

DATA STRUCTURES & ALGORITHMS LABORATORY

Student Manual

2021-2022



Pravara Rural College of Engineering, Loni.
Department of Computer Engineering
Data Structures & Algorithms Laboratory
INDEX

Sr. No	Title of Experiments	Page No	Date of		Remark	Signature
			Performance	Submission		
Group A:						
1	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					
2	Implement all the functions of a dictionary (ADT) using hashing and handle collisions using chaining with / without replacement. 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)					
Group B:						
3	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.					

4	<p>Beginning with an empty binary search tree, Construct binary search tree by inserting the values in the order given. After constructing a binary tree -</p> <ul style="list-style-type: none"> i. Insert new node ii. Find number of nodes in longest path from root 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. 					
5	<p>A Dictionary stores keywords and 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 Binary Search Tree for implementation</p>					

Group C:						
6	Represent a given graph using adjacency matrix/list to perform DFS and using adjacency list to perform BFS. Use the map of the area around the college as the graph. Identify the prominent land marks as nodes and perform DFS and BFS on that.					
7	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 takes 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. Justify the storage representation used.					
Group D:						
8	Given sequence $k = k_1 < k_2 < \dots < k_n$ of n sorted keys, with a search probability p_i for each key k_i . Build the Binary search tree that has the least search cost given the access probability for each key.					
9	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					

Group E:						
10	Read the marks obtained by students of second year in an online examination of particular subject. Find out maximum and minimum marks obtained in that subject. Use heap data structure. Analyze the algorithm.					
Group F:						
11	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 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.					
12	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					
Mini-Projects/ Case Study						
28	Design a mini project to implement Snake and Ladders Game using Python.					
OR						
29	Design a mini project to implement a Smart text editor.					



CERTIFICATE

This is to certify that Mr. /Ms. _____

Roll no-_____Examination seat no-_____has performed above mentioned practical's in the college.

Faculty In-charge

Dr. M.R. Bendre
Head of the Department

Experiment No.: 1**Title:**

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

Learning Objectives:

- To understand concept of Hashing
- To understand to find record quickly using hash function.
- To understand collision handling techniques.

Learning Outcome:

- Understand & implement concept of hash table.
- Understand collision handling techniques.

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:

```
employees[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:

