DATA STRUCTURE AND FILES LABORATORY

SUBJECT CODE: 214455

CLASS: SECOND YEAR SEMESTER - II

LAB MANUAL

DEPARTMENT OF INFORMATION TECHNOLOGY,

SAVITRIBAI PHULE PUNE UNIVERSITY, PUNE

DEPARTMENT OF INFORMATION TECHNOLOGY

SAVITRIBAI PHULE PUNE UNIVERSITY, PUNE

PROGRAM OUTCOME (PO)

PO1	To study data structures and their implementations using OOP (C++) and their Applications
PO2	To study some advanced data structures such as trees, graphs and tables
PO3	To learn different file organizations

PROGRAM SPECIFIC OUTCOME (PSO)

PSO1	Apply and implement algorithm to illustrate use of linear data structures such as stack, queue
PSO2	Apply and implement algorithms to create/represent and traverse non-linear data structures such as trees, graphs etc.
PSO3	Apply and implement algorithms to create and manipulate database using different file organizations
PSO4	Learn and apply the concept of hashing in database creation and manipulation

214455: DATA STRUCTURE AND FILES LABORATORY

Teaching Scheme: Practical: 4 Hours/Week 02 **Credits: 02**

Examination Scheme: Term Work: 25 Marks
Practical: 50 Marks

Prerequisites: Fundamentals of Data Structures, Discrete Structures

	hashing function for telephone number and implement chaining with and without replacement.	
10	A business house has several offices in different countries; they want to lease phone lines to connect them with each other and the phone company charges different rent to connect different pairs of cities. Business house want to connect all its offices with a minimum total cost. Solve the problem by suggesting appropriate data structures	48
11	Department maintains a student information. The file contains roll number, name, division and address. Write a program to create a sequential file to store and maintain student data. It should allow the 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 student record is found it should display the student details.	56
12	Implement direct access file using hashing (chaining without replacement) perform following operations on it a. Create Database b. Display Database c. Add a record d. Search a record e. Modify a record	63

REALIZATION OF PO'S AND PSO'S

Sr. No	List of Assignments	PO'S REALIZED	PSO'S REALIZED
1	Implement stack as an abstract data type using linked list and use this ADT for conversion of infix expression to postfix, prefix and evaluation of postfix/prefix expression.	PO1	PSO1
2	Implement priority queue as ADT using single linked list for servicing patients in an hospital with priorities as i) Serious (top priority) ii) medium illness (medium priority) iii) General (Least priority)	PO1	PSO1
3	Create Binary tree and perform following operations: a. Insert b. Display c. Depth of a tree d. Display leaf-nodes e. Create a copy of a tree	PO1, PO2	PSO2
4	Construct and expression tree from postfix/prefix expression and perform recursive and non- recursive In-order, pre-order and post-order traversals.	PO1, PO2	PSO2
5	Implement binary search tree and perform following operations: a. Insert b. Delete c. Search d. Mirror image e. Display f. Display level wise	PO1, PO2	PSO2
6	Consider a friends' network on face book social web site. Model it as a graph to represent each node as a user and a link to represent the fried relationship between them. Store data such as date of birth, number of comments for each user. 1. Find who is having maximum friends 2. Find who has post maximum and minimum comments 3. Find users having birthday in this month Hint: (Use adjacency list representation and perform DFS and BFS traversals)	PO1, PO2	PSO2
7	Represent any real world graph using adjacency list /adjacency matrix find minimum spanning tree using Kruskal's algorithm.	PO1, PO2	PSO2
8	Represent a given graph using adjacency matrix /adjacency list and find the shortest path using Dijkstra's algorithm (single source all destination).	PO1, PO2	PSO2
9	Store data of students with telephone no and name in the structure using hashing function for telephone number and	PO1, PO2	PSO4

	implement chaining with and without replacement.		
10	A business house has several offices in different countries; they want to lease phone lines to connect them with each other and the phone company charges different rent to connect different pairs of cities. Business house want to connect all its offices with a minimum total cost. Solve the problem by suggesting appropriate data structures	PO1, PO2	PSO2
11	Department maintains a student information. The file contains roll number, name, division and address. Write a program to create a sequential file to store and maintain student data. It should allow the 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 student record is found it should display the student details.	PO1, PO3	PSO3
12	Implement direct access file using hashing (chaining without replacement) perform following operations on it a. Create Database b. Display Database c. Add a record d. Search a record e. Modify a record	PO1, PO3	PSO3

214455: DATA STRUCTURE AND FILES LABORATORY

Objective:

The objective of Data Structure and Files Laboratory (DSFL) is to enlighten students with several data structures and algorithms to perform numerous operations on these data structures. This lab complements the Data Structure and Files (DSF) theory course. The Students will gain applied knowledge by studying and implementing data structures and their Applications using OOP (C++). The students will not only learn linear data structures such as stack, queue but they will implement some advanced data structures such as trees, graphs and tables also. The students will gain knowledge about creating and manipulating database using different file organizations.

OUTCOMES:

Upon the completion of DSFL practical course, the student will be able to:

- **Acquire** knowledge about different data structures.
- **Recognize** the appropriate data structure for given problem / Application.
- **Understand** role, purpose and importance of various data structure.
- **Comprehend** the applications of data structures.
- **Write** complex applications using object oriented programming methods.
- **Relate** diverse ways of implementations of data structures and to distinguish the advantages and disadvantages of them.
- **Analyze** the time and space complexity of the different operations on numerous data structures.
- **Acquiring** knowledge of effective and efficient programming.

Guidelines for Student's Lab Journal

- The laboratory assignments are to be submitted by student in the form of journal. The Journal consists of prologue, Certificate, table of contents, and handwritten write-up of each assignment (Title, Objectives, Problem Statement, Outcomes, software & Hardware requirements, Date of Completion, Assessment grade/marks and assessor's sign, Theory- Concept, algorithms, printouts of the code written using coding standards, sample test cases etc.

- Practical Examination will be based on the term work submitted by the student in the form of journal
- Candidate is expected to know the theory involved in the experiment
- The practical examination should be conducted if the journal of the candidate is completed in all respects and certified by concerned faculty and head of the department
- All the assignment mentioned in the syllabus must be conducted

Guidelines for Lab /TW Assessment

- Examiners will assess the term work based on performance of students considering the parameters such as timely conduction of practical assignment, methodology adopted for implementation of practical assignment, timely submission of assignment in the form of handwritten write-up along with results of implemented assignment, attendance etc.

1. Implementation of Stack

AIM: Write a program to implement stack as an abstract data type using linked list and use this ADT for conversion of infix expression to postfix, prefix and evaluation of postfix/prefix expression.

OBJECTIVE:

1) To understand the concept of abstract data type.
2) How different data structures such as arrays and a stacks are represented as an ADT.

THEORY:

1) What is an abstract data type?

An Abstract Data type is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects. There are no standard conventions for defining them. A broad division may be drawn between "imperative" and "functional" definition styles. In general terms, an abstract data type is a *specification* of the values and the operations that has two properties:

- It specifies everything you need to know in order to use the data type
- It makes absolutely no reference to the manner in which the data type will be implemented.

When we use abstract data types, our programs divide into two pieces:

- The Application: The part that uses the abstract data type.
- The Implementation: The part that implements the abstract data type.

2) What is stack? Explain stack operations with neat diagrams.

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

Operations

An abstract data type (ADT) consists of a data structure and a set of **primitive**

- **Push** adds a new element
- **Pop** removes a element

Additional primitives can be defined:

- **IsEmpty** reports whether the stack is empty
- **IsFull** reports whether the stack is full
- **Initialise** creates/initialises the stack
- **Destroy** deletes the contents of the stack (may be implemented by re-initialising
- the stack)

3) Explain how stack can be implemented as an ADT.

User can Add, Delete, Search, and Replace the elements from the stack. It also checks for Overflow/Underflow and returns user friendly errors. You can use this stack implementation to perform many useful functions. In graphical mode, this C program displays a startup message and a nice graphic to welcome the user.

The program performs the following functions and operations:

- **Push**: Pushes an element to the stack. It takes an integer element as argument. If the stack is full then error is returned.
- **Pop**: Pop an element from the stack. If the stack is empty then error is returned. The element is deleted from the top of the stack.
- **DisplayTop** : Returns the top element on the stack without deleting. If the stack is empty then error is returned.

4) **With an example explain how an infix expression can be converted to prefix and postfix form with the stack. (Ans to be written by students)**

ALGORITHM:

Abstract Data Type Stack:

Define Structure for stack(Data, Next Pointer)

Stack Empty:

Return True if Stack Empty else False.
Top is a pointer of type structure stack.
Empty(Top)
Step 1: If Top = = NULL
Step 2: Return 1;
Step 3: Return 0;

Push Opertion:

Top & Node pointer of structure Stack.
Push(element)
Step 1: Node->data=element;
Step 2; Node->Next = Top;
Step 3: Top = Node

Step 3: Stop.

Pop Operation:

Top & Temp pointer of structure Stack.
Pop()
Step 1:If Top != NULL
Then
i) Temp = Top;
ii) element=Temp->data;
iiI) Top = (Top)->Next;
iv) delete temp;
Step 2: Else Stack is Empty.
Step 3: return element;

Infix to Prefix Conversion:

String is an array of type character which store infix expression.
This function return Prefix expression.
InfixToPrefix(String)
Step 1: I = strlen(String);
Step 2: Decrement I;
Step 3: Do Steps 4 to 9 while I >= 0
Step 4: If isalnum(String[I] Then PreExpression[J++] = String[I];
Step 5: Else If String[I]=='('
Then a) Temp=Pop(&Top);//Pop Operator from stack
b) Do Steps while Temp->Operator!=')' and Temp!=NULL
i) PreExpression[J++] = Temp->Operator;
ii) Temp = Pop(&Top);}

Step 6: Else If String[I]==')' Then Push(&Top,Node);//push ‘)’ in a stack
Step 7: Else

a) Temp = Pop(&Top);//Pop operator from stack
b) DO Steps while Temp->Operator != ')' And Temp != NULL
And Priority(String[I])<Priority(Temp->Operator)
i) PreExpression[J++] = Temp->Operator;
ii) Temp = Pop(&Top);}
c) If Temp!=NULL And Temp->Operator==')'
Or Priority(String[I])>=Priority(Temp->Operator))
Then Push(&Top,Temp);
Step 8: Push(&Top,Node);//Push String[I] in a stack;

Step 9: Decrement I;

Step 10: Temp = Pop(&Top);//pop remaining operators from stack
Step 11: Do Steps while Temp != NULL //Stack is not Empty
i) PreExpression[J++] = Temp->Operator;
ii)Temp = Pop(&Top);}
Step 12: PreExpression[J] = NULL;

Step 13: Reverse PreExpression;
Step 14: Return PreExpression.

