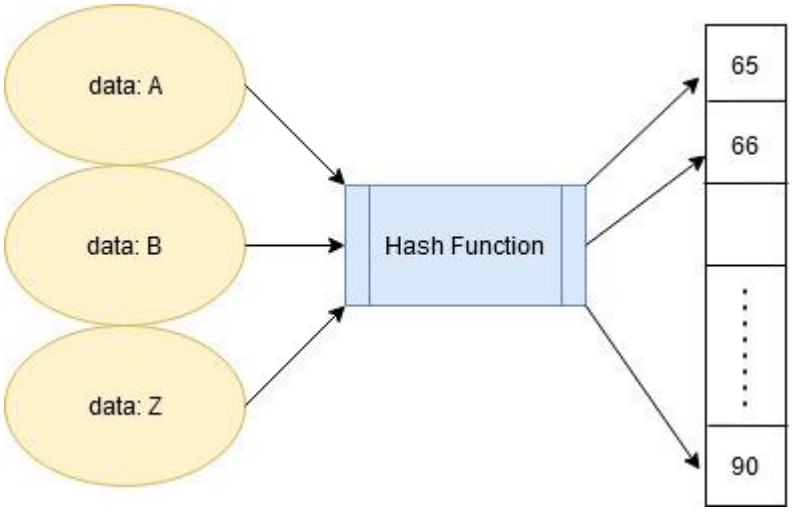
- **Dictionary create()**
  creates empty dictionary
- **boolean isEmpty(Dictionary d)**
  tells whether the dictionary **d** is empty
- **put(Dictionary d, Key k, Value v)**
  associates key **k** with a value **v;**
  if key **k** already presents in the dictionary
  old value is replaced by **v**
- **Value get(Dictionary d, Key k)**
  returns a value, associated with key **k**
  or null, if dictionary contains no such key

- **remove(Dictionary d, Key k)**
  removes key **k** and associated value

- **destroy(Dictionary d)**
  destroys dictionary **d** // Binary Search Tree operations in C++

8b) n this article, we have explored the idea of collision in hashing and explored different collision resolution techniques such as:

- Open Hashing (Separate chaining)
- Closed Hashing (Open Addressing)
- Liner Probing
- Quadratic probing
- Double hashing



9b) ***1) The maximum number of nodes at level 'l' of a binary tree is 2^l.***
Here level is the number of nodes on the path from the root to the node (including root and node). Level of the root is 0.
This can be proved by induction.
For root, l = 0, number of nodes = $2^0 = 1$
Assume that the maximum number of nodes on level 'l' is 2^l
Since in Binary tree every node has at most 2 children, next level would have twice nodes, i.e. $2 * 2^l$
2) The Maximum number of nodes in a binary tree of height 'h' is $2^h - 1$.
Here the height of a tree is the maximum number of nodes on the root to leaf path. Height of a tree with a single node is considered as 1.
This result can be derived from point 2 above. A tree has maximum nodes if all levels have maximum nodes. So maximum number of nodes in a binary tree of height h is $1 + 2 + 4 + .. + 2^{h-1}$. This is a simple geometric series with h

terms and sum of this series is $2^h - 1$.
In some books, the height of the root is considered as 0. In this convention, the above formula becomes $2^{h+1} - 1$

3) In a Binary Tree with N nodes, minimum possible height or the minimum number of levels is $Log_2(N+1)$.

There should be at least one element on each level, so the height cannot be more than N. A binary tree of height 'h' can have maximum $2^h - 1$ nodes (previous property). So the number of nodes will be less than or equal to this maximum value.

10a)

