

DR. D.Y. PATIL INSTITUTE OF TECHNOLOGY
PIMPRI, PUNE-411018

DEPARTMENT
OF
COMPUTER ENGINEERING

LAB MANUAL

PRACTICAL- DATA STRUCTURES AND ALGORITHMS LAB (210256)

S.E 2019 Pattern

1. Vision of the Institute:

"Empowerment through Knowledge"

2. Mission of the Institute:

Developing human potential to serve the Nation by

- Dedicated efforts for quality education
- Yearning to promote research and development
- Persistent endeavor to imbibe moral and professional ethics
- Inculcating the concept of emotional intelligence
- Emphasizing extension work to reach out to the society
- Treading the path to meet the future challenges

3. Vision of the Department:

To produce globally competitive computer professionals, enriched with knowledge and power of innovation.

4. Mission of the Department:

- Imparting quality education using state-of-art facilities to meet the global challenges.
- Enhancing the potential of aspiring students and faculty for higher education and lifelong learning.
- Imbibing ethical values and developing leadership skills those lead to professionals with strong commitments.

5. Program Education Outcomes:

PEO1:

Have strong fundamental concepts in mathematics, science and engineering to address technological challenges.

PEO2:

Possess knowledge and skills in the field of Computer Engineering for analyzing, designing and implementing novel software products in a dynamic environment for successful career and pursue higher studies.

PEO3:

Demonstrate multidisciplinary approach and leadership skills that augment their professional competency.

PEO4:

Exhibit commitment to ethical practices, societal contributions and lifelong learning

6. Program Specific Outcomes

- a.** The ability to understand, analyze and develop computer programs in the areas related to computer engineering.
- b.** The ability to understand the evolutionary changes in computing, apply standard practices and strategies in software project development using open-ended programming environments to deliver a quality product for business success, real world problems and meet the challenges of the society related to computer engineering.
- c.** The ability to employ modern computer languages, environments, and platforms in creating innovative career paths to be an entrepreneur, lifelong learning and a zest for higher studies, innovation and research and also to act as a good citizen by inculcating in them moral values & ethics.

7. Programme Outcomes:

- a.** An ability to apply knowledge of computing, mathematics, science and engineering fundamentals appropriate to Computer Engineering.
- b.** An ability to define the problems and provide solutions by designing and conducting experiments, interpreting, and analysing data.
- c.** An ability to design, implement and evaluate a system, process, component, and programme to meet desired needs within realistic constraints.
- d.** An ability to investigate, formulate, analyze, and provide appropriate solution to the engineering problems.
- e.** An ability to use modern engineering tools and technologies necessary for engineering practice.
- f.** An ability to analyze the local and global impact of computing on individuals, organizations, and society.
- g.** An ability to understand the environmental issues and provide the sustainable system.
- h.** An ability to understand professional and ethical responsibility.

i. An ability to function effectively as an individual or as a team member to accomplish the goal.

j. An ability to communicate effectively at different levels.

k. An ability to keep abreast with contemporary technologies through lifelong learning.

l. An ability to understand engineering, management, financial aspects, performance, optimizations and time complexity necessary for professional practice.

8. Graduate Attributes and Program Outcomes

Graduate Attributes	Program Outcomes
1. Engineering Knowledge	a. An ability to apply knowledge of computing, mathematics, science, and engineering fundamentals appropriate to Computer Engineering.
2. Problem Analysis	b. An ability to define the problems and provide solutions by designing and conducting experiments, interpreting and analyzing data.
3. Design & Development of Solutions	c. An ability to design, implement and evaluate a system, process, component, and program to meet desired needs within realistic constraints.
4. Investigation of Complex Problem	d. An ability to investigate, formulate, analyze, and provide appropriate solution to the engineering problems.
5. Modern Tools Usage	e. An ability to use modern engineering tools and technologies necessary for engineering practices.
6. Engineer and Society	f. An ability to analyze the local and global impact of computing on individuals, organizations, and society.
7. Environment & Sustainability	g. An ability to understand the environmental issues and provide the sustainable system.
8. Ethics	h. An ability to understand professional and ethical responsibility.

9. Individual & Teamwork	i. An ability to function effectively as an individual or as a team member to accomplish the goal.
10. Communication	j. An ability to communicate effectively at different levels.
11. Lifelong Learning	k. An ability to keep abreast with contemporary technologies through lifelong learning.
12. Project management & Finance	l. An ability to understand engineering, management, financial aspects, performance, optimization, and time complexity necessary for profession practice.

DATA STRUCTURES AND ALGORITHMS LAB (210256)

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:

- 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.

LIST OF ASSIGNMENTS

Teaching Scheme
Practical: 4 hrs/week

Examination Scheme:
Term Work: 25 Marks
Practical: 25 Marks

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

Sr.No	Name of the assignment	CO's Addressed	PO's Addressed
1.	Consider telephone book database of N clients. Make use of a hash table implementation to quickly look up client's telephone number. Make use of two collision handling techniques and compare them using number of comparisons required to find a set of telephone numbers	CO1	a, b, c, d, e
2.	Implement all the functions of a dictionary (ADT) using hashing and handle collisions using chaining with / without replacement. Data: Set of (key, value) pairs, Keys are mapped to values, Keys must be comparable, Keys must be unique. Standard Operations: Insert (key, value), Find(key), Delete(key)	CO1	a, b, c, d, e
3.	A book consists of chapters, chapters consist of sections and sections consist of subsections. Construct a tree and print the nodes. Find the time and space requirements of your method.	CO2, CO3	a, b, c, d, e
4.	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	CO2, CO3	a, b, c, d, e
5.	Convert given binary tree into threaded binary tree. Analyze time and space complexity of the algorithm.	CO1	a, b, c, d, e
6.	There are flight paths between cities. If there is a flight between city A and city B then there is an edge between the cities. The cost of the edge can be the time that flight take to reach city B from A, or the amount of fuel used for the journey. Represent this as a graph. The node can be represented by airport name or name of the city. Use adjacency list representation of the graph or use adjacency matrix representation of the graph. Check whether the graph is connected or not. Justify the storage representation used.	CO2, CO3	a, b, c, d, e

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.	CO4, CO5	a, b, c, d, e
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.	CO1	a, b, c, d, e
9.	Implement the Heap/Shell sort algorithm implemented in Java demonstrating heap/shell data structure with modularity of programming language.	CO4, CO5	a, b, c, d, 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.	CO4, CO5	a, b, c, d, e
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.	CO4, CO5	a, b, c, d, e
12.	Company maintains employee information as employee ID, name, designation, and salary. Allow user to add, delete information of employee. Display information of particular employee. If employee does not exist an appropriate message is displayed. If it is, then the system displays the employee details. Use index sequential file to maintain the data.	CO4	a, b, c, d, e
13.	Design a mini project to implement Snake and Ladders Game using Python.	CO4, CO5	a, b, c, d, e

References:

1. Horowitz, Sahani, Dinesh Mehata, “Fundamentals of Data Structures in C++”l, Galgotia Publisher, ISBN: 8175152788, 9788175152786.
2. M Folk, B Zoellick, G. Riccardi, “File Structuresl, Pearson Education”, ISBN:81-7758-37-5
3. Peter Brass, “Advanced Data Structures”l, Cambridge University Press, ISBN: 978-1-107-43982-5.
4. Aho, J. Hopcroft, J. Ulman, “Data Structures and Algorithms”l, Pearson Education, 1998,ISBN-0-201-43578-0.
5. Michael J Folk, “File Structures an Object Oriented Approach with C++l”, Pearson Education, ISBN: 81-7758-373-5.
6. Sartaj Sahani, “Data Structures, Algorithms and Applications in C++”l, Second Edition, University Press, ISBN:81-7371522 X.
7. G A V Pai, “Data Structures and Algorithms”l, McGraw-Hill Companies, ISBN -9780070667266. Goodrich,
8. Tamassia, Goldwasser, “Data Structures and Algorithms in Java”l, Wiley Publication, ISBN: 9788126551903

Annexure: The Programme Outcomes are better aligned with Graduate Attributes

GA \ PO	PO											
	a	b	c	d	e	f	g	h	i	j	K	l
1. Engineering Knowledge	√	√	√	√	√	√		√	√	√	√	√
2. Problem Analysis	√	√	√	√	√	√			√	√	√	√
3. Design & Development of Solutions	√	√	√	√	√	√	√	√	√	√	√	√
4. Investigation of Complex Problem	√	√	√	√	√	√			√	√	√	√
5. Modern Tools Usage	√	√	√	√	√	√			√		√	√
6. Engineer and Society	√	√	√	√	√	√	√	√	√	√	√	√
7. Environment & Sustainability						√	√					√
8. Ethics					√	√	√	√	√			√
9. Individual & Team work	√	√	√	√	√				√		√	√
10. Communication					√	√	√	√	√	√		√
11. Lifelong Learning	√	√	√	√	√			√			√	√
12. Project management & Finance	√	√	√	√	√	√	√	√	√	√	√	√

ASSIGNMENT NO.1

Title:

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

Learning 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 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. In C++, you can take advantage of the STL map container for keyed arrays implemented using binary trees, but this article will give you some of the theory behind how a hash table works.

Keyed Arrays vs. Indexed Arrays

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

```
employees [50];
```

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

```
employees ["Brown, John"];
```

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

Hashing Functions

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

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

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

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

A --> 0

B --> 1

C --> 2

D --> 3

...

and so on until Z --> 25.

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

Since we have 260 elements, we can multiply the first letter of the last name by 10. So, when a key like "Smith" is given, the key would be transformed to the index 180 (S is the 19 letters 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 would not 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 driver output for inserting phone number, display and look up function.
e.g

Menu
1.Create Telephone book
2.Display
3.Look up

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

Assignment -2

Title:

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

Data: Set of (key, value) pairs, Keys are mapped to values, Keys must be comparable, Keys must be unique

Standard Operations: Insert (key, value), Find(key), Delete(key)

Learning Objectives:

- ✓ To understand Dictionary (ADT)
- ✓ To understand concept of hashing
- ✓ To understand concept & features like searching using hash function.

Learning Outcome:

- ✓ Define class for Dictionary using Object Oriented features.
- ✓ Analyze working of hash function.

Theory:

Dictionary ADT

Dictionary (map, association list) is a data structure, which is generally an association of unique keys with some values. One may bind a value to a key, delete a key (and naturally an associated value) and lookup for a value by the key. Values are not required to be unique. Simple usage example is an explanatory dictionary. In the example, words are keys and explanations are values.

Dictionary Operations

- **Dictionary create()**
creates empty dictionary
- **boolean isEmpty(Dictionary d)**
- **put(Dictionary d, Key k, Value v)**