Infix to Postfix Conversion:

String is an array of type character which store infix expression.
This function return Postfix expression.
InfixToPostfix(String)
Step 1: I = 0;
Step 2: Do Steps 3 to 8 while String[I] != NULL
Step 3: If isalnum(String[I] Then PostExpression[J++] = String[I];
Step 4: Else If String[I]==')'
Then a) Temp=Pop(&Top);//Pop Operator from stack
b) Do Steps while Temp->Operator!='(' and Temp!=NULL
i) PostExpression[J++] = Temp->Operator;
ii) Temp = Pop(&Top);}

Step 5: Else If String[I]=='(' Then Push(&Top,Node);//push '(' in a stack
Step 6: Else
a) Temp = Pop(&Top);//Pop operator from stack
b) DO Steps while Temp->Operator != '(' And Temp != NULL
And Priority(String[I]) >= Priority(Temp->Operator)
iii) PreExpression[J++] = Temp->Operator;
iv) Temp = Pop(&Top);}
c) If Temp!=NULL And Temp->Operator=='('
Or Priority(String[I]) < Priority(Temp->Operator))
Then Push(&Top,Temp);
Step 7: Push(&Top,Node);//Push String[I] in a stack;
Step 8: Increment I;
Step 9: Temp = Pop(&Top);//pop remaining operators from stack
Step 10: Do Steps while Temp != NULL //Stack is not Empty
iii) PostExpres88(&Top);}
Step 11: PostExpression[J] = NULL;
Step 12: Return PostExpression.

PostFix Expression Evaluation:

String is an array of type character which store infix expression.
Postfix_Evaluation(String)
Step 1: Do Steps 2 & 3 while String[I] != NULL
Step 2: If String[I] is operand
Then Push it into Stack
Step 3: If it is an operator Then
Pop two operands from Stack;
Stack Top is 1st Operand & Top-1 is 2nd Operand;
Perform Operation;
Step 4: Return Result.

INPUT:

Test Case O/P

If Stack Empty	Display message “Stack Empty”
Stack Empty	Display message “Stack Full”

INPUT:	**POSTFIX OUTPUT:**
(A+B) * (C-D)	AB+CD-*
A$B*C-D+E/F/(G+H)	AB$C*D-EF/GH+/+
((A+B)*C-(D-E))$(F+G)	AB+C*DE—FG+$
A-B/(C*D$E)	ABCDE$*/-
A^B^C	ABC^^

INPUT:	**PREFIX OUTPUT:**
(A+B) * (C-D)	*+AB-CD
A$B*C-D+E/F/(G+H)	+-*$ABCD//EF+GH
((A+B)*C-(D-E))$(F+G)	$-*+ABC-DE+FG
A-B/(C*D$E)	-A/B*C$DE
A^B^C	^A^BC

NOTE: Here $ AND ^ are used as raised to operator

FAQ:

1. What is data structure?
2. Types of data structure?
3. Examples of linear data structure & Non-linear data structure?
4. What are the operations can implement on stack?
5. Explain recursion using stack.
6. Explain how a string can be reversed using stack.
7. How does a stack similar to list? How it is different?
8. List advantages and disadvantages of postfix and prefix expression over infix expression.

2. Implementation of Priority Queue

AIM: Implement priority queue as ADT using single linked list for servicing patients in an hospital with priorities as i) Serious (top priority) ii) medium illness (medium priority) iii) General (Least priority)

OBJECTIVE:

1) To understand the concept of priority Queue.
2) How data structures Queue is represented as an ADT.

THEORY:

1) What is Queue? Explain Queue operations with neat diagrams?

A queue is a particular kind of collection in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position and removal of entities from the front terminal position. This makes the queue a First-In-First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed. This is equivalent to the requirement that once an element is added, all elements that were added before have to be removed before the new element can be invoked. A queue is an example of a linear data structure.

Queues provide services in computer science, transport, and operations research where various entities such as data, objects, persons, or events are stored and held to be processed later. In these contexts, the queue performs the function of a buffer. Queues are common in computer programs, where they are implemented as data structures coupled with access routines, as an abstract data structure or in object-oriented languages as classes. Common implementations are circular buffers and linked lists. Queue is a data structure that maintain "First In First Out" (FIFO) order. And can be viewed as people queueing up to buy a ticket. In programming, queue is usually used as a data structure for BFS (Breadth First Search).

Operations on queue:

1. enqueue - insert item at the back of queue Q
2. dequeue - return (and virtually remove) the front item from queue Q
3. displayfront - return (without deleting) the front item from queue Q
4. displayrear - return (without deleting) the reart item from queue Q

2) Explain how Queue can be implemented as an ADT.

Theoretically, one characteristic of a queue is that it does not have a specific capacity. Regardless of how many elements are already contained, a new element can always be

added. It can also be empty, at which point removing an element will be impossible until a new element has been added again. A practical implementation of a queue, e.g. with pointers, of course does have some capacity limit, that depends on the concrete situation it is used in. For a data structure the executing computer will eventually run out of memory, thus limiting the queue size. Queue overflow results from trying to add an element onto a full queue and queue underflow happens when trying to remove an element from an empty queue. A bounded queue is a queue limited to a fixed number of items.

3) What is Priority Queue ? Explain with example.

priority queue is an abstract data type in computer programming that supports the following three operations:

- insertWithPriority: add an element to the queue with an associated priority
- getNext: remove the element from the queue that has the highest priority, and return it (also known as "PopElement(Off)", or "GetMinimum")
- peekAtNext (optional): look at the element with highest priority without removing it.

The rule that determines who goes next is called a queueing discpline. The simplest queueing discipline is called FIFO, for "first-in-first-out." The most general queueing discipline is priority queueing, in which each customer is assigned a priority, and the customer with the highest priority goes first, regardless of the order of arrival. The reason I say this is the most general discipline is that the priority can be based on anything: what time a flight leaves, how many groceries the customer has, or how important the customer is. Of course, not all queueing disciplines are "fair," but fairness is in the eye of the beholder.

The Queue ADT and the Priority Queue ADT have the same set of operations and their interfaces are the same. The difference is in the semantics of the operations: a Queue uses the FIFO policy, and a Priority Queue (as the name suggests) uses the priority queueing policy. As with most ADTs, there are a number of ways to implement queues Since a queue is a collection of items, we can use any of the basic mechanisms for storing collections: arrays, lists, or vectors. Our choice among them will be based in part on their performance--- how long it takes to perform the operations we want to perform--- and partly on ease of implementation.

ALGORITHM:

Define structure for Queue(Priority, Patient Info, Next Pointer).

Empty Queue:

Return True if Queue is Empty else False.
isEmpty(Front)
Front is pointer of structure,which is first element of Queue.
Step 1: If Front = = NULL
Step 2: Return 1

Step 3: Return 0

Insert Function:

Insert Patient in Queue with respect to the Priority.
Front is pointer variable of type Queue,which is 1st node of Queue.
Patient is a pointer variable of type Queue, which hold the information about new patient.

```
Insert(Front, Queue )
    Step 1: If Front = = NULL //Queue Empty
              Then  Front = Patient;
    Step 2: Else if Patient->Priority >  Front->Priority
              Then i) Patient->Next = Front;
             ii) Front=Patient;
    Step 3: Else A) Temp = Front;
                 B)   Do Steps a while Temp != NULL And
                          Patient->Priority <= Temp->Next->Priority
                                a)  Temp=Temp->Next;
                 c) Patient->Next = Temp->Next;
                      Temp->Next = Patient;
    Step 4: Stop.
```

Delete Patient details from Queue after patient get treatment:

Front is pointer variable of type Queue,which is 1st node of Queue.
Delete Node from Front.

```
Delete( Front )
Step 1: Temp = Front;
Step 2: Front = Front->Next;
Step 3: return Temp
```

Display Queue Front:

Front is pointer variable of type Queue,which is 1st node of Queue.

```
Display( Front )
Step 1: Temp = Front;
Step 2: Do Steps  while Temp != NULL
        a) Display Temp Data
        b)        If  Priority 1 Then “General Checkup”;
          Else If Priority 2 Then Display “ Non-serious";
          Else If Priority 3 Then Display "Serious"
                   Else Display "Unknown";
           c) Temp = Temp->Next;
Step 3: Stop.
```

Display Queue rear:

Front is pointer variable of type Queue,which is 1st node of Queue.

```
Display( Rear )
Step 1: Temp = Rear;
Step 2: Do Steps  while Temp != NULL
        a) Display Temp Data
        b)        If  Priority 1 Then “General Checkup”;
```

Else If Priority 2 Then Display " Non-serious";
Else If Priority 3 Then Display "Serious"
Else Display "Unknown";
c) Temp = Temp->Next;

Step 3: Stop.

INPUT:

Test Case	O/P
Queue Empty	Display message "Queue Empty"
Queue Full	Display message "Queue Full"

Name of patients & category of patient like

a) Serious (top priority), b) non-serious (medium priority), c) General Checkup (Least priority).

E.g. Enter patient arrival in the hospital with following priorities 1, 3, 2, 2, 1, 3

OUTPUT:

Priority queue cater services to the patients based on priorities.
Patient should be given service in the following order 3, 3, 2, 2, 1, 1

Note: 3 means top priority.

FAQ:

1. What are the types of data structure?
2. What are the operations can implement on queue?
3. What is circular queue?
4. What is Multiqueue?
5. Explain importance of stack in recursion
6. Explain Implicit & Explicit Stack.
7. Applications of stack and Queue as a data structure

3. Binary Tree

AIM: Create Binary Tree (BT) and perform following operations:

a. Insert
b. Display
c. Depth of a tree
d. Display leaf-nodes
e. Create a copy of a tree

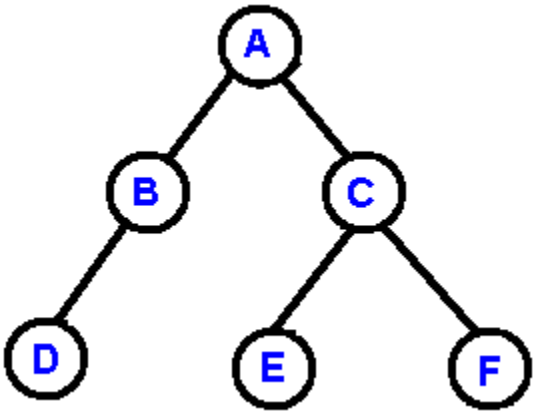
OBJECTIVE:

1. To understand the concept of binary tree as a data structure.
2. Applications and Operations of BT.

THEORY:

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

Definition: A binary tree is a finite set of elements that is either empty or is partitioned into three disjoint subsets. The first subset contains a single element called the root of the tree. The other two subsets are themselves binary trees, called the left and right subtrees of the original tree. Conventional method of picturing binary tree



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

Advantages of BT:

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

- BT reflect structural relationships in the data

- BT are used to represent hierarchies
- BT are very flexible data, allowing to move subtrees around with minimum effort

Traversals

A traversal is a process that visits all the nodes in the tree. Since a tree is a nonlinear data structure, there is no unique traversal. Two broad categories of traversals are

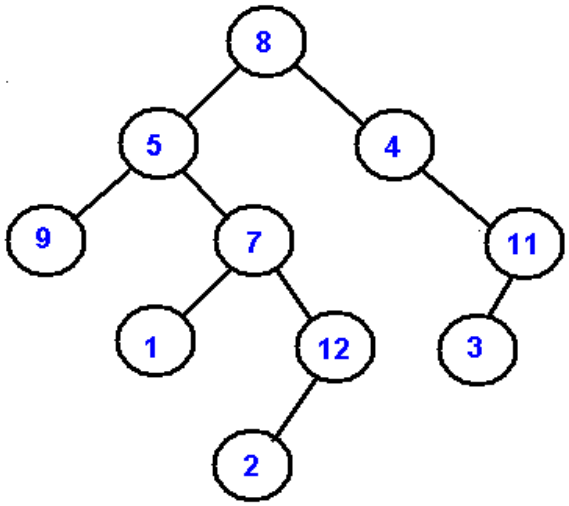
- depth-first traversal
- breadth-first traversal

There are three different types of depth-first traversals:

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

  As an example consider the following tree and its four traversals:

  PreOrder - 8, 5, 9, 7, 1, 12, 2, 4, 11, 3
  InOrder - 9, 5, 1, 7, 2, 12, 8, 4, 3, 11
  PostOrder - 9, 1, 2, 12, 7, 5, 3, 11, 4, 8
  LevelOrder - 8, 5, 4, 9, 7, 11, 1, 12, 3, 2



There is only one kind of breadth-first traversal--the level order traversal. This traversal visits nodes by levels from top to bottom and from left to right.

