Pimpri-Chinchwad Educational Trusts

Pimpri-Chinchwad College of Engineering And Research,

Ravet, Pune

DEPARTMENT OF COMPUTER ENGINEERING

LAB MANUAL

DATA STRUCTURES AND ALGORITHMS LABORATORY

Course Code: 210256

Vision – Mission of the Institute

Vision

To be a Premier institute of technical education & research to serve the need of society and all the stakeholders.

Mission

To establish state-of-the-art facilities to create an environment resulting in individuals who are technically sound having professionalism, research and innovative aptitude with high moral and ethical values.

Vision – Mission of the Computer Department

Vision

To strive for excellence in the field of Computer Engineering and Research through Creative Problem Solving related to societal needs

Mission:

1. Establish strong fundamentals, domain knowledge and skills among the students with analytical thinking, conceptual knowledge, social awareness and expertise in the latest tools & technologies to serve industrial demands
2. Establish leadership skills, team spirit and high ethical values among the students to serve industrial demands and societal needs
3. Guide students towards Research and Development, and a willingness to learn by connecting themselves to the global society

Program Educational Objectives (PEO)

1. To prepare globally competent graduates having strong fundamentals, domain knowledge, upd with modern technology to provide the effective solutions for engineering problems.

2. To prepare the graduates to work as a committed professional with strong professional ethics values, sense of responsibilities, understanding of legal, safety, health, societal, cultural environmental issues.

3. To prepare committed and motivated graduates with research attitude, lifelong learı investigative approach, and multidisciplinary thinking.

4. To prepare the graduates with strong managerial and communication skills to work effectivel individual as well as in teams.

Program Specific Outcomes (PSO)

A graduate of the Computer Engineering Program will demonstrate-

PSO1: Problem-Solving Skills- The ability to apply standard practices and strategies in software project development using open-ended programming environments to deliver a quality project.

PSO2: Professional Skills-The ability to understand, analyze and develop computer programs in the areas related to algorithms, software testing, application software, web design, data analytics, IOT and networking for efficient design of computer-based systems.

PSO3: Successful Career and Entrepreneurship- The ability to employ modern computer languages, environments, and platforms in creating innovative career paths to be an entrepreneur, and a zest for higher studies, and to generate IPR & Deliver a quality project.

Companion Course: Data Structures and Algorithms (210252), Principles of Programming Language (210255)

Course Objectives:

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

Course Outcomes:
On completion of the course, student will be able to–

CO	Statements
C217.1	**Understand** the ADT/libraries, hash tables and dictionary to **implement** algorithms for aspecific problem.
C217.2	**Choose** appropriate non-linear data structures and **apply** algorithms for graphical solutions of the problems.
C217.3	**Apply** and **analyze** nonlinear data structures to solve real world complex problems.
C217.4	**Apply** and **analyze** algorithm design techniques for indexing, sorting, multi-way searching, file organization and compression.
C217.5	**Analyze** the efficiency of most appropriate data structure for creating efficient solutions for engineering design situations.

Index

		representation of the graph. Check whether the graph is connected or not. Justify the storage representation used		
	7	You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. Solve the problem by suggesting appropriate data structures.	**C217.2**	**35**
Group D	**8**	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?	**C217.3**	**43**
	9	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	**C217.3**	**47**
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	**C217.3**	**52**
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 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	**C217.4**	**58**
	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	**C217.4**	61

		system displays the employee details. Use index sequential file to maintain the data.		

References:

1. Steven S S. Skiena, "The Algorithm Design Manual", Springer, 2nd ed. 2008 Edition, ISBN-13: 978-1849967204, ISBN-10: 1849967202.
2. Allen Downey, Jeffery Elkner, Chris Meyers, "How to think like a Computer Scientist: Learning with Python", Dreamtech Press, ISBN: 9789351198147.
3. M. Weiss, "Data Structures and Algorithm Analysis in C++", 2nd edition, Pearson Education, 2002, ISBN-81-7808-670-0.
4. Brassard and Bratley, "Fundamentals of Algorithmic", Prentice Hall India/Pearson Education, ISBN 13-9788120311312.

Rubrics for Continuous Assessment in the laboratory

Parameters	Marks	Points				
		2 Marks	1 Marks	0 Marks		
Timely completion of laboratory journal	02	Students Completed laboratory journal within time.	Students completed laboratory journal after a week.	Students completed laboratory journal at the end of Semester.		
		4 Marks	3 Marks	2 Marks	1 Marks	0 Marks
Understanding of the assignment	04	Students performed the experiment/ executed the given program, extend the program by adding new features and answered all the questions.	Students performed the experiment/ executed the given program and answered all the questions.	Students performed the experiment / executed the given program with the help of peers and answered only a few questions.	Students performed the experiment/ executed the given program with the help of peers and did not answer any questions.	Students did not perform the experiment and did not answer any questions.
		4 Marks	3 Marks	2 Marks	1 Marks	0 Marks
Presentation / Clarity of journal writing	04	Written in an extraordinary style and voice, Well organized, Very informative.	Written in an interesting style and voice, Somewhat informative and organized.	Written in little style, Gives some new information but poorly organized.	Written in bad style, Gives some new information but poorly organized.	Written in bad style, no new information and very poorly organized.

Abbreviation:
T- Timely completion of laboratory journal
U- Understanding of the assignment
P- Presentation / Clarity of journal writing
S- Sum

CERTIFICATE

This is to certify that Mr./Miss.____

Roll No.:_______Exam. Seat No.:_______________of SE/TE/BE Computer has carried out above practical /term work within PCCOER as prescribed by Savitribai Phule Pune University, Pune during the academic year 20 -20 . His/Her performance is satisfactory and attendance is_______%.

Date:	**Faculty I/C**	**HOD**	**Principal**

Assignment No. 01

Problem Statement:

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

Learning Objectives:

To learn concept of hashing.

To implement program for above problem using hash function and handle collisions.

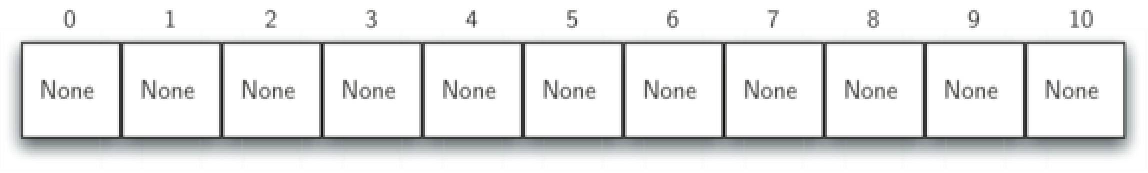
Learning Outcome: Students implement the program for above problem.

Theory:

By knowing that a list was ordered, we could search in logarithmic time using a binary search. In this, we will attempt to go one step further by building a data structure that can be searched in O(1) time. This concept is referred to as hashing.

A hash table is a collection of items which are stored in such a way as to make it easy to find them later. Each position of the hash table, often called a slot, can hold an item and is named by an integer value starting at 0. For example, we will have a slot named 0, a slot named 1, a slot named 2, and so on. Initially, the hash table contains no items so every slot is empty. We can implement a hash table by using a list with each element initialized to the special Python value None.

Figure shows a hash table of size m=11. In other words, there are m slots in the table, named 0 through 10.



The mapping between an item and the slot where that item belongs in the hash table is called the hash function. The hash function will take any item in the collection and return an integer in the range of slot names, between 0 and m-1. Assume that we have the set of integer items 54, 26, 93, 17, 77, and 31. Our first hash function, sometimes referred to as the "remainder method," simply takes an item and divides it by the table size, returning the remainder as its hash value (h(item)=item%11). Table gives all of the hash values for our example items. Note that this remainder method (modulo arithmetic) will typically be present in some form in all hash functions, since the result must be in the range of slot names.

Item	Hash Value
54	10
26	4
93	5
17	6
77	0
31	9

Once the hash values have been computed, we can insert each item into the hash table at the designated position.

Hash Functions

Given a collection of items, a hash function that maps each item into a unique slot is referred to as a perfect hash function. If we know the items and the collection will never change, then it is possible to construct a perfect hash function (refer to the exercises for more about perfect hash functions). Unfortunately, given an arbitrary collection of items, there is no systematic way to construct a perfect hash function. Luckily, we do not need the hash function to be perfect to still gain performance efficiency.

One way to always have a perfect hash function is to increase the size of the hash table so that each possible value in the item range can be accommodated. This guarantees that each item will have a unique slot. Although this is practical for small numbers of items, it is not feasible when the number of possible items is large. For example, if the items were nine-digit Social Security numbers, this method would require almost one billion slots. If we only want to store data for a class of 25 students, we will be wasting an enormous amount of memory.

Our goal is to create a hash function that minimizes the number of collisions, is easy to compute, and evenly distributes the items in the hash table. There are a number of common ways to extend the simple remainder method. We will consider a few of them here.

The folding method for constructing hash functions begins by dividing the item into equal-size pieces (the last piece may not be of equal size). These pieces are then added together to give the resulting hash value. For example, if our item was the phone number 436-555-4601, we would take he digits and divide them into groups of 2 (43,65,55,46,01). After the addition, 43+65+55+46+01

, we get 210. If we assume our hash table has 11 slots, then we need to perform the extra step of dividing by 11 and keeping the remainder. In this case 210 % 11 is 1, so the phone number 436-555-4601 hashes to slot 1. Some folding methods go one step further and reverse every other piece before the addition. For the above example, we get 43+56+55+64+01=219 which gives 219 % 11=10.

Another numerical technique for constructing a hash function is called the mid-square method. We first square the item, and then extract some portion of the resulting digits. For example, if the item were 44, we would first compute 442=1,936. By extracting the middle two digits, 93, and performing the remainder step, we get 5 (93 % 11).