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)**

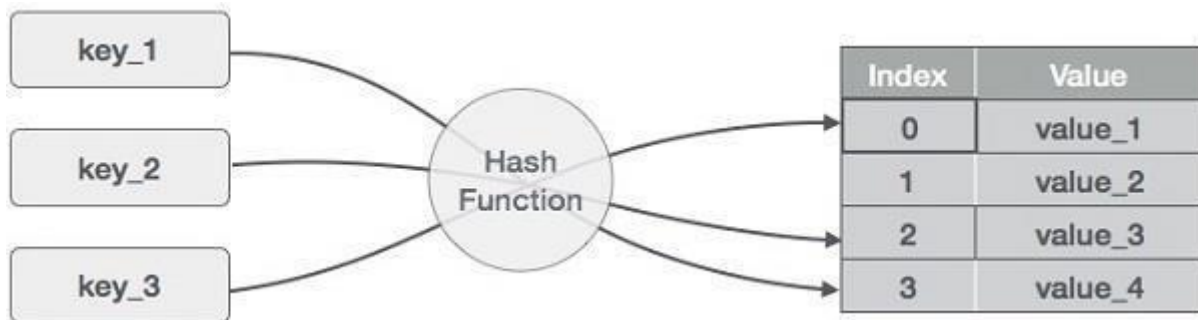
destroys dictionary **d**

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;
}
```

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

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

Assignment -3

Title:

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

Learning Objectives:

- ✓ To understand concept of 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 graphically is represented most commonly as on *Picture 1*. The circles are the nodes, and the edges are the links between them.

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

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

1. The height of a tree with no elements is 0
2. The height of a tree with 1 element is 1
3. The height of a tree with > 1 element is equal to 1 + the height of its tallest subtree.

The depth of a node is the length of the path to its root (i.e., its root path). Every child node is always one level lower than his parent.

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

In some trees, such as heaps, the root node has special properties.

A subtree is a portion of a tree data structure that can be viewed as a complete tree in itself. Any node in a tree T , together with all the nodes below his height, that are reachable from the node, comprise a subtree of T . The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree (in analogy to the term proper subset).

Every node in a tree can be seen as the root node of the subtree rooted at that node.

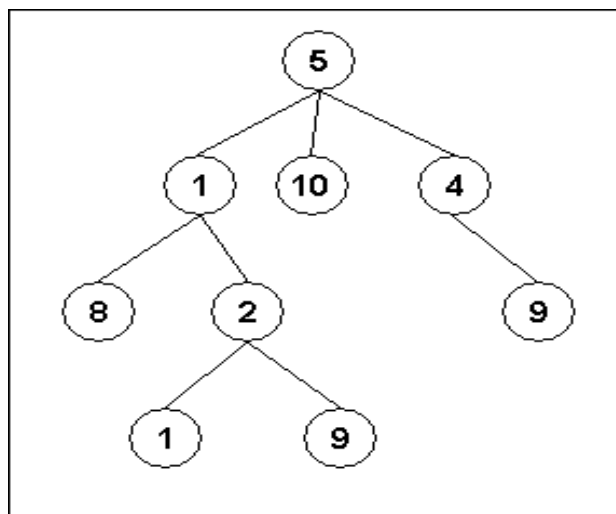


Fig1. An example of a tree

An internal node or inner node is any node of a tree that has child nodes and is thus not a leaf node.

There are two basic types of trees. In an unordered tree, a tree is a tree in a purely structural sense — that is to say, given a node, there is no order for the children of that node. A tree on which an order is imposed — for example, by assigning different natural numbers to each child of each node — is called an ordered tree, and data structures built on them are called ordered tree data structures. Ordered trees are by far the most common form of tree data structure. Binary search trees are one kind of ordered tree.

Important Terms

Following are the important terms with respect to tree.

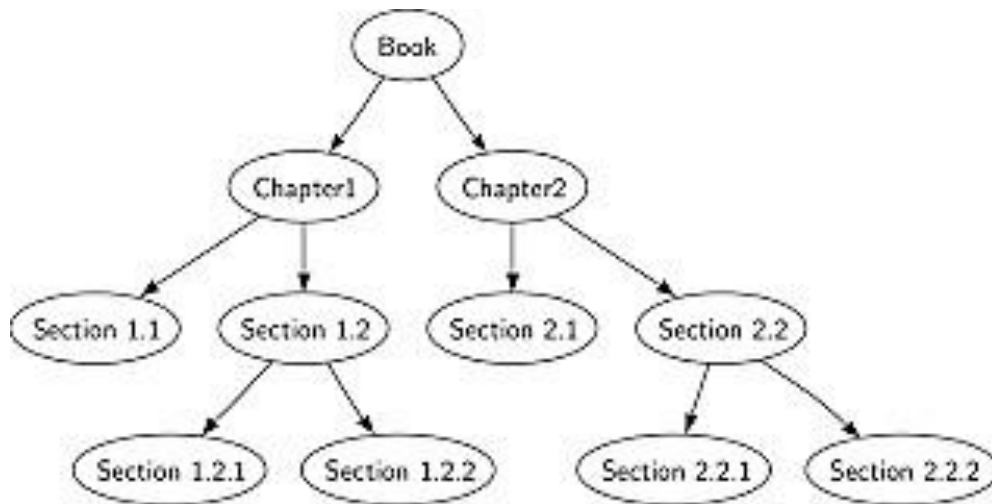
- **Path** – Path refers to the sequence of nodes along the edges of a tree.
- **Root** – The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** – Any node except the root node has one edge upward to a node called parent.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

Advantages of trees

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

- Trees reflect structural relationships in the data.
- Trees are used to represent hierarchies.
- Trees provide an efficient insertion and searching.
- Trees are very flexible data, allowing to move subtrees around with minimum effort.

this assignment we are considering the tree as follows.



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

Conclusion: This program gives us the knowledge 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?

Assignment –4

Title:

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

- i. Insert new node
- ii. Find number of nodes in longest path
- iii. Minimum data value found in the tree
- iv. Change a tree so that the roles of the left and right pointers are swapped at every node
- v. Search a value

Learning Objective:

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

Learning Outcome:

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

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 behavior. 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 –

$$\text{left_subtree (keys)} \leq \text{node (key)} \leq \text{right_subtree (keys)}$$

Tree Node

Following Structure is used for Node creation

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

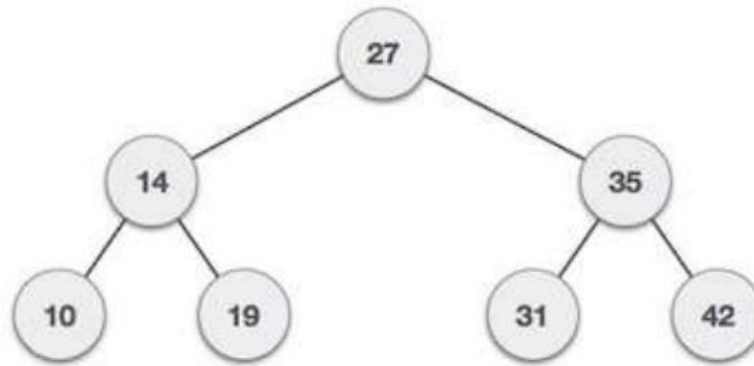


Fig: Binary

Search TreeBST 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.

Post 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.

Expected Output:

Formation of binary search tree structure with its basic Operation Insert, Delete and search, Traverse in tree.

Conclusion: This program gives us the knowledge binary search tree data structure and its basic operations.

Assignment -5

Title:

Convert given binary tree into threaded binary tree. Analyze time and space complexity of the algorithm.

Learning Objective:

To understand the basic concept of Non-Linear Data Structure “threaded binary tree” and its use in Data structure.

Learning Outcome:

To convert the Binary Tree into threaded binary tree and analyze its time and space complexity.

Theory –

Threaded Binary Tree

Inorder traversal of a Binary tree is either be done using recursion or with the use of a auxiliary stack. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).

There are two types of threaded binary tree.

Single Threaded:

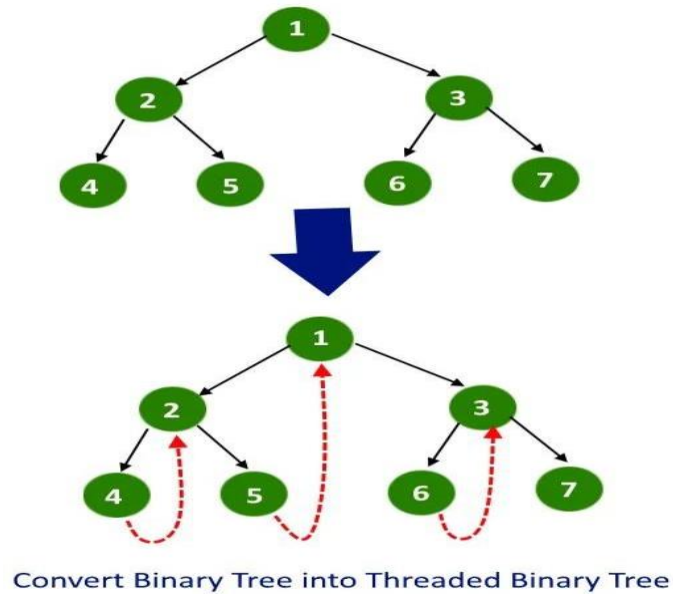
Where a NULL right pointer is made to point to the inorder successor (if successor exists)

Double Threaded:

Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor, respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

The threads are also useful for fast accessing ancestors of a node.

Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.



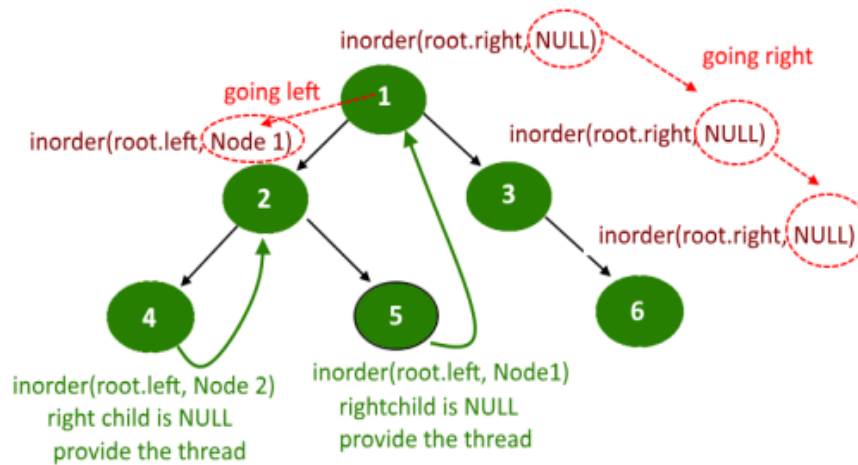
C representation of a Threaded Node

Following is C representation of a single threaded node.