```
employees["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;
}

```

Collision:

Since a hash function gets us a small number for a key which is a big integer or string, there is a possibility that two keys result in the same value. The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique.

How to handle Collisions?

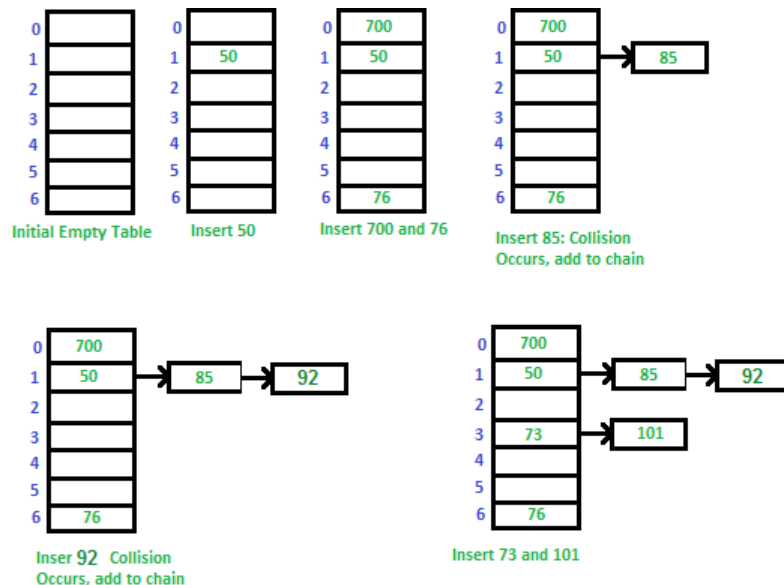
There are mainly two methods to handle collision:

- 1) Separate Chaining
- 2) Open Addressing

Separate Chaining:

The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

Let us consider a simple hash function as “**key mod 7**” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

**Advantages:**

- 1) Simple to implement.
- 2) Hash table never fills up, we can always add more elements to the chain.
- 3) Less sensitive to the hash function or load factors.
- 4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

- 1) Cache performance of chaining is not good as keys are stored using a linked list. Open

addressing provides better cache performance as everything is stored in the same table.

2) Wastage of Space (Some Parts of hash table are never used)

3) If the chain becomes long, then search time can become $O(n)$ in the worst case.

4) Uses extra space for links.

Open Addressing

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, the size of the table must be greater than or equal to the total number of keys. Open Addressing is done in the following ways:

a) Linear Probing: In linear probing, we linearly probe for next slot. For example, the typical gap between two probes is 1 as taken in below example also.

let **hash(x)** be the slot index computed using a hash function and **S** be the table size

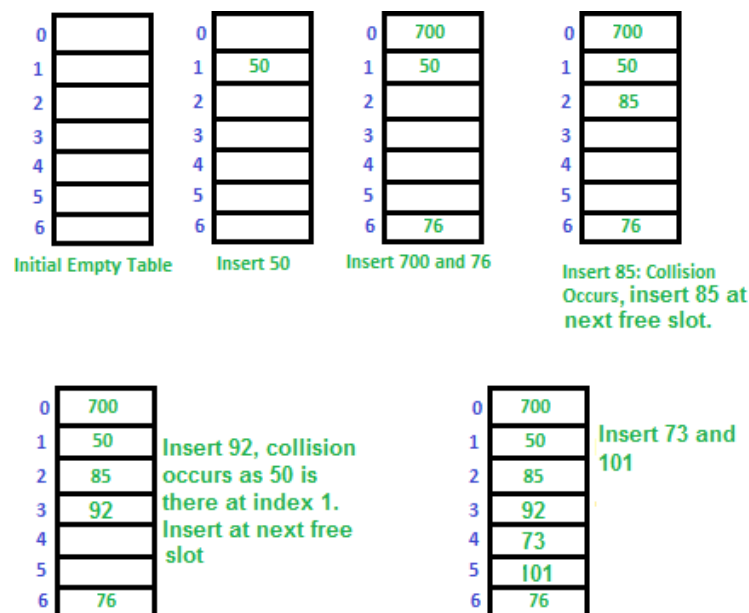
If slot $\text{hash}(x) \% S$ is full, then we try $(\text{hash}(x) + 1) \% S$

If $(\text{hash}(x) + 1) \% S$ is also full, then we try $(\text{hash}(x) + 2) \% S$

If $(\text{hash}(x) + 2) \% S$ is also full, then we try $(\text{hash}(x) + 3) \% S$

.....
.....

Let us consider a simple hash function as “key mod 7” and a sequence of keys as 50, 700, 76, 85, 92, 73, 101.



Challenges in Linear Probing:

- 1. Primary Clustering:** One of the problems with linear probing is primary clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.

- 2. Secondary Clustering:** Secondary clustering is less severe, two records do only have the same collision chain (Probe Sequence) if their initial position is the same.

b) Quadratic Probing We look for i^2 'th slot in i 'th iteration.

let $\text{hash}(x)$ be the slot index computed using hash function.

If slot $\text{hash}(x) \% S$ is full, then we try $(\text{hash}(x) + 1*1) \% S$

If $(\text{hash}(x) + 1*1) \% S$ is also full, then we try $(\text{hash}(x) + 2*2) \% S$

If $(\text{hash}(x) + 2*2) \% S$ is also full, then we try $(\text{hash}(x) + 3*3) \% S$

.....

.....

c) Double Hashing We use another hash function $\text{hash2}(x)$ and look for $i*\text{hash2}(x)$ slot in i 'th rotation.

let $\text{hash}(x)$ be the slot index computed using hash function.

If slot $\text{hash}(x) \% S$ is full, then we try $(\text{hash}(x) + 1*\text{hash2}(x)) \% S$

If $(\text{hash}(x) + 1*\text{hash2}(x)) \% S$ is also full, then we try $(\text{hash}(x) + 2*\text{hash2}(x)) \% S$

If $(\text{hash}(x) + 2*\text{hash2}(x)) \% S$ is also full, then we try $(\text{hash}(x) + 3*\text{hash2}(x)) \% S$

.....

.....

Comparison of above three:

Linear probing has the best cache performance but suffers from clustering. One more advantage of Linear probing is easy to compute.

Quadratic probing lies between the two in terms of cache performance and clustering.

Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

Sr. No.	Separate Chaining	Open Addressing
1.	Chaining is Simpler to implement.	Open Addressing requires more computation.
2.	In chaining, Hash table never fills up, we can always add more elements to chain.	In open addressing, table may become full.
3.	Chaining is Less sensitive to the hash function or load factors.	Open addressing requires extra care for to avoid clustering and load factor.
4.	Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.	Open addressing is used when the frequency and number of keys is known.