Collision Resolution

We now return to the problem of collisions. When two items hash to the same slot, we must have a systematic method for placing the second item in the hash table. This process is called collision resolution. As we stated earlier, if the hash function is perfect, collisions will never occur. However, since this is often not possible, collision resolution becomes a very important part of hashing.

One method for resolving collisions looks into the hash table and tries to find another open slot to hold the item that caused the collision. A simple way to do this is to start at the original hash value position and then move in a sequential manner through the slots until we encounter the first slot that is empty. Note that we may need to go back to the first slot (circularly) to cover the entire hash table. This collision resolution process is referred to as open addressing in that it tries to find the next open slot or address in the hash table. By systematically visiting each slot one at a time, we are performing an open addressing technique called linear probing.

Figure shows an extended set of integer items under the simple remainder method hash function (54,26,93,17,77,31,44,55,20). Table 4 above shows the hash values for the original items. When we attempt to place 44 into slot 0, a collision occurs. Under linear probing, we look sequentially, slot by slot, until we find an open position. In this case, we find slot 1.

Again, 55 should go in slot 0 but must be placed in slot 2 since it is the next open position. The final value of 20 hashes to slot 9. Since slot 9 is full, we begin to do linear probing. We visit slots 10, 0, 1, and 2, and finally find an empty slot at position 3.

Once we have built a hash table using open addressing and linear probing, it is essential that we utilize the same methods to search for items. Assume we want to look up the item 93. When we compute the hash value, we get 5. Looking in slot 5 reveals 93, and we can return True. What if we are looking for 20? Now the hash value is 9, and slot 9 is currently holding 31. We cannot simply return False since we know that there could have been collisions. We are now forced to do a sequential search, starting at position 10, looking until either we find the item 20 or we find an empty slot.

A disadvantage to linear probing is the tendency for clustering; items become clustered in the table. This means that if many collisions occur at the same hash value, a number of surrounding slots will be filled by the linear probing resolution. This will have an impact on other items that are being inserted, as we saw when we tried to add the item 20 above. A cluster of values hashing to 0 had to be skipped to finally find an open position.

One way to deal with clustering is to extend the linear probing technique so that instead of looking sequentially for the next open slot, we skip slots, thereby more evenly distributing the items that have caused collisions. This will potentially reduce the clustering that occurs. Figure shows the items when collision resolution is done with a "plus 3" probe. This means that once a collision occurs, we will look at every third slot until we find one that is empty.

The general name for this process of looking for another slot after a collision is rehashing. With simple linear probing, the rehash function is newhashvalue=rehash(oldhashvalue)

where rehash(pos)=(pos+1)% sizeoftable. The "plus 3" rehash can be defined as rehash(pos)=(pos+3)%sizeoftable. In general, rehash(pos)=(pos+skip)%sizeoftable. It is important to note that the size of the "skip" must be such that all the slots in the table will eventually be visited. Otherwise, part of the table will be unused. To ensure this, it is often suggested that the table size be As before, we will have a result for both a successful and an unsuccessful search. For a successful search using open addressing with linear probing, the average number of comparisons is approximately 1/2(1+ 1/ 1−λ) and an unsuccessful search gives 1 / 2(1+(1 / 1−λ)2) If we are using chaining, the average number of comparisons is 1+λ2 for the successful case, and simply λ comparisons if the search is unsuccessful. a prime number. This is the reason we have been using 11 in our examples.

A variation of the linear probing idea is called quadratic probing. Instead of using a constant "skip" value, we use a rehash function that increments the hash value by 1, 3, 5, 7, 9, and so on.

This means that if the first hash value is h, the successive values are h+1, h+4, h+9, h+16, and so on. In other words, quadratic probing uses a skip consisting of successive perfect squares.

An alternative method for handling the collision problem is to allow each slot to hold a reference to a collection (or chain) of items. Chaining allows many items to exist at the same location in the hash table. When collisions happen, the item is still placed in the proper slot of the hash table. As more and more items hash to the same location, the difficulty of searching for the item in the collection increases.

We stated earlier that in the best case hashing would provide a O(1), constant time search technique. However, due to collisions, the number of comparisons is typically not so simple. Even though a complete analysis of hashing is beyond the scope of this text, we can state some well-known results that approximate the number of comparisons necessary to search for an item. The most important piece of information we need to analyze the use of a hash table is the load factor, λ. conceptually, if λ is small, then there is a lower chance of collisions, meaning that items are more likely to be in the slots where they belong. If λ is large, meaning that the table is filling up, then there are more and more collisions. This means that collision resolution is more difficult, requiring more comparisons to find an empty slot. With chaining, increased collisions means an increased number of items on each chain.

As before, we will have a result for both a successful and an unsuccessful search. For a successful search using open addressing with linear probing, the average number of comparisons is approximately $1/2(1+ 1/ 1-\lambda)$ and an unsuccessful search gives $1 / 2(1+(1 / 1-\lambda)^2)$ If we are using chaining, the average number of comparisons is $1+\lambda/2$ for the successful case, and simply λ comparisons if the search is unsuccessful.

Conclusion:

Student implemented the program.

Questions

1. Which hash function is used in your program?
2. What is size of hash table used in your program?
3. Which algorithm is used to resolve collision in your program?
4. What is the space and time complexity of your program?
5. If you are applying key on text value then how would you use string in your hash function.

Assignment No. 02

Problem Statement:

To create ADT that implement the "set" concept.

a. Add (newElement) -Place a value into the set

b. Remove (element) Remove the value

c. Contains (element) Return true if element is in collection

d. Size () Return number of values in collection Iterator () Return an iterator used to loop over collection

e. Intersection of two sets

f. Union of two sets

g. Difference between two sets

h.Subset

Learning Objectives:

To learn concept of set operations.

To implement program for above problem using python.

Learning Outcome: Students implement the program for above problem.

Theory:

Python:

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.

Set Operations:

We have to perform here different set operations like Union, Intersections, Difference, Symmetric Difference.

Universal set U:

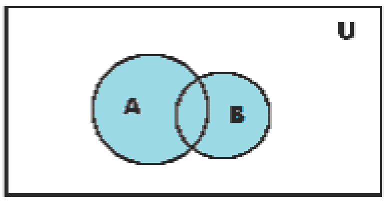
Often a discussion involves subsets of some particular set called the universe of discourse (or briefly universe), universal set or space. The elements of a space are often called the points of the space. We denote the universal set by U.

Example. The set of all even integers could be considered a subset of a universal set consisting of all the integers. Or they could be considered a subset of a universal set consisting of all the rational numbers. Or of all the real numbers.

Often the universal set may not be explicitly stated and it may be unclear as to just what it is. At other times it will be clear.

1. Union Operation: In set theory the union of collection of sets is the set of all distinct elements in the collection. The union of two sets A and B is the set consisting of all elements in A plus all elements in B and is denoted by A∪B or A + B.

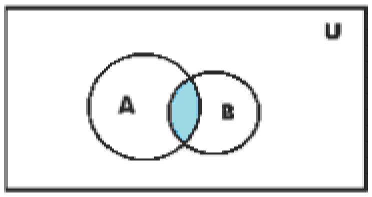
   Example. If A = {a, b, c, d} and B = {b, c, e, f, g} then A∪B = {a, b, c, d, e, f, g}.



A U B

2. Intersection operation: In set theory intersection is a operation where we collect common elements of different sets. The intersection of two sets A and B is the set consisting of all elements that occur in both A and B (i.e. all elements common to both) and is denoted by A∩B, A · B or AB.

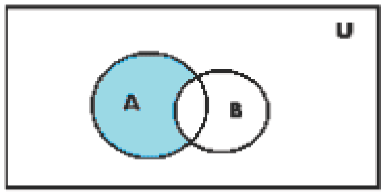
   Example. If A = {a, b, c, d} and B = {b, c, e, f, g} then
   A∩B = {b, c}.



A ∩ B

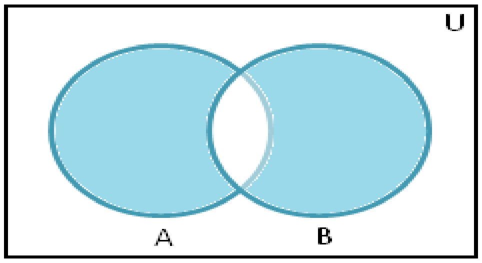
3. Difference Operation: It is a generalization of the idea of the compliment of a set and as a such is sometimes called the relative compliment of T with respect to S where T and s are two sets. The set consisting of all elements of a set A that do not belong to a set B is called the difference of A and B and denoted by A - B.

Example. If A = {a, b, c, d} and B = {b, c, e, f, g} then A - B = {a, d}.



A - B

4. Symmetric Difference: The symmetric difference between two sets S and t is the union of S-T and T-S. The symmetric difference using Venn diagram of two subsets A and B is a sub set of U, denoted by A ▵ B and is defined by A ▵ B = (A – B) ∪ (B – A)



A △ B

Software required: Open Source Python, Programming tool like Jupyter Notebook, Pycharm, Spyder.

Conclusion: Thus, we have studied use of set operations using python.

Assignment No. 03

Problem Statement:

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 learn concept of tree.

To implement program for above problem using tree.

Learning Outcome: Students implement the program for above problem.

