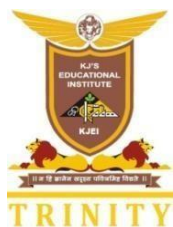# K J College of Engineering & Management Research
(Accredited by NAAC)
Sr. No. 25 & 27, Kondhwa - Saswad Road, Near Bopdev Ghat, Pune 411048.
Department of Computer Engineering

Savitribai Phule Pune University
Second Year of Computer Engineering (2019 Course)
210252: Data Structures and Algorithms

Teaching Scheme **Practical:** 04 Hours/Week	Credit:02	Examination Scheme and Marks Term work: 25 Marks **Practical:** 25 Marks

Companion Course: Data Structures and Algorithms (210252)

Subject Incharge	Prof. S.A.Nirve
Academic Year	2022-2023
Semester	II

K J College of Engineering & Management Research
(Accredited by NAAC)
Sr. No. 25 & 27, Kondhwa - Saswad Road, Near Bopdev Ghat, Pune 411048.
Department of Computer Engineering

210256: Data Structures and Algorithms Laboratory

Teaching Scheme: Practical: 04 Hours/Week

Credit: 02

Examination Scheme and Marks:

Term work: 25 Marks

Practical: 25 Marks

Course Objectives:

- To understand practical implementation and usage of non linear data structures for solving problems of different domain.
- To strengthen the ability to identify and apply the suitable data structure for the given real world problems.
- To analyze advanced data structures including hash table, dictionary, trees, graphs, sorting algorithms and file organization.

Course Outcomes:

On completion of the course, learners should be able to –

CO1: Understand the ADT/libraries, hash tables and dictionary to design algorithms for a specific problem.

CO2: Choose most appropriate data structures and apply algorithms for graphical solutions of the problems.

CO3: Apply and analyze nonlinear data structures to solve real world complex problems.

CO4: Apply and analyze algorithm design techniques for indexing, sorting, multi-way searching, file organization and compression.

CO5: Analyze the efficiency of most appropriate data structure for creating efficient solutions for engineering design situations.

7	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 accessprobability for each key?	33
8	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.	37
9	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.	47
10	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.	51
11	Implement the Heap/Shell sort algorithm implemented in Java demonstrating heap/shell data structure with modularity of programming language.	55
12	Write a Java program which will demonstrate a concept of Interfaces and packages: In this assignment design and use of customized interfaces and packages for a specificapplication are expected.	59
13	Write a Java program which will demonstrate a concept of cohesion and coupling of the various modules in the program.	65

Experiment No. 1

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.

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 graphicaly is represented most commonly as on *Picture 1*. The circles are the nodes and the edges are the links between them.

Trees are usualy used to store and represent data in some hierarhical 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 childs 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.
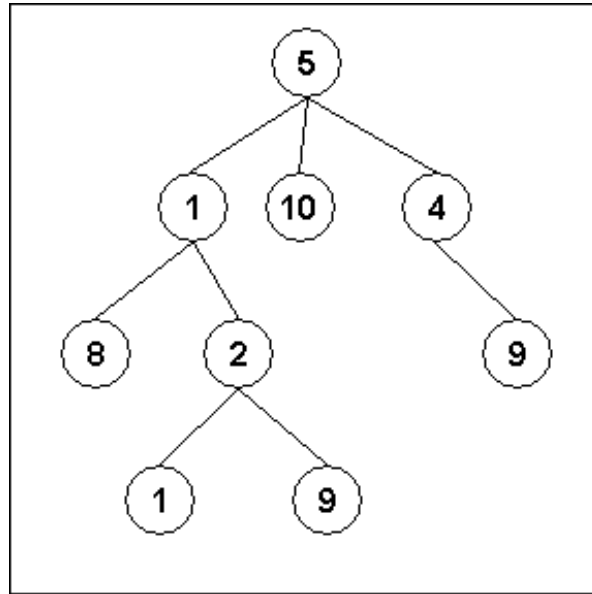
Fig1. An example of a tree

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 minumum effort

For this assignment we are considering the tree as follows.
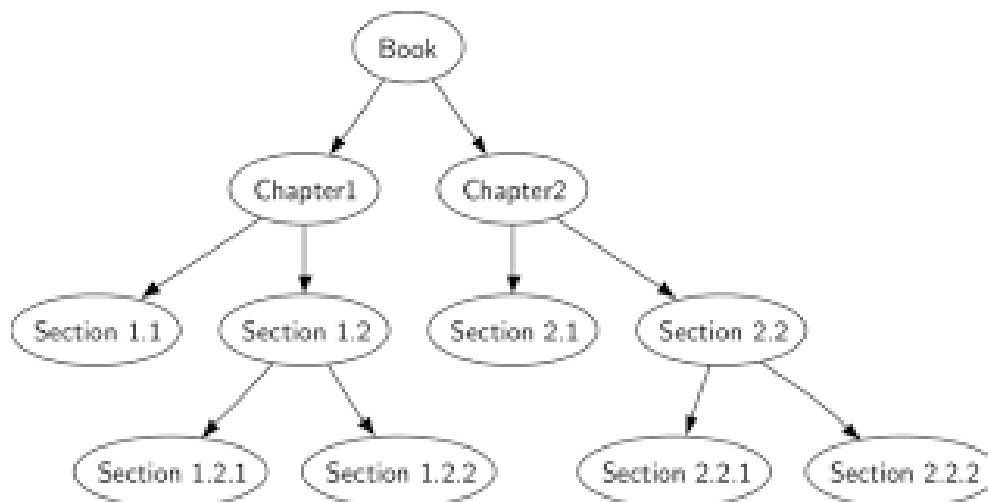
Software Required: g++ / gcc compiler- / 64 bit Fedora, eclipse IDE

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

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

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.

Questions asked in university exam.

1. What is class, object and data structure?

2. What is tree data structure?
3. Explain different types of tree?

Experiment No. 2

Aim: To illustrate the various Binary Tree functions.

Problem Statement : For given expression eg. a-b*c-d/e+f construct inorder sequence and traverse it using postorder traversal(non recursive).

Learning Objectives:

To understand concept of Tree & Binary Tree.

To analyze the working of various Tree operations.

Learning Outcome: Students will be able to use various set of operations on Binary   search.

Theory:

Tree

Tree represents the nodes connected by edges also a class of graphs that is acyclic is termed as trees. Let us now discuss an important class of graphs called trees and its associated terminology. Trees are useful in describing any structure that involves hierarchy. Familiar examples of such structures are family trees, the hierarchy of positions in an organization, and so on.

Binary Tree

A binary tree is made of nodes, where each node contains a "left" reference, a "right" reference, and a data element. The topmost node in the tree is called the root.

