**Dr. D. Y. Patil Technical Campus**

**Dr. D. Y. Patil School of Engineering**

Dr. D. Y. Patil Knowledge City, Charholi (Bk), Lohegaon, Pune – 412 105

**Department of Computer Engineering**



**Data Structures and Algorithms**

**(S. E. Computer) 2019 Course**

**LABORATORY MANUAL**

|  |  |
| --- | --- |
| Prepared By  **Prof. Nilesh Mali**  **Prof. Ashwini Pandagale**  (Asst.Professor) | Approved By  **Dr.Pankaj Agarkar**  (H. O. D.) |

**Assignment 01**

**Problem Statement:**

Consider telephone book database of N clients. Make use of a hash table implementation to quickly look up client‘s telephone number. Make use of two collision handling techniques and compare them using number of comparisons required to find a set of telephone numbers

**Theory:**

Hash tables are an efficient implementation of a keyed array data structure, a structure sometimes known as an associative array or map. If you're working in C++, you can take advantage of the STL map container for keyed arrays implemented using binary trees, but this article will give you some of the theory behind how a hash table works.

**Keyed Arrays vs. Indexed Arrays**

One of the biggest drawbacks to a language like C is that there are no keyed arrays. In a normal C array (also called an indexed array), the only way to access an element would be through its index number. To find element 50 of an array named "employees" you have to access it like this:

1employees[50];

In a keyed array, however, you would be able to associate each element with a "key," which can be anything from a name to a product model number. So, if you have a keyed array of employee records, you could access the record of employee "John Brown" like this:

1employees["Brown, John"];

One basic form of a keyed array is called the hash table. In a hash table, a key is used to find an element instead of an index number. Since the hash table has to be coded using an indexed array, there has to be some way of transforming a key to an index number. That way is called the hashing function.

**Hashing Functions**

A hashing function can be just about anything. How the hashing function is actually coded depends on the situation, but generally the hashing function should return a value based on a key and the size of the array the hashing table is built on. Also, one important thing that is sometimes overlooked is that a hashing function has to return the same value every time it is given the same key.

Let's say you wanted to organize a list of about 200 addresses by people's last names. A hash table would be ideal for this sort of thing, so that you can access the records with the people's last names as the keys.

First, we have to determine the size of the array we're using. Let's use a 260 element array so that there can be an average of about 10 element spaces per letter of the alphabet.>

Now, we have to make a hashing function. First, let's create a relationship between letters and numbers:

A --> 0

B --> 1

C --> 2

D --> 3

...

and so on until Z --> 25.

The easiest way to organize the hash table would be based on the first letter of the last name.

Since we have 260 elements, we can multiply the first letter of the last name by 10. So, when a key like "Smith" is given, the key would be transformed to the index 180 (S is the 19 letter of the alphabet, so S --> 18, and 18 \* 10 = 180).

Since we use a simple function to generate an index number quickly, and we use the fact that the index number can be used to access an element directly, a hash table's access time is quite small. A linked list of keys and elements wouldn't be nearly as fast, since you would have to search through every single key-element pair.

**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;

}

**Expected Output**

Menu

1.Create Telephone book

2.Display

3.Look up

Enter Choice1

how many entries2

enter Namea

enter number1234567890

enter Named

enter number3216549876

do u want to continue?(1 for continue)1

Menu

1.Create Telephone book

2.Display

3.Look up

Enter Choice2

a 1234567890

0

0

do u want to continue?(1 for continue)1

Menu

1.Create Telephone book

2.Display

3.Look up

Enter Choice3

enter Name to searchd

found at 0

no of comparision1

do u want to continue?(1 for continue)0\*/

**Conclusion:** In this way we have implemented Hash table for quick lookup using Python.

**Assignment 02**

**Problem Statement:**

To create ADT that implement the "set" concept.

* Add (new Element) -Place a value into the set,
* Remove (element) Remove the value
* Contains (element) Return true if element is in collection,
* Size () Return number of values in collection Iterator () Return an iterator used to loop over collection,
* Intersection of two sets,
* Union of two sets,
* Difference between two sets,
* Subset

**Theory**

**Set Operations**

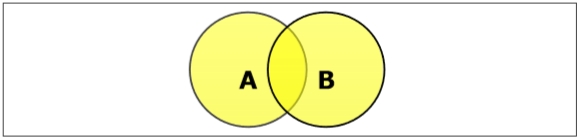
[Sets](https://www.edureka.co/blog/sets-in-python/) are a collection of unordered elements that are unique. Meaning that even if the data is repeated more than one time, it would be entered into the set only once. It resembles the sets that you have learnt in arithmetic. The operations also are the same as is with the arithmetic sets.

Set Operations include Set Union, Set Intersection, Set Difference, Complement of Set, and Cartesian Product.

**Set Union**

The union of sets A and B (denoted by A ∪ B) is the set of elements that are in A, in B, or in both A and B. Hence, A ∪ B = { x | x ∈ A OR x ∈ B }.

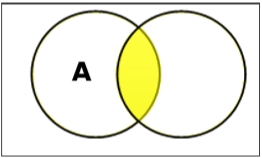
**Example** − If A = { 10, 11, 12, 13 } and B = { 13, 14, 15 }, then A ∪ B = { 10, 11, 12, 13, 14, 15 }. (The common element occurs only once)



### Set Intersection

The intersection of sets A and B (denoted by A ∩ B) is the set of elements which are in both A and B. Hence, A ∩ B = { x | x ∈ A AND x ∈ B }.

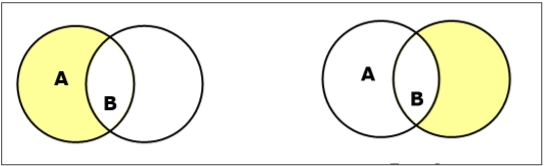
**Example** − If A = { 11, 12, 13 } and B = { 13, 14, 15 }, then A ∩ B = { 13 }.



### Set Difference/ Relative Complement

The set difference of sets A and B (denoted by A – B) is the set of elements that are only in A but not in B. Hence, A - B = { x | x ∈ A AND x ∉ B }.

**Example** − If A = { 10, 11, 12, 13 } and B = { 13, 14, 15 }, then (A - B) = { 10, 11, 12 } and (B - A) = { 14, 15 }. Here, we can see (A - B) ≠ (B - A)

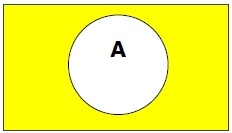


### Complement of a Set

The complement of a set A (denoted by A’) is the set of elements which are not in set A. Hence, A' = { x | x ∉ A }.

