**LABORATORY MANUAL**

## DATA STRUCTURES AND ALGORITHMS LABORATORY (210246)

### (S.E. COMPUTER ENGINEERING 2019)

****

### PREPARED BY PROF.S.V.SHINDE

### DEPARTMENT OF COMPUTER ENGINEERING COLLEGE OF ENGINEERING MANJARI (BK), PUNE

**DSA-Lab Manuals Contents**

[Group A - Assignment No. A-1 3](#_bookmark0)

[Group A : Assignment No. A-2 6](#_bookmark1)

[Group B - Assignment No. B-5 10](#_bookmark2)

[Group B - Assignment No. B-6 13](#_bookmark3)

[Group B - Assignment No. B-7 16](#_bookmark4)

[Group C – Assignment No. C-13 23](#_bookmark5)

[Group C : Assignment No. C-15 25](#_bookmark6)

[Group D : Assignment No. D-18 28](#_bookmark7)

[Group D : Assignment No. D-19 31](#_bookmark8)

[Group E - Assignment No. E-21 36](#_bookmark9)

[Group F : Assignment No. F-23 39](#_bookmark10)

[Group F : Assignment No. F-24 44](#_bookmark11)

# Group A - Assignment No. A-1

Consider telephone book database of N clients. Make use of a hash table implementation to quickly look up client‘s telephone number.

#### Title:

Consider telephone book database of N clients. Make use of a hash table implementation to quickly

look up client‘s telephone number

#### Objectives:

1. To understand concept of Hashing
2. To understand to find record quickly using hash function.
3. To understand concept & features of object oriented programming.

#### Learning Objectives

* + To understand concept of hashing.
  + To understand operations like insert and search record in the database. Learning Outcome
  + Learn object oriented Programming features
  + Understand & implement concept of hash table.

#### 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;

}

**Conclusion:** In this way we have implemented Hash table for quick lookup using C++. Assignment Questions:

1. What is Hash Function?
2. what is Good Hash function?
3. How many ways are there to implement hash function?
4. What are the Collision Resolution Strategies?
5. What is the Hash Table Overflow?

# Group A : Assignment No. A-2

Implement all the functions of a dictionary (ADT) using hashing.

Data: Set of (key, value) pairs, Keys are mapped to values, Keys must be comparable, Keys must be unique Standard Operations: Insert (key, value), Find(key), Delete(key)

#### Title:

Implement all the functions of a dictionary (ADT) using hashing.

Data: Set of (key, value) pairs, Keys are mapped to values, Keys must be comparable, Keys must be unique Standard Operations: Insert (key, value), Find(key), Delete(key) **Objectives:**

1. To understand Dictionary (ADT)
2. To understand concept of hashing
3. To understand concept & features like searching using hash function. Learning Objectives:
   * To understand Dictionary (ADT)
   * To understand concept of hashing
   * To understand concept & features like searching using hash function.

#### Learning Outcome:

* + Define class for Dictionary using Object Oriented features.
  + Analyze working of hash function.

#### Theory:

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 Operations

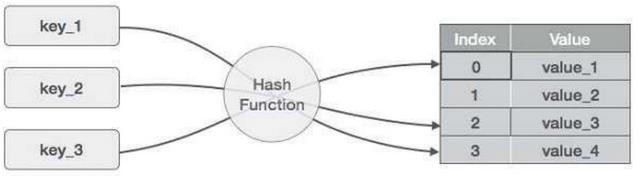
* Dictionary create() creates empty dictionary
* boolean isEmpty(Dictionary d) tells whether the dictionary d is empty
* put(Dictionary d, Key k, Value v) associates key k with a value v; if key k already presents in the dictionary old value is replaced by v
* Value get(Dictionary d, Key k) returns a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

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



#### Basic Operations of hash table

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;

}

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

**Input:** No. of. elements with key and value pair.

**Output:** Create dictionary using hash table and search the elements in table.

**Conclusion:** This program gives us the knowledge of dictionary(ADT).

OUTCOME

Upon completion Students will be able to:

ELO1: Learn object oriented Programming features.

ELO2: Understand & implement Dictionary (ADT) using hashing.

# Group B - Assignment No. B-5

A book consists of chapters, chapters consist of sections and sections consist of subsections. Construct a tree and print the nodes. Find the time and space requirements of your method.

#### Title:

A book consists of chapters, chapters consist of sections and sections consist of subsections. Construct a tree and print the nodes. Find the time and space requirements of your method. **Objectives:**

1. To understand concept of tree data structure
2. To understand concept & features of object oriented programming. Learning Objectives:
   * To understand concept of class
   * To understand concept & features of object oriented programming.
   * To understand concept of tree data structure.

#### Learning Outcome:

* Define class for structures using Object Oriented features.
* Analyze tree data structure.

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

**Input:** Book name & its number of sections and subsections along with name.

**Output:** Formation of tree structure for book and its sections.

#### Theory:

#### Introduction to Tree:

Definition:

A tree T is a set of nodes storing elements such that the nodes have a parent-child relationship that satisfies the following

* if T is not empty, T has a special tree called the root that has no parent
* each node v of T different than the root has a unique parent node w; each node with parent w is a child of w

#### Recursive definition

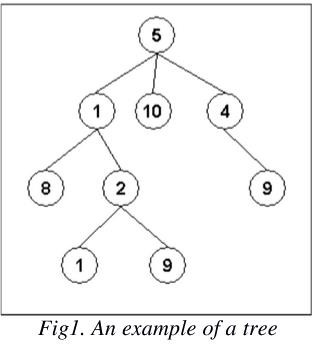
* T is either empty
* or consists of a node r (the root) and a possibly empty set of trees whose roots are the children of r Tree is a widely-used data structure that emulates a tree structure with a set of linked nodes. The tree graphically is represented most commonly as on Picture 1. The circles are the nodes and the edges are the links between them.

Trees are usually used to store and represent data in some hierarchical order. The data are stored in the nodes, from which the tree is consisted of.

A node may contain a value or a condition or represent a separate data structure or a tree of its own. Each node in a tree has zero or more child nodes, which are one level lower in the tree hierarchy (by convention, trees grow down, not up as they do in nature). A node that has a child is called the child's parent node (or ancestor node, or superior). A node has at most one parent. A node that has no child is called a leaf, and that node is of course at the bottommost level of the tree. The height of a node is the length of the longest path to a leaf from that node. The height of the root is the height of the tree. In other words, the "height" of tree is the "number of levels" in the tree. Or more formaly, the height of a tree is defined as follows:

1. The height of a tree with no elements is 0
2. The height of a tree with 1 element is 1
3. The height of a tree with > 1 element is equal to 1 + the height of its tallest subtree. The depth of a node is the length of the path to its root (i.e., its root path). Every child node is always one level lower than his parent.