Every node (excluding a root) in a tree is connected by a directed edge from exactly one other node. This node is called a parent. On the other hand, each node can be connected to arbitrary number of nodes, called children. Nodes with no children are called leaves, or external nodes. Nodes which are not leaves are called internal nodes. Nodes with the same parent are called siblings.
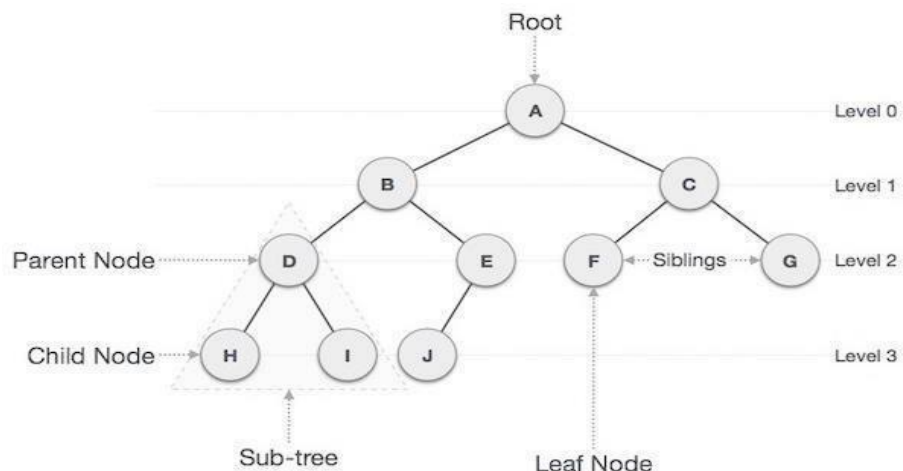
Insert Operation

The very first insertion creates the tree. Afterwards, whenever an element is to be inserted, first locate its proper location. Start searching from the root node, then if the data is less than the key value, search for the empty location in the left subtree and insert the data. Otherwise, search for the empty location in the right subtree and insert the data.

Traversals

A traversal is a process that visits all the nodes in the tree. Since a tree is a nonlinear data structure, there is no unique traversal. We will consider several traversal algorithms with we group in the following two kinds

- depth-first traversal
- breadth-first traversal

There are three different types of depth-first traversals, :

- PreOrder traversal - visit the parent first and then left and right children;
- InOrder traversal - visit the left child, then the parent and the right child;
- PostOrder traversal - visit left child, then the right child and then the parent;

There is only one kind of breadth-first traversal--the level order traversal. This traversal visits nodes by levels from top to bottom and from left to right. As an example consider the following tree and its four traversals:
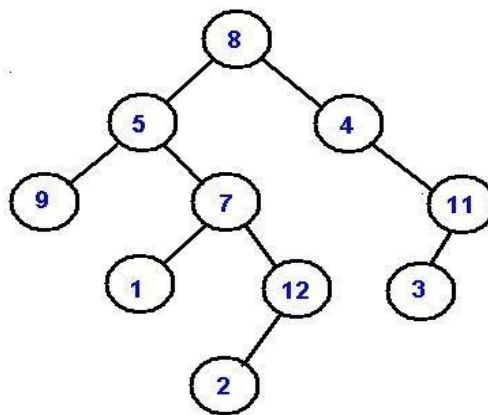
PreOrder - 8, 5, 9, 7, 1, 12, 2, 4, 11, 3
InOrder - 9, 5, 1, 7, 2, 12, 8, 4, 3, 11
PostOrder - 9, 1, 2, 12, 7, 5, 3, 11, 4, 8
LevelOrder - 8, 5, 4, 9, 7, 11, 1, 12, 3, 2

Algorithm:

•

Algorithm to insert a node :

Step 1 - Search for the node whose child node is to be inserted. This is a node at some level i, and a node is to be inserted at the level i +1 as either its left child or right child. This is the node after which the insertion is to be made.

Step 2 - Link a new node to the node that becomes its parent node, that is, either the Lchild or the Rchild.

Algorithm to traverse a tree :
• **Inorder traversal**

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
Step 3 − Recursively traverse right subtree.

• **Preorder**

Until all nodes are traversed
− Step 1 − Visit root node.

Step 2 − Recursively traverse left subtree.
Step 3 − Recursively traverse right subtree.

• **Postorder**

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.

Algorithm to copy one tree into another tree :

Step 1 – if (Root == Null)
Then return Null

Step 2 −Tmp = new TreeNode

Step 3 – Tmp->Lchild = TreeCopy(Root->Lchild); Step 4 – Tmp->Rchild = TreeCopy(Root->Rchild); Step 5 – Tmp-Data =
       Then return
•**S**<u>oftware required:</u> g++ / gcc compiler- / 64 bit fedora.

Outcome

Learn object oriented programming features.
Understand & implement different operations on tree & binary tree.

Conclusion : Thus we have studied the implementation of various Binary tree operations

Experiment No. 3

Aim: To illustrate the concept of graph.

Problem Statement :

Write a function to get the number of vertices in an undirected graph and its edges. You may assume that no edge is input twice.

iii.   Use adjacency list representation of the graph and find runtime of the function

iv. Use adjacency matrix representation of the graph and find runtime of the function

Learning Objectives:

To understand directed and undirected graph.

To implement program to represent graph using adjacency matrix and list.

Learning Outcome:

Student 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.
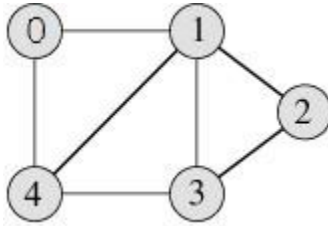
Following two are the most commonly used representations of graph.

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 x 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:

Adjacency Matrix Representation

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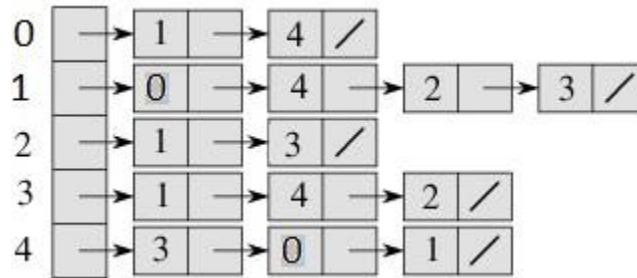
Pros: Saves space O(|V|+|E|) . In the worst case, there can be C(V, 2) number of edges in a graph thus consuming O(V^2) space. Adding a vertex is easier.

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

Conclusion:

Student implemented program for graph presentation in adjacency matrix and list.