```
struct Node
{
int data;
Node *left, *right; bool rightThread;
}
```

Conversion of Binary Tree to Threaded Binary Tree

- ✓ Do the reverse inorder traversal, means visit right child first.
- ✓ In recursive call, pass additional parameter, the node visited previously.
- ✓ whenever you will find a node whose right pointer is NULL and previous visited node is not NULL then make the right of node points to previous visited node and mark the boolean right Threaded as true.
- ✓ Whenever making a recursive call to right subtree, do not change the previous visited not and when making a recursive call to left subtree then pass the actual previous visited node.



Similarly for other nodes as well.

Expected Outcome:

The binary tree into threaded binary tree in one single traversal with no extra space required.

Conclusion: Able to convert the Binary Tree into threaded binary tree and analyze its time and space complexity.

Assignment -6

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.

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:

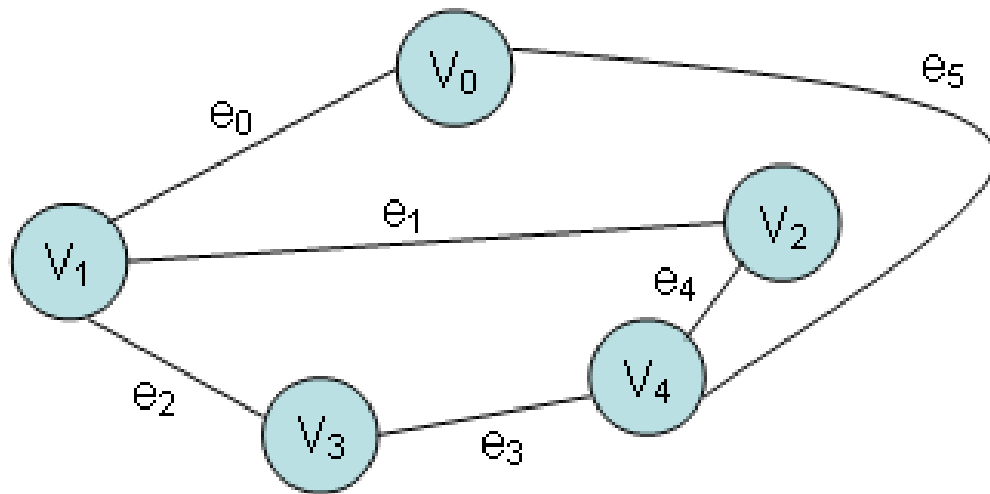
Graphs are the most general data structure. They are also commonly used data structures.

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

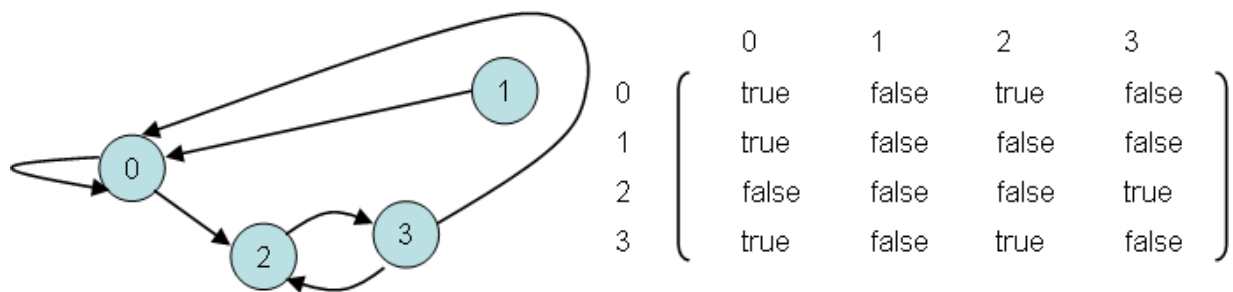


Fig: Graph and adjacency matrix

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

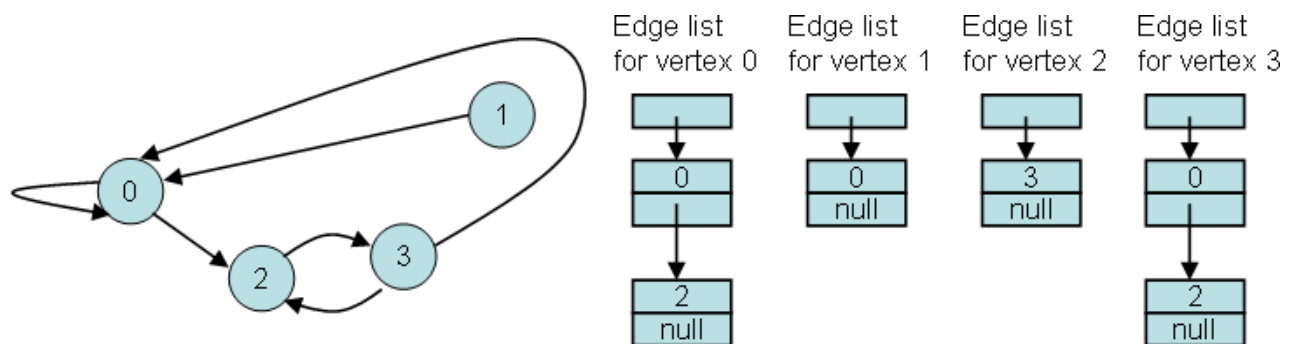


Fig: Graph and adjacency list for each node

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.

Expected Output: Create Adjacency matrix to represent path between various cities.

Conclusion: This program gives us the knowledge of adjacency matrix graph.

Assignment -7

Title:

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

Learning Objective:

- ✓ To understand the concept and basic of spanning.
- ✓ To understand Graph in Data structure.

Learning Outcome:

- ✓ To implement the spanning
- ✓ To find the minimum distance between the vertices of Graph in Data structure.

Theory

Properties of a Greedy Algorithm:

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

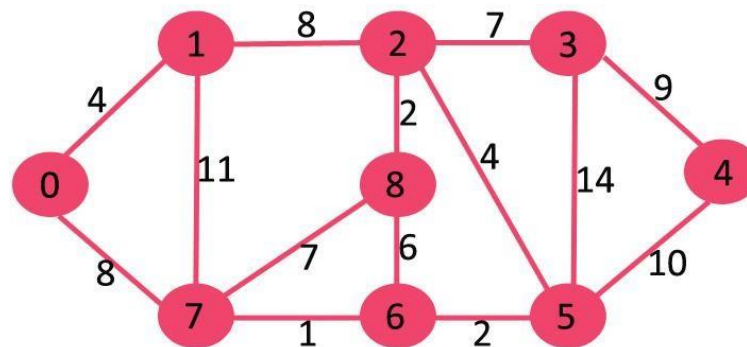
Prim's Algorithm has the following properties:

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.

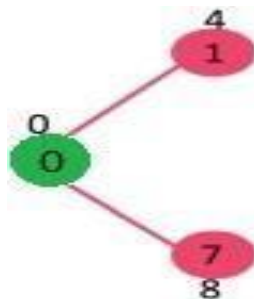
Algorithm:

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. Notethat, we don't have to form cycles.
3. Stop when $n - 1$ edges have been added to the tree



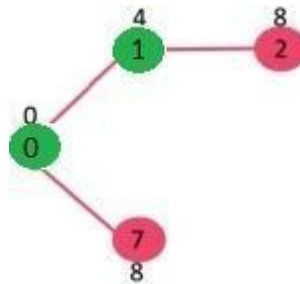
Example

- ✓ The set *mstSet* is initially empty and keys assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where INF indicates infinite.
- ✓ Pick the vertex with the minimum key value.
- ✓ The vertex 0 is picked, include it in *mstSet*. So *mstSet* becomes {0}.
- ✓ After including to *mstSet*, update key values of adjacent vertices.
- ✓ Adjacent vertices of 0 are 1 and 7.
- ✓ The key values of 1 and 7 are updated as 4 and 8.
- ✓ Following subgraph shows vertices and their key values, only the vertices with finite key values are shown.
- ✓ The vertices included in MST are shown in green color.

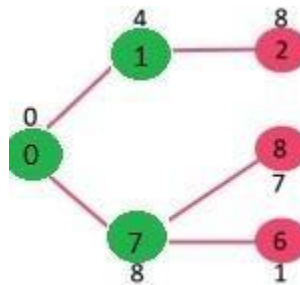


- ✓ Pick the vertex with minimum key value and not already included in MST (not in *mstSet*).
- ✓ The vertex 1 is picked and added to *mstSet*. So *mstSet* now becomes {0, 1}. Update the key values of adjacent vertices of 1.

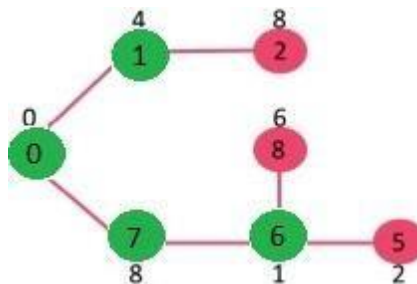