The topmost node in a tree is called the root node. Being the topmost node, the root node will not have parents. It is the node at which operations on the tree commonly begin (although some algorithms begin with the leaf nodes and work up ending at the root). All other nodes can be reached from it by following edges or links. (In the formal definition, a path from a root to a node, for each different node is always unique).

In diagrams, it is typically drawn at the top. In some trees, such as heaps, the root node has special properties. A subtree is a portion of a tree data structure that can be viewed as a complete tree in itself. Any node in a tree T, together with all the nodes below his height, that are reachable from the node, comprise a subtree of T. The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree (in analogy to the term proper subset). Every node in a tree can be seen as the root node of the subtree rooted at that node.

An internal node or inner node is any node of a tree that has child nodes and is thus not a leaf node. There are two basic types of trees. In an unordered tree, a tree is a tree in a purely structural sense — that is to say, given a node, there is no order for the children of that node. A tree on which an order is imposed — for example, by assigning different natural numbers to each child of each node — is called an ordered tree, and data structures built on them are called ordered tree data structures. Ordered trees are by far the most common form of tree data structure. Binary search trees are one kind of ordered tree.

#### Important Terms:

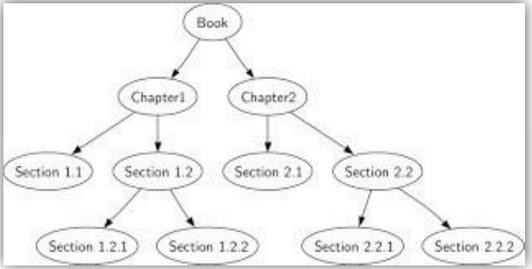
Following are the important terms with respect to tree.

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

#### Advantages of trees

Trees are so useful and frequently used, because they have some very serious advantages:

* + Trees reflect structural relationships in the data
  + Trees are used to represent hierarchies
  + Trees provide an efficient insertion and searching
  + Trees are very flexible data, allowing to move subtrees around with minimum effort For this assignment we are considering the tree as follows.



**Conclusion:** This program gives us the knowledge tree data structure.

### OUTCOME

Upon completion Students will be able to:

ELO1: Learn object oriented Programming features. ELO2: Understand & implement tree data structure.

# Group B - Assignment No. B-6

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

1. Insert new node
2. Minimum data value found in the tree
3. Search a value iv print tree **Title:**

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

1. Insert new node
2. Minimum data value found in the tree
3. Search a value iv print tree **Objectives:**
   1. To understand concept of tree data structure
   2. To understand concept & features of object oriented programming. Learning Objectives:
      * To understand concept of class
      * To understand concept & features of object oriented programming.
      * To understand concept of tree data structure.

#### Learning Outcome:

* Define class for structures using Object Oriented features.
* Analyze tree data structure.

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

**Input:** Book name & its number of sections and subsections along with name.

**Output:** Formation of tree structure for book and its sections.

#### Theory:

Binary Search Tree (BST)

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 or equal to its parent node's key.

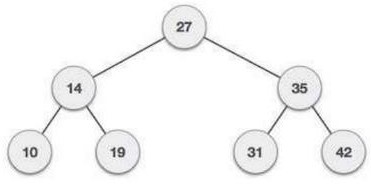
Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −

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 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;

}

}

}

}

}

**Conclusion:** This program gives us the knowledge tree data structure.

### OUTCOME

Upon completion Students will be able to:

ELO1: Learn object oriented Programming features. ELO2: Understand & implement tree data structure.

# Group B - Assignment No. B-7

Construct an expression tree from the given prefix expression eg. +--a\*bc/def and traverse it using post order traversal (non recursive) and then delete the entire tree

#### Title:

Construct an expression tree from the given prefix expression eg. +--a\*bc/def and traverse it using post order traversal (non recursive) and then delete the entire tree.

#### Objectives:

1. To Understand the concept of expression tree and binary tree.
2. To Understand the different type of traversals (recursive & non-recursive).
3. To understand the different notations like prefix, infix and postfix and the conversion between them using data structures (like stack and tree). **Learning Objectives:**
   * To understand concept of class
   * To understand concept & features of object oriented programming.
   * To understand and implement concepts of tree data structure.
   * To implement stack data structure
   * Ability to perform conversions between the notations

#### Learning Outcome:

* Define class for structures using Object Oriented features.
* Analyze stack / tree data structure.

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

**Input:** prefix expression

Output: corresponding postfix expression

#### Theory :

Expression:

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

1. Infix Notation
2. Prefix (Polish) Notation
3. 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.

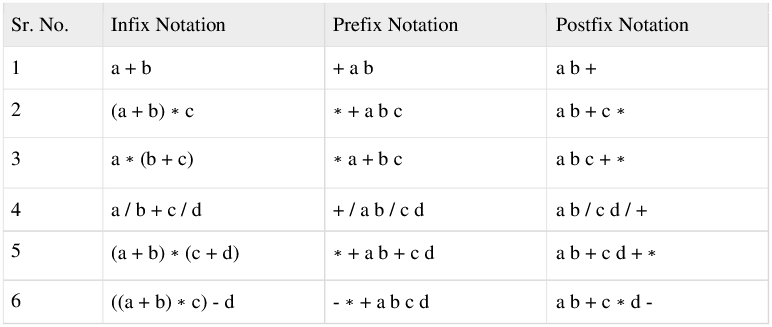
* Prefix Notation (operator then operands)

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

* Postfix Notation

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

Examples of notations



#### Approach 1: Using Stack Data structure:

In computer science, a stack is a last in, first out (LIFO) abstract data type and data structure. A stack can have any abstract data type as an element, but is characterized by only two fundamental operations: push and pop. The push operation adds an item to the top of the stack, hiding any items already on the stack, or initializing the stack if it is empty. A pop either reveals previously concealed items, or results in an empty stack. A stack is a restricted data structure, because only a small number of operations are performed on it. The nature of the pop and push operations also mean that stack elements have a natural order. Elements are removed from the stack in the reverse order to the order of their addition: therefore, the lower elements are those that have been on the stack the longest. A collection of items in which only the most recently added item may be removed. The latest added item is at the top. Basic operations are push and pop. Often top and isEmpty are available, too. Also known as "last-in, first-out" or LIFO. **Operations**

* Push : adds a new element
* Pop : removes a element Additional primitives can be defined:
* IsEmpty reports whether the stack is empty
* IsFull reports whether the stack is full
* Initialise creates/initialises the stack
* Destroy deletes the contents of the stack (may be implemented by re-initialising the stack)
* Push : Pushes an element to the stack. It takes an integer element as argument. If the stack is full then error is returned.
* Pop : Pop an element from the stack. If the stack is empty then error is returned. The element is deleted from the top of the stack.
* DisplayTop : Returns the top element on the stack without deleting. If the stack is empty then error is returned.