Questions

1. An undirected graph having n edges, then find out no. Of vertices that graph have?

2. Define data structure to represent graph.

3. What are the methods to display graph.

4. Where you apply directed and undirected grap?

5. What is complexity of your graph to represent it in adjacency matrix and list?

Experiment No.4

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 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.
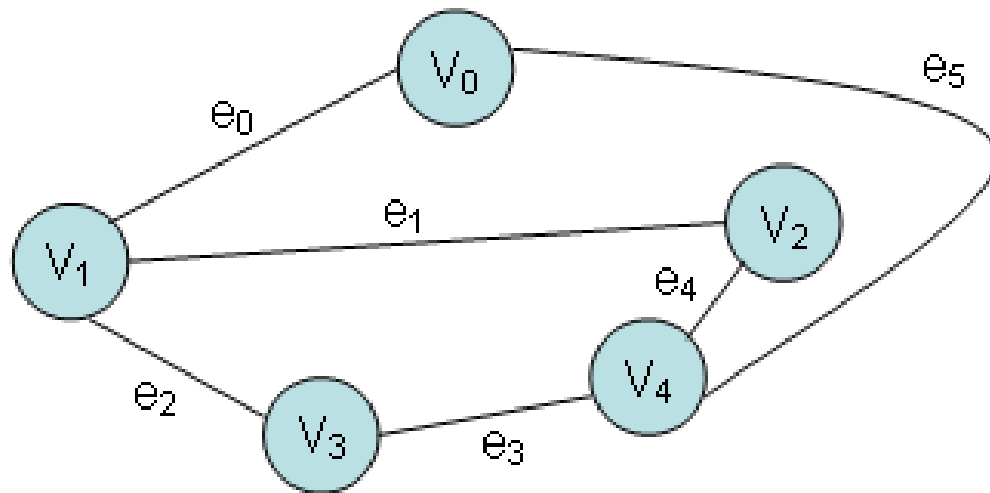
Graph Implementation:

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.
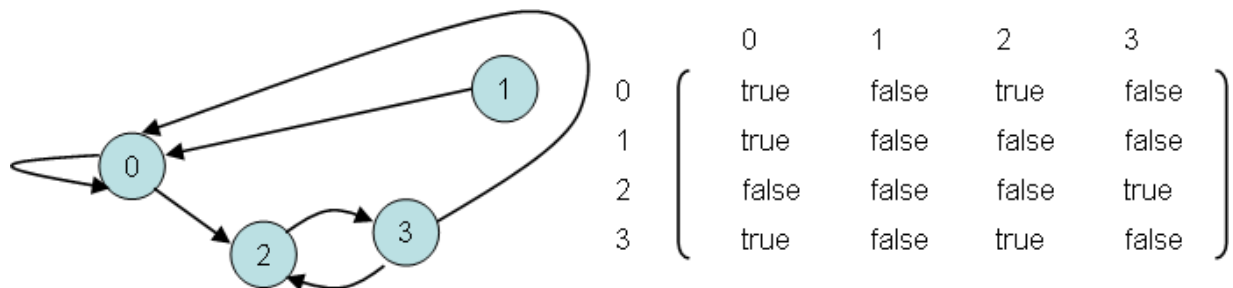
Representing Graphs with an Adjacency Matrix

	0	1	2	3
0	true	false	true	false
1	true	false	false	false
2	false	false	false	true
3	true	false	true	false

Fig: Graph and adjacency matrix

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.

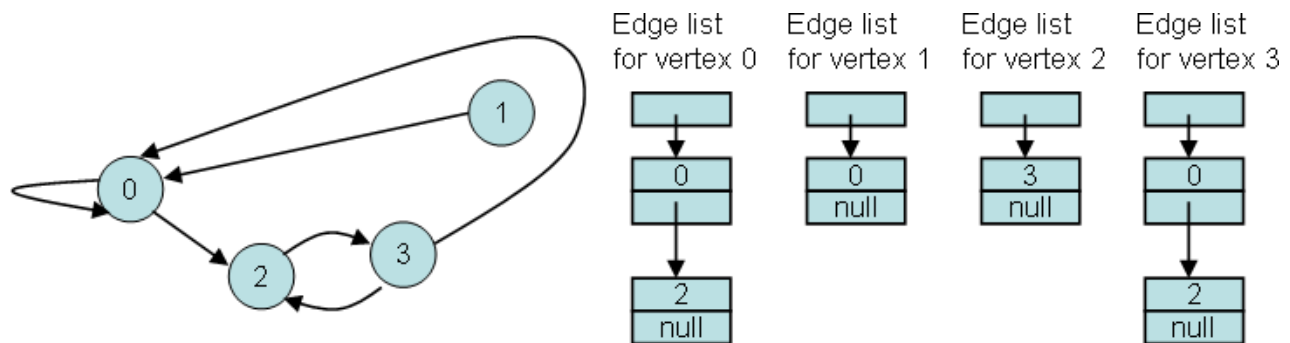## Representing Graphs with Edge Lists

Fig: Graph and adjacency list for each node

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?

Expected Output

Menu
1.Create Graph using adjacency List
2.Display Graph
Enter Choice1
No of Cities ?4
No of Flights?3

Please enter source city and destination city starting from A upto number of cities like A,B,C,D
Edge no -> 1
Source city->A
Destination city-> B
cost->23

Edge no -> 2
Source city->B
Destination city-> C
cost->35

Edge no -> 3
Source city->C
Destination city-> D
cost->56

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

Menu
1.Create Graph using adjacency List
2.Display Graph
Enter Choice2

A--> B & cost23
|
B--> C & cost35
|
C--> D & cost56
|
D

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

Experiment No. 5

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;
}
```

Delete Operation

Whenever an element is to be deleted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing to get the element ahead if an

element is not found at the computed hash code. When found, store a dummy item there to keep the performance of the hash table intact.

Example

```
struct DataItem* delete(struct DataItem* item) {
  int key = item->key;

  //get the hash
  int hashIndex = hashCode(key);

  //move in array until an empty
  while(hashArray[hashIndex] !=NULL) {

    if(hashArray[hashIndex]->key == key) {
      struct DataItem* temp = hashArray[hashIndex];
      //assign a dummy item at deleted position
      hashArray[hashIndex] = dummyItem;
      return temp;
    }

    //go to next cell
    ++hashIndex;

    //wrap around the table
    hashIndex %= SIZE;
  }

  return NULL;
}
```

Expected Output

Menu
1.Create Telephone book
2.Display
3.Look up
Enter Choice1
how many entries2
enter Namea
enter number1234567890
enter Named
enter number3216549876

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

Menu
1.Create Telephone book

2.Display
3.Look up
Enter Choice2

d 3216549876
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
a 1234567890
 0
 0
do u want to continue?(1 for continue)1

Menu
1.Create Telephone book
2.Display
3.Look up
Enter Choice3
enter Name to searchd
found at 0
no of comparision1
do u want to continue?(1 for continue)0*/

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

Experiment No. 6

<u>Title:</u>
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)

<u>Objectives:</u>

1. To understand Dictionary (ADT)

2. To understand concept of hashing

3. To understand concept & features like searching using hash function.

<u>Learning Objectives:</u>
- ✓ To understand Dictionary(ADT)

- ✓ To understand concept of hashing

- ✓ To understand concept & features like searching using hash function.

<u>Learning Outcome:</u>

- ✓ Define class for Dictionary using Object Oriented features.

- ✓ Analyze working of hash function.

<u>Theory:</u>

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.
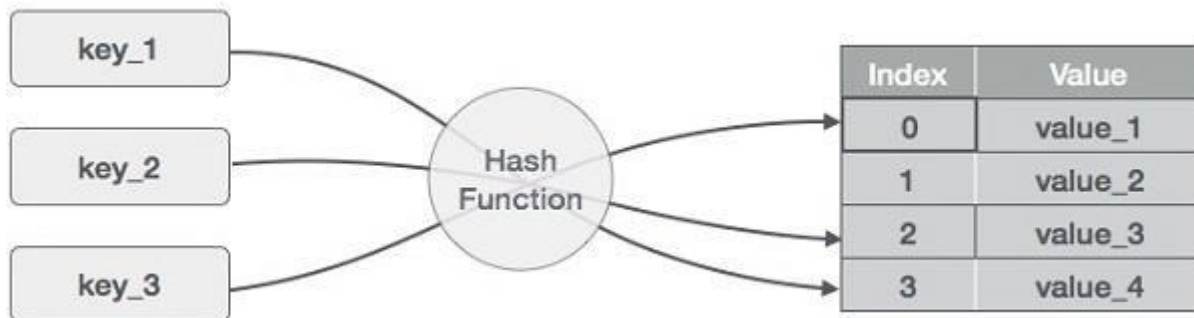
Basic Operations of hash table

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.

1. 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;
};
```