Theory:

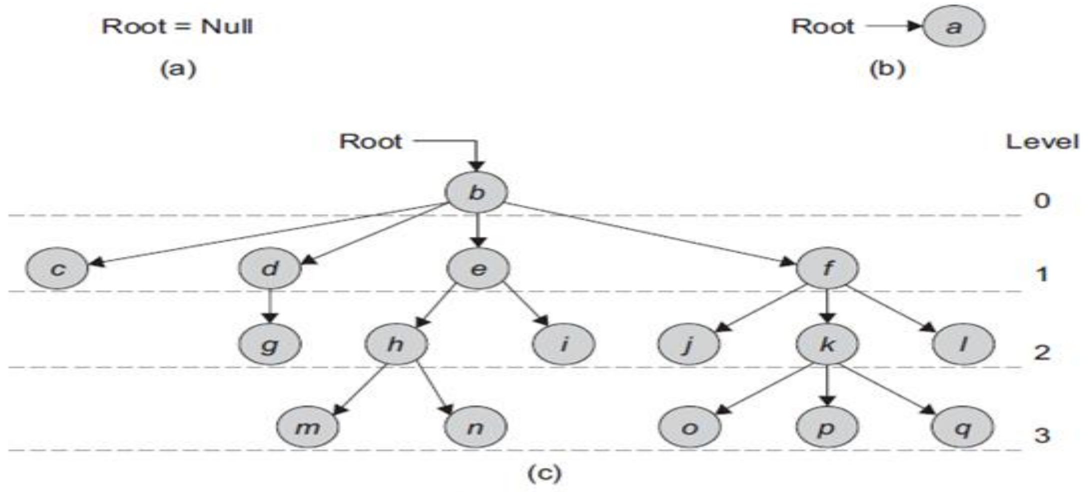
General Tree

A tree *T* is defined recursively as follows:

1. A set of zero items is a tree, called the *empty tree* (or null tree).

2. If *T*1, *T*2, ..., *Tn* are *n* trees for *n* > 0 and *R* is a *node*, then the set *T* containing *R* and the trees *T*1, *T*2, ..., *Tn* are a tree. Within *T*, *R* is called the *root* of *T*, and *T*1, *T*2, ..., *Tn* are called *subtrees*.

The tree in Fig. (a) is the empty tree since there are no nodes. The tree in Fig. (b) has only one node, the *root*. The tree in Fig. (c) has 16 nodes. The root node has four subtrees. The roots of these subtrees are called the *children* of the root. There are 16 nodes in the tree, so there are 15 non-empty subtrees. The nodes with no subtrees are called *terminal nodes* or more commonly, *leaves*. These are 10 leaves in the tree in Fig. (c).



Degree of a tree (a) Empty tree—degree undefined (b) Tree with a single node—degree 0 (c) Tree of height 3—degree 4

The degree of a node is the number of subtrees it has. Thus, the degree of the nodes in Fig. (c) ranges from zero to four. By definition, the degree of each leaf node is zero. The *degree of a tree* is the maximum degree of a node in the tree. As the tree in Fig. (a) has no nodes, there is no maximum degree of a node, and the degree of the tree is not defined. The tree in Fig. (b) has degree zero, and the tree in Fig. (c) has degree four. Since family relationships can be modelled as trees, we often call the root of a tree (or subtree) the *parent*, and the roots of the subtrees the *children*. Consequently, the children of the same node are called *siblings*.

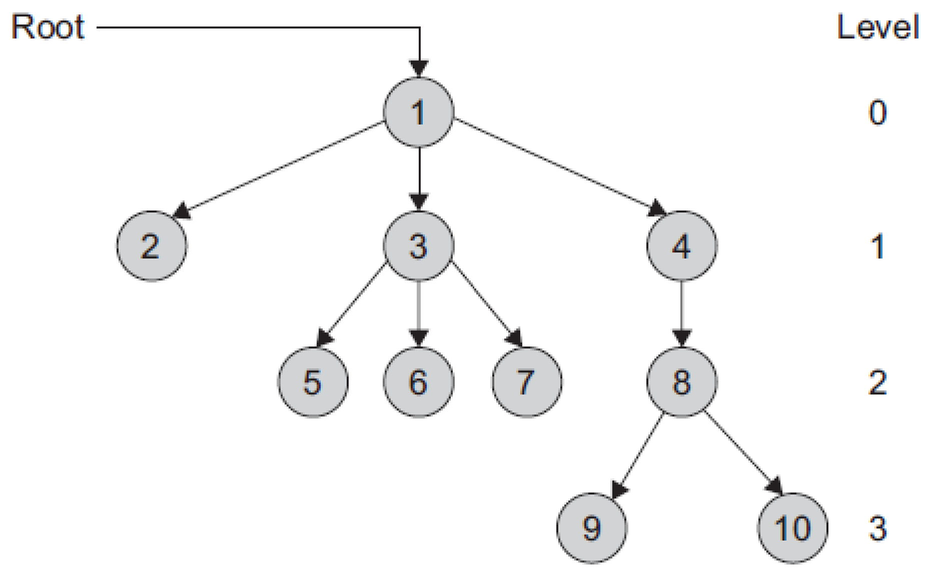
For trees with more than one node, the following statements are true:

1. The preorder list contains the root followed by the preorder list of nodes of the subtrees of the root from left to right.

2. The inorder list contains the inorder list of the leftmost subtree, the root, and the inorder list of each of the other subtrees from left to right.

3. The postorder list contains the postorder list of subtrees of the root from left to right followed by the root.

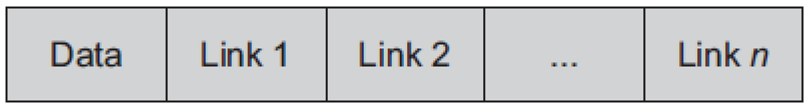
Figure shows a tree whose nodes are labelled with numbers rather than letters.



Sample tree

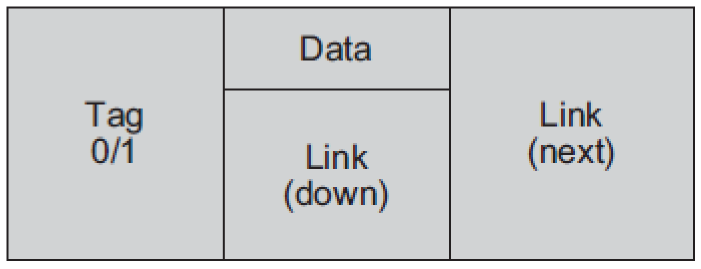
Representation of a General Tree

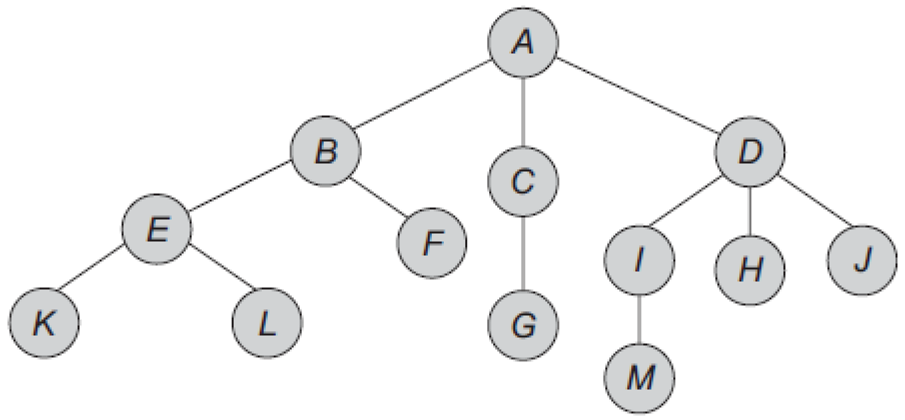
We can use either a sequential organization or a linked organization for representing a tree. If we wish to use a generalized linked list, then a node must have a varying number of fields depending upon the number of branches. However, it is simpler to use algorithms for the data where the node size is fixed.



For a fixed size node, we can use a node with data and pointer fields as in a generalized

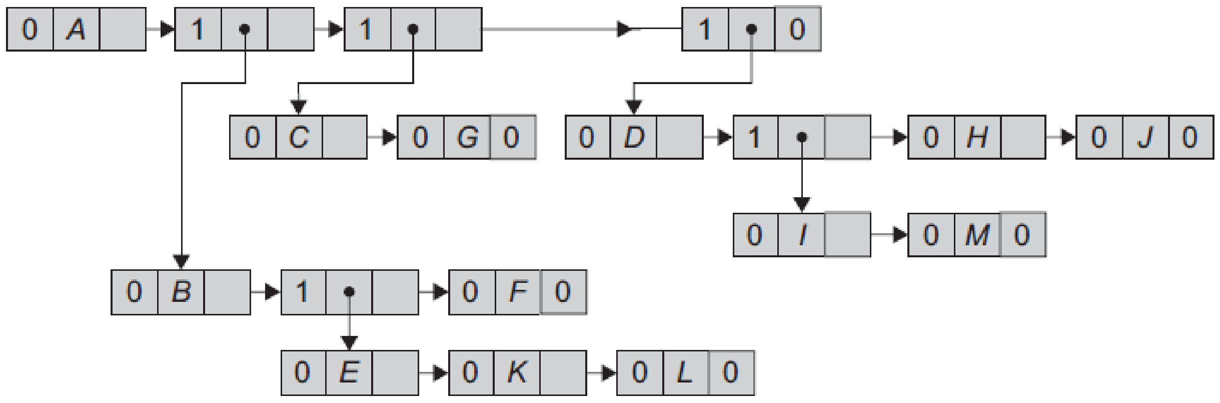
linked list.





Sample tree

The list representation of this tree is shown below



List representation

Conclusion:

Student implemented the program.

Questions

6. What do you mean by depth, height, sibling of a tree

7. How the representation of general tree is takes place?

8. How to insert new node in general tree?

Assignment No. 04

Problem Statement:

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

i. Insert new node

ii. Find number of nodes in longest path 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

Objective:

To understand the basic concept of Non-Linear Data Structure “TREE” and its basic operation in Data structure.

Outcome:

To implement the basic concept of Binary Search Tree to store a number in it. Also perform basic Operation Insert, Delete and search, Traverse in tree in Data structure.

Software & Hardware Requirements:

1. 64-bit Open source Linux or its derivative

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

Theory:

Tree represents the nodes connected by edges. Binary Tree is a special data structure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.

Binary Search Tree Representation: Binary Search tree exhibits a special behaviour. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned

properties −

- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than or equal to its parent node's key.