**ALGORITHM:** Abstract Data Type Stack: Define Structure for stack(Data, Next Pointer)

Stack Empty: Return True if Stack Empty else False. Top is a pointer of type structure stack.

Empty(Top)

Step 1: If Top = = NULL Step 2: Return 1;

Step 3: Return 0; Push Opertion:

Top & Node pointer of structure Stack. Push(element)

Step 1: Node->data=element; Step 2; Node->Next = Top; Step 3: Top = Node

Step 4: Stop.

Pop Operation:

Top & Temp pointer of structure Stack. Pop()

Step 1:If Top != NULL Then

1. Temp = Top;
2. element=Temp->data;
3. Top = (Top)->Next;
4. delete temp; Step 2: Else Stack is Empty. Step 3: return element;

Infix to Prefix Conversion:

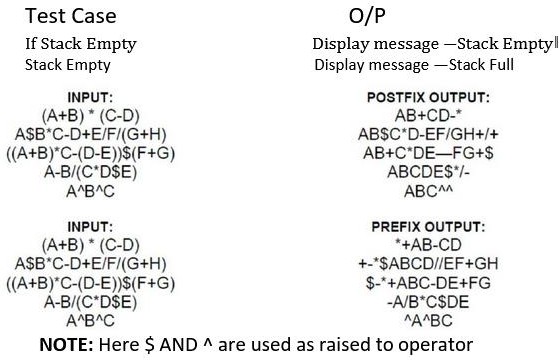
1. First, reverse the given infix expression.
2. Scan the characters one by one.
3. If the character is an operand, copy it to the prefix notation output.
4. If the character is a closing parenthesis, then push it to the stack.
5. If the character is an opening parenthesis, pop the elements in the stack until we find the corresponding closing parenthesis.
6. If the character scanned is an operator
   * If the operator has precedence greater than or equal to the top of the stack, push the operator to the stack.
   * If the operator has precedence lesser than the top of the stack, pop the operator and output it to the prefix notation output and then check the above condition again with the new top of the stack.
7. After all the characters are scanned, reverse the prefix notation output.
8. Infix to Postfix Conversion:
9. : Scan the Infix Expression from left to right.
10. : If the scanned character is an operand, append it with final Infix to Postfix string.
11. : Else,
    1. : If the precedence order of the scanned(incoming) operator is greater than the precedence order of the operator in the stack (or the stack is empty or the stack contains a ‘(‘ or ‘[‘ or ‘{‘), push it on stack.
    2. Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
12. If the scanned character is an ‘(‘ or ‘[‘ or ‘{‘, push it to the stack.
13. If the scanned character is an ‘)’or ‘]’ or ‘}’, pop the stack and and output it until a ‘(‘ or ‘[‘ or ‘{‘ respectively is encountered, and discard both the parenthesis.
14. Repeat steps 2-6 until infix expression is scanned.
15. Print the output
16. Pop and output from the stack until it is not empty.

#### Prefix to postfix expression:

* Read the prefix expression in reverse order. (from right to left)
* If the symbol is an operand, then push it into the stack.
* If the symbol is an operator, then pop two operand from the stack. create a string by concatenating two operands and the operator after them.

#### string = operand 1 + operand 2 + operator

And push the resultant string back into the stack



FAQs :

1. What are the operations can implement on stack?
2. Explain recursion using stack.
3. Explain how a string can be reversed using stack.
4. How does a stack similar to list? How it is different?
5. List advantages and disadvantages of postfix and prefix expression over infix expression.

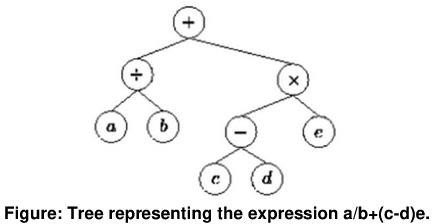
**Approach 2:** Using Tree (Expresion Tree) Data structure :

Theory:

1. Definition of an expression tree with diagram. Algebraic expressions such as: **a/b + (c-d) e**

The terminal nodes (leaves) of an expression tree are the variables or constants in the expression (a, b, c, d, and e). The non-terminal nodes of an expression tree are the operators (+, -, , and ).

Notice that the parentheses which appear in Equation do not appear in the tree. Nevertheless, the tree representation has captured the intent of the parentheses since the subtraction is lower in the tree than the multiplication.



#### Show the different type of traversals with example

To traverse a non-empty binary tree in preorder,

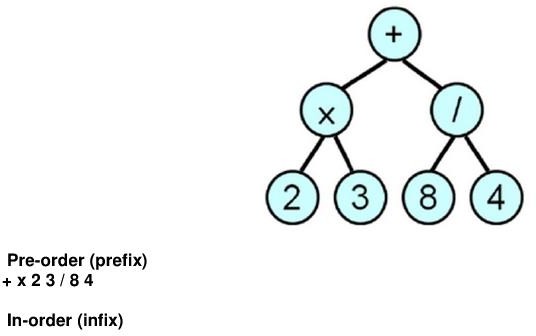
1. Visit the root.
2. Traverse the left subtree.
3. Traverse the right subtree.

To traverse a non-empty binary tree in inorder:

1. Traverse the left subtree.
2. Visit the root.
3. Traverse the right subtree.

To traverse a non-empty binary tree in postorder, 24

1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the root



2 x 3 + 8 / 4

Post-order (postfix) 2 3 x 8 4 / +

### ALGORITHM:

Define structure for Binary Tree (Information, Left Pointer & Right Pointer)

#### Create Expression Tree:

#### CreateTree()

Root& Node pointer variable of type structure. Stack is an pointer array of type structure. String is character array which contains postfix expression. Top & I are variables of type integer.