More specifically, A'= (U - A) where **U** is a universal set that contains all objects.

**Example** − If A = { x | x belongs to set of odd integers } then A' = { y | y does not belong to set of odd integers }



### Cartesian Product / Cross Product

The Cartesian product of n number of sets A1, A2, ... An denoted as A1 × A2 ... × An can be defined as all possible ordered pairs (x1, x2, ... xn) where x1 ∈ A1, x2 ∈ A2, ... xn ∈ A\_n

**Example** − If we take two sets A = { a, b } and B = { 1, 2 },

The Cartesian product of A and B is written as − A × B = { (a, 1), (a, 2), (b, 1), (b, 2)}

The Cartesian product of B and A is written as − B × A = { (1, a), (1, b), (2, a), (2, b)}

#### **Creating a set**

Sets are created using the flower braces but instead of adding key-value pairs, you just pass values to it.

|  |  |
| --- | --- |
| 1  2 | my\_set = {1, 2, 3, 4, 5, 5, 5} #create set  print(my\_set) |

**Output:**  
{1, 2, 3, 4, 5}

#### **Adding elements**

To add elements, you use the add() function and pass the value to it.

|  |  |
| --- | --- |
| 1  2  3 | my\_set = {1, 2, 3}  my\_set.add(4) #add element to set  print(my\_set) |

**Output:**  
{1, 2, 3, 4}

#### **Operations in sets**

The different operations on set such as union, intersection and so on are shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | my\_set = {1, 2, 3, 4}  my\_set\_2 = {3, 4, 5, 6}  print(my\_set.union(my\_set\_2), '----------', my\_set | my\_set\_2)  print(my\_set.intersection(my\_set\_2), '----------', my\_set & my\_set\_2)  print(my\_set.difference(my\_set\_2), '----------', my\_set - my\_set\_2)  print(my\_set.symmetric\_difference(my\_set\_2), '----------', my\_set ^ my\_set\_2)  my\_set.clear()  print(my\_set) |

* The union() function combines the data present in both sets.
* The intersection() function finds the data present in both sets only.
* The difference() function deletes the data present in both and outputs data present only in the set passed.
* The symmetric\_difference() does the same as the difference() function but outputs the data which is remaining in both sets.

**Output:**  
{1, 2, 3, 4, 5, 6} ———- {1, 2, 3, 4, 5, 6}  
{3, 4} ———- {3, 4}  
{1, 2} ———- {1, 2}  
{1, 2, 5, 6} ———- {1, 2, 5, 6}  
set()

**Conclusion:** In this way we have implemented set concept using Python.

**Assignment 03**

**Problem Statement:**

A book consists of chapters, chapters consist of sections and sections consist of subsections. Construct a tree and print the nodes.

**Theory**

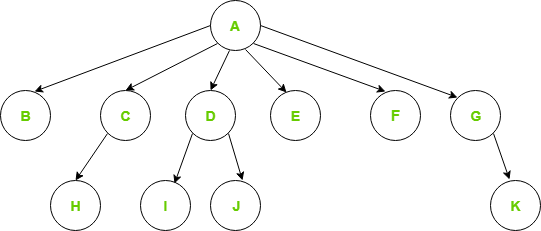
Generic trees are a collection of nodes where each node is a data structure that consists of records and a list of references to its children(duplicate references are not allowed). Unlike the linked list, each node stores the address of multiple nodes. Every node stores address of its children and the very first node’s address will be stored in a separate pointer called root.

The Generic trees are the N-ary trees which have the following properties:

            1. Many children at every node.

            2. The number of nodes for each node is not known in advance.

**Example:** 

**

*Generic Tree*

To represent the above tree, we have to consider the worst case, that is the node with maximum children (in above example, 6 children) and allocate that many pointers for each node.  
The node representation based on this method can be written as:

//Node declaration

struct Node{

int data;

struct Node \*firstchild;

struct Node \*secondchild;

struct Node \*thirdchild;

struct Node \*fourthchild;

struct Node \*fifthchild;

struct Node \*sixthchild;

}

**Disadvantages of the above representation are:**

1. **Memory Wastage** – All the pointers are not required in all the cases. Hence, there is lot of memory wastage.
2. **Unknown number of children** – The number of children for each node is not known in advance.

**Simple Approach:**

For storing the address of children in a node we can use an array or linked list. But we will face some issues with both of them.

1. In **Linked list**, we can not randomly access any child’s address. So it will be expensive.
2. In **array**, we can randomly access the address of any child, but we can store only fixed number of children’s addresses in it.

**Better Approach:**

We can use [Dynamic Arrays](https://www.geeksforgeeks.org/how-do-dynamic-arrays-work/) for storing the address of children’s address. We can randomly access any child’s address and the size of the vector is also not fixed.

//Node declaration

struct Node{

int data;

vector<Node\*> children;

}

#### Efficient Approach:

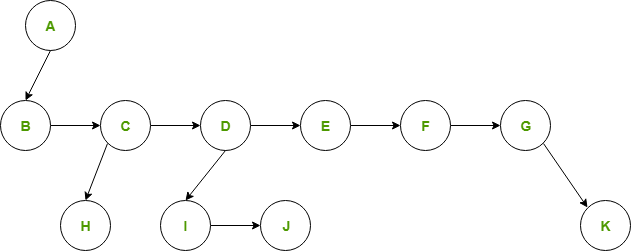
First child / Next sibling representation

 In the first child/next sibling representation, the steps taken are:

At each node-link the children of the same parent(siblings) from left to right.

* Remove the links from parent to all children except the first child.

Since we have a link between children, we do not need extra links from parents to all the children. This representation allows us to traverse all the elements by starting at the first child of the parent.

**

*FIRST CHILD/NEXT SIBLING REPRESENTATION*

The node declaration for first child / next sibling representation can be written as:   
//Node declaration

struct Node{

int data;

struct Node \*firstChild;

struct Node \*nextSibling;

}

**Advantages:**

* Memory efficient – No extra links are required, hence a lot of memory is saved.
* Treated as binary trees – Since we are able to convert any generic tree to binary representation, we can treat all generic trees with a first child/next sibling representation as binary trees. Instead of left and right pointers, we just use firstChild and nextSibling.
* Many algorithms can be expressed more easily because it is just a binary tree.
* Each node is of fixed size no auxiliary array or vector is required.