ALGORITHM:

Insert Node:

Insertion operation means inserting a new node into any position of the binary tree. The insertion operation can take place in any one of the following two positions

1. At the external node
2. At internal node

Here , Algorithm of only insertion of node as a external node is considered. The insertion operation involves the following steps

1. Search for a node after which an insertion has to be made
2. Establish a link for the new node

Steps:

1. Accept the **KEY** (or data item) which has to be searched in a BT after which the new data item has to be placed
2. Accept the item to be inserted, **ITEM**
3. Start from the ROOT node, search KEY in the binary tree. The address of the node with KEY as the data will be in the **ptr.**
4. Check whether ptr contains NULL or not. If ptr is NULL then Search is Unsuccessful.
5. Otherwise

    a. Node with data item **KEY** is there in the binary tree (i.e in **ptr**)
    b. Now check whether the left child of **ptr** is empty or the right child of **ptr** is empty. If the left child of the ptr is empty then do steps (i) to (v)

        (i) Allocate memory space for the new node (say **new)**
        (ii) Place the item to be inserted (ITEM) in the new node.
        (iii) Make the left child and right child fields of the new node to be NULL
        (iv) Make the new node as left child of **ptr**.

    c. Otherwise check whether the right child of ptr is empty. If the right child of the ptr is empty then do steps (i) to (v)

        (i) Allocate memory space for the new node (say **new)**
        (ii) Place the item to be inserted (ITEM) in the new node.
        (iii) Make the left child and right child fields of the new node to be NULL
        (iv) Make the new node as right child of **ptr**.

    d. Otherwise Print “Insertion is not possible. The KEY node already has a child

6. Stop

Display:

In order to display content of each node in exactly once, a BT can be traversed in either Pre-Order, Post- Order, In-Order.
Here Pre-order Traversal is considered

1. Visit the root node R First
2. Then visit the left sub tree of R in Pre-Order fashion
3. Finally, visit the right sub tree of R in Pre-Order fashion

Depth of a Tree:

This function finds the depth of the linked binary tree.

1. Start from the ROOT node and store the address of the root node in **ptr.**
2. Check whether **ptr** contains NULL or not. If **ptr** is NULL then tree is empty.
3. Otherwise

    a. Get the height of left sub tree, say leftHeight.
    b. Get the height of right sub tree, say rightHeight.
    c. Take the Max(leftHeight, rightHeight) and add 1 for the root and return.
    d. Call recursively.

Display Leaf-Nodes:

This function displays all the leaf nodes in the linked binary tree.

1. Start from the ROOT node and store the address of the root node in **ptr.**
2. Check whether **ptr** contains NULL or not. If **ptr** is NULL then tree is empty.
3. Otherwise

    a. Check whether the left child and right child of **ptr** are empty or not. If the left child and right child of **ptr** are NULL then display the content of the node.
    b. Traverse the left subtrees and the right subtrees and repeat step a for all the nodes.

4. Stop

Create a Copy of a Tree:

This function will make a copy of the linked binary tree. The function should allocate memory for necessary nodes and copy respective contents in it.

1. Start from the ROOT node. The address of the root node will be in the **ptr.** Let newroot be the root of the new tree after the copy.

2. Check whether **ptr** contains NULL or not. If **ptr** is NULL then newroot = NULL
3. Otherwise

    (i) Allocate memory space for the new node
    (ii) Copy the data from current node from the old tree to node in the new tree.
    (iii) Traverse each node in the left subtrees from the old tree and repeat step (i) and (ii).
    (iv) Traverse each node in the right subtrees from the old tree and repeat step (i) and (ii).

4. Stop

INPUT:

Test Case	O/P
Tree Empty	Display message “Tree Empty”
Insertion is not possible.	The KEY node already has a child
Binary Tree	

OUTPUT:

Height of a Binary Tree

FAQ:

1. What is a Binary Tree?
2. List the different types of Binary Tree?
3. List the different methods of visiting the nodes of a binary tree?
4. What are the various operations that can be performed on Binary Tree?
5. List the different methods of converting general tree to Binary Tree.

4. Expression tree Traversals

AIM: Construct and expression tree from postfix/prefix expression and perform recursive and non- recursive In-order, pre-order and post-order traversals.

OBJECTIVE:

1. Understand the concept of expression tree and binary tree.
2. Understand the different type of traversals (recursive & non-recursive).

THEORY:

1. Definition of an expression tree with diagram.

Algebraic expressions such as a/b + (c-d) e

The terminal nodes (leaves) of an expression tree are the variables or constants in the expression (a, b, c, d, and e). The non-terminal nodes of an expression tree are the operators (+, -, ×, and ÷). Notice that the parentheses which appear in Equation do not appear in the tree. Nevertheless, the tree representation has captured the intent of the parentheses since the subtraction is lower in the tree than the multiplication.

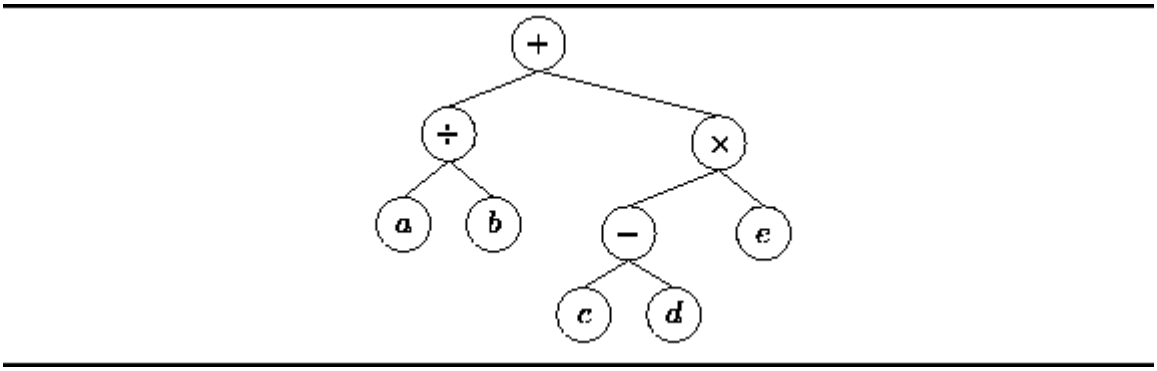


Figure: Tree representing the expression a/b+(c-d)e.

2. Show the different type of traversals with example

To traverse a non-empty binary tree in **preorder**,

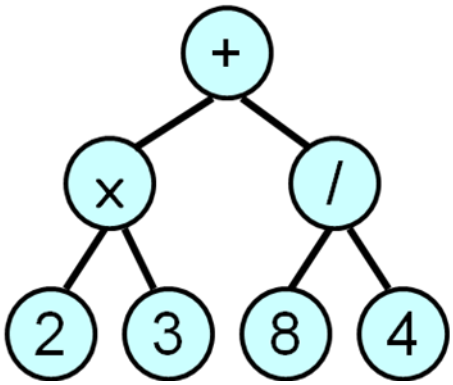
1. Visit the root.
2. Traverse the left subtree.
3. Traverse the right subtree.

To traverse a non-empty binary tree in **inorder**:

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

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

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



- **Pre-order (prefix)**
  + × 2 3 / 8 4
- **In-order (infix)**
  2 × 3 + 8 / 4
- **Post-order (postfix)**
  2 3 × 8 4 / +

ALGORITHM:

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

Create Expression Tree:

CreateTree()

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

Step 1: Top = -1 , I = 0;
Step 2: Do Steps 3,4,5,6 While String[I] != NULL
Step 3: Create Node of type of structure
Step 4: Node->Data=String[I];
Step 5: If isalnum(String[I])
Then Stack[Top++] = Node;
Else
Node->Right = Stack [--Top];
Node->Left = Stack[--Top];
Stack[Top++] = Node;
Step 6: Increment I;
Step 7: Root = Stack[0];
Step 8: Return Root

Inorder Traversal Recursive :

Tree is pointer of type structure.
InorderR(Tree)
Step 1: If Tree != NULL

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

Postorder Traversal Recursive:

Tree is pointer of type structure.
PostorderR(Tree)
Step 1: If Tree != NULL
Step 2: PostorderR(Tree->Left)
Step 3: PostorderR(Tree->Right)
Step 4: Print Tree->Data

Preorder Traversal Recursive:

Tree is pointer of type structure.
PreorderR(Tree)
Step 1: If Tree != NULL
Step 2: Print Tree->Data
Step 3: PreorderR(Tree->Left)
Step 4: PreorderR(Tree->Right)

Postorder Traversal Nonrecursive :

NonR_Postorder(Tree)
Tree, Temp is pointer of type structure. Stack is pointer array of type structure.Top variable of type integer.
Step 1: Temp = Tree // current pointer pointing to root
Step 2: if Temp != NULL then push current pointer along with its initial flag value 'L' on to the stack and traverse on the left side.
Step 3: Otherwise if the stack is not empty then pop an address from stack along with its flag value.
Step 4: For the current pointer if the flag value is 'L' then change it to 'R' and push current pointer along with its flag value 'R' on to the stack and traverse on the right side.
Step 5: otherwise, For the current pointer if the flag value is 'R' then display the element and make the current pointer NULL (i.e.temp=NULL)
Step 6: repeat steps 2, 3, 4, 5 until the stack becomes empty.

Preorder Traversal Nonrecursive :

NonR_Preorder(Tree)
Tree, Temp is pointer of type structure. Stack is pointer array of type structure. Top variable of type integer.
Step 1: Temp = Tree
Step 2: Do Steps3,4,5,6,7,& 8 While Temp != NULL And Stack is not Empty
Step 3: Do Steps 4,5&6 While Temp != NULL
Step 4: Print Temp->Data
Step 5: Stack[++ Top] = Temp //Push Element
Step 6: Temp = Temp->Left
Step 7: Temp = Stack [Top --] //Pop Element
Step 8: Temp = Temp->Right

Inorder Traversal Nonrecursive :

NonR_Inorder(Tree)
Tree, Temp is pointer of type structure. Stack is pointer array of type structure. Top variable of type integer.
Step 1: Temp = Tree
Step 2: Do Steps3,4,5,6,7,&8 While Temp != NULL And Stack is not Empty
Step 3: Do Steps 4,5 While Temp != NULL
Step 4: Stack[++ Top] = Temp;//Push Element
Step 5: Temp = Temp->Left
Step 6: Temp = Stack[Top --]//Pop Element
Step 7: Print Temp->Data
Step 8: Temp = Temp->Right

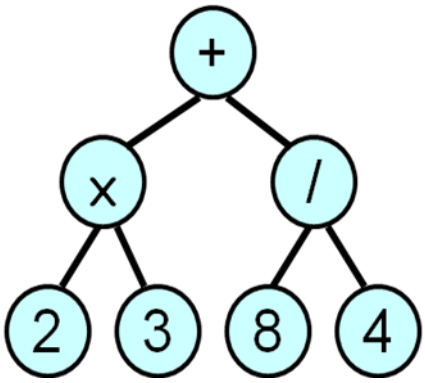
INPUT:

Postfix Expression: **2 3 × 8 4 / +**

OUTPUT:

Display result of each operation with error checking.