- 9c)

```
#include <iostream>
using namespace std;

struct node {
  int key;
  struct node *left, *right;
};

// Create a node
struct node *newNode(int item) {
  struct node *temp = (struct node *)malloc(sizeof(struct node));
  temp->key = item;
  temp->left = temp->right = NULL;
  return temp;
}

// Inorder Traversal
void inorder(struct node *root) {
  if (root != NULL) {
    // Traverse left
    inorder(root->left);

    // Traverse root
    cout << root->key << " -> ";

    // Traverse right
```

```
    inorder(root->right);
  }
}

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

  // Traverse to the right place and insert the node
  if (key < node->key)
    node->left = insert(node->left, key);
  else
    node->right = insert(node->right, key);

  return node;
}

// Find the inorder successor
struct node *minValueNode(struct node *node) {
  struct node *current = node;

  // Find the leftmost leaf
  while (current && current->left != NULL)
    current = current->left;

  return current;
}

// Deleting a node
struct node *deleteNode(struct node *root, int key) {
  // Return if the tree is empty
  if (root == NULL) return root;

  // Find the node to be deleted
  if (key < root->key)
    root->left = deleteNode(root->left, key);
  else if (key > root->key)
    root->right = deleteNode(root->right, key);
  else {
    // If the node is with only one child or no child
    if (root->left == NULL) {
```

```
      struct node *temp = root->right;
      free(root);
      return temp;
    } else if (root->right == NULL) {
      struct node *temp = root->left;
      free(root);
      return temp;
    }

    // If the node has two children
    struct node *temp = minValueNode(root->right);

    // Place the inorder successor in position of the node to be deleted
    root->key = temp->key;

    // Delete the inorder successor
    root->right = deleteNode(root->right, temp->key);
  }
  return root;
}

// Driver code
int main() {
  struct node *root = NULL;
  root = insert(root, 8);
  root = insert(root, 3);
  root = insert(root, 1);
  root = insert(root, 6);
  root = insert(root, 7);
  root = insert(root, 10);
  root = insert(root, 14);
  root = insert(root, 4);

  cout << "Inorder traversal: ";
  inorder(root);

  cout << "\nAfter deleting 10\n";
  root = deleteNode(root, 10);
  cout << "Inorder traversal: ";
  inorder(root);
}
```

10a) The term 'tree traversal' means traversing or visiting each node of a tree. There is a single way to traverse the linear data structure such as linked list, queue, and stack. Whereas, there are multiple ways to traverse a tree that are listed as follows -

- Preorder traversal
- Inorder traversal
- Postorder traversal

So, in this article, we will discuss the above-listed techniques of traversing a tree. Now, let's start discussing the ways of tree traversal.

Preorder traversal

This technique follows the 'root left right' policy. It means that, first root node is visited after that the left subtree is traversed recursively, and finally, right subtree is recursively traversed. As the root node is traversed before (or pre) the left and right subtree, it is called preorder traversal.

So, in a preorder traversal, each node is visited before both of its subtrees.

The applications of preorder traversal include -

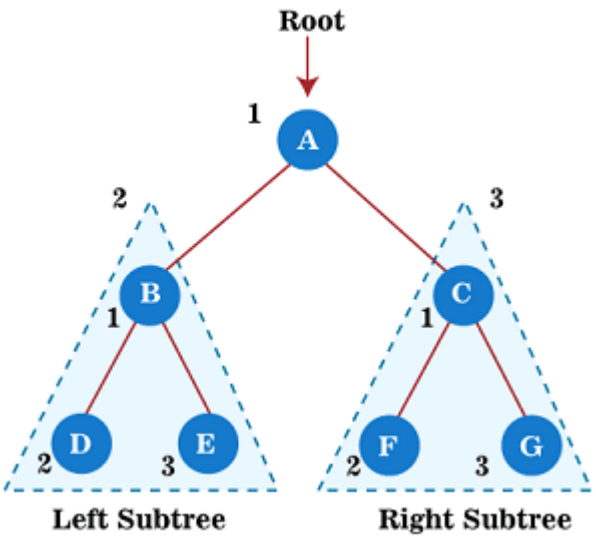
- It is used to create a copy of the tree.
- It can also be used to get the prefix expression of an expression tree.

Algorithm

1. Until all nodes of the tree are not visited
2.
3. Step 1 - Visit the root node
4. Step 2 - Traverse the left subtree recursively.
5. Step 3 - Traverse the right subtree recursively.

Example

Now, let's see the example of the preorder traversal technique.



Now, start applying the preorder traversal on the above tree. First, we traverse the root node **A;** after that, move to its left subtree **B**, which will also be traversed in preorder.

So, for left subtree B, first, the root node **B** is traversed itself; after that, its left subtree **D** is traversed. Since node **D** does not have any children, move to right subtree **E**. As node E also does not have any children, the traversal of the left subtree of root node A is completed.

Now, move towards the right subtree of root node A that is C. So, for right subtree C, first the root node **C** has traversed itself; after that, its left subtree **F** is traversed. Since node **F** does not have any children, move to the right subtree **G**. As node G also does not have any children, traversal of the right subtree of root node A is completed.

Therefore, all the nodes of the tree are traversed. So, the output of the preorder traversal of the above tree is -

A → B → D → E → C → F → G

To know more about the preorder traversal in the data structure, you can follow the link Preorder traversal.