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

left_subtree (keys) ≤ node (key) ≤ right_subtree (keys)

Tree Node

Following Structure is used for Node creation

```
Struct node
{
        Int data ;
        Struct node *leftChild;
        Struct node *rightChild;
};
```

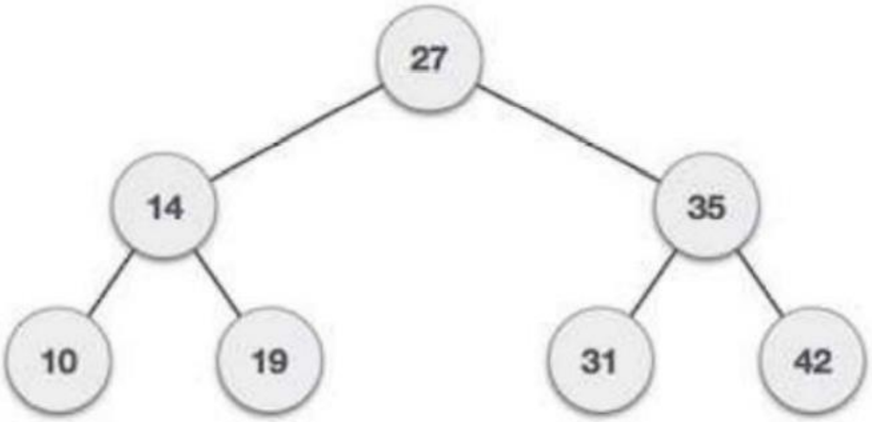


Fig: Binary Search Tree

BST Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following −

- Insert− Inserts an element in a tree/create a tree.
- Search− Searches an element in a tree.
- Traversal− A traversal is a systematic way to visit all nodes of T -Inorder, Preprder, Postorder,

  a. pre-order: Root, Left, Right

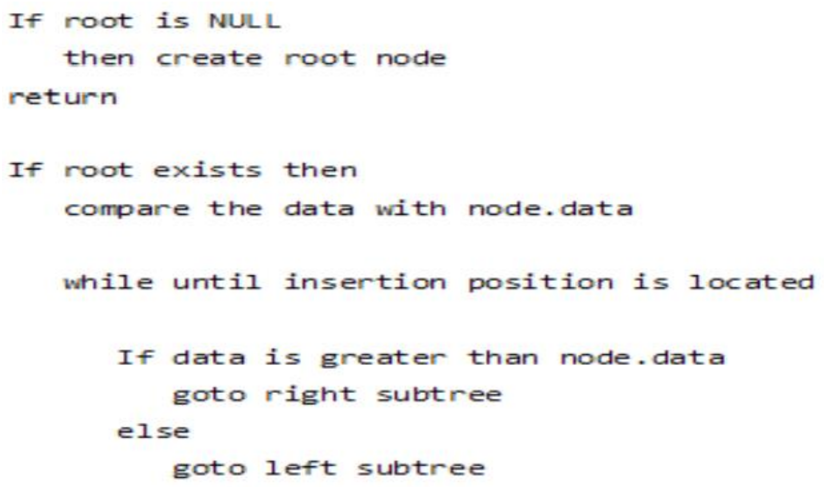
  Parent comes before children; overall root first

  B.post-order: Left, Right, Root

  Parent comes after children; overall root last

  c. In Order: In-order: Left, Root, Right,

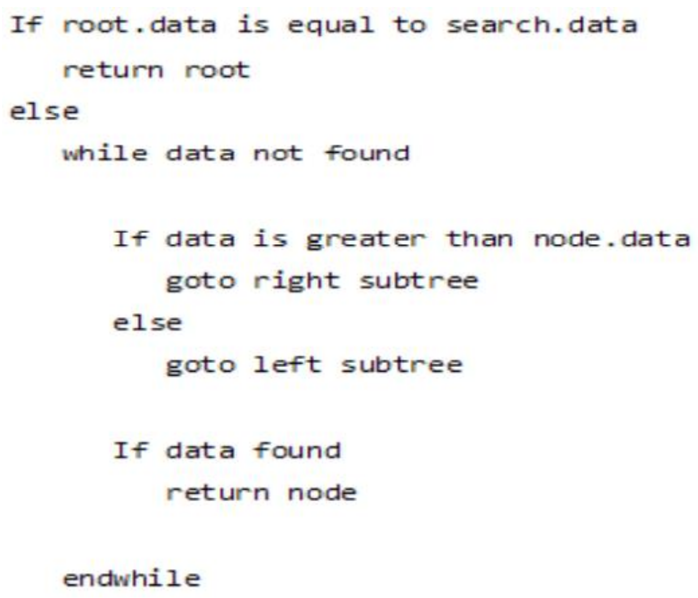
Insert Operation: Algorithm



```
If root is NULL
   then create root node
return

If root exists then
   compare the data with node.data

   while until insertion position is located

      If data is greater than node.data
         goto right subtree
      else
         goto left subtree
```

Search Operation:

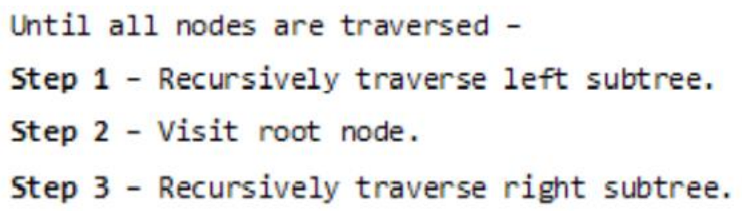
Algorithm



```
If root.data is equal to search.data
   return root
else
   while data not found

      If data is greater than node.data
         goto right subtree
      else
         goto left subtree

      If data found
         return node

   endwhile
```

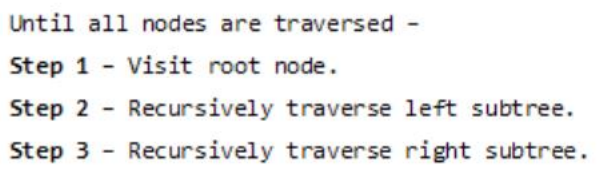
Tree Traversal

In order traversal algorithm



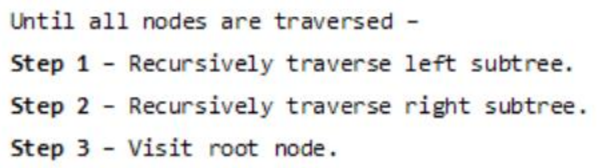
```
Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
Step 3 − Recursively traverse right subtree.
```

Pre order traversal Algorithm



```
Until all nodes are traversed −
Step 1 − Visit root node.
Step 2 − Recursively traverse left subtree.
Step 3 − Recursively traverse right subtree.
```

Pre order traversal Algorithm



```
Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.
```

Deleting in a BST

case 1: delete a node with zero child

if x is left of its parent, set parent(x).left = null

else set parent(x).right = null

case 2: delete a node with one child

link parent(x) to the child of x

case 3: delete a node with 2 children

Replace inorder successor to deleted node position.

Conclusion:

We are able to implement Binary search Tree and its operation in Data Structure.

Assignment No. 05

Problem Statement:

A Dictionary stores keywords & its meaning. 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 understand concept of Tree & Binary Search Tree.

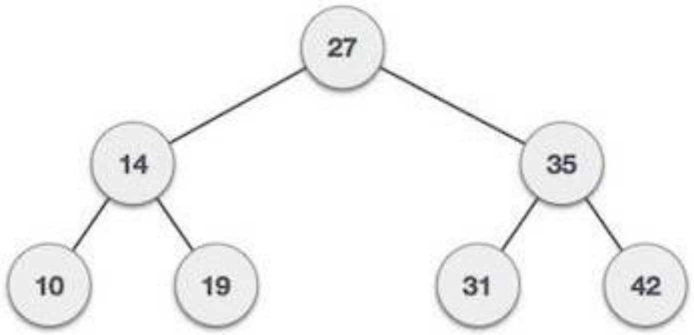
To analyze the working of various Binary Search Tree operations.

Learning Outcome: Students will be able to use various set of operations on Binary Search Tree

Theory:

Binary Search Tree

Binary Search tree exhibits a special behaviour. A node's left child must have a value less



than its parent's value and the node's right child must have a value greater than its parent value.

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.

Algorithm:

```
If root is NULL
   then create root node
return
```

- **Algorithm to insert a node**

```
If root exists then
   compare the data with node.data
   while until insertion position is located
      If data is greater than node.data
         goto right subtree
      else
         goto left subtree
   endwhile
   insert data
end If
```

- **Algorithm to delete a node**

```
If root exists then
   Search a BST node to be deleted & its parents
If temp->data= deldata then
      Found
If deldata < temp->data then
      Parent=temp
      Temp=temp->lchild
Else
      Parent=temp
      Temp=temp->rchild
If temp=Null then
      Return Null
Else if temp->rchild !=Null then
      Find leftmost of right BST node
      Parent=temp
      X=temp->rchild
      While x->lchild != Null
               Parent = x
               X=x->lchild
      Temp->data = x->data
      Temp=x
      If temp->lchild =Nulll and tenp->rchild = Null then
               If parent->lchild = temp
```

```
                        Parent->rchild = Null
               Else
                        Parent->lchild = Null
      Else root = Null

      Delete temp

Else
If temp->lchild != Null and temp->rchild =Null then

      If temp != root then

               If parent->lchild = temp then
                        Parent->lchild = temp->lchild
               Else
                        Parent->rchild = temp->rchild
      Else
               Root = temp->lchild

               Delete temp
```

- **Algorithm to display data in ascending order**

```
Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 – Visit root node.
Step 3 − Recursively traverse right subtree.
```

- **Algorithm to display data in descending order**

```
Until all nodes are traversed −
Step 1 − Recursively traverse right subtree.
Step 2 – Visit root node.
Step 3 − Recursively traverse left subtree.
```

Software required: g++ / gcc compiler- / 64 bit.

Outcome

Learn object-oriented programming features.

Understand & implement different operations on Binary Search Tree.