2. 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;
}
```

3. 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;
}
```

4. 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

}
```

5. 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.

Experiment No. 7

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:
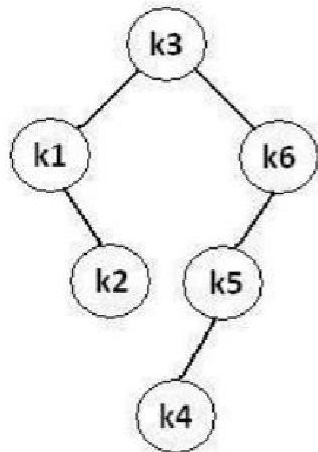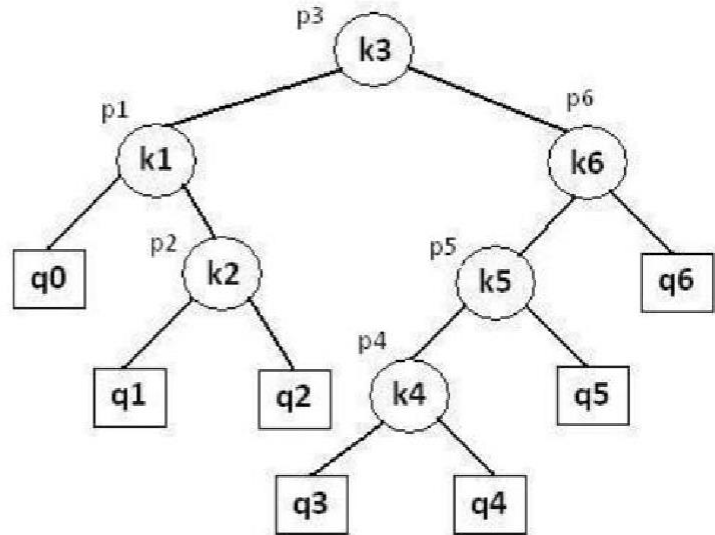
Binary search tree   Extended binary search tree

In the extended tree:

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

GENERALIZATION:

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

ALGORITHMS

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

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

begin

for i = 0 to n do

C (i, i) ←0

W (i, i) ←q(i)

for m = 0 to n

do

for i = 0 to (n – m) do

j ←i + m

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

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

tree cost

end

The following function builds an optimal binary sea

rch tree

FUNCTION CONSTRUCT(R, i, j)

begin

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

k ←R (i, j)

f i = k then

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

else

*N" ←CONSTRUCT(R, i, k)

*N" is the left child of node N

if k = (j – 1) then

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

else

*N"" ←CONSTRUCT(R, k + 1, j)

*N"" is the right child of node N

return N

end

COMPLEXITY ANALYSIS:

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

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

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

      2. key values

      3. Key Probability

Output: Create binary search tree having optimal searching cost.

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

OUTCOME

Upon completion Students will be able to:

ELO1: Learn object oriented Programming features.

ELO2: Understand & implement extended binary search tree.

ExperimentNo. 8

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 −
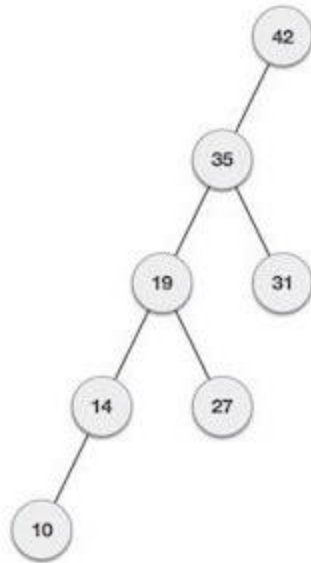
If input 'appears' non-increasing manner   If input 'appears' in non-decreasing manner
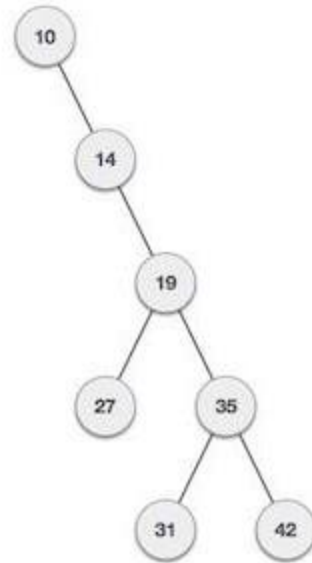
It is observed that BST's worst-case performance is closest to linear search algorithms, that is O(n). In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

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 −
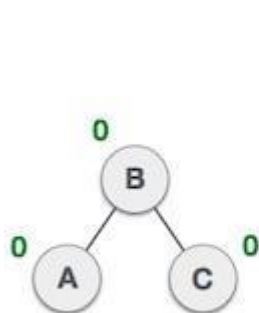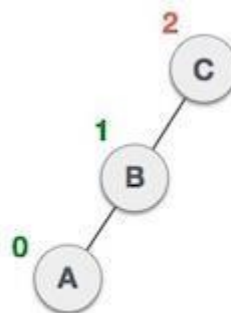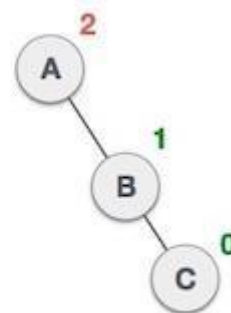
Balanced   Not balanced   Not balanced

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 −

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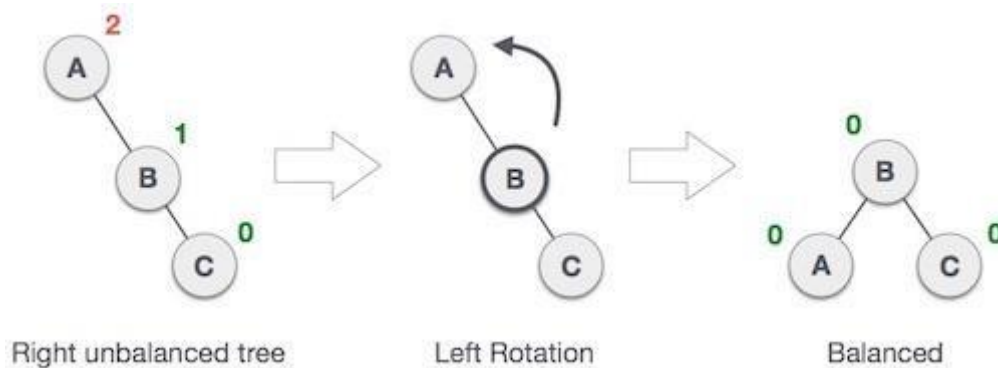
In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.

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.

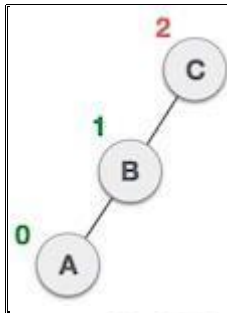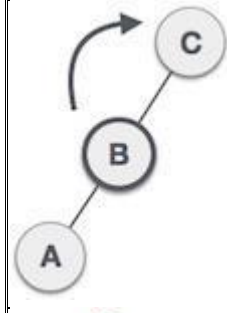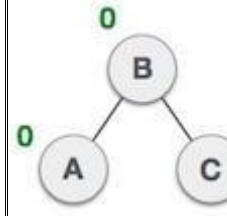Left unbalanced Tree   Right Rotation   Balanced Tree
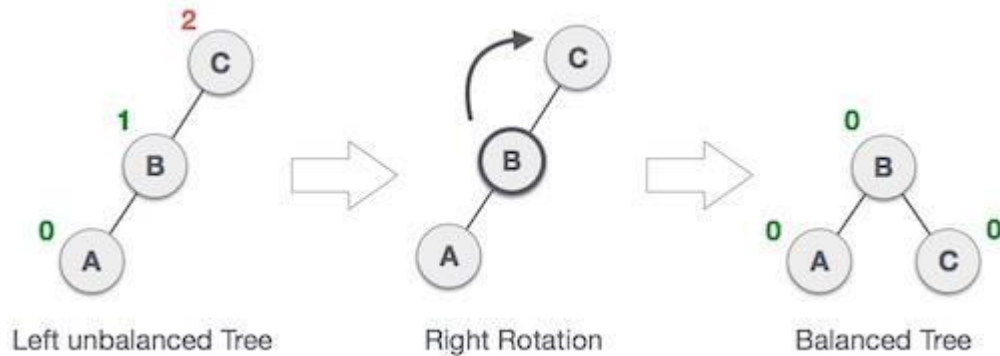
As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.
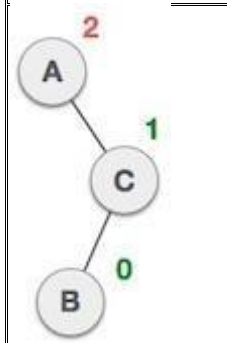
Left-Right Rotation

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.

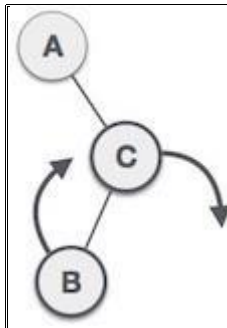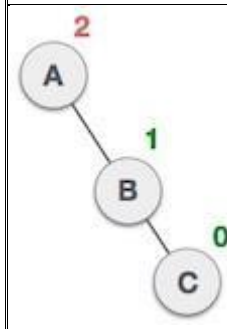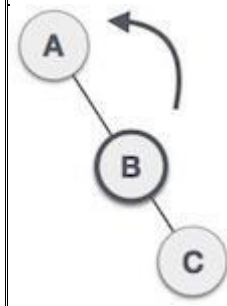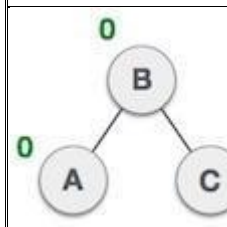| State	Action
	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.

	Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
	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.
	The tree is now balanced.

Right-Left Rotation

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

State	Action
	A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.

	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**.
	Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
	A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**.
	The tree is now balanced.

Algorithm AVL TREE:
Insert:-
1. If P is NULL, then
      I. P = new node
      II. P ->element = x
      III. P ->left = NULL
      IV. P ->right = NULL
      V. P ->height = 0
2. else if x>1 => x<P ->element
      a.) insert(x, P ->left)

b.) if height of P->left -height of P ->right =2
      1. insert(x, P ->left)
      2. if height(P ->left) -height(P ->right) =2
            if x<P ->left ->element
                  P =singlerotateleft(P)
            else
                  P =doublerotateleft(P)

3. else
      if x<P ->element
            a.) insert(x, P -> right)
            b.) if height (P -> right) -height (P ->left) =2
                if(x<P ->right) ->element
                    P =singlerotateright(P)
                else
                    P =doublerotateright(P)
4. else
Print already exits
5. int m, n, d.
6. m = AVL height (P->left)
7. n = AVL height (P->right)
8. d = max(m, n)
9. P->height = d+1
10. Stop

RotateWithLeftChild(AvlNode k2)

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

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

doubleWithLeftChild(AvlNode k3)
- ➢ k3.left = rotateWithRightChild(k3.left);
- ➢ return rotateWithLeftChild(k3);

doubleWithRightChild(AvlNode k1)

> - k1.right = rotateWithLeftChild(k1.right);
> - return rotateWithRightChild(k1);

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

Input: Dictionary word and its meaning.

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

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

OUTCOME

Upon completion Students will be able to:
      ELO1: Learn height balanced binary tree in data structure.

      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

CODE:

Function to get max element:

```
int getmax(int h1,int h2)
{
if(h1>h2)
  return h1;
return h2;
}avlnode * dict::rrotate(avlnode *y)
  {
  cout<<endl<<"rotating right - "<< y->key;
  avlnode *x=y->left;
  avlnode *t=x->right;
  x->right= y;
  y->left=t;
  y->ht=getmax(getht(y->left),getht(y->right))+1;
  x->ht=getmax(getht(x->left),getht(x->right))+1;
  return x;
  }
  avlnode * dict::lrotate(avlnode *x)
```

```
      {
      cout<<endl<<"rotating left - "<< x->key;

avlnode *y=x->right;
avlnode *t=y->left;
      x->right= t;
      y->left=x;
      y->ht=getmax(getht(y->left),getht(y->right))+1;
      x->ht=getmax(getht(x->left),getht(x->right))+1;
      return y;
      }
```

OUTPUT:
[@localhost ~]$ ./a.outMenu
1.Insert node
2.Inorder Display tree
Enter Choice1
enter key and meaning(single char)a
a
enter key and meaning(single char)b
b
height - 2
node bal - -1 current key is b
enter key and meaning(single char)cc
height - 2
node bal - -1 current key is c
height - 3
node bal - -2 current key is c
rotating left - a
enter key and meaning(single char)z
z

height - 2
node bal - -1 current key is z
height - 3
node bal - -1 current key is z
do u want to continue?(1 for continue)1

Menu
1.Insert node
2.Inorder Display tree
Enter Choice2
 a a b b c c z z
do u want to continue?(1 for continue)0

[@localhost ~]$ */

ExperimentNo. 9

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 a that subject. Use heap data structure. Analyze the algorithm.

Objectives:

1. To understand concept of heap

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

Learning 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:

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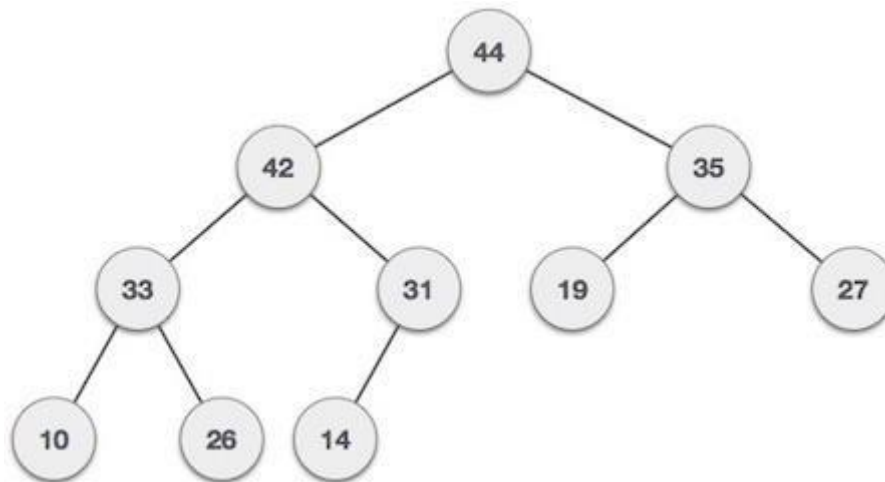
$$key(α) \geq key(β)$$

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

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

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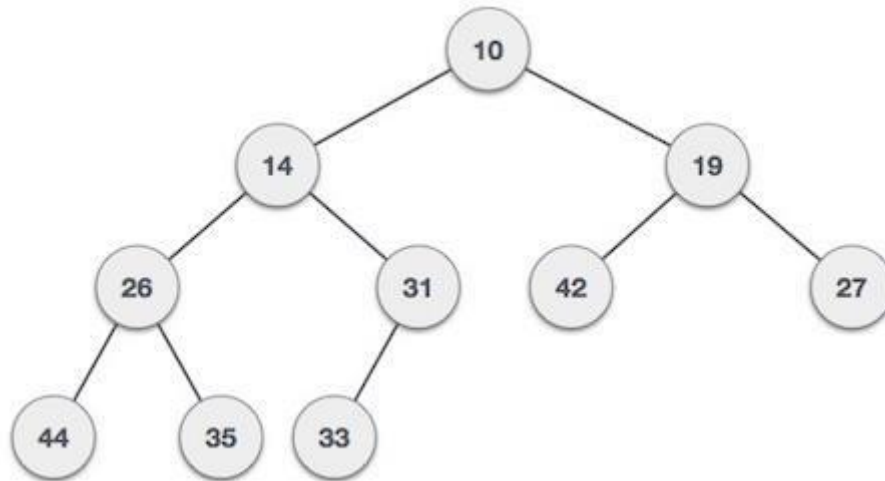
Max-Heap − Where the value of the root node is greater than or equal to either of its children.

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

Max Heap Construction Algorithm

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

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

Step 1 − Create a new node at the end of heap.
Step 2 − Assign new value to the node.
Step 3 − Compare the value of this child node with its parent.
Step 4 − If value of parent is less than child, then swap them.

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

Note − In Min Heap construction algorithm, we expect the value of the parent node to be less than that of the child node.
Let's understand Max Heap construction by an animated illustration. We consider the same input sample that we used earlier.