**Assignment 04**

**Problem Statement:**

Beginning with an empty binary search tree, Construct binary search tree by inserting the values in the order given. After constructing a binary tree -

i. Insert new node

ii. Find number of nodes in longest path

iii. Minimum data value found in the tree

iv. Change a tree so that the roles of the left and right pointers are swapped at every node

v. Search a value

Implementation:

A picture containing chart

Description automatically generated Diagram

Description automatically generated

A node in Binary Search tree will be represented as follows:



* Declare a structure to represent a node in the binary tree

Struct BinTree

{

struct BinTree \*left;

char data;

struct BinTree \*right;

};

* Display a menu with follwing options:
  1. Recursive Create
  2. Non – recursive create
  3. Insert New Node
  4. Find Height of the tree
  5. Smallest node value in the tree
  6. Mirror Image of the tree
  7. Search Value
     + Read the choice & switch according to that
     + If the choice is 1 or 2 create the root node & then give call to create function
     + Otherwise pass the root node as the input parameter to the function

**Rcreate ( ) function:**

* Main ( ) function has created the root node.
* Display the data for root node.
* Ask user if the new node is to be added to the left.
* If choice is yes
  + Create a new node
  + Read the data for new node
  + Initialize left & right pointers of the new node to NULL.
  + Attach this new node to the left of the root
  + Give recursive call to Rcreate ( ) function with root->left as the new root.
* Display the data for root node.
* Ask user if the new node is to be added to the right.
* If choice is yes
  + Create a new node
  + Read the data for new node
  + Initialize left & right pointers of the new node to NULL.
  + Attach this new node to the right of the root
  + Give recursive call to Rcreate ( ) function with root->right as the new root.
* End of Rcreate()

**Nrcreate ( ) function**

* Main ( ) function has created the root node.
* Let temp & new node be two node structures

while(1)

begin

initialize temp to point to Root

create a new node

initialize left & right pointer of new node to NULL

accept data for new node

while(1)

begin

If newnode->data < temp->data

If temp->left != NULL then

temp = temp->left

else

temp->left = newnode

break

If newnode->data > temp->data

If temp->right != NULL then

temp = temp->right

else

temp->right = newnode

break

End

Ask user if more nodes are to be added to the tree

If the ‘no’ break;

End

* End of Nrcreate( )

## Find Height of the tree-

There are two conventions to define height of Binary Tree  
1) Number of nodes on longest path from root to the deepest node.  
2) Number of edges on longest path from root to the deepest node.

In this post, the first convention is followed. For example, height of the below tree is 3.

Diagram

Description automatically generated

Recursive method to find height of Binary Tree is discussed [here](http://www.geeksforgeeks.org/write-a-c-program-to-find-the-maximum-depth-or-height-of-a-tree/). How to find height without recursion? We can use level order traversal to find height without recursion. The idea is to traverse level by level. Whenever move down to a level, increment height by 1 (height is initialized as 0). Count number of nodes at each level; stop traversing when count of nodes at next level is 0.

**Recursive Function-**

int btree::nheight(node \*root)

{

int i, j, max=0;

i=1,j=1;

if(root!=NULL)

{

i=i+nheight(root->left);

j=j+nheight(root->right);

if(i>j)

max=i;

else

max=j;

}

return(max);

}

**Non-recursive function-**

int BinaryTree :: TreeHeight(TreeNode \*Root)

{

int heightL, heightR;

if(Root == Null)

return 0;

if(Root->Lchild == Null && Root->Rchild == Null)

return 0;

heightL = TreeHeight(Root->Lchild);

heightR = TreeHeight(Root->Rchild);

if(heightR > heightL)

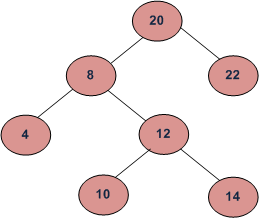
return(heightR + 1);

return(heightL + 1);

}

## Find the node with minimum value in a Binary Search Tree-

This is quite simple. Just traverse the node from root to left recursively until left is NULL. The node whose left is NULL is the node with minimum value.



For the above tree, we start with 20, then we move left 8, we keep on moving to left until we see NULL. Since left of 4 is NULL, 4 is the node with minimum value.

In binary search tree, the smallest node is in the left side and the largest node is in the right side. To find the smallest node, the process will check the parent node. In case that the parent node is not empty, if it doesn't have a left child node, the smallest node is the parent node; otherwise the smallest node is its left child node.

**Find Smallest Node function-( Non-recursive)**

void btree::smallest(node\* root)

{

node \*temp;

temp=root;

while(temp->left!=NULL)

temp=temp->left;

cout<<"\nMinimum data value="<<temp->data;

}

**Find Largest Node function-( Non-recursive)**

void btree::largest(node\* root)

{

node \*temp;

temp=root;

while(temp->right!=NULL)

temp=temp->right;

cout<<"\nMaximum data value="<<temp->data;

}

## Getting Mirror, Replica, or Tree Interchange of Binary Tree

The Mirror() operation finds the mirror of the tree that will interchange all left and right

subtrees in a linked binary tree.

A picture containing watch

Description automatically generated

**Recursive Function**

void BinaryTree :: Mirror(TreeNode \*Root)

{

TreeNode \*Tmp;

if(Root != Null)

{

Tmp = Root->Lchild;

Root->Lchild = Root->Rchild;

Root->Rchild = Tmp;

Mirror(Root->Lchild);

Mirror(Root->Rchild);

}

}

**Mirror( ) Function (Non-Recursive) using Queue-**

Initialize a queue of nodes as empty

Initialize a pointer temp = root

Add(temp) to queue

while( queue not empty)

begin

temp = delete

if (temp->left != NULL)

add(temp->left)

if(temp->right != NULL)

add (temp->right)

change = temp->left

temp->Left = temp->right

temp->right = change

end

end

**Searching for a Key-**

To search for a target key, we first compare it with the key at the root of the tree. If it is the same, then the algorithm ends. If it is less than the key at the root, search for the target key in the left subtree, else search in the right subtree. Let us, for example, search for the key ‘Saurabh’ in following Figure

Diagram

Description automatically generated

We first compare ‘Saurabh’ with the key of the root, ‘Jyoti’. Since ‘Saurabh’ comes after ‘Jyoti’ in alphabetical order, we move to the right side and next compare it with the key ‘Rekha’. Since ‘Saurabh’ comes after ‘Rekha’, we move to the right again and compare with ‘Teena’. Since ‘Saurabh’ comes before ‘Teena’, we move to the left. Now the question is to identify what event will be the terminating condition for the search. The solution is if we find the key, the function finishes successfully. If not, we continue searching until we hit an empty subtree.