Step 1: Top = -1 , I = 0;

Step 2: Do Steps 3,4,5,6 While String[I] != NULL Step 3: Create Node of type of structure

Step 4: Node->Data=String[I]; Step 5: If isalnum(String[I]) Then Stack[Top++] = Node; Else

Node->Right = Stack [--Top ]; Node->Left = Stack[ --Top ]; Stack[ Top++ ] = Node;

Step 6: Increment I; Step 7: Root = Stack[0]; Step 8: Return Root

Inorder Traversal Recursive :

Tree is pointer of type structure. InorderR(Tree)

Step 1: If Tree != NULL 25 Step 2: InorderR( Tree->Left ) Step 3: Print Tree->Data

Step 4: InorderR( Tree->Right ) Postorder Traversal Recursive:

Tree is pointer of type structure. PostorderR(Tree)

Step 1: If Tree != NULL

Step 2: PostorderR( Tree->Left ) Step 3: PostorderR( Tree->Right ) Step 4: Print Tree->Data

Preorder Traversal Recursive:

Tree is pointer of type structure. PreorderR(Tree)

Step 1: If Tree != NULL Step 2: Print Tree->Data

Step 3: PreorderR( Tree->Left ) Step 4: PreorderR( Tree->Right ) Postorder Traversal Nonrecursive :

NonR\_Postorder(Tree)

Tree, Temp is pointer of type structure. Stack is pointer array of type structure. Top variable of type integer.

Step 1: Temp = Tree // current pointer pointing to root

Step 2: if Temp != NULL then push current pointer along with its initial flag value ‗L‘ on to the stack and traverse on the left side.

Step 3: Otherwise if the stack is not empty then pop an address from stack along with its flag value.

Step 4: For the current pointer if the flag value is ‗L‘ then change it to ‗R‘ and push current pointer along with its flag value ‗R‘ on to the stack and traverse on the right side.

Step 5: otherwise, For the current pointer if the flag value is ‗R‘ then display the element and make the current pointer NULL (i.e.temp=NULL)

Step 6: repeat steps 2, 3, 4, 5 until the stack becomes empty.

Preorder Traversal Nonrecursive : NonR\_Preorder(Tree)

Tree, Temp is pointer of type structure. Stack is pointer array of type structure. Top variable of type integer.

Step 1: Temp = Tree

Step 2: Do Steps3,4,5,6,7,& 8 While Temp != NULL And Stack is not Empty Step 3: Do Steps 4,5&6 While Temp != NULL

Step 4: Print Temp->Data

Step 5: Stack[ ++ Top ] = Temp //Push Element Step 6: Temp = Temp->Left

Step 7: Temp = Stack [ Top -- ] //Pop Element Step 8: Temp = Temp->Right

Inorder Traversal Nonrecursive :

NonR\_Inorder(Tree)

Tree, Temp is pointer of type structure. Stack is pointer array of type structure. Top variable of type integer.

Step 1: Temp = Tree

Step 2: Do Steps3,4,5,6,7,&8 While Temp != NULL And Stack is not Empty Step 3: Do Steps 4,5 While Temp != NULL

Step 4: Stack[ ++ Top ] = Temp;//Push Element Step 5: Temp = Temp->Left

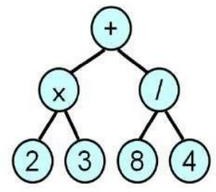
Step 6: Temp = Stack[ Top -- ]//Pop Element Step 7: Print Temp->Data

Step 8: Temp = Temp->Right INPUT

:

Postfix Expression: 2 3 x 8 4 / + OUTPUT:

Display result of each operation with error checking.



FAQS:

1. What is tree? What are properties of trees?
2. What is Binary tree, Binary search tree, Expression tree & General tree?
3. What are the members of structure of tree & what is the size of structure?
4. What are rules to construct binary tree?
5. What is preorder, postorder, inorder traversal?
6. Difference between recursive & Nonrecursive traversal?
7. What are rules to construct binary search tree?
8. What are rules to construct expression tree?
9. How binary tree is constructed from its traversals?

# Group C – Assignment No. C-13

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. Justify the storage representation used.

#### Title:

There are flight paths between cities. If there is a flight between city A and city B then here 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. Justify the storage representation used.

#### Objectives:

1. To understand concept of Graph data structure
2. To understand concept of representation of graph.

#### Learning Objectives:

* + To understand concept of Graph data structure
  + To understand concept of representation of graph.

#### Learning Outcome:

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

#### Theory:

Graphs are the most general data structure. They are also commonly used data structures. Graph definitions:

* A non-linear data structure consisting of nodes and links between nodes. Undirected graph definition:
* An undirected graph is a set of nodes and a set of links between the nodes.
* Each node is called a vertex, each link is called an edge, and each edge connects two vertices.
* The order of the two connected vertices is unimportant.

An undirected graph is a finite set of vertices together with a finite set of edges. Both sets might be empty, which is called the empty graph

#### Graph Implementation:

Different kinds of graphs require different kinds of implementations, but the fundamental concepts of all graph implementations are similar. We'll look at several representations for one particular kind of graph: directed graphs in which loops are allowed.

#### Definition:

* An adjacency matrix is a square grid of true/false values that represent the edges of a graph.
* If the graph contains n vertices, then the grid contains n rows and n columns.
* For two vertex numbers i and j, the component at row i and column j is true if there is an edge from vertex i to vertex j; otherwise, the component is false.

We can use a two-dimensional array to store an adjacency matrix: boolean[][] adjacent = new boolean[4][4];

Once the adjacency matrix has been set, an application can examine locations of the matrix to determine which edges are present and which are missing.

Definition:

* A directed graph with n vertices can be represented by n different linked lists.
* List number i provides the connections for vertex i.
* For each entry j in list number i, there is an edge from i to j. Loops and multiple edges could be allowed.

#### Representing Graphs with Edge Sets

To represent a graph with n vertices, we can declare an array of n sets of integers. For example:

IntSet[] connections = new IntSet[10]; // 10 vertices

A set such as connections[i] contains the vertex numbers of all the vertices to which vertex i is connected.

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

**Input:** 1.Number of cities.

2.Time required to travel from one city to another.

**Output:** Create Adjacency matrix to represent path between various cities. **Conclusion:** This program gives us the knowledge of adjacency matrix graph. **OUTCOME**

Upon completion Students will be able to:

ELO1: Learn concept of graph data structure.

ELO2: Understand & implement adjacency matrix for graph. Questions asked in university exam.

1. What are different ways to represent the graph? Give suitable example.
2. What is time complexity of function to create adjacency matrix?

# Group C : Assignment No. C-15

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 different a minimum total cost. Solve the problem by suggesting appropriate data structures.

#### Title:

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 different a minimum total cost. Solve the problem by suggesting appropriate data structures.

#### Objectives:

1. To understand concept of graph data structure.
2. To understand concept & features of object oriented programming.

#### Learning Objectives:

* + To understand concept of class.
  + To understand concept & features of object oriented programming.
  + To understand concept of graph data structure.