5.	Cache performance of chaining is not good as keys are stored using linked list.	Open addressing provides better cache performance as everything is stored in the same table.
6.	Wastage of Space (Some Parts of hash table in chaining are never	In Open addressing, a slot can be used even if an input doesn't map to it.

	used).	
7.	Chaining uses extra space for links.	No links in Open addressing

Conclusion:-

Hence we have studied successfully the use of a hash table & implementing it

Staff Signature & Date

Experiment No.: 2

Title:

Implement all the functions of a dictionary (ADT) using hashing and handle collisions using chaining with / without replacement.

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)

Learning Objectives:

1. To understand Dictionary (ADT)
2. To understand concept of hashing and collision handling.
3. 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 **k** or null, if dictionary contains no such key
- **remove(Dictionary d, Key k)** removes
key **k** and associated value

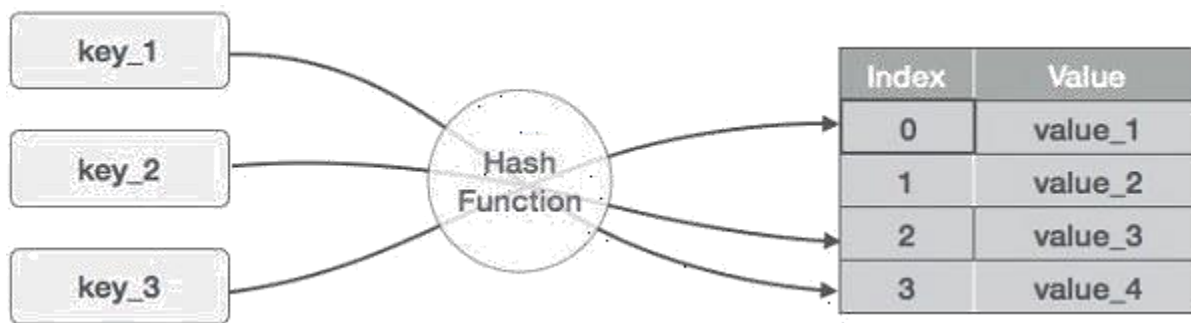
- **destroy(Dictionary d)** destroys dictionary **d**

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.



Conclusion :-

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

Staff Signature & Date

Experiment No.: 3

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.

Learning Objectives:

- To understand concept of tree data structure
- To understand concept & features of object oriented programming.
- To understand concept of class
- To understand concept of tree data structure.

Learning Outcome:

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

Theory:

Tree:

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 shown in figure below. 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 formally, 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.

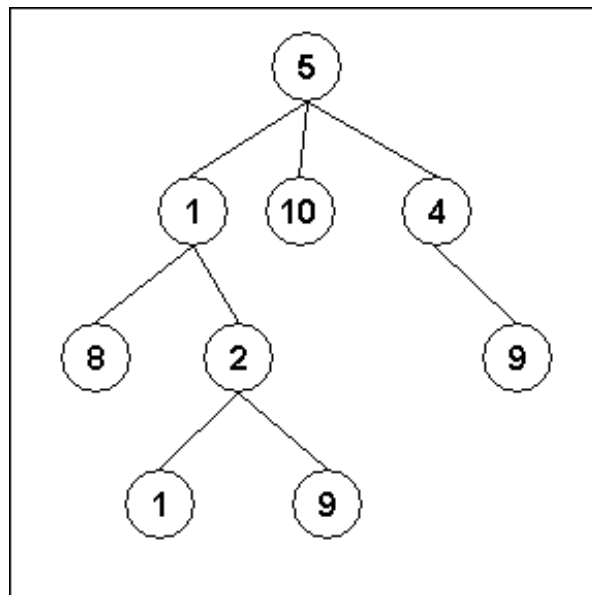


Fig1. An example of a tree

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

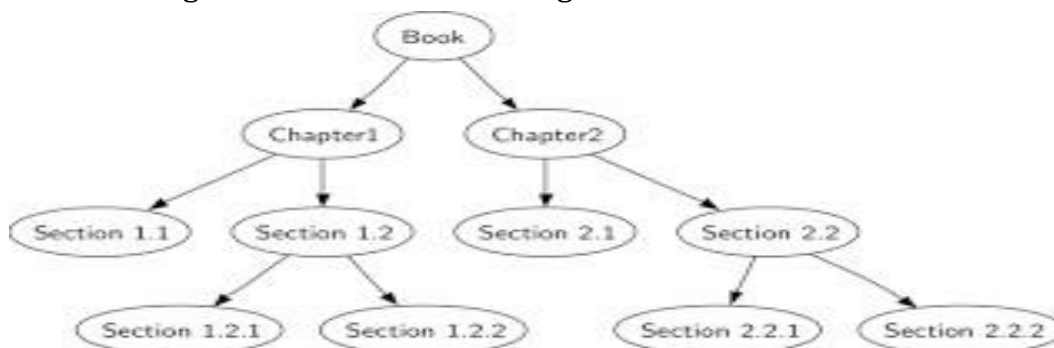
Following are the important terms with respect to tree.

- **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.

Staff Signature & Date

Experiment No.: 4

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 -

- vi. Insert new node
- vii. Find number of nodes in longest path from root
- viii. Minimum data value found in the tree
- ix. Change a tree so that the roles of the left and right pointers are swapped at every node
- x. Search a value.

Outcome: Classify the Tree data structures & apply it for problem solving

Objective: To study Different operations on binary tree

Theory:

A binary tree is composed of nodes connected by edges. Some binary tree is either empty or consists of a single root element with two distinct binary tree child elements known as the left subtree and the right subtree of . As the name binary suggests, a node in a binary tree has a maximum of children.

An important special kind of binary tree is the binary search tree (BST). In a BST, each node stores some information including a unique key value, and perhaps some associated data. A binary tree is a BST iff, for every node n in the tree:

- All keys in n 's left subtree are less than the key in n , and
- all keys in n 's right subtree are greater than the key in n .

Algorithm:

1. Create a new BST node and assign values to it.

2. insert(node, key)

i) If root == NULL,

return the new node to the calling function.

ii) if root=>data < key

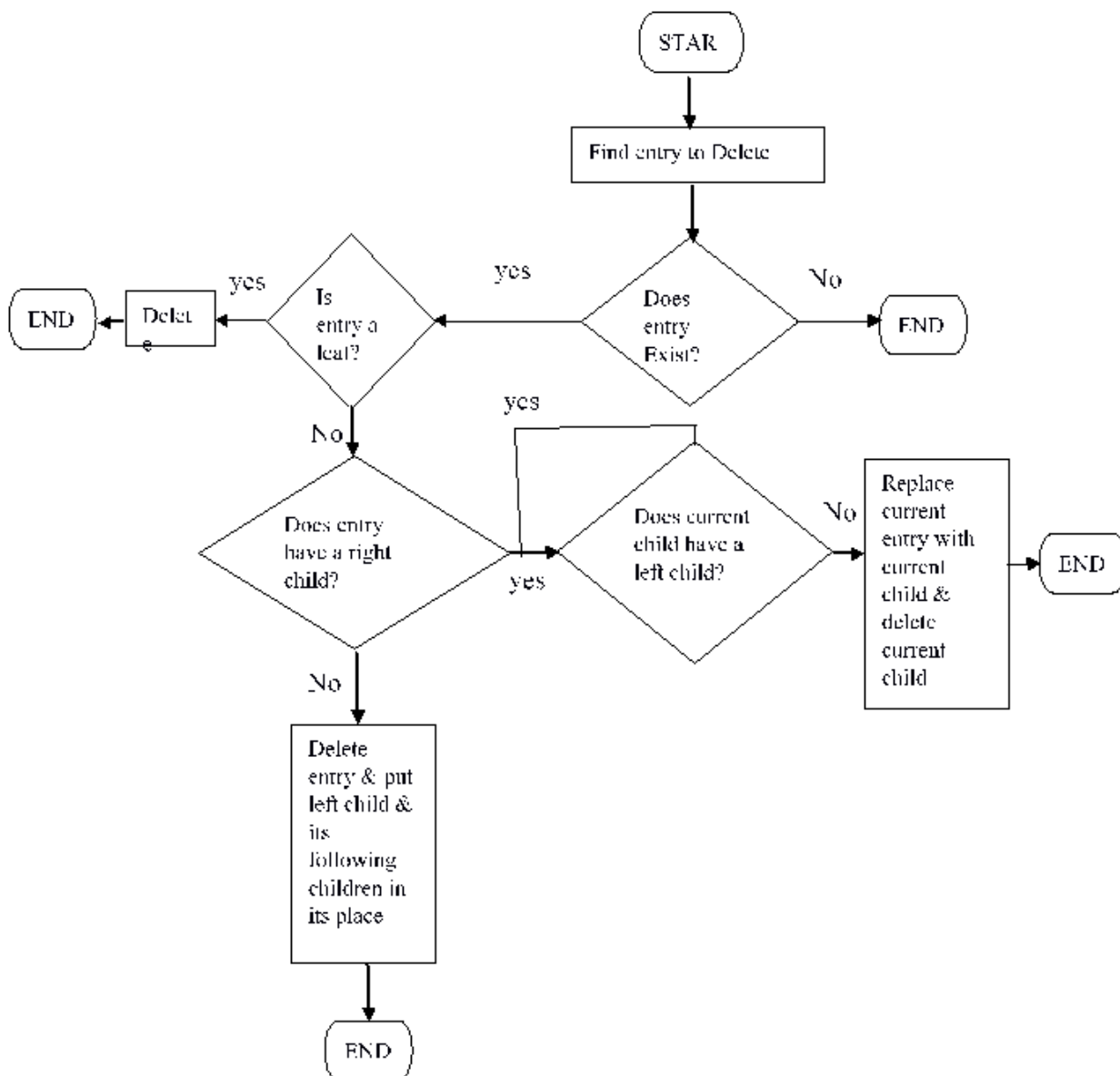
call the insert function with root=>right and assign the return value in root=>right. root->right = insert(root=>right,key)

iii) if root=>data > key

call the insert function with root->left and assign the return value in root=>left. root=>left = insert(root=>left,key)

3. Finally, return the original root pointer to the calling function

Flowchart:



Test Cases:

Sr. No	Test ID	Steps	Input	Expected Result	Actual Result	Status (Pass/Fail)
1	ID01	To insert new node in the BST	NULL tree & a key value	Node to be inserted in the tree	Not possible to insert	FAIL