Program Code shows the implementation of search () function, both nonrecursive and recursive implementations.

**Non-recursive-**

TreeNode \*BSTree :: Search(int Key)

{

TreeNode \*Tmp = Root;

while(Tmp)

{

if(Tmp->Data == Key)

return Tmp;

else if(Tmp->data < Key)

Tmp = Tmp->Lchild;

else

Tmp = Tmp->Rchild;

}

return NULL;

}

**Recursive-**

TreeNode \*BSTree :: Rec\_Search(TreeNode \*root, int key)

{ if(root == Null)

return(root);

else

{

if(root->Data < Key)

root = Rec\_Search(root->Lchild);

else if(root->data > Key)

root = Rec\_Search(root->Rchild);

}

}

**Conclusion:** Successfully implemented Binary search tree as an ADT using linked list in C++ language.

**Assignment : 5**

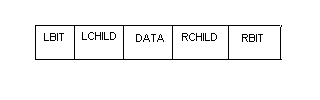
**Problem Statement:**

Write a program in C++ to create a binary inorder threaded tree & perform following tree traversals

1. In – order traversals
2. Pre – order traversals
3. Post – order traversals

**Implementation:**

A node in threaded binary tree is represented as follows:



LBIT & RBIT are the two extra fields used to distinguish between threads & normal pointers.

LBIT(P) = 1 if LCHILD is a normal pointer

LBIT(P) = 0 if LCHILD is a thread

RBIT(P) = 1 if RCHILD is a normal pointer

RBIT(P) = 0 if RCHILD is a thread

Two NULL links LCHILD of leftmost node & RCHILD of right most node point to head node.

We consider that the complete tree is left subtree of head node.

Graphical user interface, application

Description automatically generated

In Threaded Binary Tree

Memory representation of Threaded inorder tree equivalent to given binary tree is:

Diagram, engineering drawing

Description automatically generated

**Create ( ) Function:**

Getnode (parent)

Accept from user Data (parent)

Left(parent) = Head

Right(parent) = Head

Lbit(parent) = Thread

Rbit(parent) = Thread

Left(Head) = parent

while(1)

begin

Ask user if more nodes are to be added to tree

If no then break

Getnode (newnode)

Accept from user Data(newnode)

Initialize Lbit (newnode) = Rbit (newnode) = Thread

Initialize parent = left(head)

while(1)

begin

Ask user if newnode is left or right of current parent

If left

If Lbit (parent) != thread then

Parent = left (parent)

Else

Left(newnode) = left(parent)

Right(newnode) = parent

Left(parent) = newnode

Lbit(parent) = pointer

Break

If right

If Rbit (parent) != thread then

Parent = Right (parent)

Else

Right (newnode) = Right (parent)

Left(newnode) = parent

Right (parent) = newnode

Rbit(parent) = pointer

Break

End

End

End of create

**Tinorder( ) function:**

Let temp be a reference to a node of threaded tree

Initialize temp to point to head

while(1)

begin

temp = insucc(temp)

if temp = head then break

print data(temp)

end

end tinorder

insucc( ) function:

Let temp be a reference to a neode

Initialize temp = right (node)

If Rbit[node] = pointer then

while (lbit[temp] != thread)

begin

temp left(temp)

end

return(temp)

end insucc

**Tpreorder( ) function:**

Let temp be a reference to a node structure

Initialize temp = left(head)

while( temp != head)

begin

while(1)

begin

print Data(temp)

if Lbit(temp) = Thraed then break

temp = left(temp)

end

while(Rbit(temp) != pointer)

begin

temp = right(temp)

end

temp = right(temp)

end

end Tpreorder

**Tpostorder( ) Function:**

Let temp be a reference to a node structure

Let op be a string

Let i be a counter

Initialize temp = left(head)

Initialize i = 0

while(1)

begin

while(1)

begin

op[i] = Data(temp)

i = i+1

if (rbit(temp) = Thread) then break

temp = right(temp)

end

while(Lbit(temp) != pointer)

begin

temp = left(temp)

end

if temp = head then break

temp = left(temp)

end

print op in reverse order

end of Tpostorder

**Conclusion:** Successfully implemented Threaded Binary tree as an ADT using linked list in C++ language.

**Assignment 6:**

**Problem Statement:**

There are flight paths between cities. If there is a flight between city A and city B then there is an edge between the cities. The cost of the edge can be the time that flight take to reach city B from A, or the amount of fuel used for the journey. Represent this as a graph. The node can be represented by airport name or name of the city. Use adjacency list representation of the graph or use adjacency matrix representation of the graph.

Check whether the graph is connected or not.

**Theory:**

**Definition:** A set of items connected by edges. Each item is called a vertex or node. Formally, a graph is a set of vertices and a binary relation between vertices, adjacency.

In computer science, a **graph** is a kind of data structure, specifically an abstract data type (ADT) that consists of a set of nodes (also called vertices) and a set of edges that establish relationships (connections) between the nodes. The graph ADT follows directly from the graph concept from mathematics.

Informally, *G=(V,E)* consists of *vertices*, the elements of *V*, which are connected by *edges*, the elements of *E*. Formally, a graph, *G*, is defined as an ordered pair, *G=(V,E)*, where *V* is a set (usually finite) and *E* is a set consisting of two element subsets of *V*.The graphs can be represented in two ways. One is adjacency matrix and adjacency list.