Postorder traversal

This technique follows the 'left-right root' policy. It means that the first left subtree of the root node is traversed, after that recursively traverses the right subtree, and finally, the root node is traversed. As the root node is traversed after (or post) the left and right subtree, it is called postorder traversal.

So, in a postorder traversal, each node is visited after both of its subtrees.

The applications of postorder traversal include -

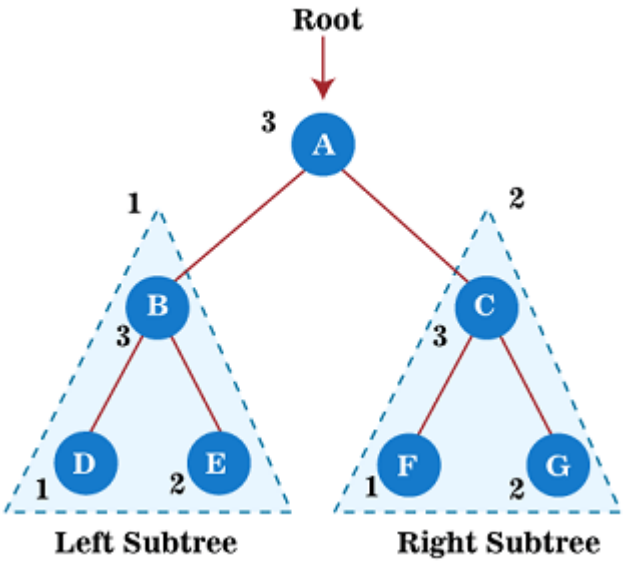
- It is used to delete the tree.
- It can also be used to get the postfix expression of an expression tree.

Algorithm

1. Until all nodes of the tree are not visited
2.
3. Step 1 - Traverse the left subtree recursively.
4. Step 2 - Traverse the right subtree recursively.
5. Step 3 - Visit the root node.

Example

Now, let's see the example of the postorder traversal technique.

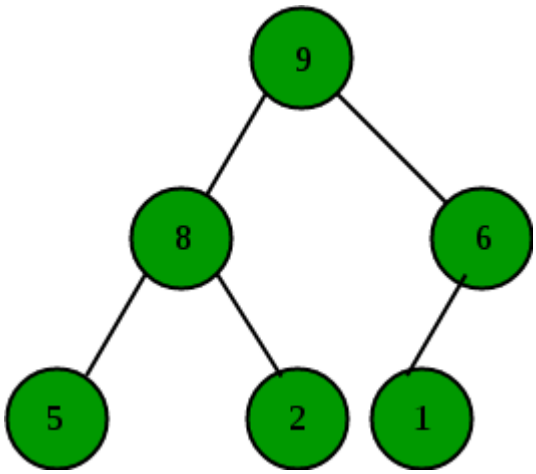


b) // Find height of a tree, defined by the root node

```
int tree_height(Node* root) {
   if (root == NULL)
      return 0;
   else {
      // Find the height of left, right subtrees
      left_height = tree_height(root->left);
```

```
    right_height = tree_height(root->right);


    // Find max(subtree_height) + 1 to get the height of the tree

    return max(left_height, right_height) + 1;

}
```

10c) A [max-heap](#) is a complete binary tree in which the value in each internal node is greater than or equal to the values in the children of that node. Mapping the elements of a heap into an array is trivial: if a node is stored an index k, then its left child is stored at index 2k + 1 and its right child at index 2k + 2.




```
include <iostream>
using namespace std;
void max_heap(int *a, int m, int n) {
  int j, t;
  t = a[m];
  j = 2 * m;
  while (j <= n) {
    if (j < n && a[j+1] > a[j])
      j = j + 1;
    if (t > a[j])
      break;
    else if (t <= a[j]) {
      a[j / 2] = a[j];
      j = 2 * j;
```

```
    }
  }
  a[j/2] = t;
  return;
}
void build_maxheap(int *a,int n) {
  int k;
  for(k = n/2; k >= 1; k--) {
    max_heap(a,k,n);
  }
}
int main() {
  int n, i;
  cout<<"enter no of elements of array
";
  cin>>n;
  int a[30];
  for (i = 1; i <= n; i++) {
    cout<<"enter elements"<<" "<<(i)<<endl;
    cin>>a[i];
  }
  build_maxheap(a,n);
  cout<<"Max Heap
";
  for (i = 1; i <= n; i++) {
    cout<<a[i]<<endl;
  }
}
```

Output

enter no of elements of array

5

enter elements 1

7

enter elements 2

6

enter elements 3

2
enter elements 4
1
enter elements 5
4
Max Heap
7
6
2
1
4