#### Learning Outcome:

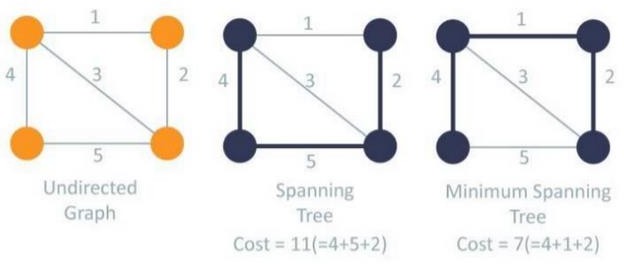
* + Learn the working of minimum spanning tree.
  + Analyze working of functions.

#### Theory:

Spanning Tree:

A spanning tree for a connected graph G is a tree T containing all the vertices of G. Minimum Spanning Tree:

A spanning tree, whose weight of edges is minimum is called as Minimum Spanning Tree.

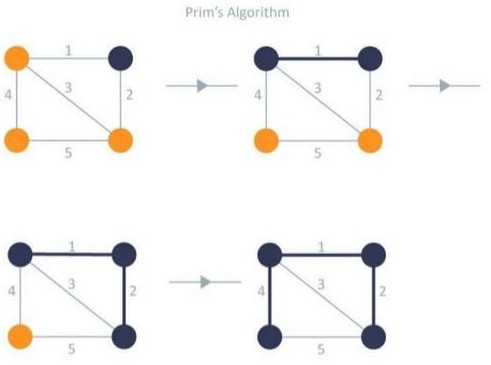


Two methods to find Minimum Spanning Tree:

#### Prim's Algorithm:

This algorithm is used to find minimum Spanning tree and minimum cost of spanning tree. Steps:

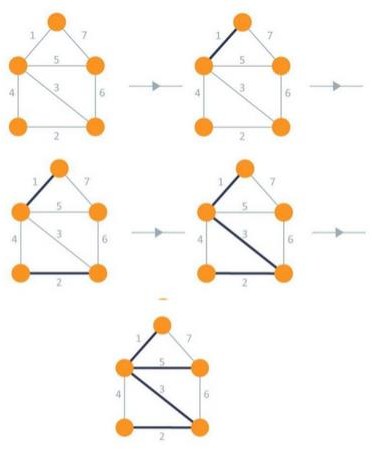
* 1. This algorithm starts with start vertex.
  2. Then it finds edge which has minimum weight.
  3. It continues the finding such that one vertex of edge is already visited & another vertex is not visited with minimum weight.
  4. These steps are repeated to cover all the vertices of graph.



#### Kruskal's Algorithm:

This algorithm is used to find minimum spanning tree and minimum cost of spanning tree. Steps:

1. Sort all edges in increasing order of their edge weights.
2. Pick the smallest edge.
3. Check if the new edge creates a cycle or loop in a spanning tree.
4. If it doesn’t form the cycle, then include that edge in MST. Otherwise, discard it.
5. Repeat from step 2 until it includes |V| - 1 edges in MST.



Applications of Minimum Spanning Tree:

Minimum spanning trees are useful in constructing networks, by describing the way to connect a set of sites using the smallest total amount of wire.

#### Software Required:

g++ / gcc compiler- / 64 bit Fedora, eclipse IDE

#### Input:

Number of vertices as offices and provide edge between vertices/offices as telephone lease lines here leased line are are nothing but vertices.

#### Output:

The path which have minimum distance.

#### Conclusion:

Prims Algorithm is successfully implemented to find out minimum distance.

#### Outcome:

Upon completion Students will be able to:

ELO1: Learn the concept of minimum spanning tree. ELO2: Understand & implement Prim's algorithm.

### FAQ:

1. What is Spanning Tree ?
2. What is Minimum Spanning ?
3. What is Prim's Algorithm?
4. What is Kruskal's Algorithm?
5. What are Applications of Minimum Spanning Tree?
6. How to decide whether to select Kruskal's algorithm or Prim's algorithm?

# Group D : Assignment No. D-18

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?

#### Title:

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.

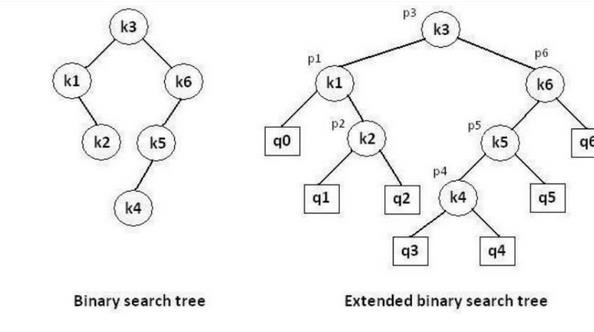
#### Learning Outcome:

* + 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 search 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.

ELO2: Understand & implement extended binary search tree.

Questions asked in university exam.

1. What is Binary Search Tree? Explain with an example?
2. Explain various operations on BST?
3. What is inorder traversal of BST?
4. What is a B tree?
5. What are threaded binary trees?
6. What is a B+ tree?
7. How do you find the depth of a binary tree?
8. Explain pre-order and in-order tree traversal.
9. Define threaded binary tree. Explain its common uses
10. Explain various operations on BST?

# Group D : Assignment No. D-19

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.

#### Title:

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:**

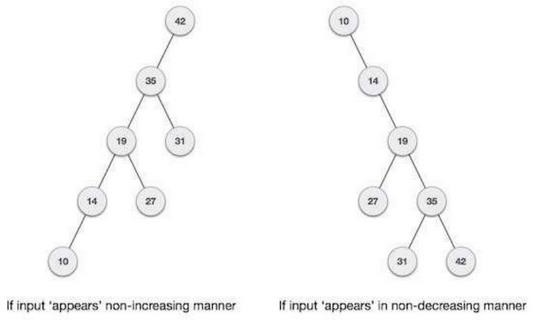
1. To understand concept of height balanced tree data structure.
2. To understand procedure to create height balanced tree. Learning Objectives:
   * To understand concept of height balanced tree data structure.
   * To understand procedure to create height balanced tree.

#### Learning Outcome:

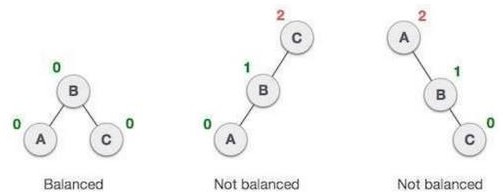
* Define class for AVL using Object Oriented features.
* Analyze working of various operations on AVL Tree .