For example, let us consider the following graph

A----------->B

| ^

| |

| |

V |

C ------------

**Adjacency Matrix**

A B C

A 0 1 1

B 0 0 0

C 0 1 0

**Adjacency List**

A ----> | B | ----> | C | ---- NULL

B ----> ---- NULL

C ----> | B | ---- NULL

**Structural Representation:**

typedef struct graph

{

int v;

struct graph \*l;

}NODE;

**Implementation:**

**Admatrix( ) Function:**

Let the adjacency matrix be M[n][n]

Initialize all elements of M to No edge

Initialize counter i =0

while( i < numedges)

begin

row = set(i)

col = set(i +1)

i = i +2

M[row][col] = edge

M[col][row] = edge

End

End

**Adlist( ) Function:**

Let Headlist be a list of head nodes

Let temp be a pointer to a node structure

Initialize counter i =0

Initialize Headlist such that

For i = 0 to i = max do

Vertex [Headlist[i]] = I

Next[Headlist[i]] = NULL

while(i < numedges)

begin

row = set[i]

col = set[i + 1]

getnode(newnode)

vertex[newnode] = col

next[newnode] = NULL

previous = Headlist[row]

temp = Next[previous]

while(temp !=NULL & vertex[temp]<=col)

begin

if vertx[temp] = col then

status = repeat

break

previous = temp

temp = next[temp]

end

if status != repeat

next[newnode] = next[previous]

next[previous] = newnode

else

status = not repeat

getnode(newnode)

vertex [newnode] = row

next[newnode] = NULL

previous = Headlist[col]

temp = next[previous]

while(temp != NULL & vertx[temp] <= row)

begin

if vertex[temp] = row then

status = repeat

break

previous = temp

temp = next[temp]

end

if status != repeat

next[newnode] = next[previos]

next[previous] newnode

else

status = not repeat

i = i + 2

end

return(Headlist)

end

**DFSdisplay( ) function:**

Initialize a visit array visit[1:n] to notvisit

Let temp be a pointer to a node

temp = Headlist(p)

while(temp!= NULL)

begin

if (visit[vertex[temp]]!= visit] then

print vertex[temp]

visit[vertex[temp]] = visited

DFSdisplay(vertex[temp]]

Temp next[temp]

End

End

**BFSdisplay( ) Function:**

Initialize a visit array visit[1:n] to notvisit

Let temp be a pointer to a node

temp = Headlist(p)

Initialize a Visit[1:n] array as notvisit for allvertices

Add(temp)

while(queue is not empty)

begin

temp = delete

temp = Headlist[vertex[temp]]

if (visit[vertex[temp]] = notvisit

print vertex[temp]

visit[vertex[temp]] = visit

temp = next [temp]

while(temp != NULL)

begin

if visit [vertex[temp]] = notvisit

add(temp)

temp next[temp]

end

end

**Input: Output: Adjacency matrix**

2

3

1

4

5

0

1

0

1

0

1

0

0

0

1

0

0

0

0

1

1

0

0

0

1

0

1

1

1

0

**Input: Output: Adjacency List**

2

3

1

4

5

aList[1] = (2,4)

aList[2] = (1,5)

aList[3] = (5)

aList[4] = (5,1)

aList[5] = (2,4,3)

**Conclusion:**

Hence we have studied representation of graph using adjacency matrix, adjacency list and traversal of graph.

**Assignment 07**

**Problem Statement:**

You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. Solve the problem by suggesting appropriate data structures.

**Theory:**

**Prim's algorithm** is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

**Implementation:**

**Prim( ) Function:**

Find edge with minimum weight

k = row no of edge with minimum weight

l = col no. of edge with minimum weight

assign k to T[1][1] and l to T[1][2]

for i-1 to n do

if cost[I][l] is less than cost[i][k]

then assign l to near[i]

else assign k to near[i]

set near[k] = near[l] = -1 and i =2

minw = 9999

for j = 1 to n

If near [j] != -1

If cost[j][near[j]] < minw then

minw = cost[j][near[j]]

minj = j

Assign j to T[i][1]

Assign near[j] to T[i][2]

Set near[j] = -1

For k =1 to n

If near[k] != -1 and if cost[k][near[k]> cost[k][j] then

near[k]= j

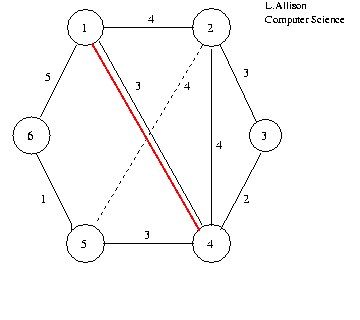
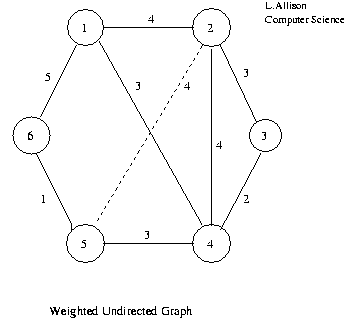
increment i by 1

If i is less than or equal to n goto step 5

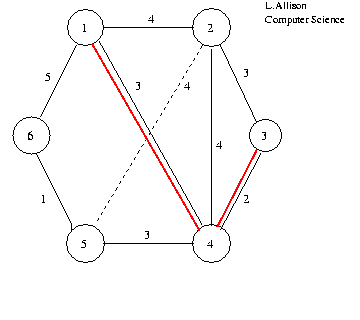
Display T arry and return

End of prim

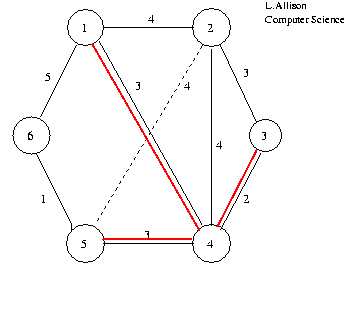
**Input/Output (Prim’s algorithm):**



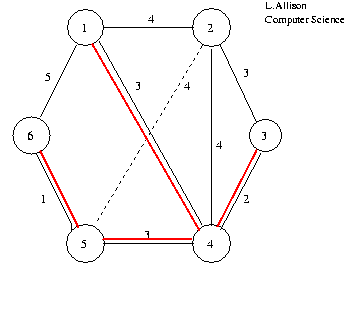
v3 is closest to tree . . .



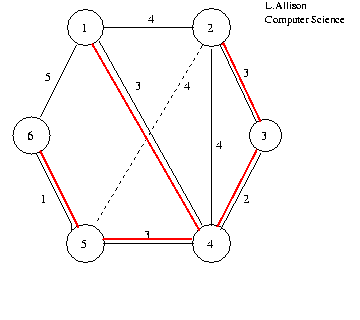
v2 and v5 are closest to tree,   pick v5, say . . .



v6 is closest to tree . . .



v2 is closest (and only remaining) vertex . . .



**Conclusion:**

Hence we have implemented Minimum spanning tree using Prim’s Algorithm

**Assignment: 08**

Given sequence k = k1 <k2 < … < kn of n sorted keys, with a search probability pi for each key ki. Build the Binary search tree that has the least search cost given the access probability for each key?

**Objectives:**

1. To understand concept of OBST.

2. To understand concept & features like extended binary search tree.

**Learning Objectives:**

* To understand concept of OBST.
* To understand concept & features like extended binary search tree.