Conclusion: Thus, we have studied the implementation of various Binary Search Tree operation

Assignment No. 06

Problem Statement:

There are flight paths between cities. If there is a flight between city A and city B then there is an edge between the cities. The cost of the edge can be the time that flight 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.

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.

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 square, two-dimensional array with one row and one column for each vertex in the graph. An entry in row *i* and column *j* is 1 if there is an edge incident between vertex *i* and vertex *j*, and is 0 otherwise. If a graph is a weighted graph, then the entry 1 is replaced with the weight. It is one of the most common and simple representations of the edges of a graph; programs can access this information very efficiently.

For a graph $G = (V, E)$, suppose $V = \{1, 2, \ldots, n\}$. The adjacency matrix for G is a two dimensional $n \setminus n$ Boolean matrix $\boldsymbol{A}$ and can be represented as

***A*[*i*][*j*]** = {1 if there exists an edge <*i*, *j*>

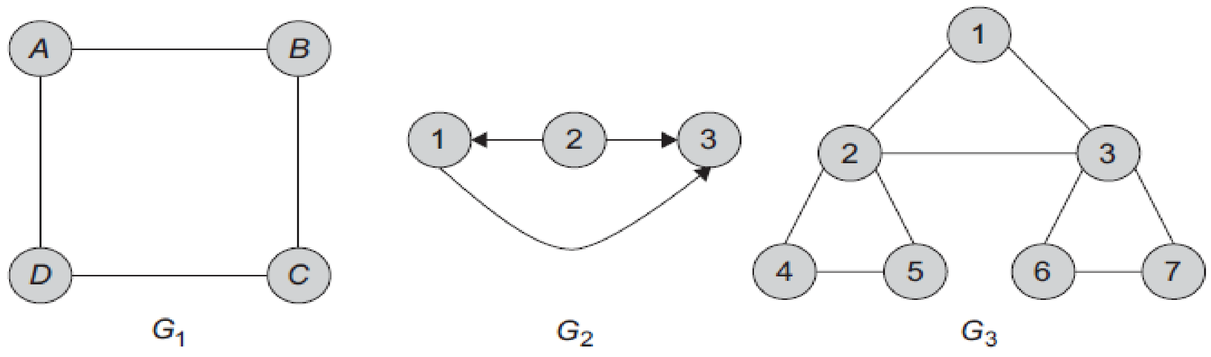
0 if edge <*i*, *j*> does not exist}

The adjacency matrix ***A*** has a natural implementation as in the following:

***A*[*i*][*j*]** is 1 (or true) if and only if vertex *i* is adjacent to vertex *j*. If the graph is undirected, then

***A*[*i*][*j*]** = ***A*[*j*][*i*]** = 1

If the graph is directed, we interpret 1 stored at ***A*[*i*][*j*]**, indicating that the edge from *i* to *j* exists and not indicating whether or not the edge from *j* to *i* exists in the graph.



Graphs G_1, G_2, and G_3

	A	B	C	D
A	0	1	0	1
B	1	0	1	0
C	0	1	0	1
D	1	0	1	0

G_1

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

G_2

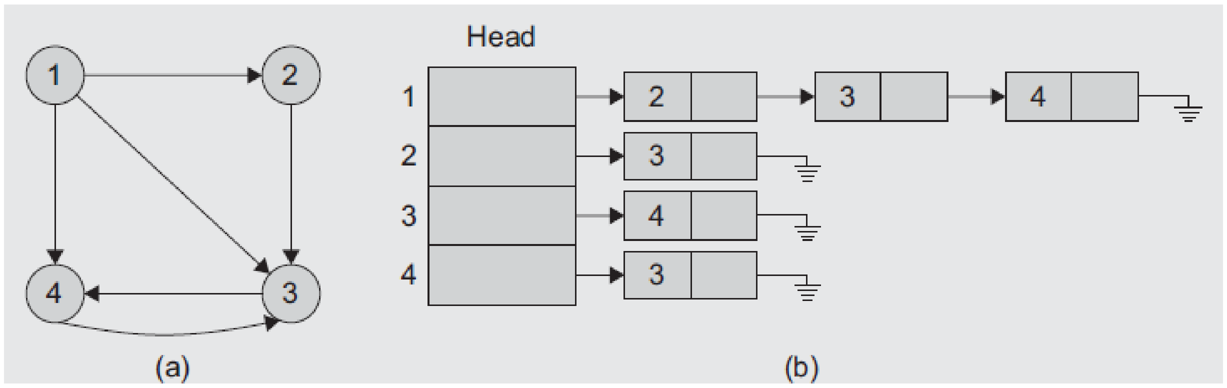
	1	2	3	4	5	6	7
1	0	1	1	0	0	0	0
2	1	0	1	1	1	0	0
3	1	1	0	0	0	1	1
4	0	1	0	0	1	0	0
5	0	1	0	1	0	0	0
6	0	0	1	0	0	0	1
7	0	0	1	0	0	1	0

G_3

Adjacency matrix for G_1, G_2, and G_3

Adjacency List:

In this representation, the *n* rows of the adjacency list are represented as *n*-linked lists, one list per vertex of the graph. The adjacency list for a vertex *i* is a list of all vertices adjacent to it. One way of achieving this is to go for an array of pointers, one per vertex. For example, we can represent the graph *G* by an array Head, where Head[i] is a pointer to the adjacency list of vertex *i*. For list, each node of the list has at least two fields: vertex and link. The vertex field contains the vertex id, and the link field stores a pointer to the next node that stores another vertex adjacent to *i*. Figure (b) shows an adjacency list representation for a directed graph in Fig. (a).



Adjacency list representation (a) Graph G_1 (b) Adjacency list for G_1

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?

Assignment No. 07

Problem Statement:

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

Objective:

To understand the concept and basic of spanning and to find the minimum distance between the vertices of Graph in Data structure.

Outcome:

To implement the concept and basic of spanning and to find the minimum distance between the vertices of Graph in Data structure.

Software & Hardware Requirements:

1. 64-bit Open source Linux or its derivative

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

Theory Concepts:

Properties of a Greedy Algorithm:

1. At each step, the best possible choice is taken and after that only the sub-problem is

solved.

2. Greedy algorithm might be depending on many choices. But, it cannot ever be depending

upon any choices of future and neither on sub-problems solutions.

3. The method of greedy algorithm starts with a top and goes down, creating greedy choices

in a series and then reduce each of the given problem to even smaller ones.

Minimum Spanning Tree:

A Minimum Spanning Tree (MST) is a kind of a sub graph of an undirected graph in which, the sub graph spans or includes all the nodes has a minimum total edge weight. To solve the problem by a prim's algorithm, all we need is to find a spanning tree of minimum length, where a spanning tree is a tree that connects all the vertices together and a minimum spanning tree is a spanning tree of minimum length.

Properties of Prim's Algorithm:

Prim's Algorithm has the following properties:

1. The edges in the subset of some minimum spanning tree always form a single tree.

2. It grows the tree until it spans all the vertices of the graph.

3. An edge is added to the tree, at every step, that crosses a cut if its weight is the minimum

of any edge crossing the cut, connecting it to a vertex of the graph.

<u>**Algorithm:**</u>

1. Begin with any vertex which you think would be suitable and add it to the tree.

2. Find an edge that connects any vertex in the tree to any vertex that is not in the tree. Note

that, we don't have to form cycles.

3. Stop when n - 1 edges have been added to the tree

<u>**Conclusion:**</u>

Implemented the spanning tree to find the minimum distance between the vertices of Graph in Data structure.

Assignment No. 08

Problem Statement:

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.

Learning Objectives:

To learn concept of OBST.

To implement program OBST

Learning Outcome: Students implement the program for above problem.

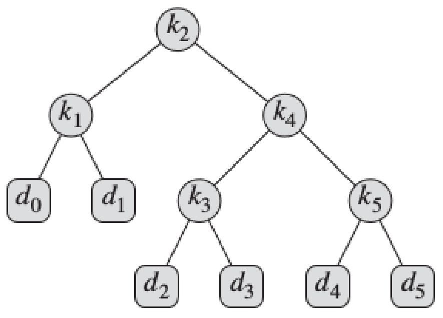
Theory and Algorithm:

We are given an ordered sequence K={k1, k2, ..., kn}of distinct keys in sorted order so that ki < k+1. We wish to build a binary search tree from these keys For each key ki we have a probability pi that a search will be for ki. Some searches may be for values not in K. So we have "dummy" keys d0, d1, ..., dn representing values not inK. In the sorted order these would look like: d0 < k1 < d1 < k2 < ..., kn < dn (meaning that di represents values in between ki and ki+1. The probability qi associated with di is the probability that the search will ask for values in between ki and ki + 1. q0 is the probability that values less than k1 will be asked for and qn is the probability that values greater than kn will be requested.

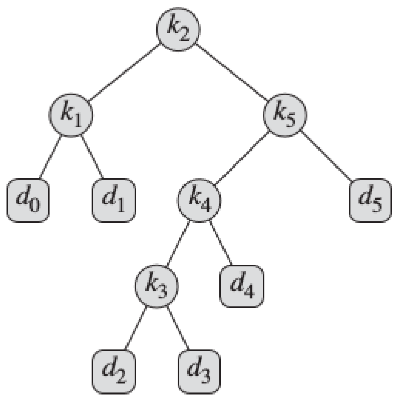
Recall, that in a binary search tree if y is a node in the left sub-tree of x and z is a node in right sub-tree of x,then key [y] ≤ key [x] ≤ key[z].If the frequencies with which these keys occur is uniform, a balanced tree is the best to minimize the average search time. If, however, they are not uniform, then it may be advantageous to have tree that is not balanced in order to minimize the expected number of comparisons. For example, consider:

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Two Binary search trees shown below for this example.