Expression tree



FAQS:

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

5. Operations on Binary search tree

AIM: Implement binary search tree and perform following operations:
a. Insert
b. Delete
c. Search
d. Mirror image
e. Display
f. Display level wise

OBJECTIVE:

1. To understand the concept of binary search tree as a data structure.
2. Applications of BST.

THEORY:

1. Definition of binary search tree

A binary tree in which each internal node x stores an element such that the element stored in the left subtree of x are less than or equal to x and elements stored in the right subtree of x are greater than or equal to x. This is called binary-search-tree property.

2. Illustrate the above operations graphically.

Searching

Searching a binary tree for a specific value can be a recursive or iterative process. This explanation covers a recursive method.We begin by examining the root node. If the tree is null, the value we are searching for does not exist in the tree. Otherwise, if the value equals the root, the search is successful. If the value is less than the root, search the left subtree. Similarly, if it is greater than the root, search the right subtree. This process is repeated until the value is found or the indicated subtree is null. If the searched value is not found before a null subtree is reached, then the item must not be present in the tree.

Insertion

Insertion begins as a search would begin; if the root is not equal to the value, we search the left or right subtrees as before. Eventually, we will reach an external node and add the value as its right or left child, depending on the node's value. In other words, we examine the root and recursively insert the new node to the left subtree if the new value is less than the root, or the right subtree if the new value is greater than or equal to the root.

Deletion

There are three possible cases to consider:

- **Deleting a leaf (node with no children):** Deleting a leaf is easy, as we can simply remove it from the tree.
- **Deleting a node with one child:** Remove the node and replace it with its child.

- **Deleting a node with two children:** Call the node to be deleted N. Do not delete N. Instead, choose either its in-order successor node or its in-order predecessor node, R. Replace the value of N with the value of R, then delete R.

As with all binary trees, a node's in-order successor is the left-most child of its right subtree, and a node's in-order predecessor is the right-most child of its left subtree. In either case, this node will have zero or one children. Delete it according to one of the two simpler cases above.

ALGORITHM:

Define structure for Binary Tree (Mnemonics, Left Pointer, Right Pointer)

Insert Node:

Insert (Root, Node)
Root is a variable of type structure, represent Root of the Tree. Node is a variable of type structure ,represent new Node to insert in a tree.
Step 1: Repeat Steps 2,3 & 4 Until Node do not insert at appropriate position.
Step 2: If Node Data is less that Root Data & Root Left Tree is NULL
Then insert Node to Left.
Else Move Root to Left
Step 3: Else If Node Data is Greater that equal that Root Data & Root Right Tree is NULL
Then insert Node to Right.
Else Move Root to Right.
Step 4: Stop.

Search Node:

Search (Root, Mnemonics)
Root is a variable of type structure, represent Root of the Tree. Mnemonics is array of character. This function search Mnemonics in a Tree.
Step 1: Repeat Steps 2,3 & 4 Until Mnemonics Not find && Root != NULL
Step 2: If Mnemonics Equal to Root Data
Then print message Mnemonics present.
Step 3: Else If Mnemonics Greater that Equal that Root Data
Then Move Root to Right.
Step 4: Else Move Root to Left.
Step 5: Stop.

Delete Node:

Dsearch(Root, Mnemonics)
Root is a variable of type structure ,represent Root of the Tree. Mnemonics is array of character. Stack is an pointer array of type structure. PTree(Parent of Searched Node),Tree(Node to be deleted), RTree(Pointg to Right Tree),Temp are pointer variable of type structure;
Step 1: Search Mnemonics in a Binary Tree
Step 2: If Root = = NULL Then Tree is NULL
Step 3: Else //Delete Leaf Node

If Tree->Left = = NULL && Tree->Right = = NULL
Then a) If Root = = Tree Then Root = NULL;
b) If Tree is a Right Child PTree->Right=NULL;
Else PTree->Left=NULL;
Step 4: Else // delete Node with Left and Right children
If Tree->Left != NULL && Tree->Right != NULL
Then a) RTree=Temp=Tree->Right;
b)Do steps i && ii while Temp->Left !=NULL
i) RTree=Temp;
ii) Temp=Temp->Left;
c)RTree->Left=Temp->Right;
d)If Root == Tree//Delete Root Node
Root=Temp;
e)If Tree is a Right Child PTree->Right=Temp;
Else PTree->Left=Temp;
f)Temp->Left=Tree->Left;
g)If RTree!=Temp
Then Temp->Right = Tree->Right;
Step 5: Else //with Right child
If Tree->Right!= NULL
Then a) If Root==Tree Root = Root->Right;
b) If Tree is a Right Child PTree->Right=Tree->Right;
Else PTree->Left=Tree->Left;
Step 6: Else //with Left child
If Tree->Left != NULL
Then a) If Root==Tree Root = Root->Left;
b) If Tree is a Right Child PTree->Right=Tree->Left;
Else PTree->Left=Tree->Left;
Step 7: Stop.

Depth First Search:

Root is a variable of type structure ,represent Root of the Tree.
Stack is an pointer array of type structure. Top variable of type integer.
DFS(Root)
Step 1: Repeat Steps 2,3,4,5,6 Until Stack is Empty
Step 2: print Root Data
Step 3: If Root->Right != NULL//Root Has a Right SubTree
Then Stack[Top++] = Tree->Right;//Push Right Tree into Stack
Step 4: Root = Root ->Left;//Move to Left
Step 5: If Root == NULL
Step 6: Root = Stack[--Top];//Pop Node from Staacl
Step 7: Stop.

Breath First Searc(Levelwise Display):

Root is a variable of type structure ,represent Root of the Tree.
Queue is an pointer array of type structure. Front & Rear variable of type integer.
BFS(Root)
Step 1:If Root == NULL Then Empty Tree;

Step 2: Else Queue[0] = Root;// insert Root of the Tree in a Queue
Step 3: Repeat Steps 4,5,6 & 7 Until Queue is Empty
Step 4: Tree=Queue[Front++]; //Remove Node From Queue
Step 5: print Root Data
Step 6: If Root->Left != NULL
Then Queue[++Rear] = Tree->Left;//insert Left Subtree in a Queue
Step 7: If Root->Right != NULL
Then Queue[++Rear] = Root->Right; //insert Left Subtree in a Queue
Else if Root->Right == NULL And Root->Left == NULL
Leaf++;//Number of Leaf Nodes
Step 8: Stop.

Mirror Image:

Root is a variable of type structure ,represent Root of the Tree.
Queue is an pointer array of type structure. Front & Rear variable of type integer.
Mirror(Root)
Step 1: Queue[0]=Root;//Insert Root Node in a Queue
Step 2: Repeat Steps 3,4,5,6,7 & 8 Until Queue is Empty
Step 3: Root = Queue[Front++];
Step 4: Temp1 = Root->Left;
Step 5: Root->Left = Root->Right;
Step 6: Root->Right = Temp1;
Step 7: If Root->Left != NULL
Then Queue[Rear++] = Tree->Left;//insert Left SubTree
Step 8: If Root->Right != NULL
Then Queue[Rear++] = Root->Right;//insert Right SubTree
Step 9: Stop.

INPUT:

Accept the nodes from the user like: add, mult, div, sub

OUTPUT:

Display result of each operation with error checking.

FAQS:

1. What is Binary search tree?
2. What are the members of structure of tree & what is the size of structure?
3. What are rules to construct binary search tree?
4. How general tree is converted into binary tree?
5. What is binary threaded tree?
6. What is use of thread in traversal?

6. BFS and DFS

Aim: Consider a friends' network on face book social web site. Model it as a graph to represent each node as a user and a link to represent the fried relationship between them. Store data such as date of birth, number of comments for each user.
1. Find who is having maximum friends
2. Find who has post maximum and minimum comments
3. Find users having birthday in this month

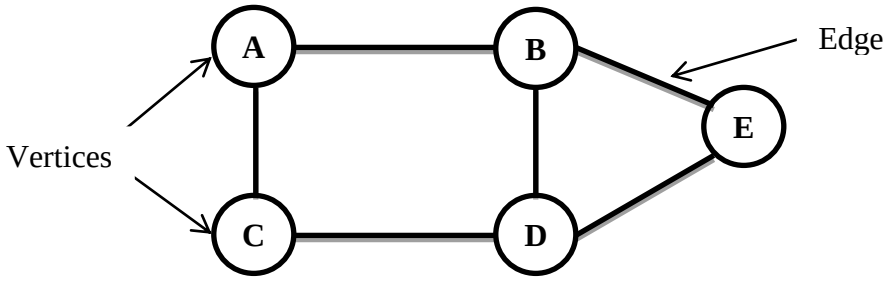
Hint: (Use adjacency list representation and perform DFS and BFS traversals)

Objective:
- To study Graph theory.
- To study different graph traversal methods.
- To understand the real time applications of graph theory.

Theory:

1) What is Graph? Explain different graph traversal methods.

GRAPH:
A graph G consist of two sets V & E where, V is finite non-empty set of vertices & E is set of pair of vertices called edges. V (G) represents set of edges in graph G.
e.g.:
This graph G can be defined as G = (V, E) Where V= {A,B,C,D,E} and E= {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



GRAPH PRESENTATION:
Graph can be represented in two ways:
- Adjacency Matrix
- Adjacency List

TRAVERSAL OF A GRAPH:
Let G= (V, E) be an undirected graph and a vertex V in V (G). There are two ways to visit all the vertices of graph:
- Depth First Search
- Breadth First Search

Depth First Search (DFS):
A depth-first search (DFS) is an algorithm for traversing a finite graph. DFS visits the child vertices before visiting the sibling vertices; that is, it traverses the depth of any particular path before exploring its breadth. A stack is generally used when implementing the algorithm.
When there are no adjacent, on visited nodes, then we proceed backwards (backtrack), wherein repeat the process.

Breadth First Search (BFS): BFS algorithm traverses a graph in a breadth ward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.
A breadth-first search (BFS) is another technique for traversing a finite graph. BFS visits the neighbor vertices before visiting the child vertices, and a queue is used in the search process. This algorithm is often used to find the shortest path from one vertex to another.

ALGORITHM:

The DFS algorithm:

```
DFS(G,v)   ( v is the vertex where the search starts )
      Stack S := {};   ( start with an empty stack )
      for each vertex u, set visited[u] := false;
      push S, v;
      while (S is not empty) do
        u := pop S;
        if (not visited[u]) then
          visited[u] := true;
          for each unvisited neighbour w of u
            push S, w;
        end if
      end while
    END DFS()
```

The BFS algorithm:

1. Accept the number of vertices from the user, say n.
2. Create a list having n nodes. This list is called as a header list. It will be connected by the down pointer whereas the adjacency list will be connected by the next pointer. Remember that these two lists will follow different structures.
3. Initialize the visited array v to zeroes.
4. Accept the graph.
    1. Accept an edge say i, j.
    2. Search in the header list for vertex i, and in the adjacency list of that vertex i, attach a node of vertex j.
    3. Search in the header list for vertex j, and in the adjacency list of that vertex j, attach a node of vertex i.
    4. If more edges then goto step (a).
5. Accept the starting vertex say i.
6. Push i to the queue, and mark it as visited, i.e. v[i] =1.