**Learning Outcome:**

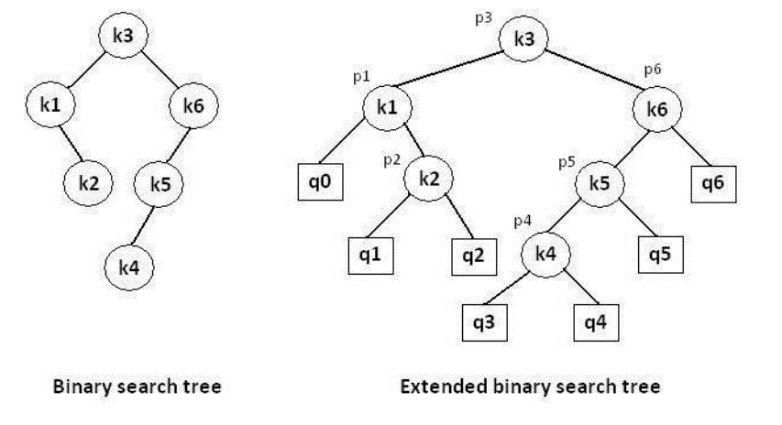
* Define class for Extended binary search tree using Object Oriented features.
* Analyze working of functions.

**Theory:**

An optimal binary search tree is a binary search tree for which the nodes are arranged on levels such that the tree cost is minimum.

For the purpose of a better presentation of optimal binary search trees, we will consider “extended binary search trees”, which have the keys stored at their internal nodes. Suppose “n” keys k1, k2, … k n are stored at the internal nodes of a binary search tree. It is assumed that the keys are given in sorted order, so that k1< k2 < … < kn.

An extended binary search tree is obtained from the binary search tree by adding successor nodes to each of its terminal nodes as indicated in the following figure by squares:

****

**In the extended tree:**

* The squares represent terminal nodes. These terminal nodes represent unsuccessful searches of the tree for key values. The searches did not end successfully, that is, because they represent key values that are not actually stored in the tree;
* The round nodes represent internal nodes; these are the actual keys stored in the tree;
* Assuming that the relative frequency with which each key value is accessed is known, weights can be assigned to each node of the extended tree (p1 … p6). They represent the relative frequencies of searches terminating at each node, that is, they mark the successful searches.
* If the user searches a particular key in the tree, 2 cases can occur:
* 1 – the key is found, so the corresponding weight ‘p’ is incremented;
* 2 – the key is not found, so the corresponding ‘q’ value is incremented.

**GENERALIZATION:**

The terminal node in the extended tree that is the left successor of k1 can be interpreted as representing all key values that are not stored and are less than k1. Similarly, the terminal node in the extended tree that is the right successor of kn, represents all key values not stored in the tree that are greater than kn. The terminal node that is successes between ki and ki-1 in an inorder traversal represent all key values not stored that lie between ki and ki - 1.

**ALGORITHMS**

We have the following procedure for determining R(i, j) and C(i, j) with 0 <= i <= j <= n:

PROCEDURE COMPUTE\_ROOT(n, p, q; R, C)

begin

for i = 0 to n do

C (i, i) ←0

W (i, i) ←q(i)

for m = 0 to n do

for i = 0 to (n – m) do

j ←i + m

W (i, j) ←W (i, j – 1) + p (j) + q (j)

\*find C (i, j) and R (i, j) which minimize the

tree cost

end

The following function builds an optimal binary sea

rch tree

FUNCTION CONSTRUCT(R, i, j)

begin

\*build a new internal node N labeled (i, j)

k ←R (i, j)

f i = k then

\*build a new leaf node N’ labeled (i, i)

else

\*N’ ←CONSTRUCT(R, i, k)

\*N’ is the left child of node N

if k = (j – 1) then

\*build a new leaf node N’’ labeled (j, j)

else

\*N’’ ←CONSTRUCT(R, k + 1, j)

\*N’’ is the right child of node N

return N

end

**COMPLEXITY ANALYSIS:**

The algorithm requires O (n2) time and O (n2) storage. Therefore, as ‘n’ increases it will run out of storage even before it runs out of time. The storage needed can be reduced by almost half by implementing the two-dimensional arrays as one-dimensional arrays.

**Software Required:** g++ / gcc compiler- / 64 bit Fedora, eclipse IDE

**Input: 1.**No.of Element.

2. key values

3. Key Probability

**Output:** Create binary search tree having optimal searching cost.

**Conclusion:** This program gives us the knowledge OBST, Extended binary search tree.

**OUTCOME**

**Upon completion Students will be able to:**

**ELO1:** Learn object oriented Programming features. A picture containing icon

Description automatically generatedDiagram, schematic

Description automatically generated

**ELO2:** Understand & implement extended binary search tree. Diagram, schematic

Description automatically generatedIcon

Description automatically generated

**Assignment 09**

**Problem Statement:**

A Dictionary stores keywords & its meanings. Provide facility for adding new keywords, deleting keywords, updating values of any entry. Provide facility to display whole data sorted in ascending/ Descending order. Also find how many maximum comparisons may require for finding any keyword. Use Height balance tree and find the complexity for finding a keyword.

**Objectives:**

1. To understand concept of height balanced tree data structure.

2. To understand procedure to create height balanced tree.

**Learning Objectives:**

* To understand concept of height balanced tree data structure.
* To understand procedure to create height balanced tree.

**Learning Outcome:**

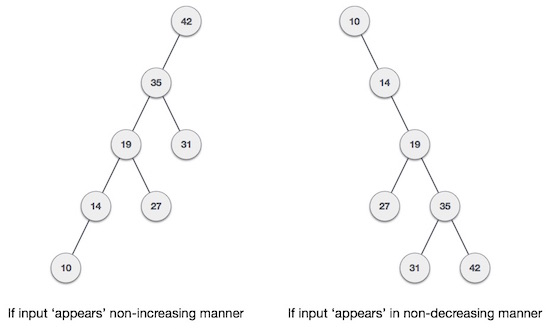
* Define class for AVL using Object Oriented features.
* Analyze working of various operations on AVL Tree .

**Theory:**

An empty tree is height balanced tree if T is a nonempty binary tree with TL and TR as its left and right sub trees. The T is height balance if and only if Its balance factor is 0, 1, -1.

**AVL (Adelson- Velskii and Landis) Tree:** A balance binary search tree. The best search time, that is O (log N) search times. An AVL tree is defined to be a well-balanced binary search tree in which each of its nodes has the AVL property. The AVL property is that the heights of the left and right sub-trees of a node are either equal or if they differ only by 1.