2	ID02	Find a node in the BST	Root of the tree & key value	Node to be found in the tree	In case if key is not present, results not found the key	FAIL
3	ID03	Delete a left child leaf node from BST	Root of the tree & key value	Node to be deleted	It can be easily deleted	PASS
4	ID04	Delete a right child leaf node from BST	Root of the tree & key value	Delete key & put left child & its following children in its place	The key can be easily deleted	PASS
5	ID05	Delete a node which is having both children	Root of the tree & key value	Replace current entry with current child & delete current child	The key can be easily deleted	PASS

Conclusion:-

Hence we have studied successfully how to delete any node from binary tree using operator overloading.

Staff Signature & Date

Experiment No.: 5**Title:**

A Dictionary stores keywords and 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 Binary Search Tree for implementation.

Learning Objectives:

- To study how trees are used to represent data in hierarchical manner.
- To define binary search tree (BST) that allow both rapid retrieval by nkey and in order traversal.

Learning Outcomes:

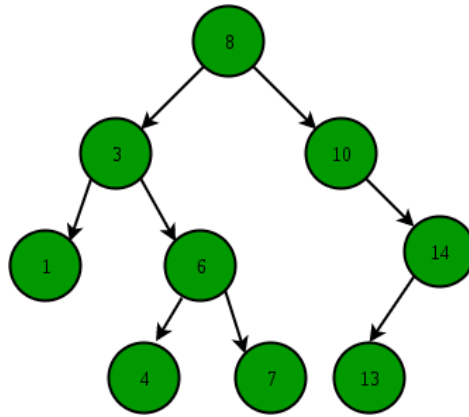
- Understand the basic concept of binary search tree.
- Understand the process of Tree traversal by visiting each node in the tree exactly once in a certain order.

Theory:

A binary search tree (BST) is a binary tree where each node has a Comparable key (and an associated value) and satisfies the restriction that the key in any node is larger than the keys in all nodes in that node's left subtree and smaller than the keys in all nodes in that node's right subtree.

Binary Search Tree A binary search tree is a rooted binary tree, whose internal nodes each store a key (and optionally, an associated value) and each have two distinguished subtrees, commonly denoted left and right. The tree additionally satisfies the binary search property, which states that the key in each node must be greater than or equal to any key stored in the left sub-tree, and less than or equal to any key stored in the right sub-tree. (The leaves (final nodes) of the tree contain no key and have no structure to distinguish them from one another. Leaves are commonly represented by a special leaf or nil symbol, a NULL pointer, etc.) Frequently, the information represented by each node is a record rather than a single data element. However, for sequencing purposes, nodes are compared according to their keys rather than any part of their associated records.

The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient; they are also easy to code. Binary search trees are a fundamental data structure used to construct more abstract data structures such as sets, multisets, and associative arrays.



The above properties of Binary Search Tree provides an ordering among keys so that the operations like search, minimum and maximum can be done fast. If there is no ordering, then we may have to compare every key to search for a given key.

To sort given input first creates a binary search tree from the elements of the input list or array and then performs an in-order traversal on the created binary search tree to get the elements in sorted order.

Algorithm:

Step 1: Take the elements input in an array.

Step 2: Create a Binary search tree by inserting data items from the array into the Binary search tree.

Step 3: Perform in-order traversal on the tree to get the elements in sorted order.

Average Case Time Complexity : $O(n \log n)$ Adding one item to a Binary Search tree on average takes $O(\log n)$ time. Therefore, adding n items will take $O(n \log n)$ time

Worst Case Time Complexity : $O(n^2)$. The worst case time complexity of Tree Sort can be improved by using a self-balancing binary search tree like Red Black Tree, AVL Tree. Using self-balancing binary tree will take $O(n \log n)$ time to sort the array in worst case.

Auxiliary Space: $O(n)$

Given a binary search tree, to sort list in decreasing order, the value of each node must be greater than the values of all the nodes at its right, and its left node must be NULL after flattening. We must do it in $O(H)$ extra space where 'H' is the height of BST.

Examples:**Input:**