7. Pop a vertex from queue, say i.
8. Print i.
9. Move in the adjacency list of i^{th} vertex, and for every node say j which is not visited, push it to the queue and mark it as visited, i.e. v[j]=1.
10. If queue is not empty repeat from step 7.
11. Now check whether all vertices are visited.
    a. Initialize j=0 and flag = -1.
    b. If j^{th} vertex is not visited, set flag to j.
    c. Increment j, and if number of vertices is not over, repeat from step b.
12. If flag! =1, the graph is not a connected graph. Otherwise it is a connected graph.
13. Stop.

Advantages of DFS

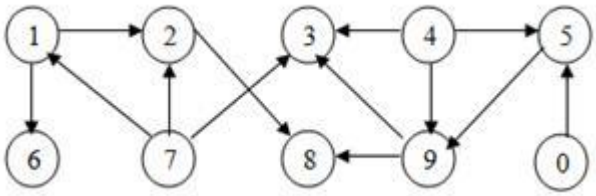
- Requires less memory than BFS since only need to remember the current path.
- If lucky, can find a solution without examining much of the state space.
- with cycle-checking, looping can be avoided

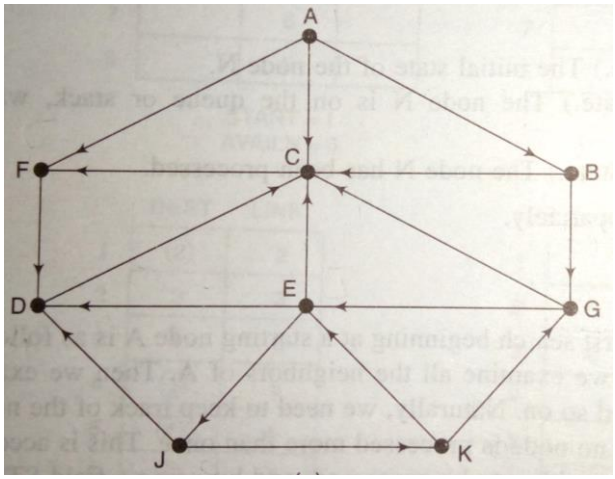
Advantages of BFS

- Guaranteed to find a solution if one exists –in addition, finds optimal (shortest) solution first
- It will not get lost in a blind alley (i.e., does not require backtracking or cycle checking)
- Can add cycle checking with very little cost

Conclusion

BFS and DFS algorithm is implemented for graph traversal.

INPUT	**OUTPUT**	Remark
	DFS – 0 5 9 8 3 BFS - 0 5 9 3 8	Starting vertex 0. The output may change depending on the adjacency list



DFS: A B G E J K D C F

BFS: A F C B D G E J K

Starting Vertex A. The output may change depending on the adjacency list

FAQS:

1. What's the difference between DFS and BFS?
2. When should we use BFS instead of DFS, and vice versa?
3. How can I do a BFS and DFS in a matrix and a List?
4. Time and Space Complexity of these algorithms

7. Implementation of Kruskal's algorithms

AIM: Represent any real world graph using adjacency list /adjacency matrix find minimum spanning tree using Kruskal's algorithm.

OBJECTIVE:

1. Learn the concepts of graph as a data structure and their applications in everyday life.
2. Understand graph representation (adjacency matrix, adjacency list, adjacency multi list)

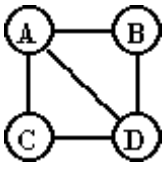
THEORY:

1. What is a graph? Various terminologies and its applications. Explain in brief.

- Definition : A graph is a triple G = (V,E, φ) where
  - V is a finite set, called the vertices of G,
  - E is a finite set, called the edges of G, and
  - φ is a function with domain E and codomain P2(V).
- **Loops**: A loop is an edge that connects a vertex to itself.
- **Degrees of vertices**: Let G = (V,E, φ) be a graph and v ∈ V a vertex. Define the degree of v, d(v) to be the number of e ∈ E such that v ∈ φ(e); i.e., e is Incident on v.
- **Directed graph:** A directed graph (or digraph) is a triple D = (V,E, φ) where V and E are finite sets and φ is a function with domain E and codomain V × V . We call E the set of edges of the digraph D and call V the set of vertices of D.
- **Path**: Let G = (V,E, φ) be a graph.
  Let e1, e2, . . . , en−1 be a sequence of elements of E (edges of G) for which there is a sequence a1, a2, . . . , an of distinct elements of V (vertices of G) such that φ(ei) = {ai, ai+1} for i = 1, 2, . . . , n − 1. The sequence of edges e1, e2, . . . , en−1 is called a path in G. The sequence of vertices a1, a2, . . . , an is called the vertex sequence of the path.
- **Circuit and Cycle**: Let G = (V, E, φ) be a graph and let e1, . . . , en be a trail with vertex sequence a1, . . . , an, a1. (It returns to its starting point.) The subgraph G′ of G induced by the set of edges {e1, . . . , en} is called a circuit of G. The length of the circuit is n.

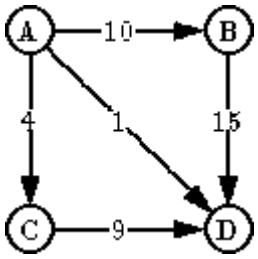
2. Different representations of graph.

- **Adjacency matrix:** Graphs $G = (V, E)$ can be represented by adjacency matrices $G\ [v_1..v_{|V|}, v_1..v_{|V|}]$, where the rows and columns are indexed by the nodes, and the entries $G\ [v_i, v_j]$ represent the edges. In the case of unlabeled graphs, the entries are just boolean values.



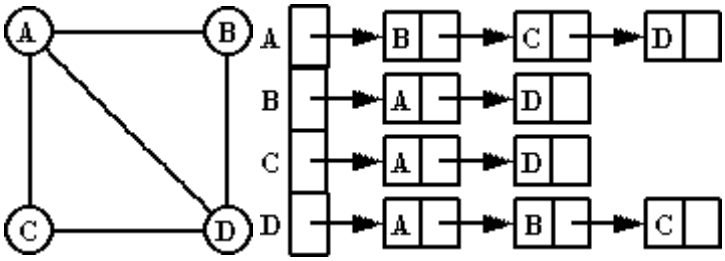
	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	1
D	1	1	1	0

In case of labeled graphs, the labels themselves may be introduced into the entries.



	A	B	C	D
A	∞	10	4	1
B	∞	∞	∞	15
C	∞	∞	∞	9
D	∞	∞	∞	∞

- **Adjacency List:** A representation of the graph consisting of a list of nodes, with each node containing a list of its neighboring nodes.



Kruskal's Algorithm:

Step 1: Choose the arc of least weight.
Step 2: Choose from those arcs remaining the arc of least weight which does not form a cycle with already chosen arcs. (If there are several such arcs, choose one arbitrarily.)
Step 3: Repeat Step 2 until *n – 1* arcs have been chosen.

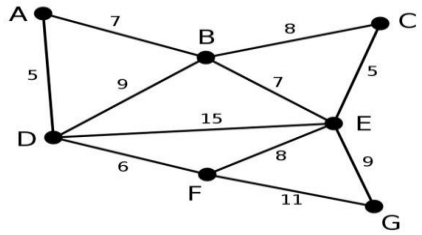
INPUT:

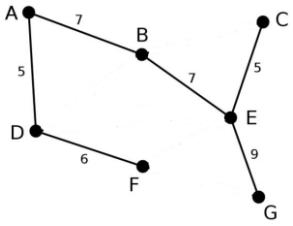
Enter the no. of nodes in graph. Create the adjacency LIST

OUTPUT:

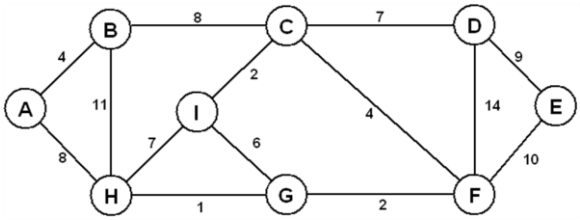
Display result of each operation with error checking.

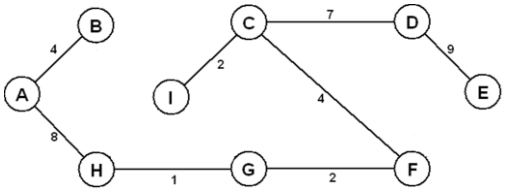
INPUT **OUTPUT** Remark





Cost of MST - 39





Cost of MST - 37

FAQS:

1. What is graph?
2. Application of Prim's & Kruskal's algorithm.
3. What are the traversal techniques?
4. What are the graph representation techniques?
5. What is adjacency Matrix?
6. What is adjacency list?
7. What is adjacency Multi-list?

8. Implementation of Dijikstra's algorithm

AIM: Represent a given graph using adjacency matrix /adjacency list and find the shortest path using Dijkstra's algorithm (single source all destination).

OBJECTIVE:

1. To understand the application of Dijkstra's algorithm

THEORY:

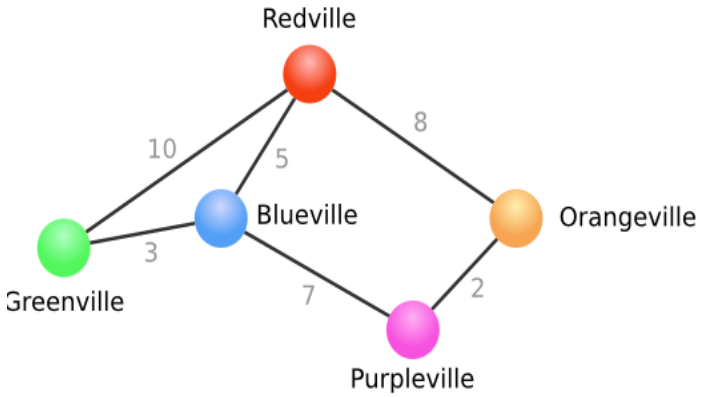
1. Explain in brief with examples how to find the shortest path using Dijkstra's algorithm.

Definition of Dijkstra's Shortest Path

1. To find the shortest path between points, the weight or length of a path is calculated as the sum of the weights of the edges in the path.
2. A path is a shortest if there is no path from x to y with lower weight.
3. Dijkstra's algorithm finds the shortest path from x to y in order of increasing distance from x. That is, it chooses the first minimum edge, stores this value and adds the next minimum value from the next edge it selects.
4. It starts out at one vertex and branches out by selecting certain edges that lead to new vertices.
5. It is similar to the minimum spanning tree algorithm, in that it is "greedy", always choosing the closest edge in hopes of an optimal solution.

Example:

It is easiest to think of the geographical distances, with the vertices being places, such as cities.