(a)



(b)

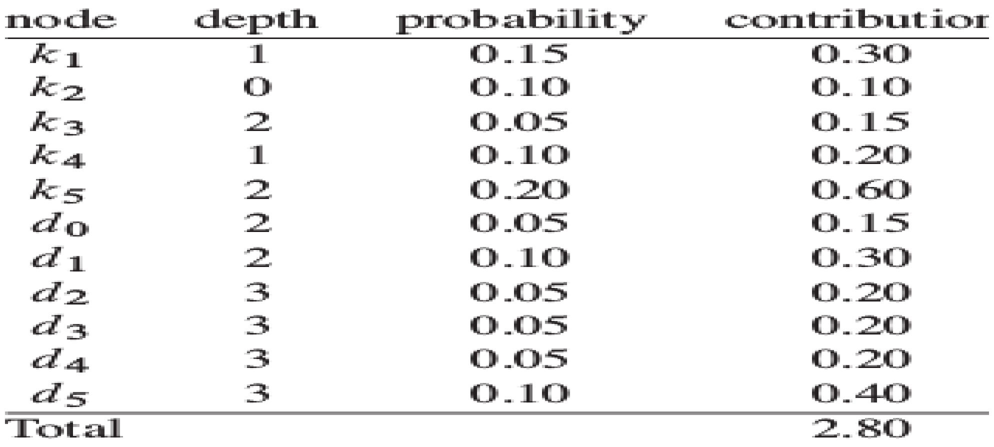
$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1 .$$

(15.10)

Because we have probabilities of searches for each key and each dummy key, we can determine the expected cost of a search in a given binary search tree T . Let us assume that the actual cost of a search equals the number of nodes examined, i.e., the depth of the node found by the search in T , plus 1. Then the expected cost of a search in T is

$$\begin{aligned} \mathrm{E}[\text{search cost in } T] &= \sum_{i=1}^{n} (\mathrm{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\mathrm{depth}_T(d_i) + 1) \cdot q_i \\ &= 1 + \sum_{i=1}^{n} \mathrm{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \mathrm{depth}_T(d_i) \cdot q_i , \qquad (15.11) \end{aligned}$$

where depth T denotes a node’s depth in the tree T . The last equality follows from equation (15.10). In Figure 15.9(a), we can calculate the expected search cost node by node:



node	depth	probability	contributio
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

For a given set of probabilities, we wish to construct a binary search tree whose expected search cost is smallest. We call such a tree an optimal binary search tree. Figure (b) shows an optimal binary search tree for the probabilities given in the figure caption; its expected cost is 2.75. This example shows that an optimal binary search tree is not necessarily a tree whose overall height is smallest. Nor can we necessarily construct an optimal binary search tree by always putting the key with the greatest probability at the root. Here, key k 5 has the greatest search probability of any key, yet the root of the optimal binary search tree shown is k 2 .

(The lowest expected cost of any binary search tree with k 5 at the root is 2.85.)

As with matrix-chain multiplication, exhaustive checking of all possibilities fails to yield an efficient algorithm. We can label the nodes of any n-node binary tree with the eys k 1 ; k 2 ; : : : ; k n to construct a binary search tree, and then add in the dummy keys as leaves. In Problem 12-4, we saw that the number of binary trees with n nodes is .4 ^{n} =n $^{3/2}$, and so we would have to examine an exponential number of binary search trees in an exhaustive search. Not surprisingly, we shall solve this problem with dynamic programming.

We are ready to define the value of an optimal solution recursively. We pick our subproblem domain as finding an optimal binary search tree containing the keys $k_i, \ldots, k_j$, where $i \geq 1$, $j \leq n$, and $j \geq i - 1$. (When $j = i - 1$, there are no actual keys; we have just the dummy key d_{i-1}.) Let us define $e[i, j]$ as the expected cost of searching an optimal binary search tree containing the keys $k_i, \ldots, k_j$. Ultimately, we wish to compute $e[1, n]$.

The easy case occurs when $j = i - 1$. Then we have just the dummy key d_{i-1}. The expected search cost is $e[i, i-1] = q_{i-1}$.

When $j \geq i$, we need to select a root k_r from among $k_i, \ldots, k_j$ and then make an optimal binary search tree with keys $k_i, \ldots, k_{r-1}$ as its left subtree and an optimal binary search tree with keys $k_{r+1}, \ldots, k_j$ as its right subtree. What happens to the expected search cost of a subtree when it becomes a subtree of a node? The depth of each node in the subtree increases by 1. By equation (15.11), the expected search cost of this subtree increases by the sum of all the probabilities in the subtree. For a subtree with keys $k_i, \ldots, k_j$, let us denote this sum of probabilities as

$$w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l \; . \tag{15.12}$$

Thus, if k_r is the root of an optimal subtree containing keys $k_i, \ldots, k_j$, we have

$$e[i, j] = p_r + (e[i, r-1] + w(i, r-1)) + (e[r+1, j] + w(r+1, j)) \; .$$

Noting that

$$w(i, j) = w(i, r-1) + p_r + w(r+1, j) \; ,$$

we rewrite $e[i, j]$ as

$$e[i, j] = e[i, r-1] + e[r+1, j] + w(i, j) \; . \tag{15.13}$$

The recursive equation (15.13) assumes that we know which node k_r to use as the root. We choose the root that gives the lowest expected search cost, giving us our final recursive formulation:

$$e[i, j] = \begin{cases} q_{i-1} & \text{if } j = i - 1 \; , \\ \min\limits_{i \leq r \leq j} \{e[i, r-1] + e[r+1, j] + w(i, j)\} & \text{if } i \leq j \; . \end{cases} \tag{15.14}$$

$\Theta(j-i)$ additions—we store these values in a table $w[1..n+1, 0..n]$. For the base case, we compute $w[i,i-1] = q_{i-1}$ for $1 \le i \le n+1$. For $j \ge i$, we compute

$$w[i,j] = w[i,j-1] + p_j + q_j \;. \tag{15.15}$$

Thus, we can compute the $\Theta(n^2)$ values of $w[i,j]$ in $\Theta(1)$ time each.

The pseudocode that follows takes as inputs the probabilities $p_1, \ldots, p_n$ and $q_0, \ldots, q_n$ and the size n, and it returns the tables e and *root*.

```
OPTIMAL-BST(p, q, n)
1   let $e[1..n+1, 0..n]$, $w[1..n+1, 0..n]$,
            and $root[1..n, 1..n]$ be new tables
2   for $i = 1$ to $n+1$
3       $e[i, i-1] = q_{i-1}$
4       $w[i, i-1] = q_{i-1}$
5   for $l = 1$ to $n$
6       for $i = 1$ to $n-l+1$
7           $j = i+l-1$
8           $e[i,j] = \infty$
9           $w[i,j] = w[i,j-1] + p_j + q_j$
10          for $r = i$ to $j$
11              $t = e[i,r-1] + e[r+1,j] + w[i,j]$
12              if $t < e[i,j]$
13                  $e[i,j] = t$
14                  $root[i,j] = r$
15  return $e$ and $root$
```

Software Required : g++ compiler.

Time Complexity :

$O(n^3)$ as we have to calculate c(i,j), w(i,j) and r(i,j).

Conclusion: Student implemented OBST successfully.

Assignment No. 09

Problem Statement:

A Dictionary stores keywords & its meaning. 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

Objective:

To understand the concept and basic of Height balanced Binary Search tree or AVL tree in Data structure.

Outcome:

To implement the basic concept of AVL tree and to perform basic operation insert, search and delete an element in AVL tree also able to apply correct rotation when tree is unbalance after insertion and deletion operation in Data structure.

Software & Hardware Requirements:

3. 64-bit Open source Linux or its derivative

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

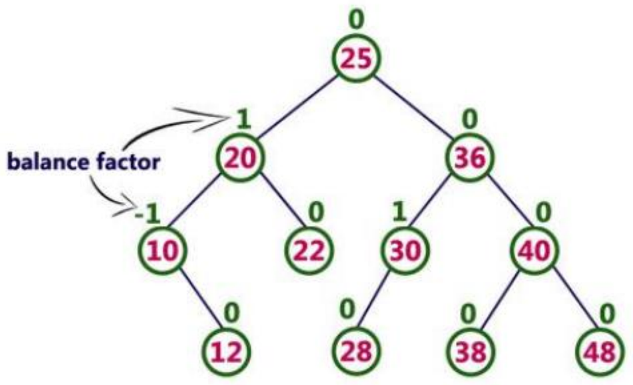
Theory Concepts:

AVL Tree

AVL tree is a self balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced, if the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if for every node, height of its children differ by at most one. In an AVL tree, every node maintains a extra information known as balance factor. The AVL tree was introduced in the year of 1962 by G.M. Adelson-Velsky and E.M. Landis.

An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

Balance factor = heightOfLeftSubtree – heightOfRightSubtree



AVL Tree

AVL tree is a self balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced, if the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if for every node, height of its children differ by at most one. In an AVL tree, every node maintains a extra information known as balance factor. The AVL tree was introduced in the year of 1962 by G.M. Adelson-Velsky and E.M. Landis.

An AVL tree is defined as follows...

An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

Balance factor of a node is the difference between the heights of left and right subtrees of that node. The balance factor of a node is calculated either height of left subtree - height of right subtree (OR) height of right subtree - height of left subtree. In the following explanation, we are calculating as follows...