- ✓ The key value of vertex 2 becomes 8.



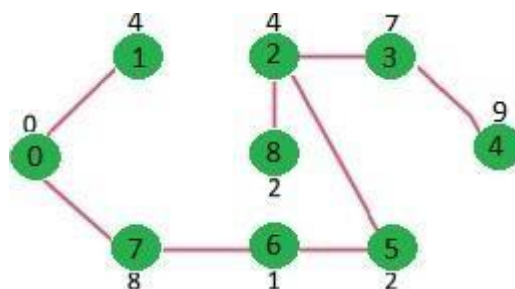
- ✓ Pick the vertex with minimum key value and not already included in MST (not in *mstSet*).
- ✓ Either pick vertex 7 or vertex 2, let vertex 7 is picked. So *mstSet* now becomes {0, 1, 7}.
- ✓ Update the key values of adjacent vertices of 7. The key value of vertex 6 and 8 becomes finite (1 and 7 respectively).



- ✓ Pick the vertex with minimum key value and not already included in MST (not in *mstSet*). Vertex 6 is picked. So *mstSet* now becomes {0, 1, 7, 6}.
- ✓ Update the key values of adjacent vertices of 6. The key value of vertex 5 and 8 are updated.



- ✓ Repeat the above steps until *mstSet* includes all vertices of given graph. Finally,
- ✓ Get the following graph.



Expected output:

Take the adjacency matrix as an input and the edges between the connected Cities should be displayed with weights.

Conclusion:

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

Assignment -8

Title:

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.

Learning Objectives:

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

Learning Outcome:

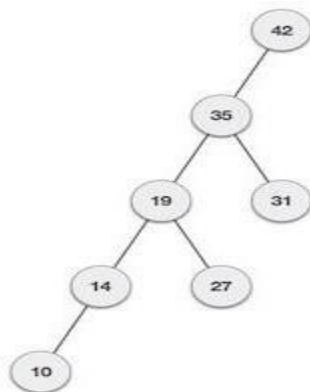
- ☐ Define class for AVL using Object Oriented features.
- ☐ Analyze working of various operations on AVL Tree.

Theory:

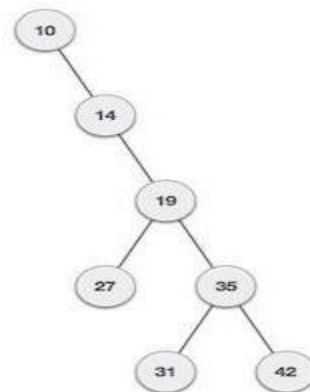
An empty tree is height balanced tree if T is a nonempty binary tree with TL and TR as its left and right sub trees. The T is height balance if and only if its balance factor is 0, 1, -1.

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

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



If input 'appears' non-increasing manner

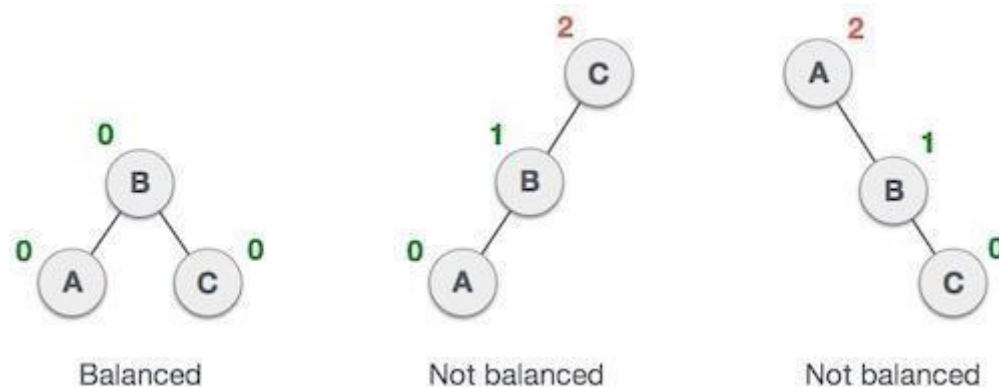


If input 'appears' in non-decreasing manner

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

Named after their inventor **Adelson, Velski & Landis**, **AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the **Balance Factor**.

Here we see that the first tree is balanced and the next two trees are not balanced –



In the second tree, the left subtree of C has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of A has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

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

If the difference in the height of left and right sub-trees is more than 1, the tree is not balanced and some rotation techniques are used.

AVL Rotations

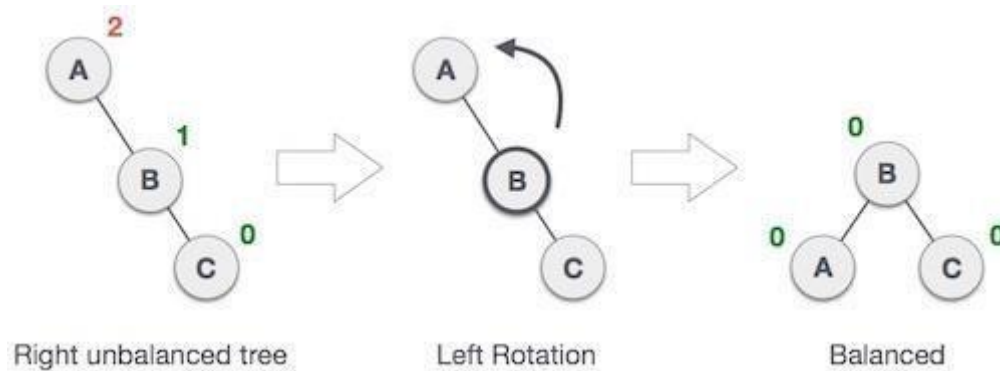
To balance itself, an AVL tree may perform the following four kinds of rotations –

- ☐ Left rotation
- ☐ Right rotation
- ☐ Left-Right rotation
- ☐ Right-Left rotation

The first two rotations are single rotations, and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

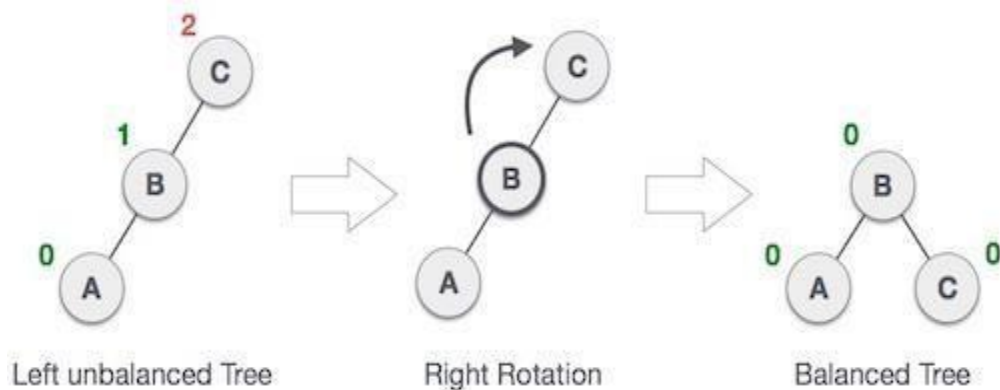
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation –



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

Right Rotation

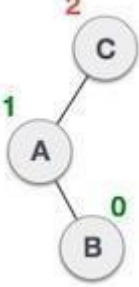
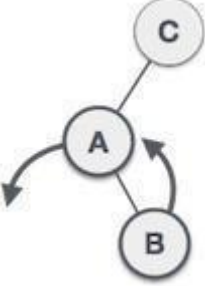
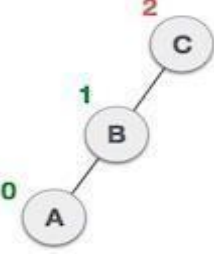
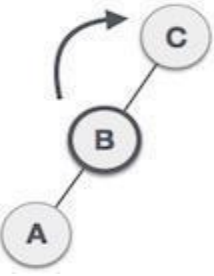
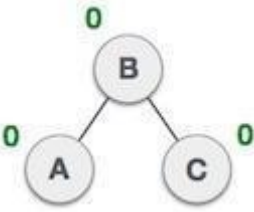
AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



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

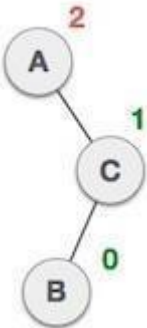
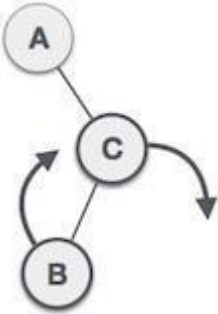
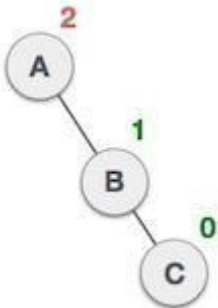
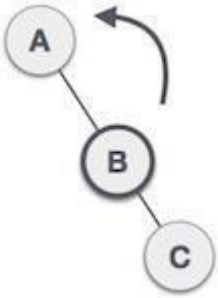