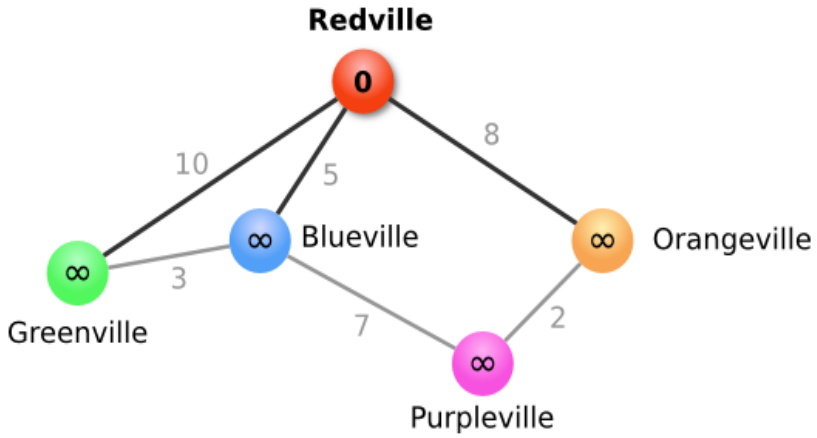
Imagine you live in Redville, and would like to know the shortest way to get to the surrounding towns: Greenville, Blueville, Orangeville, and Purpleville. You would be confronted with problems like: Is it faster to go through Orangeville or Blueville to get to Purpleville? Is it faster to take a direct route to Greenville, or to take the route that goes through Blueville? As long as you knew the distances of roads going directly from one city

to another, Dijkstra's algorithm would be able to tell you what the best route for each of the nearby towns would be.

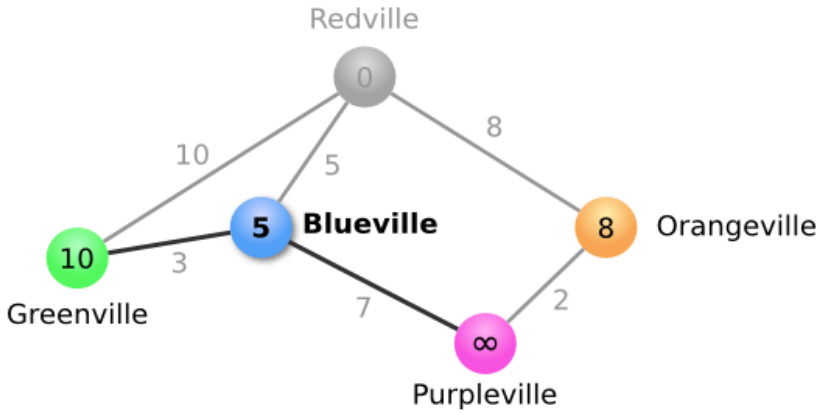
- Begin with the source node (city), and call this the current node. Set its value to 0. Set the value of all other nodes to infinity. Mark all nodes as unvisited.
- For each unvisited node that is adjacent to the current node (i.e. a city there is a direct route to from the present city), do the following. If the value of the current node plus the value of the edge is less than the value of the adjacent node, change the value of the adjacent node to this value. Otherwise leave the value as is.
- Set the current node to visited. If there are still some unvisited nodes, set the unvisited node with the smallest value as the new current node, and go to step 2. If there are no unvisited nodes, then we are done.

In other words, we start by figuring out the distance from our hometown to all of the towns we have a direct route to. Then we go through each town, and see if there is a quicker route through it to any of the towns it has a direct route to. If so, we remember this as our current best route.

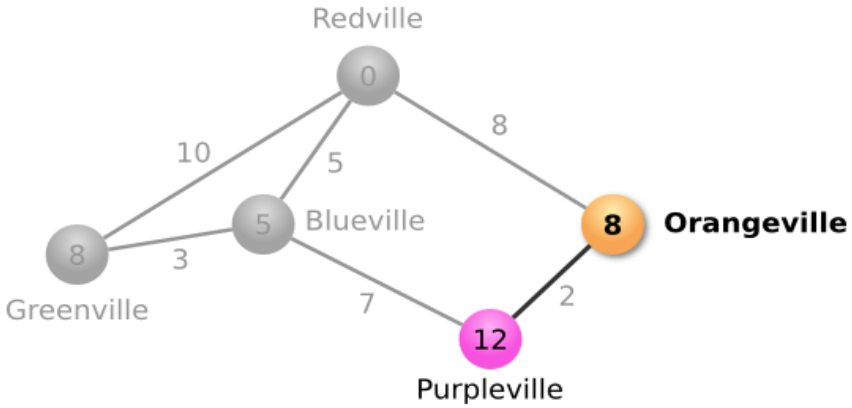
Step I:
We set Redville as our current node. We give it a value of 0, since it doesn't cost anything to get to it from our starting point. We assign everything else a value of infinity, since we don't yet know of a way to get to them.



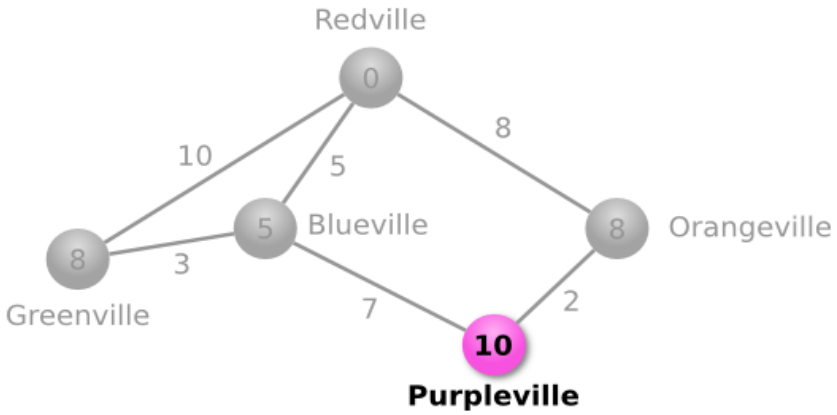
Step II:
Next, we look at the unvisited cities our current node is adjacent to. This means Greenville, Blueville and Orangeville. We check whether the value of the connecting edge, plus the value of our current node, is less than the value of the adjacent node, and if so we change the value. In this case, for all three of the adjacent nodes we should be changing the value, since all of the adjacent nodes have the value infinity. We change the value to the value of the current node (zero) plus the value of the connecting edge (10 for Greenville, 5 for Blueville, 8 for Orangeville). We now mark Redville as visited, and set Blueville as our current node since it has the lowest value of all unvisited nodes.



Step III:
The unvisited nodes adjacent to Blueville, our current node, are Purpleville and Greenville. So we want to see if the value of either of those cities is less than the value of Blueville plus the value of the connecting edge. The value of Blueville plus the value of the road to Greenvile is 5 + 3 = 8. This is less than the current value of Greenville (10), so it is shorter to go through Blueville to get to Greenville. We change the value of Greenville to 8, showing we can get there with a cost of 8. For Purpleville, 5 + 7 = 12, which is less than Purpleville's current value of infinity, so we change its value as well? We mark Blueville as visited. There are now two unvisited nodes with the lowest value (both Orangeville and Greenville have value 8). We can arbitrarily choose Greenville to be our next current node. However, there are no unvisited nodes adjacent to Greenville! We can mark it as visited without making any other changes, and make Orangeville our next current node.



Step IV:
There is only one unvisited node adjacent to Orangeville. If we check the values, Orangeville plus the connecting road is 8 + 2 = 10, Purpleville's value is 12, and so we change Purpleville's value to 10. We mark Orangeville as visited, and Purpleville is our last unvisited node, so we make it our current node. There are no unvisited nodes adjacent to Purpleville, so we're done!



All above steps can be simply put in a tabular form like this:

Current	Visited	Red	Green	Blue	Orange	Purple	Description
Red		0	Infinity	Infinity	Infinity	Infinity	Initialize Red as current, set initial values
Red		0	10	5	8	Infinity	Change values for Green, Blue, Orange
Blue	Red	0	10	5	8	Infinity	Set Red as visited, Blue as current
Blue	Red	0	8	5	8	12	Change value for Purple
Green	Red, Blue	0	8	5	8	12	Set Blue as visited, Green as current
Orange	Red, Blue, Green	0	8	5	8	12	Set Green as visited, Orange as current
Orange	Red, Blue, Green	0	8	5	8	10	Change value for Purple
Purple	Red, Blue, Green, Orange	0	8	5	8	10	Set Orange as visited, Purple as current
	Red, Blue, Green, Orange, Purple	0	8	5	8	10	Set Purple as visited

ALGORITHM:

College Area represented by Graph.
A graph G with N nodes is maintained by its adjacency matrix Cost. Dijkstra's algorithm find shortest path matrix D of Graph G.
Staring Node is 1.

Step 1: Repeat Step 2 for I = 1 to N
D[I] = Cost[1][I].
Step 2: Repeat Steps 3 & 4 for I = 1 to N
Step 3: Repeat Steps 4 for J = 1 to N
Step 4: If D[J] > D[I] + D[I][J]

Then D[J] = D[I] + D[I][J]

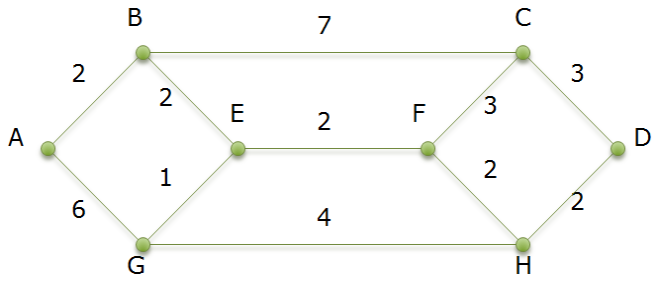
Step 5: Stop.

INPUT:

The graph in the form of edges, nodes and corresponding weights, the source node and destination node.

OUTPUT:

The shortest path and length of the shortest path

INPUT	**OUTPUT**	Remark
	Shortest Path is A-B-E-F-H-D Cost - 10	Consider Source Vertex as node A and Destination Vertex as node D

FAQs:

1. What is shortest path?
2. What are the graph representation techniques?
3. What is adjacency Matrix?
4. What is adjacency list?
5. What is adjacency Multi-list?

9. Implementation of Hash Table

AIM: Store data of students with telephone no and name in the structure using hashing function for telephone number and implement chaining with and without replacement.

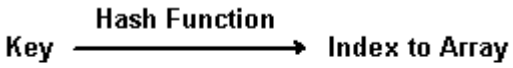
OBJECTIVE:

1. Understand the concept and applications of hashing.
2. Understand the use of hash table in files.

THEORY:

1) What is a hash table and hash function?

The organization of the file and the order in which the keys are inserted affect the number of keys that must be inspected before obtaining the desired one. Optimally we would like to have a table organization and search technique in which there are no unnecessary comparisons. Hash tables are good for doing a quick search on things. The most efficient way to organize such a table is an array. If the record keys are integers, the key themselves can serve as indicates to the array. For instance if we have an array full of data (say 100 items). If we knew the position that a specific item is stored in an array, then we could quickly access it. For instance, we just happen to know that the item we want it is at position 3; I can apply: myitem=myarray[3]; This is where hashing comes in handy. Given some key, we can apply a hash function to it to find an index or position that we want to access.



A function that transfers a key into a table index is called hash function. If *h* is a hash function and *key* is a key , *h(key)* is called the hash of key and is the index at which a record with the key *key* should be placed

1) **Characteristics of a good hash function**.

There are four main characteristics of a good hash function:

- The hash value is fully determined by the data being hashed.
- The hash function uses all the input data.
- The hash function "uniformly" distributes the data across the entire set of possible
- hash values.
- The hash function generates very different hash values for similar strings.