Balance factor = heightOfLeftSubtree - heightOfRightSubtree

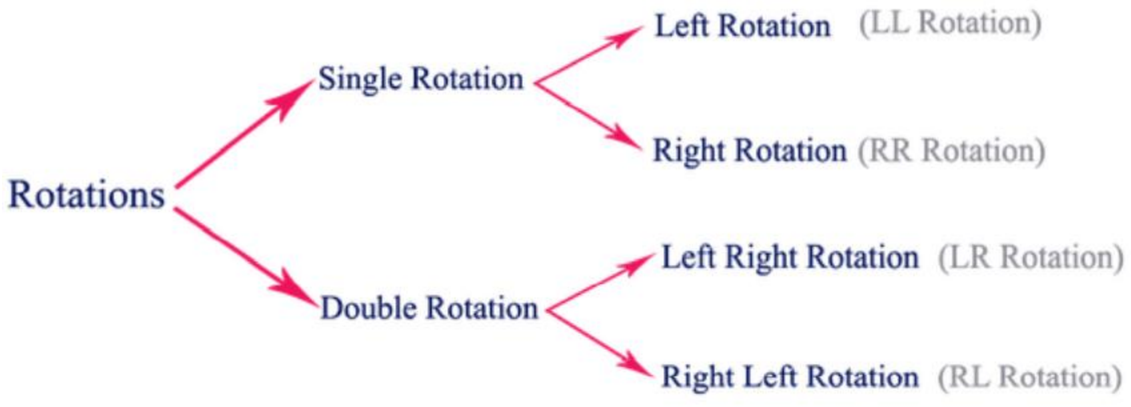
Every AVL Tree is a binary search tree but all the Binary Search Trees need not to be AVL trees.

AVL Tree Rotations

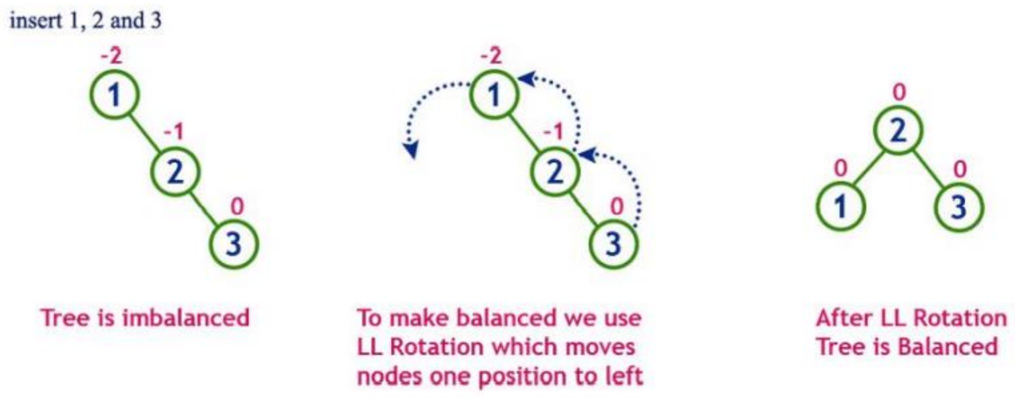
In AVL tree, after performing every operation like insertion and deletion we need to check the balance factor of every node in the tree. If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. We use rotation operations to make the tree balanced whenever the tree is becoming imbalanced due to any operation.

Rotation operations are used to make a tree balanced.

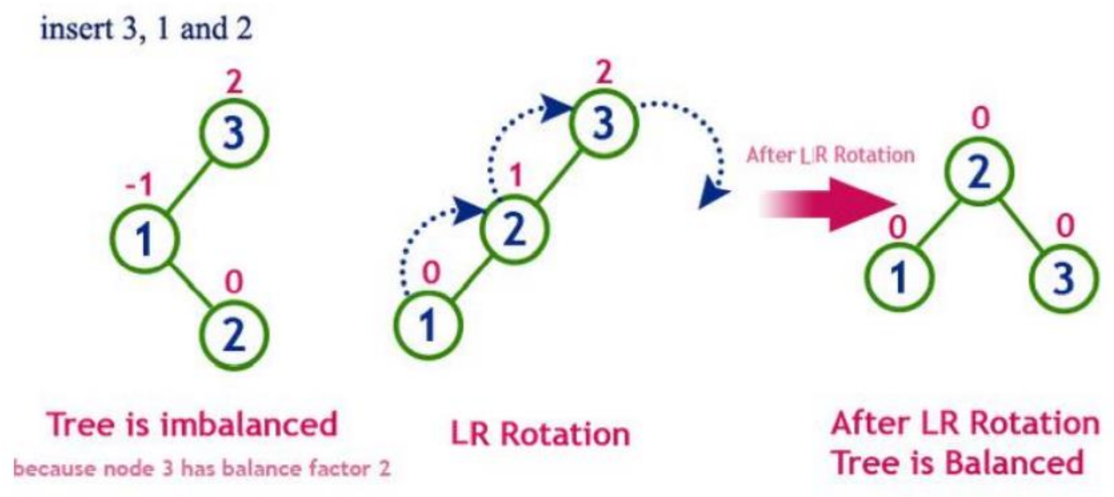
There are four rotations and they are classified into two types.


Rotations
Single Rotation
Double Rotation
Left Rotation (LL Rotation)
Right Rotation (RR Rotation)
Left Right Rotation (LR Rotation)
Right Left Rotation (RL Rotation)

Single Left Rotation (LL Rotation)


insert 1, 2 and 3
-2
1
-1
2
0
3
Tree is imbalanced
To make balanced we use LL Rotation which moves nodes one position to left
After LL Rotation Tree is Balanced

Double Rotation (LR Rotation)



Similarly, RR and RL rotation can be performed

Algorithm:

The following operations are performed on an AVL tree...

1. Search

2. Insertion

3. Deletion

Search Operation in AVL Tree

In an AVL tree, the search operation is performed with O(log n) time complexity. The search operation is performed similar to Binary search tree search operation. We use the following steps to search an element in AVL tree...

Step 1: Read the search element from the user

Step 2: Compare, the search element with the value of root node in the tree.

Step 3: If both are matching, then display "Given node found!!!" and terminate the function

Step 4: If both are not matching, then check whether search element is smaller or larger than that node value.

Step 5: If search element is smaller, then continue the search process in left subtree.

Step 6: If search element is larger, then continue the search process in right subtree.

Step 7: Repeat the same until we found exact element or we completed with a leaf node

Step 8: If we reach to the node with search value, then display "Element is found" and terminate the function.

Step 9: If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with O(log n) time complexity. In AVL Tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

Step 1: Insert the new element into the tree using Binary Search Tree insertion logic.

Step 2: After insertion, check the Balance Factor of every node.

Step 3: If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.

Step 4: If the Balance Factor of any node is other than 0 or 1 or -1 then tree is said to be imbalanced. Then perform the suitable Rotation to make it balanced. And go for next operation.

Conclusion: We are able to implement AVL Binary Search tree and maintain its heigh after every operation

Assignment No. 10

Problem Statement:

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

Objective:

To understand the basic concept of Heap Data structure.

Outcome:

To implement the concept and basic of Max Heap and Min Heap Data Structure and its use in Data structure.

Software & Hardware Requirements:

1. 64-bit Open source Linux or its derivative

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

A double ended priority queue is a data structure that support the following operation

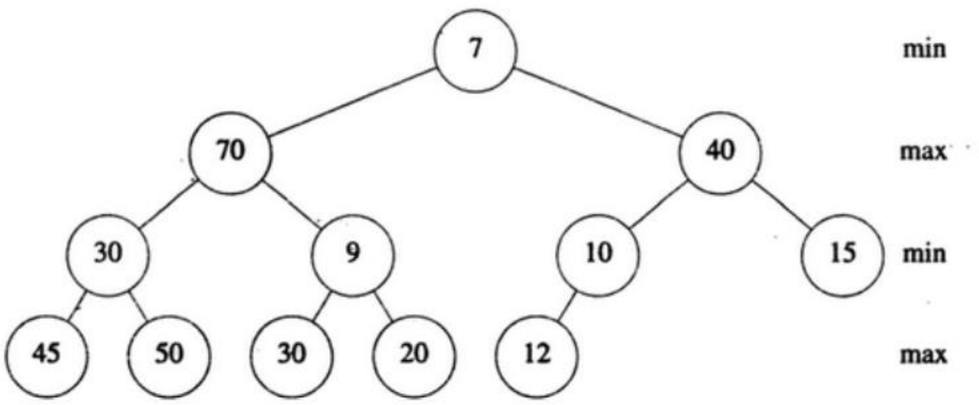
1. Inserting an element with an arbitrary key

2. Deleting an element with largest key

3. Deleting an element with smallest key

When only insertion and one of the two deletion operation re to be supported, a max heap or min heap may be used. A max-min heap support all of above operations.

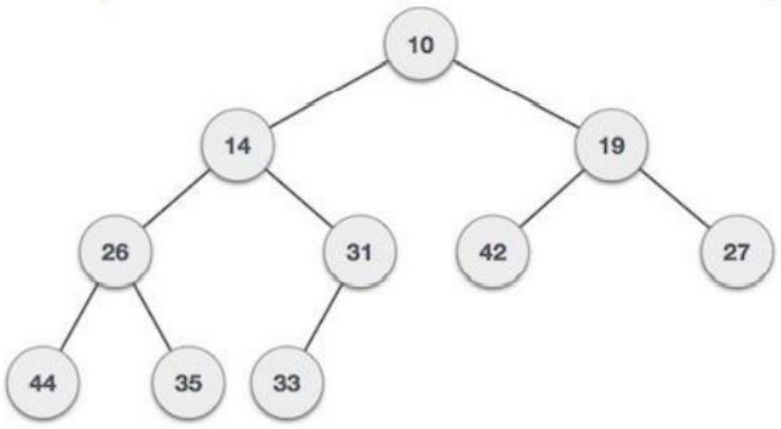
Definition: A min-Max heap is a complete binary tree such that if it is not empty, each element has a data member called key. Alternating levels of this tree are min levels and max level, respectively. the root is on min level.