#### 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 −

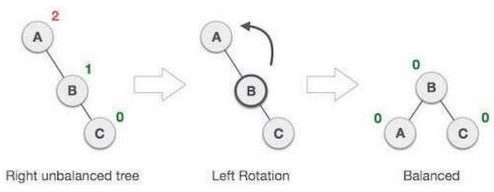
* Left rotation
* Right rotation
* Left-Right rotation
* Right-Left rotation

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 −

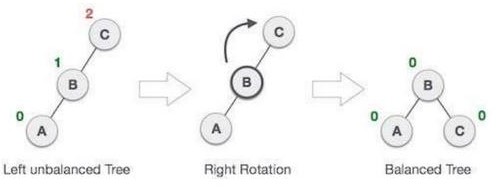
In our example, node A has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making A the left-subtree of B.



#### 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.

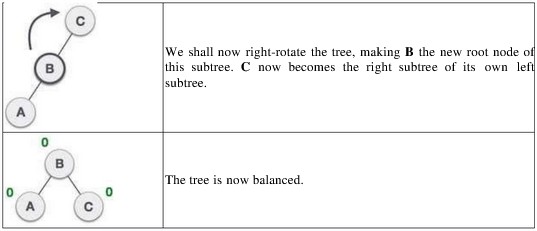
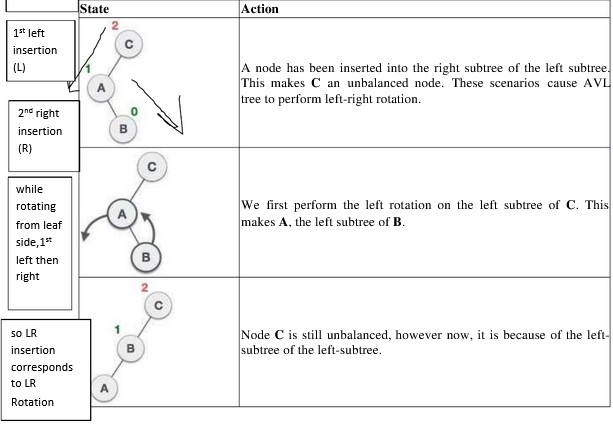
In our example, node A has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making A the left-subtree of B.

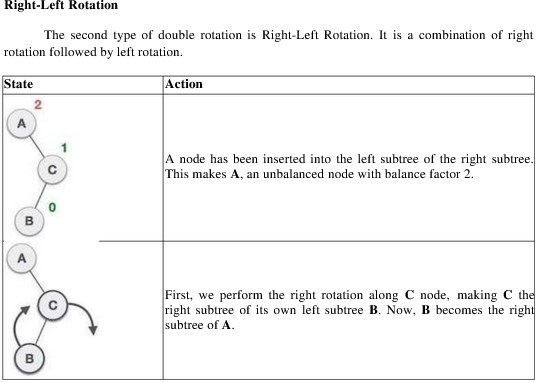


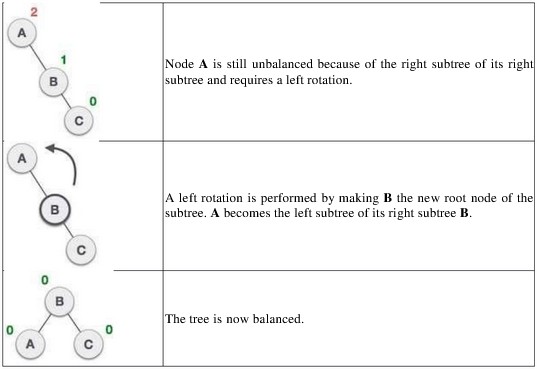
Left-Right Rotation

**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 from root first check how to perform Left-Right rotation. A left-right rotation is a combination of left side rotation followed by right rotation







Algorithm AVL TREE:

Insert:-

1. If P is NULL, then
2. P = new node
3. P ->element = x
4. P ->left = NULL
5. P ->right = NULL
6. P ->height = 0
7. else if x>1 => x<P ->element
   1. insert(x, P ->left)
   2. 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)

* + 1. else else

if x<P ->element

P =doublerotateleft(P)

* + - 1. insert(x, P -> right)
      2. if height (P -> right) -height (P ->left) =2 if(x<P ->right) ->element

P =singlerotateright(P)

* + 1. else

Print already exits

* + 1. int m, n, d. else

P =doublerotateright(P)

* + 1. m = AVL height (P->left)
    2. n = AVL height (P->right)
    3. d = max(m, n)
    4. P->height = d+1
    5. 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.

ELO2: Understand & implement rotations required to balance the tree. 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

# Group E - Assignment No. E-21

Implement the Heap/Shell sort algorithm implemented in Java demonstrating heap/shell data structure with modularity of programming language.

#### Title:

Implement the Heap/Shell sort algorithm implemented in Java demonstrating heap/shell data structure with modularity of programming language.

#### Objectives:

1. To understand concept of heap in data structure.
2. To understand concept & features of java language.

#### Learning Objectives:

* + To understand concept of heap in data structure.
  + To understand concept & features of java language.

#### Learning Outcome:

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

#### Theory:

Heap Sort:

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

#### What is Binary Heap?

Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible (Source Wikipedia)

A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater (or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

#### Why array based representation for Binary Heap?

Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index I, the left child can be calculated by 2 \* I + 1 and right child by 2 \* I + 2 (assuming the indexing starts at 0).

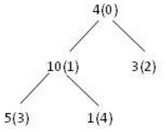
Heap Sort Algorithm for sorting in increasing order:

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps until size of heap is greater than 1.

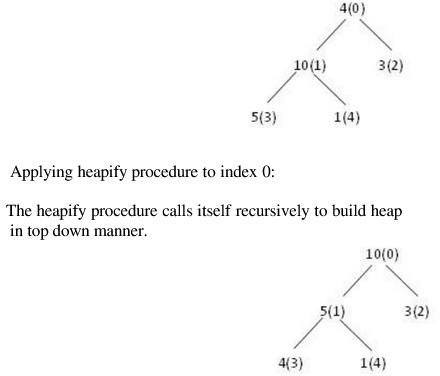
#### How to build the heap?

Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order.

Lets understand with the help of an example: Input data: 4, 10, 3, 5, 1



The numbers in bracket represent the indices in the array representation of data. Applying heapify procedure to index 1:



Algorithm:

STEP 1: Logically, think the given array as Complete Binary Tree,

STEP 2: For sorting the array in ascending order, check whether the tree is satisfying Max- heap property at each node, (For descending order, Check whether the tree is satisfying Min- heap property) Here we will be sorting in Ascending order.

STEP 3: If the tree is satisfying Max-heap property, then largest item is stored at the root of the heap. (At this point we have found the largest element in array, Now if we place this element at the end (nth position) of the array then 1 item in array is at proper place.) We will remove the largest element from the heap and put at its proper place(nth position) in array.