2) Explain in brief

a. Collision

A **collision** or **clash** is a situation that occurs when two distinct pieces of data have the same hash value. Collisions are unavoidable whenever members of a very large set (such as all possible person names, or all possible computer files) are mapped to a relatively short bit string. The problem of collision can not be eliminated but it can be minimized using hashing function.

b. Overflow

When a Hash Table becomes full then the method suggested here is to define a new table of length greater than old hash table length and then to serially re-hash into the new table the elements already defined in the original table. Subsequent entries and accessing will be made on the new table.

c. Chaining

One simple scheme is to chain all collisions in lists attached to the appropriate slot. This allows an unlimited number of collisions to be handled and doesn't require *a priori* knowledge of how many elements are contained in the collection. The tradeoff is the same as with linked lists versus array implementations of collections: linked list overhead in space and, to a lesser extent, in time.

Hashing with chaining is an application of linked lists and gives an approach to collision resolution. In hashing with chaining, the hash table contains linked lists of elements or pointers to elements (as in Figure). The lists are referred to as chains, and the technique is called chaining. This is a common technique where a straightforward implementation is desired and maximum efficiency isn't required. Each linked list contains all the elements whose keys hash to the same index.

Using chains minimizes search by dividing the set to be searched into lots of smaller pieces. There's nothing inherently wrong with linear search with small sequences; it's just that it gets slower as the sequences to be searched get longer. In this approach to resolving collisions, each sequence of elements whose keys hash to the same value will stay relatively short, so linear search is adequate.

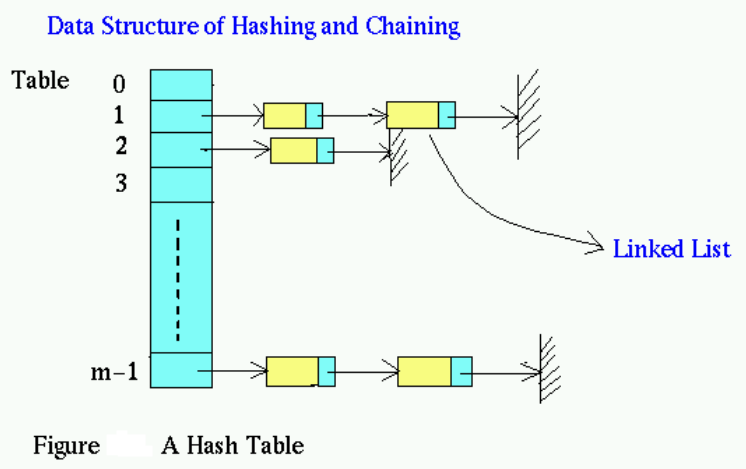


Figure A Hash Table

d. Linear probing / open addressing

Linear probing is a scheme in computer programming for resolving hash collisions of values of hash functions by sequentially searching the hash table for a free location. In that case, since we cannot insert n at h(n), which we now call h for simplicity, we try the next slot at h+1. If this is vacant, we insert n here. Otherwise, we try h+2 and so on. The sequence of locations that we probe, as far as necessary, is therefore

h, h+1, h+2, h+3, ...

The only qualification is that we must only probe *within* the table array. When this sequence would run off the end of the array, we wrap around back to the beginning at 0. The sequence of probes is therefore at

h+i % table.length

for i = 0,1,2,3,...

When we examine whether a given item d is present, we apply the same procedure. We simply search for it by calculating h(d) % table.length and then, if necessary, probe successive locations until we either find the item d, if it is present, or we find a null if it is absent.

2. Different hash function methods used to organize a hash table.

Division Method

Perhaps the simplest of all the methods of hashing an integer x is to divide x by M and then to use the remainder modulo M. This is called the division method of hashing . In this case, the hash function is $h(x) = x \bmod M.$

Generally, this approach is quite good for just aout any value of M.

In this case, M is a constant. However, an advantage of the division method is that M need not be a compile-time constant--its value can be determined at run time. In any event, the running time of this implementation is clearly a constant.

A potential disadvantage of the division method is due to the property that consecutive keys map to consecutive hash values:

$$\begin{aligned} h(i) &= i \\ h(i+1) &= i+1 \pmod M \\ h(i+2) &= i+2 \pmod M \\ &\vdots \end{aligned}$$

While this ensures that consecutive keys do not collide, it does mean that consecutive array locations will be occupied. We will see that in certain implementations this can lead to degradation in performance.

Middle Square Method

Since integer division is usually slower than integer multiplication, by avoiding division we can potentially improve the running time of the hashing algorithm. We can avoid division by making use of the fact that a computer does finite-precision integer arithmetic. E.g., all arithmetic is done modulo W where $\mathbf{W=2^n}$ is a power of two such that w is the word size of the computer.

The middle-square hashing method works as follows. First, we assume that M is a power of two, say $\mathbf{M=2^k}$ for some **k>1**. Then, to hash an integer **x**, we use the following hash function:

$$h(x) = \left\lfloor \frac{M}{W}(x^2 \bmod W) \right\rfloor .$$

Notice that since M and W are both powers of two, the ratio $\mathbf{W/M = 2^{n-k}}$ is also a power two. Therefore, in order to multiply the term $\mathbf{(x^2 \bmod W)}$ by M/W we simply shift it to the right by n-k bits! In effect, we are extracting k bits from the middle of the square of the key--hence the name of the method.

Multiplication Method

A very simple variation on the middle-square method that alleviates its deficiencies is the so-called, multiplication hashing method . Instead of multiplying the key x by itself, we multiply the key by a carefully chosen constant a, and then extract the middle k bits from the result. In this case, the hashing function is

$$h(x) = \left\lfloor \frac{M}{W}(ax \bmod W) \right\rfloor .$$

What is a suitable choice for the constant a? If we want to avoid the problems that the middle-square method encounters with keys having a large number of leading or trailing zeroes, then we should choose an a that has neither leading nor trailing zeroes.

Furthermore, if we choose an a that is relatively prime to W, then there exists another number a' such that $aa' = 1 \pmod W$. In other words, a' is the inverse of a modulo W, since the product of a and its inverse is one. Such a number has the nice property that if we take a key x, and multiply it by a to get ax, we can recover the original key by multiplying the product again by a', since axa'=aa'x=1x.

There are many possible constants which the desired properties. One possibility which is suited for 32-bit arithmetic (i.e., $W = 2^{32}$) is $a = 2\,654\,435\,769$. The binary representation of a is

$$10\,011\,110\,001\,101\,110\,111\,100\,110\,111\,001.$$

This number has neither many leading nor trailing zeroes. Also, this value of a and $W = 2^{32}$are relatively prime and the inverse of a modulo W is $a' = 340\,573\,321$.

Folding Method:

In the folding method, the key is divided into two parts that are then combined of folded together to create an index into the table. This is done by first dividing the key into two parts of the key will be the same length as the desired key.

In the shift folding method , these parts are then added together to create the index

e.g. Number: 987654321 , we divide into three parts 987,654,321, and then add these to get 1962 . Then use either division or extraction to get three digit index

In the boundary shift folding method , some parts of the key are reversed before adding e.g. Number: 987654321 , we divide into three parts 987,654,321, and then reverse the middle element 987,456,321 add these to get 11764 . Then use either division or extraction to get three digit index

ALGORITHM:

Create Hash Table and define hash Function.

Hash is a Hash function which return Hash Value(address/location)of the key in a Hash Table. Location(Hash Value) is a variable used as address of record in Hash Table.

Chaining Without Replacement:

W_Chaing_WO_Replacement(HashTable,KeyValue)

HashTable is two dimensional integer array, which store primary key & link. KeyValue is a variable of type integer,it's primary key.

HashTable[I][0] is Primary Key & HashTable[I][1] is link .

Position is integer variable, it is Empty position in Hash Table.

Step 1: Location = Hash(Key);

```
Step 2: If  HashTable[Location][0] is Empty
          Then  HashTable[Location][0] =  KeyValue;
Step 3: Else If Hash(HashTable[Location][0]) = = Hash(KeyValue)
                    I =  Location;
                Do Step A while HashTable[I][1] != -1
                     A:  I <- HashTable[I][1];
                HashTable[I][1] = Position;
                 HashTable[Position][0] = KeyValue;
                EndIf
                Else  I  =  Location+1;
                     Do Step A,B While (I%10) != Location
                          A: If I==10  Then  I=0;
                             a: If  Hash(HashTable[I][0]) = = Hash(Key)
                               Do Step A While HashTable[I][1]!=-1
                                  A: I =  HashTable[I][1];
                               EndIf
                              b: HashTable[I][1] <- Position;
                               c: break
                         B: Increment I;
                        HashTable[Position][0] <- KeyValue;
                 EndElse.
          EndElse.
Step 4: Stop.
```

INPUT:

Test Cases	O/P
Hash Table full	Display message" Hash Table full"
Hash Table Empty	Display message" Hash Table Empty"
Hash Value data already present	Display message" Position already Occupied"

OUTPUT:

Hash table with records

FAQS:

1. What is hash function?
2. What is hash table?
3. What are the advantages & disadvantages of hash technique?
4. What are characteristics of good hash function?

10. Prim's Algorithm

AIM: A business house has several offices in different countries; they want to lease Phone lines to connect them with each other and the phone company charges different rent to connect different pairs of cities. Business house want to connect all Its offices with a minimum total cost. Solve the problem by suggesting appropriate data structures

OBJECTIVES:

1. To understand minimum spanning tree of a Graph
2. To understand how Prim's algorithm works

PROBLEM SPECIPICATIONS:

DATA STRUCTURES TO BE USED:

Array: Two dimensional array (adjacency matrix) to store the adjacent vertices & the weights associated edges.

One dimensional array to store an indicator for each vertex whether visited or not.

```
#define max 20
int adj_ver[max][max];
int edge_wt[max][max];
int ind[max];
```

CONCEPTS TO BE USED:

- Arrays
- Function to construct head list & adjacency matrix for a graph.
- Function to display adjacency matrix of a graph.
- Function to generate minimum spanning tree for a graph using Prim's algorithm.

THEORY:

- **Spanning Tree:**

A Spanning Tree of a graph G = (V, E) is a sub graph of G having all vertices of G and no cycles in it.

Minimal Spanning Tree: The cost of a graph is the sum of the costs of the edges in the weighted graph. A spanning tree of a graph G= (V, E) is called minimal cost spanning tree or simply the minimal spanning tree of G if its cost is minimum.

- When a graph G is connected, depth first or breadth first search starting at any vertex visits all the vertices in G.
- The edges of G are partitioned into two sets i.e. T for the tree edges & B for back edges. T is the set of tree edges and B for back edges. T is the set of edges used or traversed during the search & B is the set of remaining edges.
- The edges of G in T form a tree which includes all the vertices of graph G and this tree is called as spanning tree.

Definition: Any tree, which consists solely of edges in graph G and includes all the vertices in G, is called as spanning tree. Thus for a given connected graph there are

multiple spanning trees possible. For maximal connected graph having ‘n’ vertices the number of different possible spanning trees is equal to n.
Cycle: If any edge from set B of graph G is introduced into the corresponding spanning tree T of graph G then cycle is formed. This cycle consists of edge (v, w) from the set B and all edges on the path from w to v in T.

- Prim's algorithm: Prim’s algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm was discovered in 1930 by mathematician Vojtech Jarník and later independently by computer scientist Robert C. Prim in 1957 and rediscovered by Edsger Dijkstra in 1959. Therefore it is sometimes called the DJP algorithm, the Jarník algorithm, or the Prim-Jarník algorithm.