```
    5
   / \
  3   7
 / \ / \
2 4 6 8
```

Output: 8 7 6 5 4 3 2

Approach: A simple approach will be to recreate the BST from its 'reverse in-order' traversal.

This will take $O(N)$ extra space where N is the number of nodes in BST.

To improve upon that, let's simulate the reverse in-order traversal of a binary tree as follows:

1. Create a dummy node.
2. Create a variable called 'prev' and make it point to the dummy node.
3. Perform reverse in-order traversal and at each step.
 - Set $\text{prev} \rightarrow \text{right} = \text{curr}$
 - Set $\text{prev} \rightarrow \text{left} = \text{NULL}$
 - Set $\text{prev} = \text{curr}$

This will improve the space complexity to $O(H)$ in the worst case as in-order traversal takes $O(H)$ extra space.

Conclusion:-

Dictionary assignment has implemented successfully using binary search tree.

Staff Signature & Date

Experiment No.: 6

Title:

Represent a given graph using adjacency matrix/list to perform DFS and using adjacency list to perform BFS. Use the map of the area around the college as the graph. Identify the prominent land marks as nodes and perform DFS and BFS on that.

Learning Objectives:

- To understand directed and undirected graph.
- To implement program to represent graph using adjacency matrix and list.
- To understand DFS and BFS

Learning Outcome:

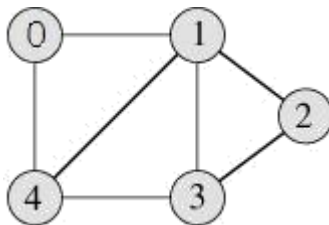
Student will be able to implement program for graph representation.

Theory:

Graph is a data structure that consists of following two components:

1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not same as (v, u) in case of directed graph(di-graph). The pair of form (u, v) indicates that there is an edge from vertex u to vertex v . The edges may contain weight/value/cost.

Graphs are used to represent many real life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, facebook. For example, in facebook, each person is represented with a vertex (or node). Each node is a structure and contains information like person id, name, gender and locale. See this for more applications of graph. Following is an example undirected graph with 5 vertices.



Following two are the most commonly used representations of graph.

1. Adjacency Matrix
2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of the graph representation is situation specific. It totally depends on the type of operations to be performed and ease of use.

Adjacency Matrix:

Adjacency Matrix is a 2D array of size $V \times V$ where V is the number of vertices in a graph. Let the 2D array be $adj[][]$, a slot $adj[i][j] = 1$ indicates that there is an edge from vertex i to vertex j . Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If $adj[i][j] = w$, then there is an edge from vertex i to vertex j with weight w .

The adjacency matrix for the above example graph is:

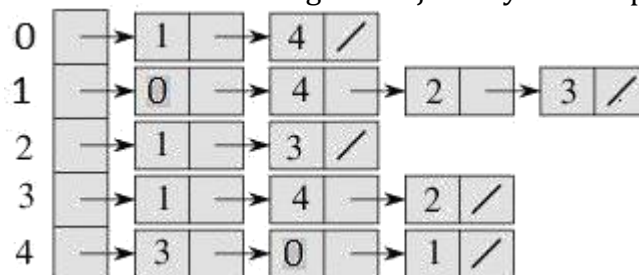
	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Pros: Representation is easier to implement and follow. Removing an edge takes $O(1)$ time. Queries like whether there is an edge from vertex 'u' to vertex 'v' are efficient and can be done in $O(1)$.

Cons: Consumes more space $O(V^2)$. Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is $O(V^2)$ time.

Adjacency List:

An array of linked lists is used. Size of the array is equal to number of vertices. Let the array be $array[]$. An entry $array[i]$ represents the linked list of vertices adjacent to the i th vertex. This representation can also be used to represent a weighted graph. The weights of edges can be stored in nodes of linked lists. Following is adjacency list representation of the above graph.



Pros: Saves space $O(|V| + |E|)$. In the worst case, there can be $C(V, 2)$ number of edges in a graph thus consuming $O(V^2)$ space. Adding a vertex is easier.

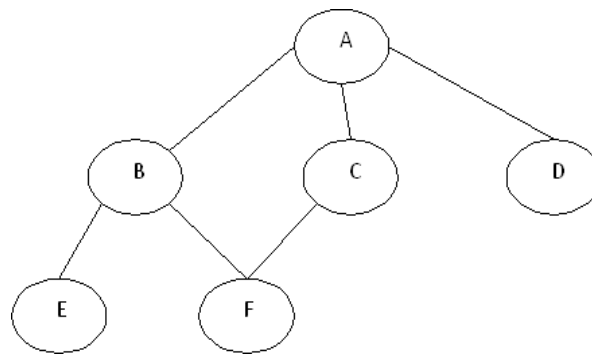
Cons: Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done $O(V)$.

Graph Traversal:

The breadth first search (BFS) and the depth first search (DFS) are the two algorithms used for traversing and searching a node in a graph. They can also be used to find out whether a node is reachable from a given node or not.

Depth First Search (DFS)

The aim of DFS algorithm is to traverse the graph in such a way that it tries to go far from the root node. Stack is used in the implementation of the depth first search. Let's see how depth first search works with respect to the following graph:



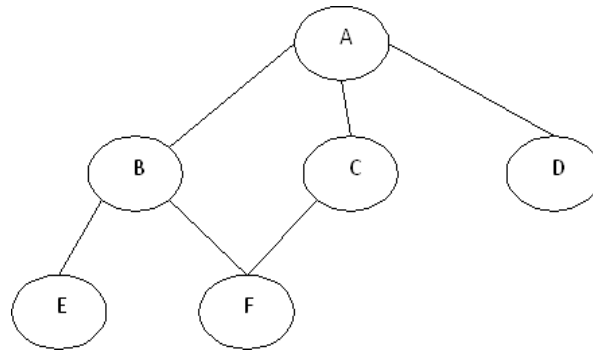
As stated before, in DFS, nodes are visited by going through the depth of the tree from the starting node. If we do the depth first traversal of the above graph and print the visited node, it will be "A B E F C D". DFS visits the root node and then its children nodes until it reaches the end node, i.e. E and F nodes, then moves up to the parent nodes.