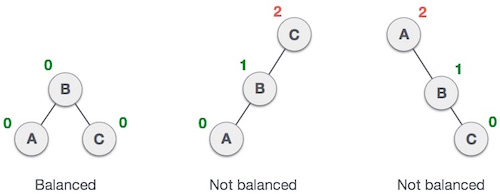
What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this −



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** |
| Description: 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. |
| Description: Left Rotation | We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**. |
| Description: Left Rotation | Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree. |
| Description: 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. |
| Description: 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** |
| Description: 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. |
| Description: 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**. |
| Description: Right Unbalanced Tree | Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation. |
| Description: 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**. |
| Description: Balanced AVL Tree | The tree is now balanced. |

**Algorithm AVL TREE:**

**Insert:-**

1. If P is NULL, then

I. P = new node

II. P ->element = x

III. P ->left = NULL

IV. P ->right = NULL

V. P ->height = 0

2. else if x>1 => x<P ->element

a.) insert(x, P ->left)

b.) if height of P->left -height of P ->right =2

1. insert(x, P ->left)

2. if height(P ->left) -height(P ->right) =2

if x<P ->left ->element

P =singlerotateleft(P)

else

P =doublerotateleft(P)

3. else

if x<P ->element

a.) insert(x, P -> right)

b.) if height (P -> right) -height (P ->left) =2

if(x<P ->right) ->element

P =singlerotateright(P)

else

P =doublerotateright(P)

4. else

Print already exits

5. int m, n, d.

6. m = AVL height (P->left)

7. n = AVL height (P->right)

8. d = max(m, n)

9. P->height = d+1

10. Stop

**RotateWithLeftChild( AvlNode k2 )**

* AvlNode k1 = k2.left;
* k2.left = k1.right;
* k1.right = k2;
* k2.height = max( height( k2.left ), height( k2.right ) ) + 1;
* k1.height = max( height( k1.left ), k2.height ) + 1;
* return k1;

**RotateWithRightChild( AvlNode k1 )**

* AvlNode k2 = k1.right;
* k1.right = k2.left;
* k2.left = k1;
* k1.height = max( height( k1.left ), height( k1.right ) ) + 1;
* k2.height = max( height( k2.right ), k1.height ) + 1;
* return k2;

**doubleWithLeftChild( AvlNode k3)**

* k3.left = rotateWithRightChild( k3.left );
* return rotateWithLeftChild( k3 );

**doubleWithRightChild( AvlNode k1 )**

* k1.right = rotateWithLeftChild( k1.right );
* return rotateWithRightChild( k1 );

**Software Required:** g++ / gcc compiler- / 64 bit Fedora, eclipse IDE

**Input:** Dictionary word and its meaning.

**Output:** Allow Add, delete operations on dictionary and also display data in sorted order.

**Conclusion:** This program gives us the knowledge height balanced binary tree.

**OUTCOME**

**Upon completion Students will be able to:**

**ELO1:** Learn height balanced binary tree in data structure. A picture containing icon

Description automatically generatedDiagram, schematic

Description automatically generated

**ELO2:** Understand & implement rotations required to balance the tree. Diagram, schematic

Description automatically generatedIcon

Description automatically generated

**Questions asked in university exam.**

1. What is AVL tree?
2. In an AVL tree, at what condition the balancing is to be done
3. When would one want to use a balance binary search tree (AVL) rather than an array data structure

**Assignment 10:**

**Problem Statement:**

Read the marks obtained by students of second year in an online examination of particular subject. Find out maximum and minimum marks obtained in a subject. Use heap data structure. Analyze the algorithm.

**Objectives:**

1. To understand concept of heap

2. To understand concept & features like max heap, min heap.

**Learning Outcome:**

* Define class for heap using Object Oriented features.
* Analyze working of functions.

**Theory:**

Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly. If **α** has child node **β** then −

**key(α) ≥ key(β)**

As the value of parent is greater than that of child, this property generates **Max Heap**. Based on this criteria, a heap can be of two types −

For Input → 35 33 42 10 14 19 27 44 26 31

**Min-Heap** − Where the value of the root node is less than or equal to either of its children.



**Max-Heap** − Where the value of the root node is greater than or equal to either of its children.



Both trees are constructed using the same input and order of arrival.

## Max Heap Construction Algorithm

We shall use the same example to demonstrate how a Max Heap is created. The procedure to create Min Heap is similar but we go for min values instead of max values.

We are going to derive an algorithm for max heap by inserting one element at a time. At any point of time, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.

**Step 1** − Create a new node at the end of heap.

**Step 2** − Assign new value to the node.

**Step 3** − Compare the value of this child node with its parent.

**Step 4** − If value of parent is less than child, then swap them.

**Step 5** − Repeat step 3 & 4 until Heap property holds.

**Note** − In Min Heap construction algorithm, we expect the value of the parent node to be less than that of the child node.

Let's understand Max Heap construction by an animated illustration. We consider the same input sample that we used earlier.

INPUT:35,33,42,10,14,19,27,44,16,31

## Max Heap Deletion Algorithm

Let us derive an algorithm to delete from max heap. Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value.

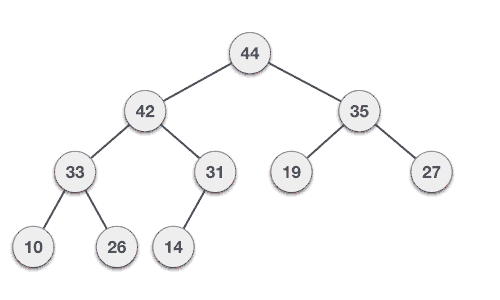
**Step 1** − Remove root node.

**Step 2** − Move the last element of last level to root.

**Step 3** − Compare the value of this child node with its parent.

**Step 4** − If value of parent is less than child, then swap them.

**Step 5** − Repeat step 3 & 4 until Heap property holds.



**Software Required:** g++ / gcc compiler- / 64 bit Fedora, eclipse IDE

**Input:** Marks obtained by student..

**Output:** Find min and max marks ontained.

**Conclusion:** This program gives us the knowledge of heap and its types.

**OUTCOME**

**Upon completion Students will be able to:**

**ELO1:** Learn object oriented Programming features. A picture containing icon

Description automatically generatedDiagram, schematic

Description automatically generated

**ELO2:** Understand & implement Heap data structure. Diagram, schematic

Description automatically generatedIcon

Description automatically generated

**Conclusion:** In this way we have implemented heap creation using C++.

**Assignment 11:**

**Problem Statement:**

Department maintains a student information. The file contains roll number, name, division and address. Allow user to add, delete information of student. Display information of particular employee. If record of student does not exist an appropriate message is displayed. If it is, then the system displays the student details. Use sequential file to maintain the data.