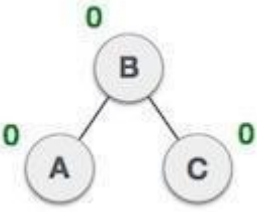
Left-Right Rotation

Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

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

Right-Left Rotation

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

State	Action
	<p>A node has been inserted into the left subtree of the right subtree. This makes A, an unbalanced node with balance factor 2.</p>
	<p>First, we perform the right rotation along C node, making C the right subtree of its own left subtree B. Now, B becomes the right subtree of A.</p>
	<p>Node A is still unbalanced because of the right subtree of its right subtree and requires a left rotation.</p>
	<p>A left rotation is performed by making B the new root node of the subtree. A becomes the left subtree of its right subtree B.</p>
	<p>The tree is now balanced.</p>

Algorithm AVL TREE:

Insert:-

1. If P is NULL, then
 - I. P = new node
 - II. P ->element = x
 - III. P ->left = NULL
 - IV. P ->right = NULL
 - V. P ->height = 0
2. else if $x > P \rightarrow \text{element}$
 - 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 $x < P \rightarrow \text{left} \rightarrow \text{element}$
P = singlerotateleft(P)
else
P = doublerotateleft(P)
3. else
if $x < P \rightarrow \text{element}$
 - a.) insert(x, P -> right)
 - b.) if height (P -> right) - height (P ->left) = 2
if $(x < P \rightarrow \text{right}) \rightarrow \text{element}$
P = singlerotateright(P)
else
P = doublerotateright(P)
4. else
Print already exists
5. int m, n, d.
6. m = AVL height (P->left)
7. n = AVL height (P->right)
8. d = max(m, n)
9. P->height = d+1
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);

Expected 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.

Assignment -9

Title:

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

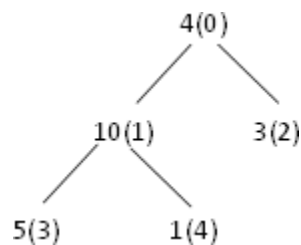
1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps until size of heap is greater than 1.

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.

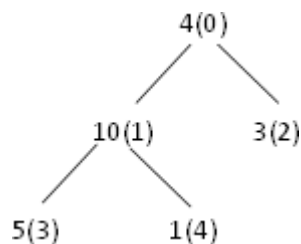
Lets understand with the help of an example:

Input data: 4, 10, 3, 5, 1



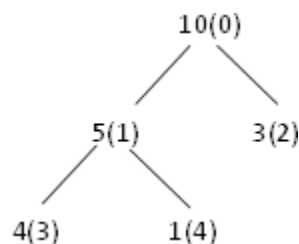
The numbers in bracket represent the indices in the array representation of data.

Applying heapify procedure to index 1:



Applying heapify procedure to index 0:

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



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, Thenew 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.)

Expected Output: Elements in sorted order.

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

Assignment-10

Title:

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

Learning Objectives:

- ✓ To understand concept of heap
- ✓ To understand concept & features like max heap, min heap.

Learning Outcome:

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

Theory:

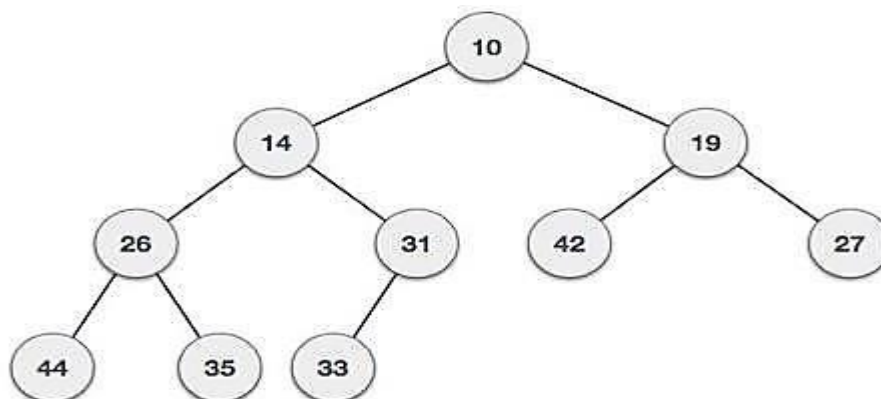
Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly. If α has child node β then –

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

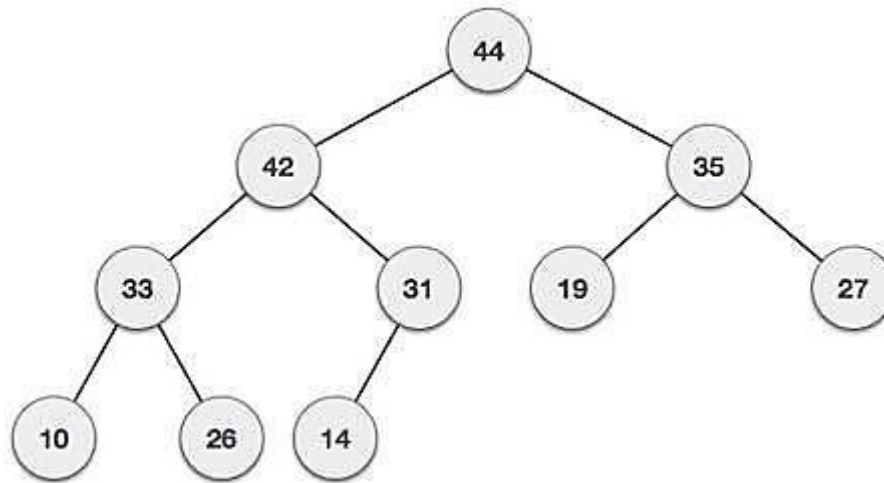
As the value of parent is greater than that of child, this property generates **Max Heap**. Based on this criterion, a heap can be of two types –

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

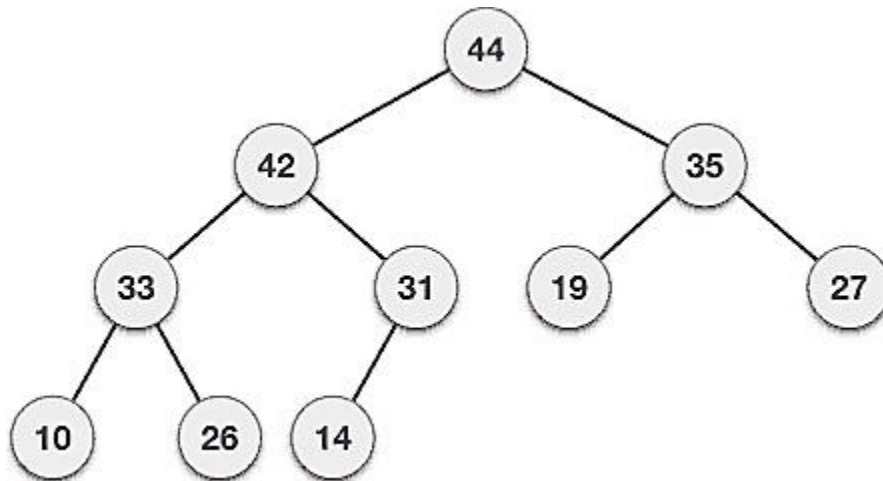
Let us derive an algorithm to delete from max heap. Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value.