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

Max Heap Deletion Algorithm
Let us derive an algorithm to delete from max heap. Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value.
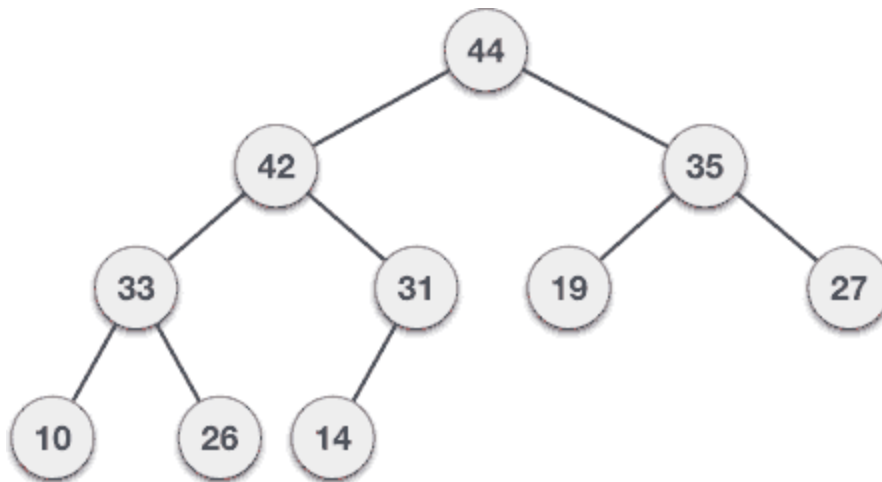**Step 1** − Remove root node.
Step 2 − Move the last element of last level to root.
Step 3 − Compare the value of this child node with its parent.
Step 4 − If value of parent is less than child, then swap them.
Step 5 − Repeat step 3 & 4 until Heap property holds.

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

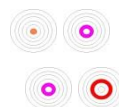**Input:** Marks obtained by student..

Output: Find min and max marks ontained.

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

OUTCOME
Upon completion Students will be able to:
    ELO1: Learn object oriented Programming features.

ELO2: Understand & implement Heap data structure.

Experiment No. 10

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.

Objectives:

1. To understand concept of file organization in data structure.

2. To understand concept & features of sequential file organization.

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.
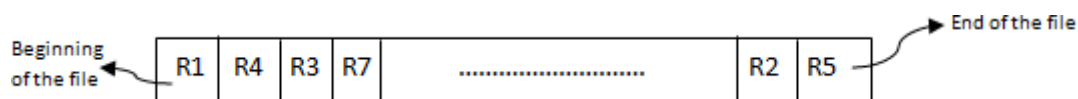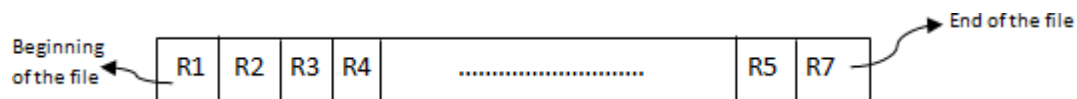

Inserting a new record:
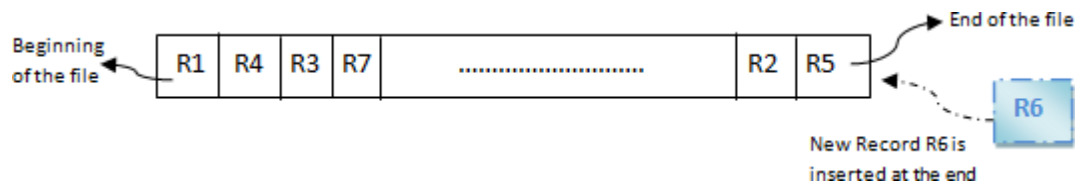
In the diagram above, R1, R2, R3 etc are the records. They contain all the attribute of a row. i.e.; when we say student record, it will have his id, name, address, course, DOB etc. Similarly R1, R2, R3 etc can be considered as one full set of attributes.

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:
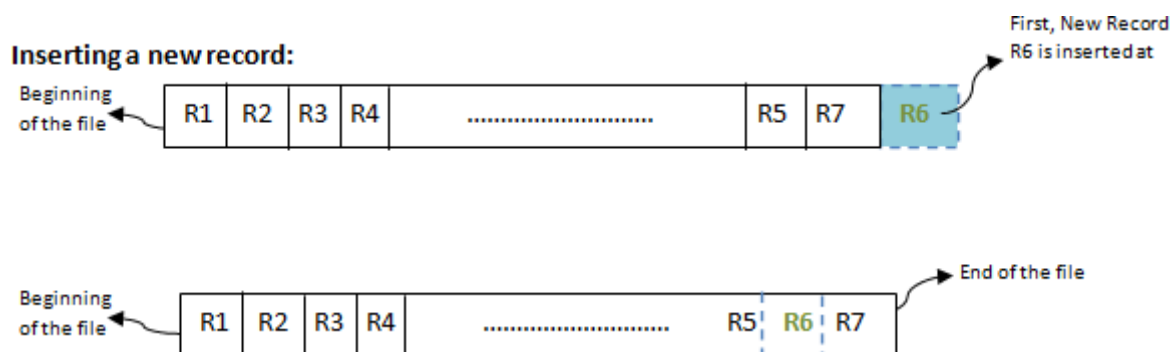
Advantages:

- 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.

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

Input: Details of student like roll no, name, address division etc.

Output: If record of student does not exist an appropriate message is displayed otherwise the student details are displayed.

Conclusion: This program gives us the knowledge sequential file organization..

OUTCOME

Upon completion Students will be able to:

ELO1: Learn File organization in data structure.

ELO2: Understand & implement sequential file and operation on it.

Experiment No. 11

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.
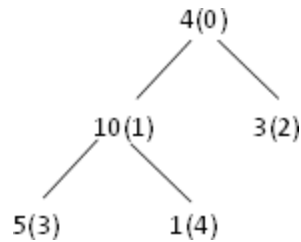
Lets understand with the help of an example:
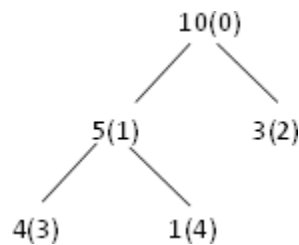
Input data: 4, 10, 3, 5, 1
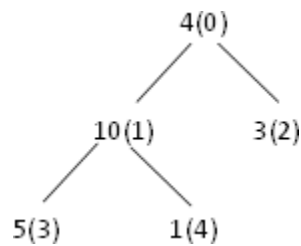
The numbers in bracket represent the indices in the array
representation of data.
Applying heapify procedure to index 1:

```
                                    4(0)
                                   /    \
                             10(1)       3(2)
                            /     \
                        5(3)       1(4)
```

Applying heapify procedure to index 0:

The heapify procedure calls itself recursively to build heap
in top down manner.

```
                                   10(0)
                                  /     \
                              5(1)       3(2)
                             /    \
                         4(3)      1(4)
```

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.

# Experiment No. 12

## Title:
Write a Java program which will demonstrate a concept of Interfaces and packages: In this assignment design and use of customized interfaces and packages for a specific application are expected.

## Objectives:

1. To understand concept of Interface in java language.

2. To understand concept of Package in java language.

## Learning Objectives:
✓ To understand concept of Interface in java language.

✓ To understand concept of Package in  java language.

## Learning Outcome:

- Define  and use interface in java.
- Define  and use package in java.

## Theory:

### 1. Interface:

An **interface in java** is a blueprint of a class. It has static constants and abstract methods only.

The interface in java is **a mechanism to achieve fully abstraction**. There can be only abstract methods in the java interface not method body. It is used to achieve fully abstraction and multiple inheritance in Java. It cannot be instantiated just like abstract class.

### Why use Java interface?

There are mainly three reasons to use interface. They are given below.

- It is used to achieve fully abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

In other words, Interface fields are public, static and final by default, and methods are public and abstract.

Printable.java

Printable.class

## Understanding relationship between classes and interfaces:

As shown in the figure given below, a class extends another class, an interface extends another interface but a class implements an interface.



## Simple example of Java interface:

In this example, Printable interface have only one method, its implementation is provided in the A class.

```
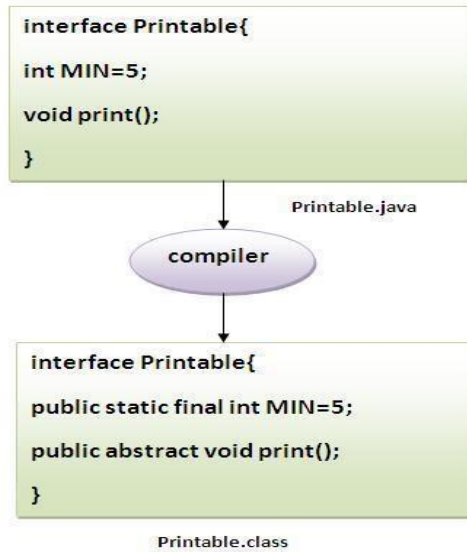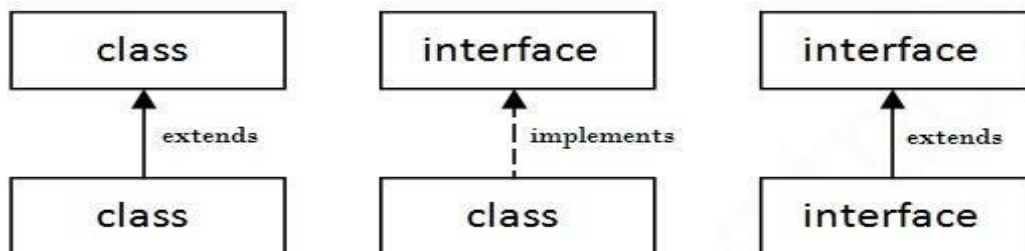interface printable{
void print();
}

class A6 implements printable{
public void print(){System.out.println("Hello");}
```

```
   public static void main(String args[]){
   A6 obj = new A6();
   obj.print();
    }

}
```

OUTPUT: Hello

Multiple inheritance in Java by interface:

If a class implements multiple interfaces, or an interface extends multiple interfaces i.e. known as multiple inheritance.

Fig.: Multiple Inheritance in java

2.Package:

A **java package** is a group of similar types of classes, interfaces and sub-packages. Package in java can be categorized in two form, built-in package and user-defined package. There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection.
3) Java package removes naming collision.

Simple example of java package:

The package keyword is used to create a package in java.