Algorithmic Steps

1. **Step 1:** Push the root node in the Stack.
2. **Step 2:** Loop until stack is empty.
3. **Step 3:** Peek the node of the stack.
4. **Step 4:** If the node has unvisited child nodes, get the unvisited child node, mark it as traversed and push it on stack.
5. **Step 5:** If the node does not have any unvisited child nodes, pop the node from the stack.

Breadth First Search (BFS)

This is a very different approach for traversing the graph nodes. The aim of BFS algorithm is to traverse the graph as close as possible to the root node. Queue is used in the implementation of the breadth first search. Let's see how BFS traversal works with respect to the following graph:



If we do the breadth first traversal of the above graph and print the visited node as the output, it will print the following output. "A B C D E F". The BFS visits the nodes level by level, so it will start with level 0 which is the root node, and then it moves to the next levels which are B, C and D, then the last levels which are E and F.

Algorithmic Steps

1. **Step 1:** Push the root node in the Queue.
2. **Step 2:** Loop until the queue is empty.
3. **Step 3:** Remove the node from the Queue.
4. **Step 4:** If the removed node has unvisited child nodes, mark them as visited and insert the unvisited children in the queue.

Conclusion:-

Hence we have successfully studied DFS and BFS in Binary Tree.

Staff Signature & Date

Experiment No.: 7

Title:

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 takes 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 or use adjacency matrix representation for graph. Check whether the graph is connected or not. Justify the storage representation used.

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.
- Analyse working of functions.

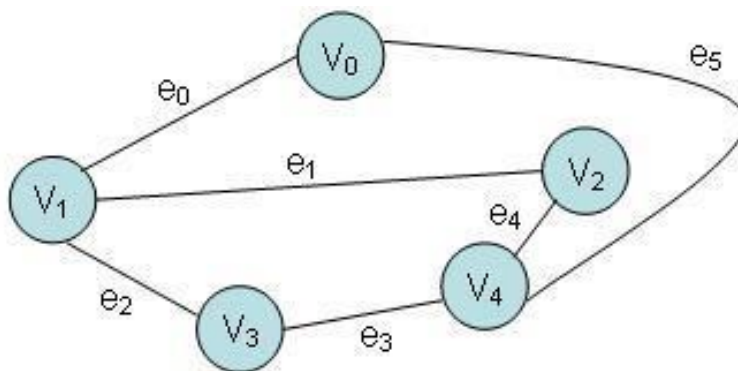
Theory:

Graph:

Graphs are the most general data structure. They are also commonly used data structures. A non-linear data structure consisting of nodes and links between nodes.

Undirected graph:

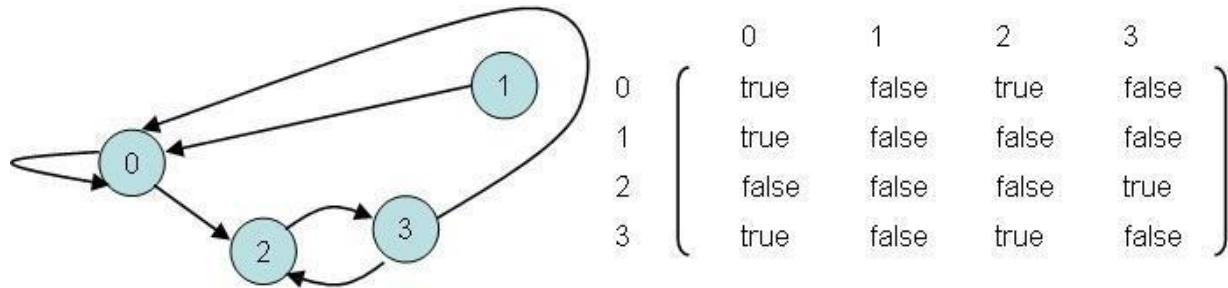
- An undirected graph is a set of nodes and a set of links between the nodes.
- Each node is called a **vertex**, each link is called an **edge**, and each edge connects two vertices.
- The order of the two connected vertices is unimportant.
- An undirected graph is a finite set of vertices together 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.

Representing Graphs with an Adjacency Matrix



Graph and adjacency matrix

Adjacency Matrix:

An adjacency matrix is a square grid of true/false values that represent the edges of a graph

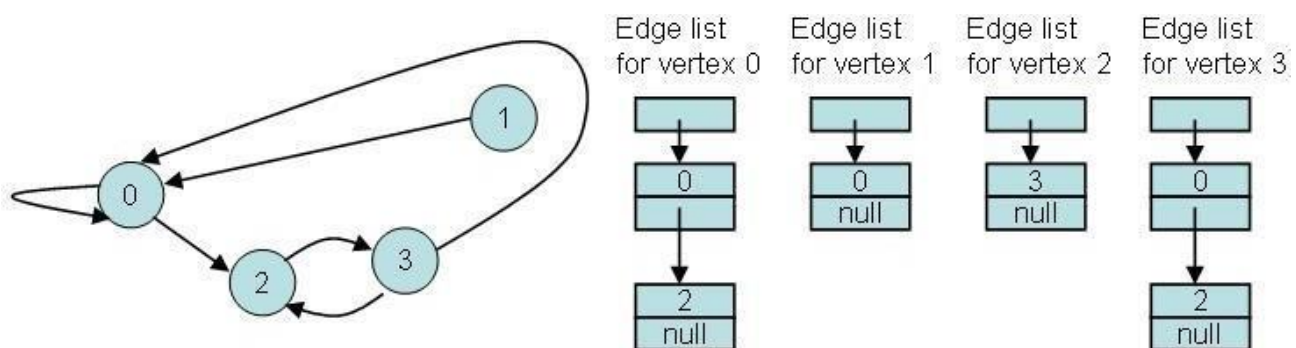
- If the graph contains n vertices, then the grid contains n rows and n columns.
- For two vertex numbers i and j , the component at row i and column j is true if there is an edge from vertex i to vertex j ; otherwise, the component is false.

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.

Representing Graphs with Edge Lists



Graph and adjacency list for each node

Adjacency List:

- 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.

Conclusion:-

Graph representation using adjacency matrix is implemented successfully.

Staff Signature & Date

Experiment No.: 8

Title:

Given sequence $k = k_1 < k_2 < \dots < k_n$ of n sorted keys, with a search probability p_i for each key k_i . Build the Binary search tree that has the least search cost given the access probability for each key?

Learning Objectives:

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

Learning Outcome:

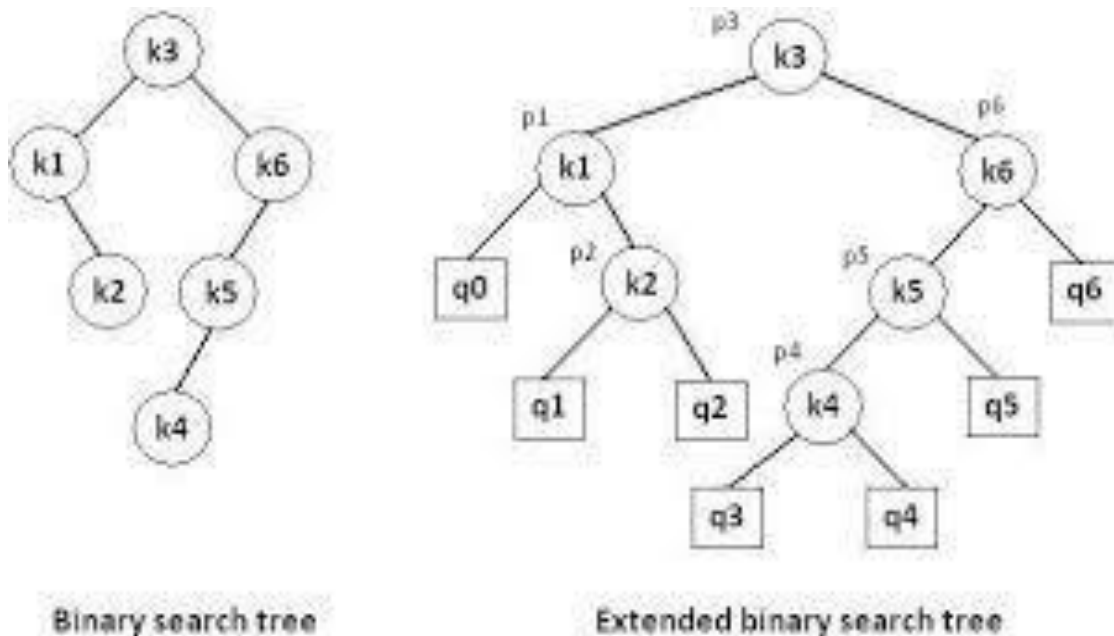
- Define class for Extended binary search tree using Object Oriented features.
- Analyse OBST.

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 k_1, k_2, \dots, 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 $k_1 < k_2 < \dots < k_n$.

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 ($p_1 \dots p_6$). 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:
 - the key is found, so the corresponding weight 'p' is incremented;
 - 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 k_1 can be interpreted as representing all key values that are not stored and are less than k_1 . Similarly, the terminal node in the extended tree that is the right successor of k_n , represents all key values not stored in the tree that are greater than k_n . The terminal node that is success between k_i and k_{i-1} in an inorder traversal represent all key values not stored that lie between k_i and k_{i-1} .

Algorithm

We have the following procedure for determining $R(i, j)$ and $C(i, j)$ with $0 \leq i \leq j \leq n$:

PROCEDURE COMPUTE_ROOT($n, p, q; R, C$)

begin