After removing the largest element, which element will take its place? We will put last element of the heap at the vacant place. After placing the last element at the root, The new tree formed may or may not satisfy max-heap property. So, If it is not satisfying max-heap property then first task is to make changes to the tree, So that it satisfies max-heap property. (Heapify process: The process of making changes to tree so that it satisfies max-heap property is called heapify) When tree satisfies max-heap property, again largest item is stored at the root of the heap. We will remove the largest element from the heap and put at its proper place(n-1 position) in array. Repeat step 3 until size of array is 1 (At this point all elements are sorted.)

**Software Required:** java, 64 bit Fedora, eclipse IDE **Input:** Number of elements to be sort and element values. **Output:** Elements in sorted order.

**Conclusion:** This program gives us the knowledge of heap data structure.

### OUTCOME

Upon completion Students will be able to: ELO1: Learn heap data structure.

ELO2: Understand procedure for heap sort. Questions asked in university exam.

1. What is Binary Heap?
2. Why array based representation for Binary Heap?
3. How to build the heap?

# Group F : Assignment No. F-23

Department maintains a student information. The file contains roll number, name, and division and address. Allow user to add, delete information of student. Display information of particular student. 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.

#### Title:

Department maintains a student information. The file contains roll number, name, and division and address. Allow user to add, delete information of student. Display information of particular student. 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 main the data.

#### Objective:

To understand the concept and basic of sequential file and its use in Data structure

#### Outcome:

To implement the concept and basic of sequential file and to perform basic operation as adding record, display all record, search record from sequential file and its use in Data structure.

#### Software & Hardware Requirements:

64- bit Open source Linux or its derivative , Open Source C++ Programming tool like G++/GCC

#### Theory:

Types of File Organization-

1. Sequential access file organization
2. Indexed sequential access file organization
3. Direct access file organization

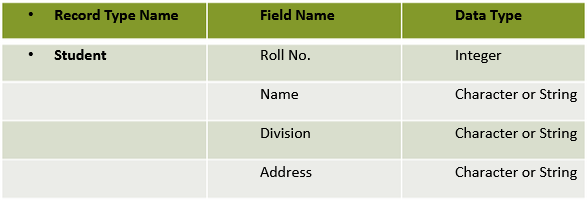
#### Sequential access file organization

* + Storing and sorting in contiguous block within files on tape or disk is called as sequential access file organization.
  + In sequential access file organization, all records are stored in a sequential order. The records are arranged in the ascending or descending order of a key field.
  + Sequential file search starts from the beginning of the file and the records can be added at the end of the file.
  + In sequential file, it is not possible to add a record in the middle of the file without rewriting the file.
  + Primitive Operations on Sequential files :
  + Open—This opens the file and sets the file pointer to immediately before the first record
  + Read-next—This returns the next record to the user. If no record is present, then EOF condition will be set.
  + Close—This closes the file and terminates the access to the file.

Write-next—File pointers are set to next of last record and write the record to the file.

* + EOF—If EOF condition occurs, it returns true, otherwise it returns false.
  + Search—Search for the record with a given key.
  + Update—Current record is written at the same position with updated values

Sample Input Set



#### Sample Output Set:

\*\*\*\*\*\*\*Student Records System\*\*\*\*\*\*\*\*\*\*\*\*\*

1. Add new Record
2. Display All Records
3. Delete record by roll no.
4. Search a Record by Roll no.
5. Enter your choice 1

Enter a new Record- Student Roll No: 210054 Student Name: Sangram Kulkarni Division: SE A Address: Pune

Initialization of Input Set parameters:

class Student

{

int rollno;

char name[20],address\_city[20]; char div; int year; public: Student()

{

strcpy(name,“ ");

strcpy(address\_city,“ "); rollno=year=div=0;

//==========File Operations =========== class FileOperations

{

fstream file; public:

FileOperations(char\* filename)

#### Algorithms:

1. Algorithm for main function MAIN FUNCTION ()

S1: Read the filenames from user from database.

S2: Read the operations to be performed from the keyboard

S3: If the operation specified is create go to the create function, if the operation specified is display go to the display function, if the operation specified is add go to the

add function, if the operation specified is delete go to delete function, if the operation specified is display particular record go to the search function, if the operation specified is exit go to step 4.

S4: Stop

//==========File Operations =========== class FileOperations

{

fstream file; public:

FileOperations(char\* filename)

//==========File Operations =========== class FileOperations

{

fstream file; public:

FileOperations(char\* filename)

#### Algorithm for create function

S1: Open the file in the write mode, if the file specified is not found or unable to open then display error message and go to step5, else go to step2.

S2: Read the no: of records N to be inserted to the file. S3: Repeat the step4 N number of times.

S4: Read the details of each student from the keyboard and write the same to the file. S5: Close the file.

S6: Return to the main function

#### Algorithm for add a record

S1: Open the file in the append mode,if the file specified is not found or unable to open then display error message and go to step5 , else go to step2

S2: Scan all the student details one by one from file until end of file is reached. S3: Read the details of the form the keyboard and write the same to the file

S4: Close the file.

S5: Return to the main function

void insertRecord(int rollno, char name[MAX],int year, char div,char city[MAX])

{

Student s1(rollno,name,year,div,city); file.seekp(0,ios::end); file.write((char \*)&s1,sizeof(Student));

file.clear();

}

void deleteRecord(int rollno)

{

ofstream outFile("new.dat",ios::binary); file.seekg(0,ios::beg); bool flag=false;

Student s1;

while(file.read((char \*)&s1, sizeof(Student)))

{

if(s1.getRollNo()==rollno)

{

flag=true; continue;

}

outFile.write((char \*)&s1, sizeof(Student));

}

if(!flag)

#### Algorithm for deleting a record

S1: Open the file in the append mode, if the file specified is not found or unable to open then display error message and go to step5, else go to step2

S2: Accept the roll no from the user to delete the record

S3: Search for the roll no in file. If roll no. exits, copy all the records in the file except the one to be deleted in another temporary file.

S4: Close both files

S5: Now, remove the old file & name the temporary file with name same as that of old filename

void deleteRecord(int rollno)

{

ofstream outFile("new.dat",ios::binary); file.seekg(0,ios::beg); bool flag=false;

Student s1;

while(file.read((char \*)&s1, sizeof(Student)))

{

if(s1.getRollNo()==rollno)

{

flag=true; continue;

}

outFile.write((char \*)&s1, sizeof(Student));

}

if(!flag) cout<<"\nRecord of "<<rollno<<" is not present.";

}

#### Algorithm for displaying particular record (search)

S1: Open the file in the read mode, if the file specified is not found or unable to open then display error message and go to step6, else go to step2.

S2: Read the roll number of the student whose details need to be displayed. S3: Read each student record from the file.