```
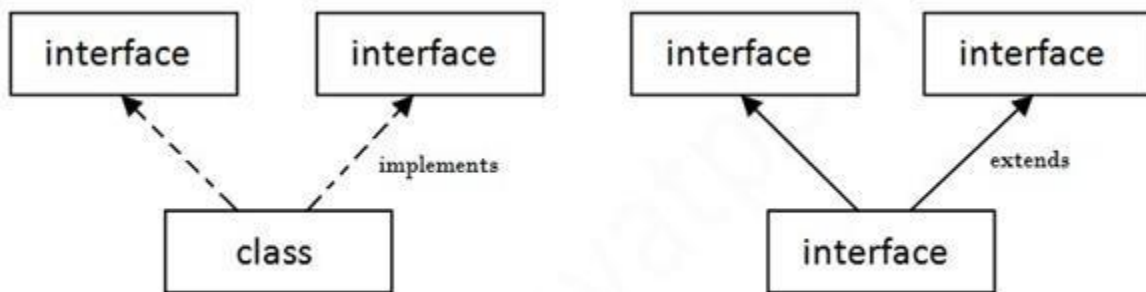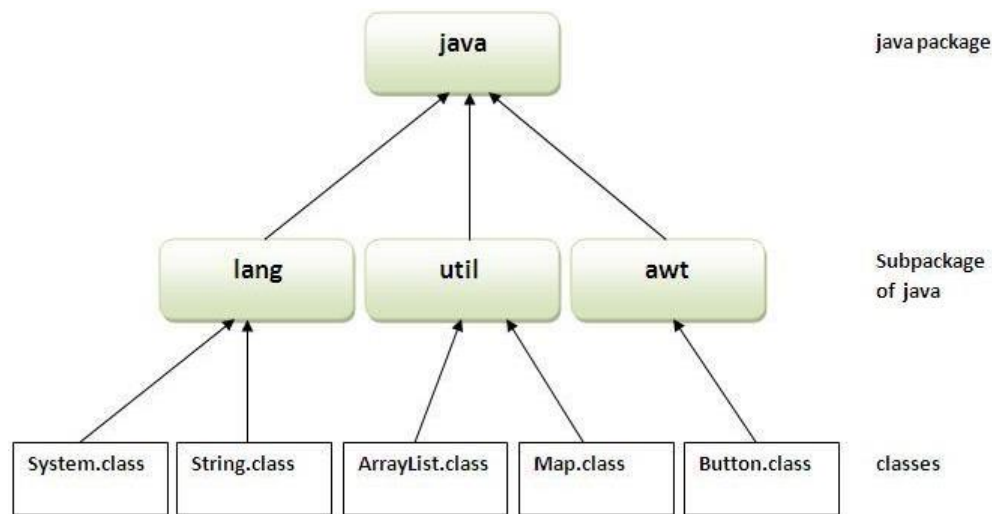//save as Simple.java
package mypack;
public class Simple
{
    public static void main(String args[])
    {
            System.out.println("Welcome to package");
    }
}
```

How to compile java package
If you are not using any IDE, you need to follow the syntax given below:
    1. javac -d directory javafilename
For example
    1. javac -d . Simple.java
The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To Run: java mypack.Simple
Output: Welcome to package

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

How to access package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

Program 1.
```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
```

Program 2.

```
//save by B.java
package mypack;
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```
Output: Hello

2) Using packagename.classname

If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

Program 1.

```
//save by A.java

package pack;
public class A{
 public void msg(){System.out.println("Hello");}
}
```

Program 2.

```
//save by B.java
package mypack;
import pack.A;

class B{
 public static void main(String args[]){
  A obj = new A();
  obj.msg();
 }
}
```
Output:Hello

3) Using fully qualified name

If you use fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

Program 1.
```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
```

Program 2.
```
//save by B.java
package mypack;
class B{
 public static void main(String args[]){
  pack.A obj = new pack.A();//using fully qualified name
  obj.msg();
 }
}
```
Output: Hello

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 a Java package and how is it used?
2. Name few classes of package java.io?
3. Which package stores all the standard java classes?
4. What is an interface in Java?
5. Can we create non static variables in an interface?
6. Can we declare interface members as private or protected?

Experiment No. 13

Title:

        Write a Java program which will demonstrate a concept of cohesion and coupling of the various modules in the program.

Objectives:

1. To understand concept of cohesion in java language.

2. To understand concept coupling in java language.

Learning Objectives:

- ✓ To understand concept of cohesion in java language.

- ✓ To understand concept coupling in java language.

Learning Outcome:

- Define classes to show the concept of cohesion and coupling.

- Analyze working of functions.

Theory:

COUPLING

An indication of the strength of interconnections between program units.

By definition coupling is the degree to which one class has knowledge of another or in other words one class has a dependency upon another. Tight coupling occurs when a dependent concrete class contains a pointer to another concrete class that provides the required behavior and should be avoided. The problem here is that any changes to one class could impact the other and the person making the changes may be completely unaware of this and thus unknowingly break the class. So how do we avoid this scenario? We *design by contract* by using an interface to specify an API for other classes to use as discussed in the OO Concepts - Interfaces lesson or by using encapsulation as discussed in the OO Concepts - Encapsulation. The following code is an example of tight coupling and should be avoided:

```
/*
 Tight coupling example
*/
class A {
   int i;
   B b = new B();
   i = b.value;     // No encapsulation of this variable in class B!
}
class B {
```

```
    public int value; // Should be private and be accessed through public getters and setters
}
```

COHESION

Measure of how well module fits together.

Cohesion is the degree to which components of a class belong together to fit a particular role. What we want to avoid is low cohesion where a class incorporates several different aspects. A class that tries to do many things comes with higher maintenance and lower reusability.

```
/*
 Low cohesion example
*/
class AllInStaff {
   void getStaffSalary();
   void getStaffDetails();
   void getStaffSalesReport();
}
/*
 High cohesion example
*/
class Accounts {
   void getStaffSalary();
   ...
}
class Personnel {
   void getStaffDetails();
   ...
}
class SalesReporting {
   void getStaffSalesReport();
   ...
}
```

Benefits of Higher Cohesion:

- Highly cohesive classes are much easier to maintain and less frequently changed.
- Such classes are more usable than others as they are designed with a well-focused purpose.

Software Required: java, 64 bit Fedora, eclipse IDE

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

OUTCOME

Upon completion Students will be able to:
    ELO1: Learn cohesion in java.

    ELO2: Learn coupling in java.

Questions asked in university exam.

1. What is cohesion and coupling in java
2. What is java virtual machine and what is its uses?
3. Can we override Thread's run() method?
4. What is the priority of main thread in java?? and why??