**Objectives:**

1. To understand concept of file handling in C++.

2. To understand concept of sequential file handling.

**Theory:**

So far, we have been using the iostream standard library, which provides cin and cout methods for reading from standard input and writing to standard output respectively. This practical will teach you how to read and write from a file. This requires another standard C++ library called fstream, which defines three new data types:

|  |  |
| --- | --- |
| **Data Type** | **Description** |
| ofstream | This data type represents the output file stream and is used to create files and to write information to files. |
| ifstream | This data type represents the input file stream and is used to read information from files. |
| fstream | This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files. |

To perform file processing in C++, header files <iostream> and <fstream> must be included in your C++ source file.

## Opening a File

A file must be opened before you can read from it or write to it. Either the **ofstream** or **fstream** object may be used to open a file for writing or ifstream object is used to open a file for reading purpose only.

Following is the standard syntax for open() function, which is a member of fstream, ifstream, and ofstream objects.

void open(const char \*filename, ios::openmode mode);

Here, the first argument specifies the name and location of the file to be opened and the second argument of the **open()** member function defines the mode in which the file should be opened.

|  |  |
| --- | --- |
| **Mode Flag** | **Description** |
| ios::app | Append mode. All output to that file to be appended to the end. |
| ios::ate | Open a file for output and move the read/write control to the end of the file. |
| ios::in | Open a file for reading. |
| ios::out | Open a file for writing. |
| ios::trunc | If the file already exists, its contents will be truncated before opening the file. |

You can combine two or more of these values by **OR**ing them together. For example if you want to open a file in write mode and want to truncate it in case it already exists, following will be the syntax:

ofstream outfile;

outfile.open("file.dat", ios::out | ios::trunc );

Similar way, you can open a file for reading and writing purpose as follows:

fstream afile;

afile.open("file.dat", ios::out | ios::in );

## Closing a File

When a C++ program terminates it automatically closes flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.

Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.

void close();

## INPUT AND OUTPUT OPERATION

**put() and get() function**  
the function put() writes a single character to the associated stream. Similarly, the function get() reads a single character form the associated stream.  
example :  
file.get(ch);  
file.put(ch);

**write() and read() function**  
write() and read() functions write and read blocks of binary data.  
**example:**  
file.read((char \*)&obj, sizeof(obj));  
file.write((char \*)&obj, sizeof(obj));

### ERROR HANDLING FUNCTION

|  |  |
| --- | --- |
| **FUNCTION** | **RETURN VALUE AND MEANING** |
| eof() | returns true (non zero) if end of file is encountered while reading; otherwise return false(zero) |
| fail() | return true when an input or output operation has failed |
| bad() | returns true if an invalid operation is attempted or any unrecoverable error has occurred. |
| good() | returns true if no error has occurred. |

## 

## File Pointers and Their Manipulation

All i/o streams objects have, at least, one internal stream pointer:   
ifstream, like istream, has a pointer known as the get pointer that points to the element to be read in the next input operation.

ofstream, like ostream, has a pointer known as the put pointer that points to the location where the next element has to be written.

Finally, fstream, inherits both, the get and the put pointers, from iostream (which is itself derived from both istream and ostream).   
  
These internal stream pointers that point to the reading or writing locations within a stream can be manipulated using the following member functions:

|  |  |
| --- | --- |
| seekg() | moves get pointer(input) to a specified location |
| seekp() | moves put pointer (output) to a specified location |
| tellg() | gives the current position of the get pointer |
| tellp() | gives the current position of the put pointer |

The other prototype for these functions is:

seekg(offset, refposition );   
seekp(offset, refposition );

The parameter offset represents the number of bytes the file pointer is to be moved from the location specified by the parameter refposition. The refposition takes one of the following three constants defined in the ios class.

**ios::beg**          start of the file  
**ios::cur**          current position of the pointer  
**ios::end**          end of the file

**example:**  
file.seekg(-10, ios::cur);

**Conclusion:** In this way we have implemented file handling assignment using C++.

**Assignment 12:**

**Problem Statement:**

Company maintains employee information as employee ID, name, designation and salary. Allow user to add, delete information of employee. Display information of particular employee. If employee does not exist an appropriate message is displayed. If it is, then the system displays the employee details. Use index sequential file to maintain the data.

**Theory:**

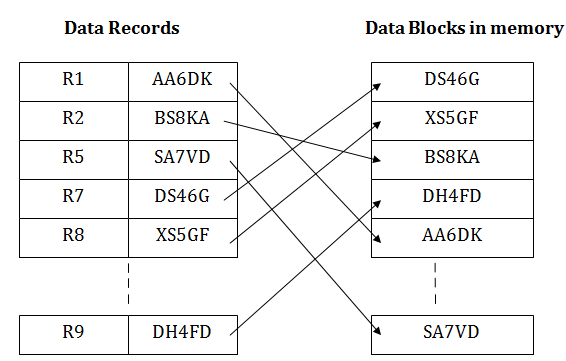
* Indexed sequential access file combines both sequential file and direct access file organization.
* In indexed sequential access file, records are stored randomly on a direct access device such as magnetic disk by a primary key.
* This file have multiple keys. These keys can be alphanumeric in which the records are ordered is called primary key.
* The data can be access either sequentially or randomly using the index. The index is stored in a file and read into memory when the file is opened.

**Advantages of Indexed sequential access file organization**

* In indexed sequential access file, sequential file and random file access is possible.
* It accesses the records very fast if the index table is properly organized.
* The records can be inserted in the middle of the file.
* It provides quick access for sequential and direct processing.
* It reduces the degree of the sequential search.

**Disadvantages of Indexed sequential access file organization**

* Indexed sequential access file requires unique keys and periodic reorganization.
* Indexed sequential access file takes longer time to search the index for the data access or retrieval.
* It requires more storage space.
* It is expensive because it requires special software.
* It is less efficient in the use of storage space as compared to other file organizations.
* ISAM method is an advanced sequential file organization. In this method, records are stored in the file using the primary key. An index value is generated for each primary key and mapped with the record. This index contains the address of the record in the file.

**Conclusion:** In this way we have implemented file handling assignment using C++.