S4: Compare the students roll number scanned from file with roll number specified by the user. S5: If they are equal then display the details of that record else display required roll number not found message and go to step6.

S6: Close the file.

S7: Return to the main function.

while(file.read((char\*)&s1,sizeof(Student)))

{

if(s1.getRollNo()==rollNo)

{

s1.displayRecord(); flag=true; break;

}

}

if(flag==false)

{

cout<<"\nRecord of "<<rollNo<<"is not present.";

}

file.clear();

}

#### Test Conditions:

1. Input valid filename.
2. Input valid record.
3. Check for opening / closing / reading / writing file errors Successful Test Conditions:

Create : Accept the records and write into the file. File created successfully Display : Display all records present in Master file.

Add : Accept the record to be add into the file at the end of the file. Record inserted successfully Delete: Accept the record to be deleted. Record deleted successfully

Search : Accept the record to be displayed by search. Display whether the record if it is present else record not present.

**Conclusion:** This program gives us the knowledge of sequential file

#### FAQ’s:

1. Define File? What are the factors affecting the file organization?
2. Compare Text and Binary File?
3. Explain the different File opening modes in C++?
4. What is indexed sequential file? Explain the primitive operations on indexed sequential file
5. Write a note on Direct Access File?
6. What are the advantages and disadvantages of indexed-sequential file organization?

# Group F : Assignment No. F-24

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

#### Title:

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.

#### Objective:

To understand the concept and basic of index sequential file and its use in Data structure

#### Outcome:

To implement the concept and basic of index sequential file and to perform basic operation as adding record, display all record, search record from index sequential file and its use in Data structure.

#### Software & Hardware Requirements:

64- bit Open source Linux or its derivative

Open Source C++ Programming tool like G++/GCC

#### Theory:

Indexed sequential access file organization

* + 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

#### Primitive operations on Index Sequential files:

* + Write (add, store): User provides a new key and record, IS file inserts the new record and key.
  + Sequential Access (read next): IS file returns the next record (in key order)
  + Random access (random read, fetch): User provides key, IS file returns the record or "not there"
  + Rewrite (replace): User provides an existing key and a new record, IS file replaces existing record with new.
  + Delete: User provides an existing key, IS file deletes existing record

#### Sample Input Set

#### Sample Output Set:

\*\*\*\*\*\*\*Employee Management System\*\*\*\*\*\*\*\*\*\*\*\*\*

1. Add new Record
2. Display All Records
3. Delete record by roll no.
4. Search a Record by Employee ID. 5.Enter your choice 1 Enter a new Record-

Employee ID: 101

Employee Name: Sangram Kulkarni Designation: HR Manager Salary: 60000\-

Algorithm:

Step 1 - Include the required header files (iostream.h, conio.h, and windows.h for colors). Step 2 - Create a class (employee) with the following members as public members. emp\_number, emp\_name, emp\_salary, as data members.

get\_emp\_details(), find\_net\_salary() and show\_emp\_details() as member functions.

Step 3 - Implement all the member functions with their respective code (Here, we have used scope resolution operator ::).

Step 4 - Create a main() method.

Step 5 - Create an object (emp) of the above class inside the main() method. Step 6 - Call the member functions get\_emp\_details() and show\_emp\_details(). Step 7 - returnn 0 to exit form the program execution.

#### Approach:

1. For storing the data of the employee, create a user define datatype which will store the information regarding Employee. Below is the declaration of the data type:
2. struct employee {
3. string name;
4. long int Employee\_id;
5. string designation;
6. int salary;
7. };

using namespace std; class employee

{

int emp\_number; char emp\_name[20]; float emp\_basic; float emp\_da;

float emp\_it;

float emp\_net\_sal; public:

void get\_emp\_details(); float find\_net\_salary(float basic, float da, float it); void show\_emp\_details();

};

#### Building the Employee's table:

For building the employee table the idea is to use the array of the above struct datatype which will use to store the information regarding employee. For storing information at index i the data is stored as:

struct employee emp[10];

emp[i].name = "PDEAsforPdeas" emp[i].code = "12345" emp[i].designation = "Organisation"

emp[i].exp = 10

emp[i].age = 10 void build()

{

cout << "Build The Table\n";

cout << "Maximum Enteries can be "<< max << "\n";

cout << "Enter the number of "<< "Enteries required"; cin >> num; if (num > 20) {

cout << "Maximum number of "<< "Enteries are 20\n"; num = 20;

}

cout << "Enter the following data:\n"; for (int i = 0; i < num; i++) {

cout << "Name "; cin >> emp[i].name;

cout << "Employee ID "; cin >> emp[i].code;

cout << "Designation "; cin >> emp[i].designation; cout << “Salary ";

cin >> emp[i].salary;

}

showMenu();

}

Deleting in the record:

Since we are using array to store the data, therefore to delete the data at any index shift all the data at that index by 1 and delete the last data of the array by decreasing the size of array by 1.

#### // Function to delete record at index i

void deleteIndex(int i)

{

for (int j = i; j < num - 1; j++) { emp[j].name = emp[j + 1].name; emp[j].code = emp[j + 1].code;

emp[j].designation = emp[j + 1].designation; emp[j].exp = emp[j + 1].exp; emp[j].age = emp[j +

1].age;

}

return;

}

#### // Function to delete record

void deleteRecord()

{

cout << "Enter the Employee ID "<< "to Delete Record"; int code; cin >> code;

for (int i = 0; i < num; i++) { if (emp[i].code == code) { deleteIndex(i);

num--; break;

}

}

showMenu();

}

#### Searching in the record:

For searching in the record based on any parameter, the idea is to traverse the data and if at any index the value parameters matches with the record stored, print all the information of that employee.

void searchRecord()

{

cout << "Enter the Employee"<< " ID to Search Record"; int code;

cin >> code;

for (int i = 0; i < num; i++) {

// If the data is found

if (emp[i].code == code) { cout << "Name "<< emp[i].name << "\n"; cout << "Employee ID "<< emp[i].code << "\n";

cout << "Designation "<< emp[i].designation << "\n"; cout << "Experience "<< emp[i].exp << "\n";

cout << "Age "<< emp[i].age << "\n"; break;

}

}

**Conclusion:** This program gives us the knowledge of index sequential file. FAQ’s:

1. Define File? What are the factors affecting the file organization?
2. Compare Text and Binary File?
3. Explain the different File opening modes in C++?
4. What is indexed sequential file? Explain the primitive operations on indexed sequential file.
5. Write a note on Direct Access File?
6. What are the advantages and disadvantages of indexed-sequential file organization?
7. What are the advantages of Sequential File Organization?