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.



Expected Output: Find min and max marks obtained.

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

Assignment-11

Title:

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

Learning Objectives:

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

Learning Outcome:

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

Theory:

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

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

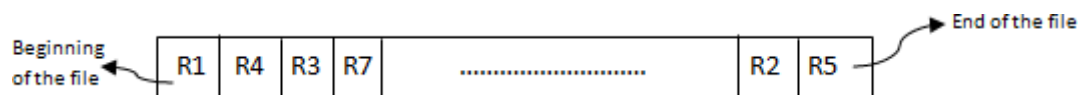
Some of the file organizations are:

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

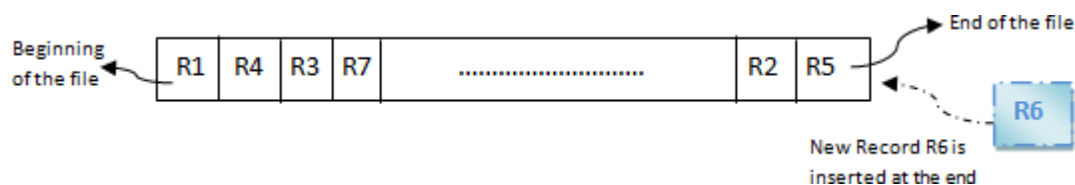
Sequential File Organization:

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

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



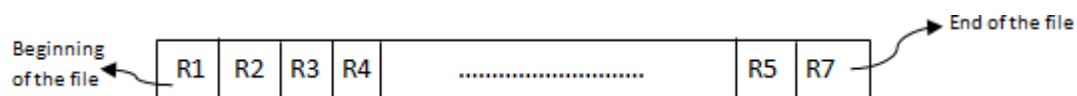
Inserting a new record:



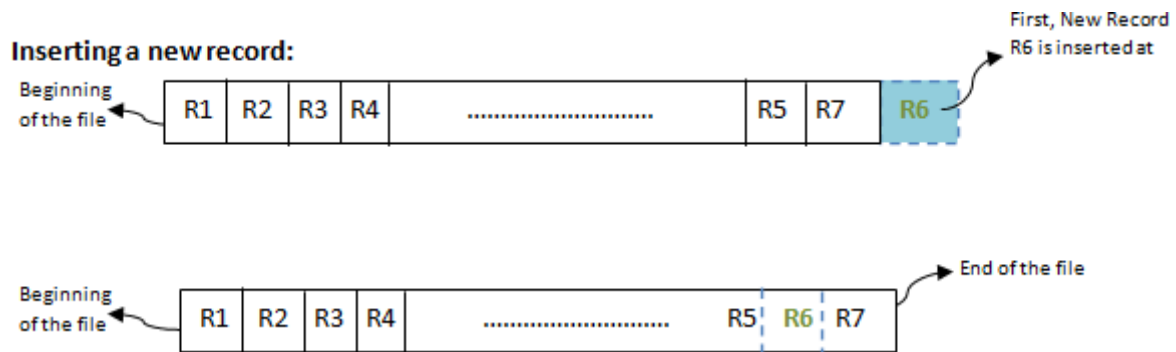
In the diagram above, R1, R2, R3 etc are the records. They contain all the attribute of a row. i.e.; when we say student record, it will have his id, name, address, course, DOB etc. Similarly R1, R2, R3 etc can be considered as one full set of attributes.



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



Inserting a new record:



Advantages:

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

Disadvantages:

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

Expected 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.

Assignment-12

Title:

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

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.
- Analyze working of various operations on Indexed Sequential Access Method

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.

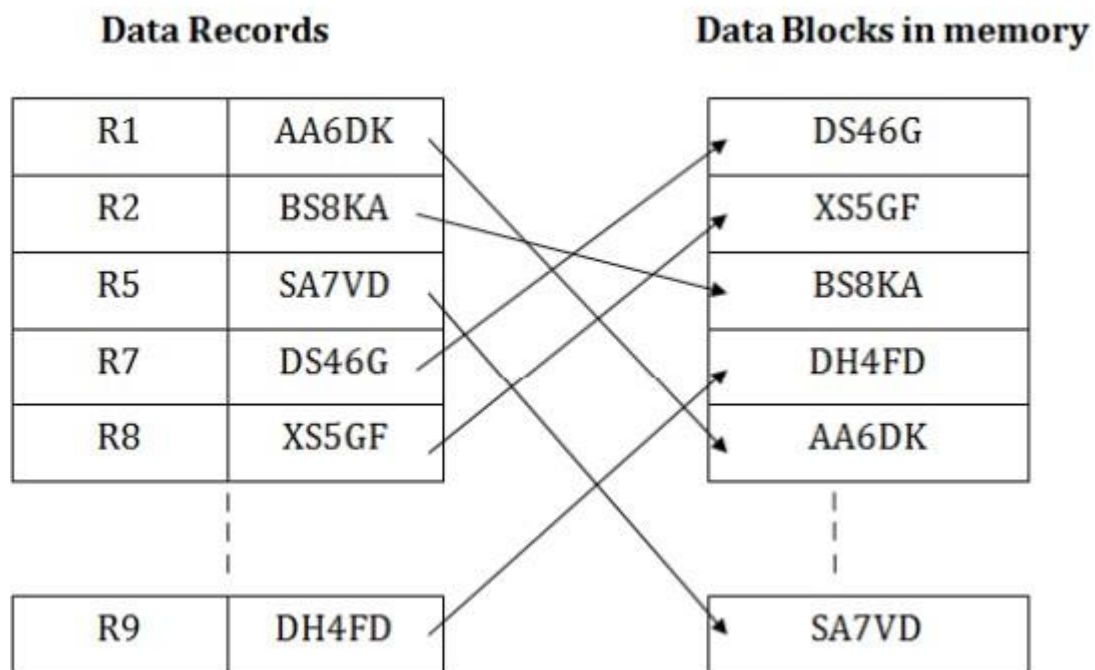
There are various methods of file organizations. These methods may be efficient for certain types of access/selection meanwhile it will turn inefficient for other selections. Hence it is up to the programmer to decide the best suited file organization method depending on his requirement.

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

Indexed sequential access method (ISAM)

ISAM method is an advanced sequential file organization. In this method, records are stored in the file using the primary key. An index value is generated for each primary key and mapped with the record. This index contains the address of the record in the file.



If any record has to be retrieved based on its index value, then the address of the data block is fetched and the record is retrieved from the memory.

Pros of ISAM:

- ✓ In this method, each record has the address of its data block, searching a record in a huge database is quick and easy.
- ✓ This method supports range retrieval and partial retrieval of records. Since the index is based on the primary key values, we can retrieve the data for the given range of value. In the same way, the partial value can also be easily searched, i.e., the student name starting with 'JA' can be easily searched.

Cons of ISAM

- ✓ This method requires extra space in the disk to store the index value. When the new records are inserted, then these files have to be reconstructed to maintain the sequence.
- ✓ When the record is deleted, then the space used by it needs to be released. Otherwise, the performance of the database will slow down.