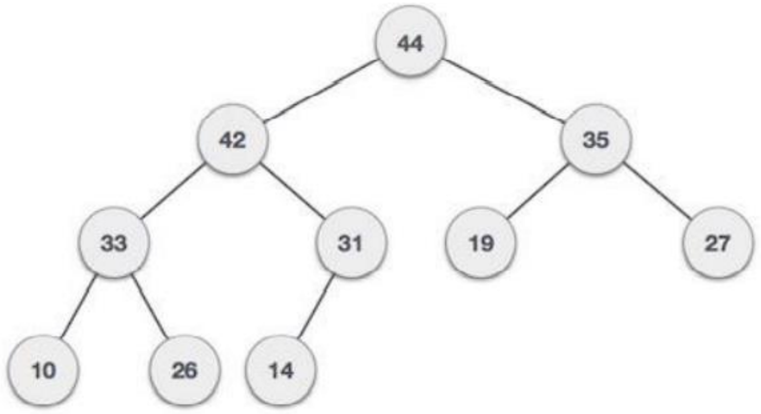
A 12 element Max-Min heap

```
template <class KeyType>
class DEPQ {
public:
    virtual void Insert(const Element<KeyType>&) = 0 ;
    virtual Element<KeyType>* DeleteMax(Element<KeyType>&) = 0 ;
    virtual Element<KeyType>* DeleteMin(Element<KeyType>&) = 0 ;
};
```

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.



Algorithm:

Max heap Construction

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

Max heap Deletion Algorithm

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.

Max-Min Heap

Insert:

To add an element to a min-max heap perform following operations:

1. Append the required key to the array representing the min-max heap. This will likely break the min-max heap properties, therefore we need to adjust the heap.
2. Compare this key with its parent:

1. If it is found to be smaller (greater) compared to its parent, then it is surely

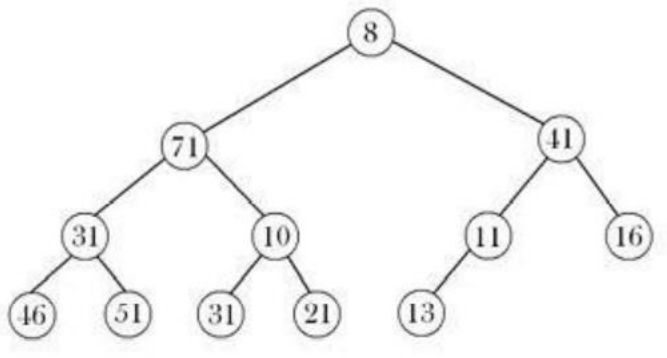
   smaller (greater) than all other keys present at nodes at max(min) level that are on the path from the present position of key to the root of heap. Now, just check for nodes on Min(Max) levels.

2. If the key added is in correct order then stop otherwise swap that key with its parent.

Example

Here is one example for inserting an element to a Min-Max Heap.

Say we have the following min-max heap and want to install a new node with value 6.



Initially, element 6 is inserted at the position indicated by j. Element 6 is less than its parent element. Hence it is smaller than all max levels and we only need to check the min levels. Thus, element 6 gets moved to the root position of the heap and the former root, element 8, gets moved down one step.

If we want to insert a new node with value 81 in the given heap, we advance similarly. Initially the node is inserted at the position j. Since element 81 is larger than its parent element and the parent element is at min level, it is larger than all elements that are on min levels. Now we only need to check the nodes on max levels.

Delete :

Delete the minimum element

To delete min element from a Min-Max Heap perform following operations.

The smallest element is the root element.

1. Remove the root node and the node which is at the end of heap. Let it be x.
2. Reinsert key of x into the min-max heap

Reinsertion may have 2 cases:

1. If root has no children, then x can be inserted into the root.

2. Suppose root has at least one child. Find minimum value (Let this is be node m). m is in one of the children or grandchildren of the root. The following condition must be considered:

    1. x.key <= h[m].key: x must be inserted into the root.

    2. x.key > h[m].key and m are child of the root L: Since m is in max level, it has no descendants. So, the element h[m] is moved to the root and x is inserted into node m.

    3. x.key > h[m].key and m is grandchild of the root: So, the element h[m] is moved to the root. Let p be parent of m. if x.key > h[p].key then h[p] and x are interchanged.

Algorithm Average Worst Case

Insert O(log 2 n) O(log 2 n)

Delete O(log 2 n) O(log 2 n)

<u>**Conclusion**</u>: we are able to implement Max /Min heap concept in Data Structure

Assignment No 11

Problem Statement:

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

Analyse 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

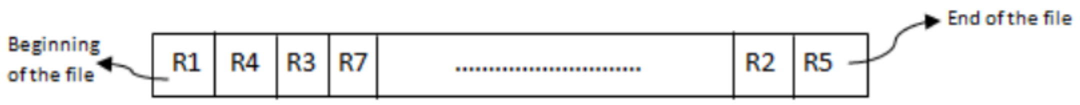
6. Cluster File Organization

Sequential File Organization:

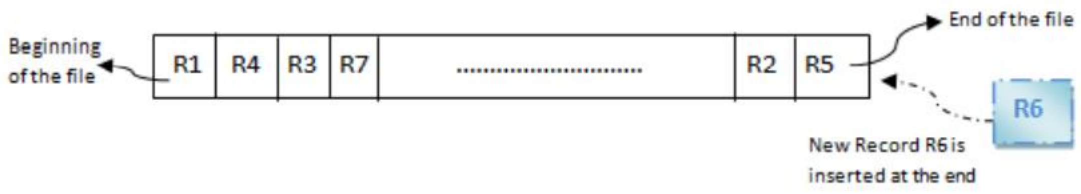
It is one of the simple methods of file organization. Here each file/records are stored one

after the other in a sequential manner. This can be achieved in two ways:

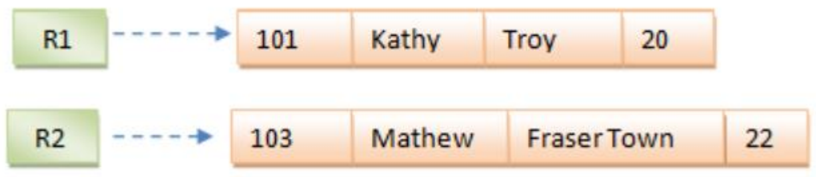
Records are stored one after the other as they are inserted into the tables. This method is called pile file method. When a new record is inserted, it is placed at the end of the file. In the case of any modification or deletion of record, the record will be searched in the memory blocks. Once it is found, it will be marked for deleting and new block of record is entered.



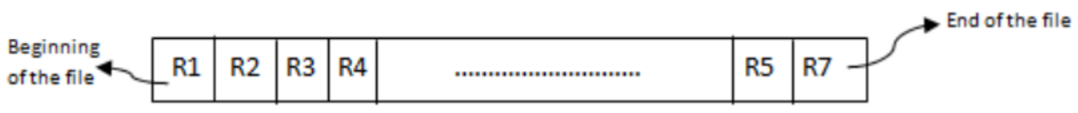
Inserting a new record:



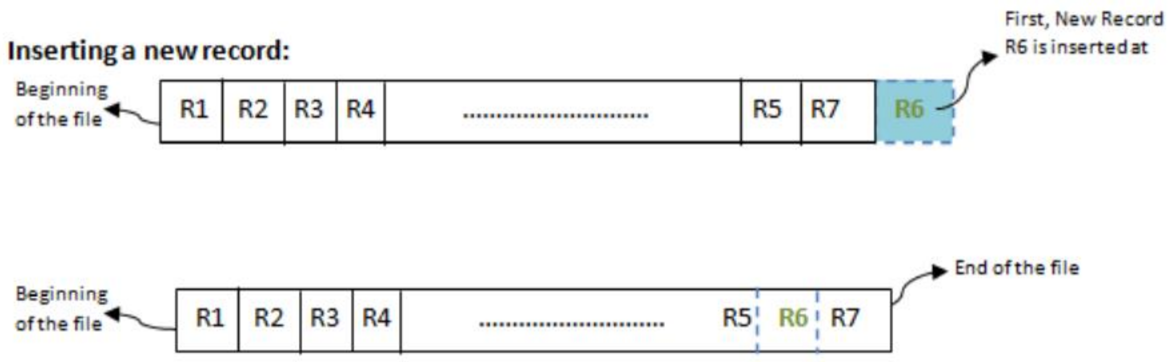
In the diagram above, R1, R2, R3 etc are the records. They contain all the attribute of a row. i.e.; when we say student record, it will have his id, name, address, course, DOB etc. Similarly R1, R2, R3 etc can be considered as one full set of attributes.



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

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:

- Learn File organization in data structure.
- Understand & implement sequential file and operation on it.

Assignment No 12

Problem Statement:

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

Objectives:

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

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

Learning Objectives:

To understand concept of file organization in data structure.

To understand concept & features of indexed sequential file organization.

Learning Outcome:

Define class for sequential file using Object Oriented features.

Analyse working of various operations on indexed sequential file.

Theory:

When there is need to access records sequentially by some key value and also to access records directly by the same key value, the collection of records may be organized in an effective manned called Indexes Sequential Organization.

You must be familiar with search process for a word in a language dictionary. The data in the dictionary is stored in sequential manner. However an index is provided in terms of thumb tabs. To search for a word we do not search sequentially. We access the index that is the appropriate thumb tab, locate an approximate location for the word and then proceed to find the word sequentially.

To implement the concept of indexed sequential file organizations, we consider an approach in which the index part and data part reside on a separate file. The index file has a tree structure and data file has a sequential structure. Since the data file is sequenced, it is not necessary for the index to have an entry for each record Following figure shows a sequential file with a two-level index.

Level 1 of the index holds an entry for each three-record section of the main file. The level 2 indexes level 1 in the same way.

When the new records are inserted in the data file, the sequence of records need to be preserved and also the index is accordingly updated.

Two approaches used to implement indexes are static indexes and dynamic indexes.

As the main data file changes due to insertions and deletions, the static index contents may change but the structure does not change. In case of dynamic indexing approach, insertions and deletions in the main data file may lead to changes in the index structure. Recall the change in height of B-Tree as records are inserted and deleted. Both dynamic and static indexing techniques are useful depending on the type of application.

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

OUTCOME:

Upon completion Students will be able to:

- Learn File organization in data structure.
- Understand & implement indexed sequential file and operation on it.