for $i = 0$ to n do

$C(i, i) \leftarrow 0$

$W(i, i) \leftarrow q(i)$

for $m = 0$ to n do

for $i = 0$ to $(n - m)$ do

$j \leftarrow i + m$

$W(i, j) \leftarrow W(i, j - 1) + p(j) + q(j)$

*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 $O(n^2)$ time and $O(n^2)$ 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.

Conclusion:-

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

Staff Signature & Date

Experiment No.: 9

Title:

A Dictionary stores keywords and 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.

Learning Objectives:

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

Learning Outcome:

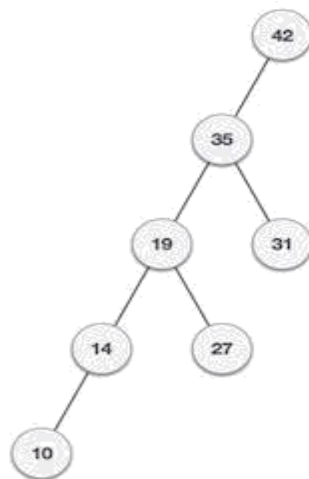
- Define class for AVL using Object Oriented features.
- Analyze working of various operations on AVL Tree .

Theory:

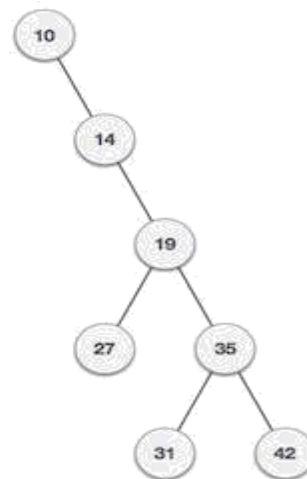
An empty tree is height balanced tree if T is a nonempty binary tree with TL and TR as its left and right sub trees. The T is height balance if and only if Its balance factor is 0, 1, -1.

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 –



If input 'appears' non-increasing manner

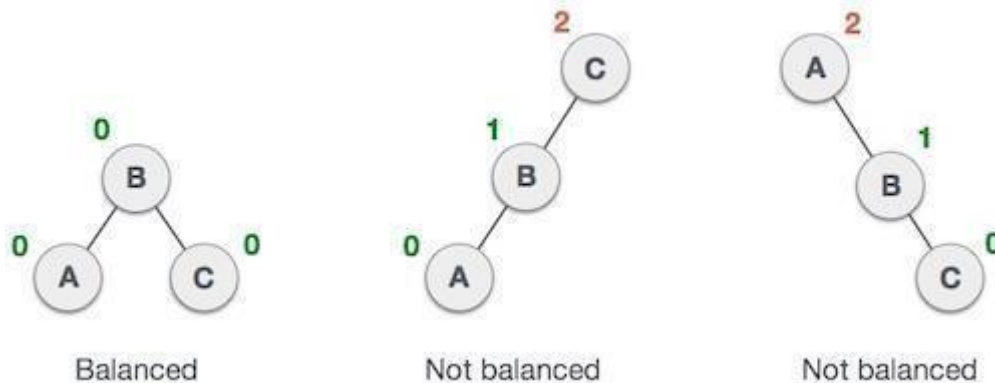


If input 'appears' in non-decreasing manner

It is observed that BST's worst-case performance is closest to linear search algorithms, that is $O(n)$. 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.

$$\text{BalanceFactor} = \text{height}(\text{left-subtree}) - \text{height}(\text{right-subtree})$$

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

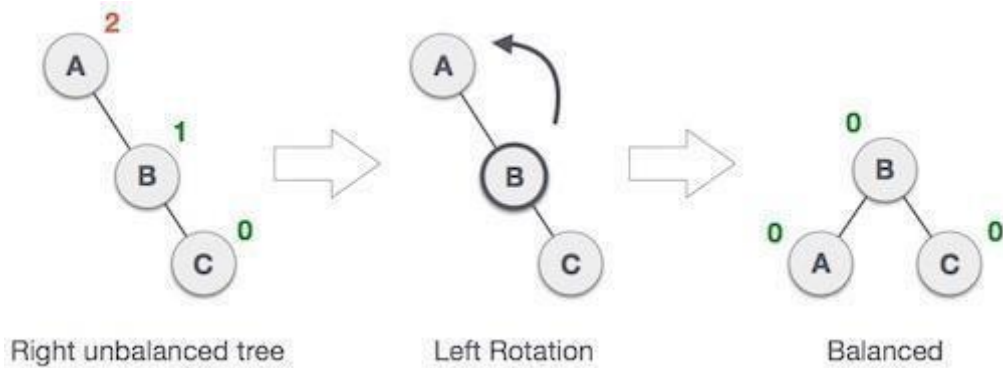
To balance itself, an AVL tree may perform the following four kinds of rotations –

- 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

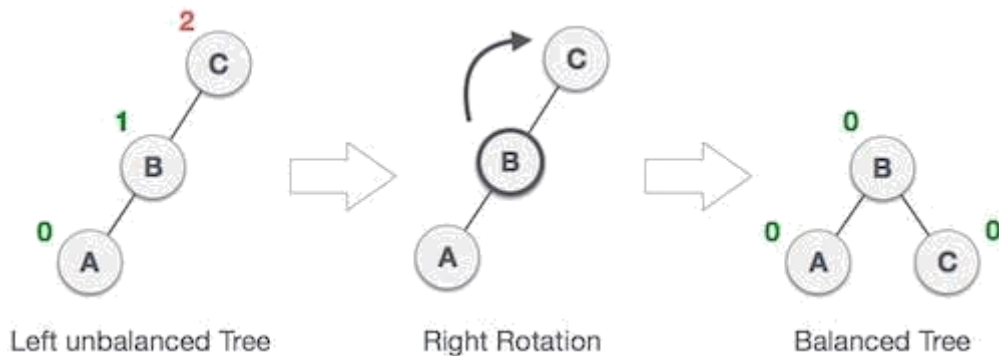
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation –



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

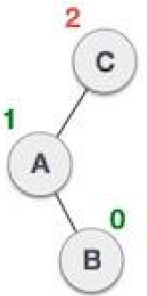
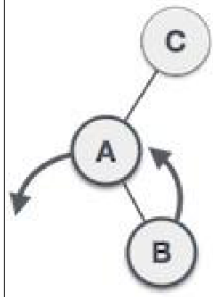
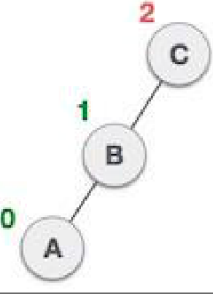
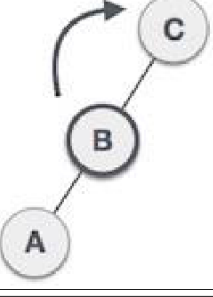
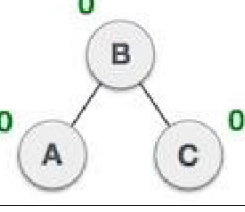
AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



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

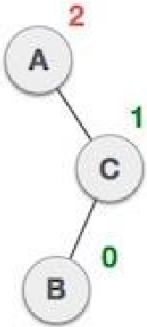
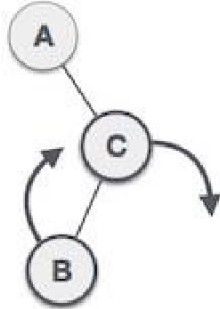
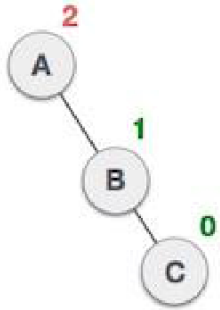
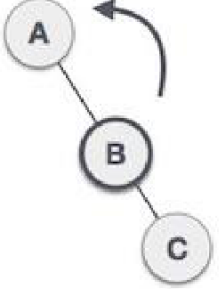
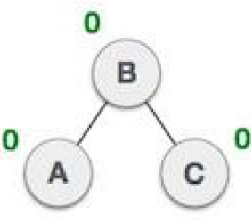
Left-Right Rotation

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
	<p>A node has been inserted into the right subtree of the left subtree.</p> <p>This makes C an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.</p>
	<p>We first perform the left rotation on the left subtree of C.</p> <p>This makes A, the left subtree of B.</p>
	<p>Node C is still unbalanced, however now, it is because of the left-subtree of the left-subtree.</p>
	<p>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.</p>
	<p>The tree is now balanced.</p>

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
	<p>A node has been inserted into the left subtree of the right subtree. This makes A, an unbalanced node with balance factor 2.</p>
	<p>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.</p>
	<p>Node A is still unbalanced because of the right subtree of its right subtree and requires a left rotation.</p>
	<p>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.</p>
	<p>The tree is now balanced.</p>

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 =singlesrotateleft(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 =singlesrotateright(P)
            else
                P =doublerotateright(P)
4. else
Print already exists
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
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);

Conclusion:-

Dictionary assignment has implemented successfully using binary search tree.

Staff Signature & Date

Experiment No.: 10**Title:**

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

Learning Objectives:

- To understand concept of heap
- 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–

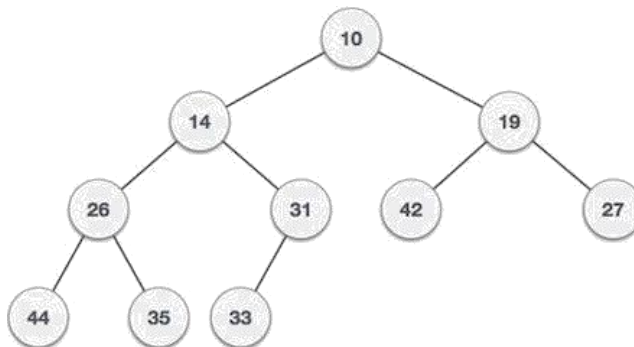
$$\text{key}(\alpha) \geq \text{key}(\beta)$$

As the value of parent is greater than that of child, this property generates **Max Heap**.

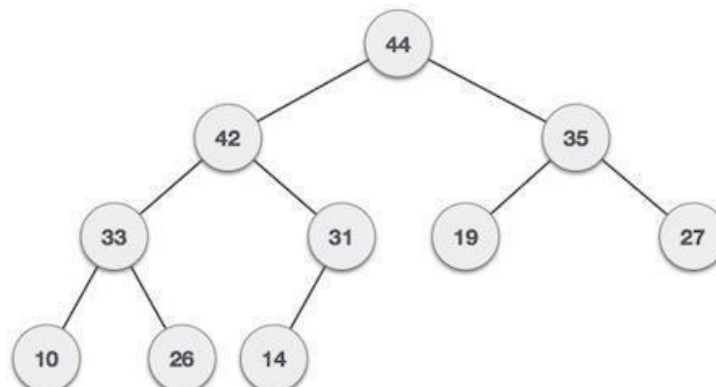
Based on this criteria, a heap can be of two types –

For Input \rightarrow 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.



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.

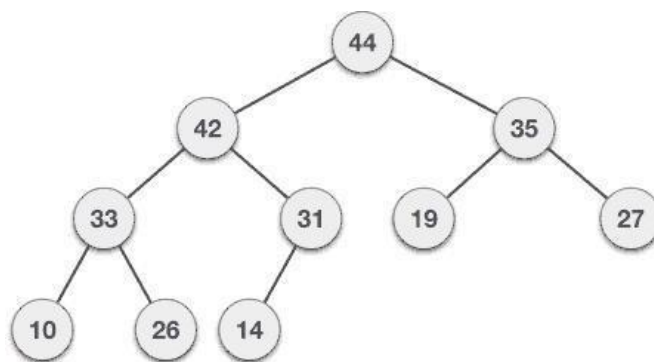
Step 1 – Remove root node.

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.



Conclusion:-

Max Heap and Min heap are implemented successfully.

Staff Signature & Date

Experiment No.: 11

Title:

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 main the data.