Expected Output: If record of employee does not exist an appropriate message is displayed otherwise the employee's details are displayed.

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

Assignment-13

Title:

Design a mini project to implement Snake and Ladders Game using Python.

Learning Objectives:

- ✓ To understand concept & features of Python Programming.

Learning Outcome:

- Analyze working of various python file scripts (snakes_ladders.py), resources files and sound files.

Theory:

Python:

- Python is a high-level, general-purpose and a very popular programming language.
- Python programming language (latest Python 3) is being used in web development, Machine Learning applications, along with all cutting-edge technology in Software Industry.
- Python Programming Language is very well suited for Beginners, also for experienced programmers with other programming languages like C++ and Java.

❖ Python Programming Language:

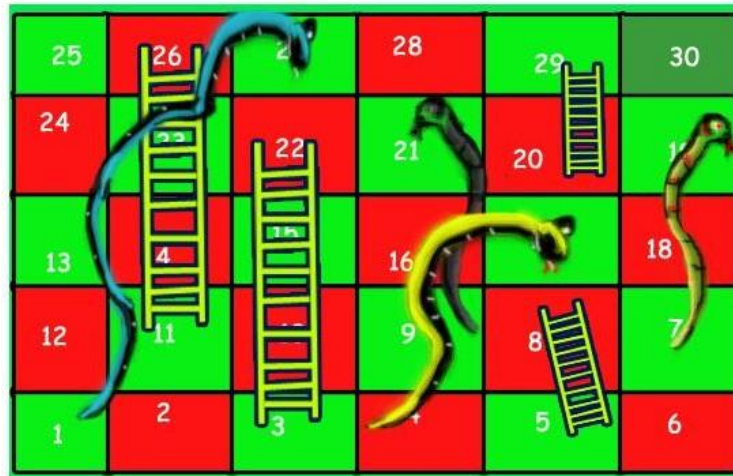
1. Python is currently the most widely used multi-purpose, high-level programming language.
2. Python allows programming in Object-Oriented and Procedural paradigms.
3. Python programs generally are smaller than other programming languages like Java. Programmers have to type relatively less and indentation requirement of the language, makes them readable all the time.
4. Python language is being used by almost all tech-giant companies like – Google, Amazon, Facebook, Instagram, Dropbox, Uber... etc.
5. The biggest strength of Python is huge collection of standard library which can be used for the following:
 - Machine Learning
 - GUI Applications (like Kivy, Tkinter, PyQt etc.)
 - Web frameworks like Django (used by YouTube, Instagram, Dropbox)
 - Image processing (like OpenCV, Pillow)
 - Web scraping (like Scrapy, BeautifulSoup, Selenium)

- Test frameworks
- Multimedia
- Scientific computing

This Snakes and Ladders Game contains python file scripts (snakes_ladders.py), resources files and sound files. The gameplay of the system, which is the user can choose an option either to play multiple participant or with the computer.

Beginning of the game, the player needs to roll the dice and in the wake of moving it the game moves the token consequently as indicated by the dice number. The interactivity is like the genuine one. Here, the player likewise gets one more opportunity to roll the dice at whatever point he/she gets 6 number.

There are quantities of stepping stools and snakes in the game which causes the player to update or minimization the square number. The player who arrives at the last square of the track is the champ.



Snakes and Ladders Game

- **Step 1: Create a project name.**

First when you finished installed the **Pycharm IDE** in your computer, open it and then create a “**project name**” after creating a project name click the “**create**” button.

- **Step 2: Create a python file.**

Second after creating a project name, “**right click**” your project name and then click “**new**” after that click the “**python file**”.

- **Step 3: Name your python file.**

Third after creating a python file, Name your python file after that click “**enter**”.

- **Step 4: The Python code.**

The actual coding of how to create **Snakes and Ladders Game in Python**

Expected Output:

Implementation of Snake and Ladders Game using Python.

Conclusion:

Implementation of a mini project using python.