Learning Objectives:

- To understand concept of file organization in data structure.
- To understand concept & features of sequential file organization.

Learning Outcome:

- Define class for sequential file using Object Oriented features.
- Analyze working of various operations on sequential file .

Theory:

File organization refers to the relationship of the key of the record to the physical location of that record in the computer file. File organization may be either physical file or a logical file. A physical file is a physical unit, such as magnetic tape or a disk. A logical file on the other hand is a complete set of records for a specific application or purpose. A logical file may occupy a part of physical file or may extend over more than one physical file.

There are various methods of file organizations. These methods may be efficient for certain types of access/selection meanwhile it will turn inefficient for other selections. Hence it is up to the programmer to decide the best suited file organization method depending on his requirement.

Some of the file organizations are

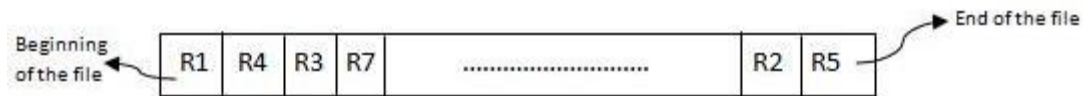
1. Sequential File Organization
2. Heap File Organization
3. Hash/Direct File Organization
4. Indexed Sequential Access Method
5. B+ Tree File Organization
6. Cluster File Organization

Sequential File Organization:

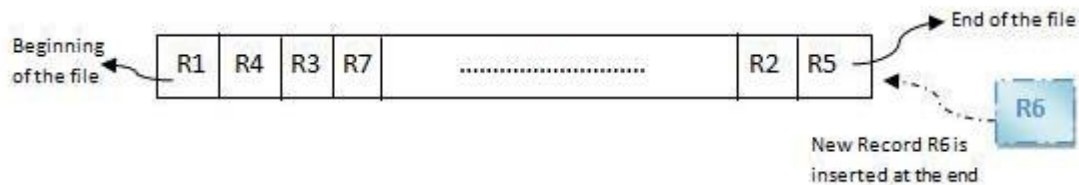
It is one of the simple methods of file organization. Here each file/records are stored one after the other in a sequential manner. This can be achieved in two ways:

- Records are stored one after the other as they are inserted into the tables. This method is called pile file method. When a new record is inserted, it is placed at the end of the

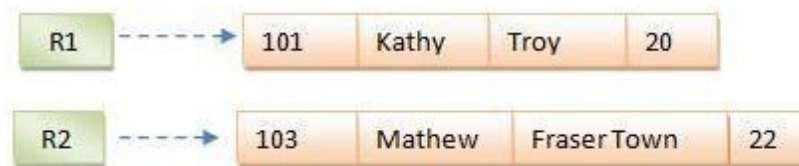
file. In the case of any modification or deletion of record, the record will be searched in the memory blocks. Once it is found, it will be marked for deleting and new block of record is entered.



Inserting a new record:



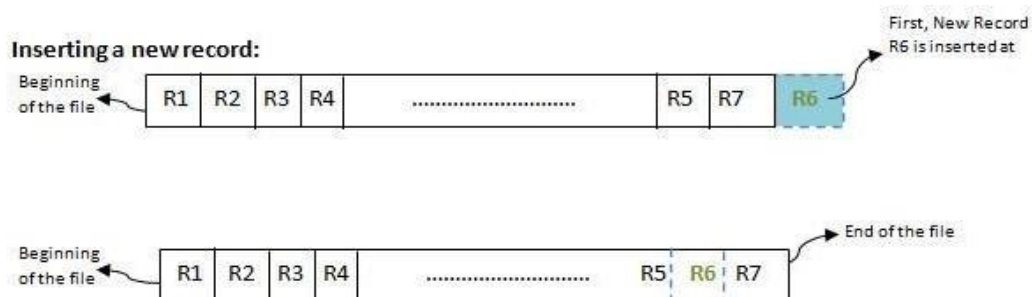
In the diagram above, R1, R2, R3 etc are the records. They contain all the attribute of a row. i.e.; when we say student record, it will have his id, name, address, course, DOB etc. Similarly R1, R2, R3 etc can be considered as one full set of attributes.



In the second method, records are sorted (either ascending or descending) each time they are inserted into the system. This method is called **sorted file method**. Sorting of records may be based on the primary key or on any other columns. Whenever a new record is inserted, it will be inserted at the end of the file and then it will sort – ascending or descending based on key value and placed at the correct position. In the case of update, it will update the record and then sort the file to place the updated record in the right place. Same is the case with delete.



Inserting a new record:



Advantages:

- Simple to understand.
- Easy to maintain and organize
- Loading a record requires only the record key.
- Relatively inexpensive I/O media and devices can be used.
- Easy to reconstruct the files.
- The proportion of file records to be processed is high.

Disadvantages:

- Entire file must be processed, to get specific information.
- Very low activity rate stored.
- Transactions must be stored and placed in sequence prior to processing.
- Data redundancy is high, as same data can be stored at different places with different keys.
- Impossible to handle random enquiries.

Conclusion:-

Sequential Files are implemented successfully for Student database system.

Staff Signature & Date

Experiment No.: 12

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

Aim: Study of Use index sequential file

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.

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 has multiple keys. These keys can be alphanumeric in which the records are ordered is called primary key.
- The data can be accessed 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

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 3 - Create a main() method.

Step 4 - Create an object (emp) of the above class inside the

main() method. Step 5 - Call the member functions

get_emp_details() and show_emp_details(). Step 6 - return 0 to

exit from 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. };

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  
= "GeeksforGeeks"  
emp[i].code = "12345"  
emp[i].designation =  
"Organisation"  
emp[i].exp = 10  
emp[i].age = 10
```

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

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.

Conclusion: Index Sequential Files are implemented successfully for Student database system.