An arbitrary node is chosen initially as the tree root (any node can be considered as root node for a graph). The nodes of the graph are then appended to the tree one at a time until all nodes of the graph are included. The node of the graph added to the tree at each point is that node adjacent to a node of the tree by an arc of minimum weight. The arc of minimum weight becomes a tree arc containing the new node to tree. When all the nodes of the graph have been added to the tree, a minimum spanning tree has been constructed for the graph.

Algorithm / Pseudo code:

- **Prim’s Algorithm:**
  All vertices of a connected graph are included in the minimum spanning tree. Prim’s algorithm starts from one vertex and grows the rest of tree by adding one vertex at a time by adding associated edge in T. This algorithm iteratively adds edges until all vertices are visited.

  void prims (vertex i)

1. Start
2. Initialize visited [ ] to 0
   for (i=0;i<n; i++)
   visited [i] = 0;
3. Find minimum edge from i
   for (j=0;j<n; j++)
   {
   if (min > a [i] [j])
   {
   min = a[i] [j]
   x = i;
   y = j;
   }
   }
4. Print the edge between i and j with weight.
5. Make visit [i++] = x
   visit [j++] = y

6. Find next minimum edge starting from nodes of visit array.
7. Repeat step 6 until all the nodes are visited.
8. End.

Trace of Prim's algorithm for graph G1 starting from vertex 1

Step No.	**Set A**	**Set (V-A)**	**Min cost Edge (u, v)**	**Cost**	**Set B**
Initial	{1}	{2,3,4,5,6,7}	--	--	{}
1	{1,2}	{3,4,5,6,7}	(1, 2)	1	{(1, 2)}
2	{1, 2, 3}	{4, 5, 6, 7}	(2, 3)	2	{(1,2},(2,3)}
3	{1,2,3,5}	{4, 6, 7}	(2, 5)	4	{(1,2),(2,3),(2,5)}
4	{1,2,3,5,4}	{6, 7}	(1,4)	4	{(1,2),(2,3),(2,5),(1,4)}
5	{1,2,3,5,4,7}	{6}	(4,7)	4	{(1,2),(2,3),(2,5),(1,4),(4,7)}
6	{1,2,3,5,4,7.6}	{ }	(7,6)	3	{(1,2),(2,3),(2,5),(1,4),(4,7),(7,6)}
		Total Cost		17	

Thus the minimum spanning tree for graph G1 is : A = { 1,2,3,4,5,7,6}
B = {(1,2),(2,3),(2,5),(1,4),(4,7),(7,6)}
Total Weight: 1+2+4+3+4+3=17

INPUT:
Graph entered as an adjacency matrix for n vertices i.e. fill an n*n matrix with the values of the weights of the edges of the graph where the row number and column number as the two vertices of one edge.

OUTPUT: Display adjacency matrix for the constructed graph. Display minimum spanning tree using Prim's algorithm.

- **Sample Input/Output:**

Input: Enter vertices of a graph: 1 2 3 4 5 6 7
Enter vertex wise adjacent vertices & cost of edges.

Vertex	**Adjacent Vertex**	**Cost**
Pune (1)	2	1
	4	4
	0	
Mumbai (2)	**1**	**1**
	4	**6**
	5	**4**
	3	**2**
	0	

Bangalore (3)	**2**	**2**
	5	**5**
	6	**8**
	0	
Hyderabad (4)	**1**	**4**
	2	**6**
	5	3
	5	3
	7	4
	0	
Chennai (5)	**2**	**4**
	3	**5**
	6	**8**
	7	**7**
	4	**3**
	0	
Delhi (6)	**3**	**8**
	5	**8**
	7	**3**
	0	
Ahmadabad (7)	**4**	**4**
	5	**7**
	6	**3**
	0	

Output: Display of adjacent matrix_adj vertices & cost of associated edges.

Pune (1): (0, 0) (2, 1) (0,0) (4, 4) (0, 0) (0, 0) (0, 0)
Mumbai (2): (1, 1) (0, 0) (3, 2) (4, 6) (5, 4) (0, 0) (0, 0)
Bangalore (3): (0, 0) (2, 2) (0, 0) (0, 0) (5, 5) (6, 8) (0, 0)
Hyderabad (4): (1, 4) (2, 6) (0, 0) (0, 0) (5, 3) (0, 0) (7, 4)
Chennai (5): (0, 0) (2, 4) (3, 5) (4, 3) (0, 0) (6, 8) (7, 7)
Delhi (6): (0, 0) (0, 0) (3, 8) (0, 0) (5, 8) (0, 0) (7, 3)
Ahmadabad (7): (0, 0) (0, 0) (0, 0) (4, 4) (5, 7) (6, 3) (0, 0)

Minimum spanning tree using Prim's algorithm:
{(1,2), (2,3), (2,5), (5,4), (4,7), (7,6)}

Total cost: 17

Testing:

Test program for following test cases

For each test case:

a) Display the total number of comparisons required to construct the graph in computer memory.

b) Display the results as given in the sample o/p above.

c) Finally conclude on time & time space complexity for the construction of the graph and for generation of minimum spanning tree using Prim's algorithm.

Time Complexity:

For the construction of an undirected graph with 'n' vertices and 'e' edges using adjacency list is O (n + e), since for every vertex 'v' in G we need to store all adjacent edges to vertex v.

- ✓ In Prim's algorithm to get minimum spanning tree from an undirected graph with 'n' vertices using adjacency matrix is $O(n^2)$.
- ✓ Using Kruskal's algorithm
  using adjacency matrix = $O(n^2)$.
  using adjacency list=O(eloge)

Pseudo code:

Structure to be used

```
typedef struct edges
{
        int v1,v2,wt;
}edge;
```

Algorithm acceptgraph (int G[][MAX], int n)

```
        declare int i,j;
        print (Enter 0 if no edge is present)
        for i=0 to n do
                for j=i+1 to n do
                        print Enter Wt. of edge(V[i], V[j]):
                        read(G[i][j])
                        G[j][i]=G[i][j]
                end
        end
end acceptgraph
```

- **Algorithm for adjacent to edges**

Algorithm int AdjToEdges(int G[][MAX], int n, edge E[])

```
        declare int i,j,k=0;
        for i=0 to n do
                for j=i+1 to n do
                        if(G[i][j])
                                E[k].v1=i;
                                E[k].v2=j;
                                E[k++].wt=G[i][j];
```

```
                end if
            end for
        end for
        return k;
end AdjToEdges
```

- **Algorithm to print edges**

```
    Algorithm PrintEdges(edge E[], int noe)
{
        int i;
        for(i=0;i<noe;i++)
        {
            printf("\n(V%d, V%d)=%d",E[i].v1,E[i].v2,E[i].wt);
        }
}
```

- **// Function to sort the edges according to weights**

```
Algorithm SortEdges(edge E[], int noe)
{
        int i,j;
        edge t;
        for(i=0;i<noe;i++)
        {
            for(j=i+1;j<noe;j++)
            {
                if(E[i].wt>E[j].wt)
                {
                    t=E[i];
                    E[i]=E[j];
                    E[j]=t;
                }
            }
        }
}
```

- **// Function to calculate total cost**

```
Algorithm int total(edge E[], int noe)
{
        int i,sum=0;
        for(i=0;i<noe;i++)
            sum=sum+E[i].wt;
        return sum;
}
```

- **Algorithm for sorting the edges according to weights**

```
algorithm int Search(int tv[], int v, int n)
begin:
        declare int i;
        for i=0 to n dp
            if(tv[i]==v)
                return(1);
```

```
        end
        return(0);
end
```

- **Algorithm PrintAdj(int G[][MAX], int nov)**

```
begin:
        declare int i,j;
        for i=0to nov do
                print V[i]
        end
        for i=0 to nov do
                print "\nV[i]"
                for j=0 to nov do
                        printf("%5d",G[i][j]);
                end
        end
end
```

- **Algorithm to generate spanning tree by Prim's Algorithm**

Algorithm prims(int S[ ][MAX],edge E[], int noe):

```
begin:
        declare int TV[MAX], visited[MAX]={0};
        declare int i,j,k=0,l,vt1,vt2,v;
        declare edge T[20];
        TV[k++]=0;
        for i=0 to noe do
                for j=0 to noe do
                        if(!visited[j])
                                vt1=E[j].v1;
                                vt2=E[j].v2;
                                if((Search(TV, vt1,k)&&!Search(TV,vt2,k)))
                                        S[vt1][vt2]=E[j].wt;
                                        S[vt2][vt1]=E[j].wt;
                                        TV[k++]=vt2;
                                        visited[j]=1;
                                        break;
                                end
                                else if(Search(TV,vt2,k)&&!Search(TV,vt1,k))
                                        S[vt1][vt2]=E[j].wt;
                                        S[vt2][vt1]=E[j].wt;
                                        TV[k++]=vt1;
                                        visited[j]=1;
                                        break;
                                end
                        end if
                end for
        end for
end Prim
```

Applications of spanning Trees:

- To find independent set of circuit equations for an electrical network. By adding an edge from set B to spanning tree we get a cycle and then Kirchoff's second law is used on the resulting cycle to obtain a circuit equation.
- Using the property of spanning trees we can select the spanning tree with (n-1) edges such that total cost is minimum if each edge in a graph represents cost.
- Analysis of project planning
- Identification of chemical compounds
- Statistical mechanics, genetics, cybernetics, linguistics, social sciences

CONCLUSION:

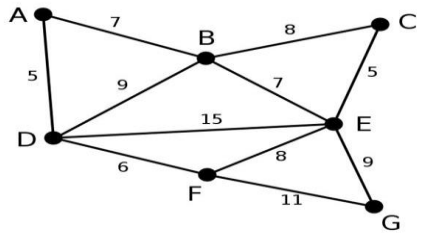
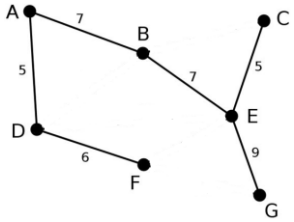
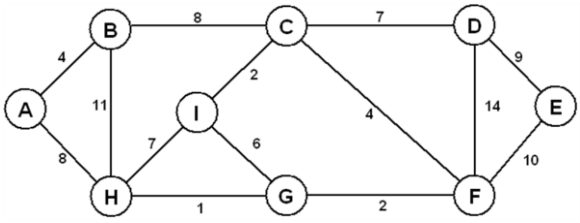
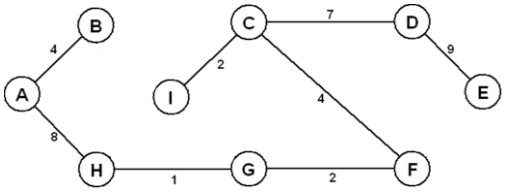
The cost of spanning tree of graph G is the sum of the costs of the edges in that tree. Any connected graph with n vertices must have at least n-1 edges and all connected graphs with n-1 edges are trees.

INPUT:

Enter the no. of nodes in graph. Create the adjacency LIST

OUTPUT:

Display result of each operation with error checking.

INPUT	**OUTPUT**	Remark
		Cost of MST - 39
		Cost of MST - 37

FAQS:

1. Explain the PRIM's algorithm for minimum spanning tree.
2. What are the traversal techniques?
3. What are the graph representation techniques?

11. Implementation of sequential file

AIM: Department maintains a student information. The file contains roll number, name, division and address. Write a program to create a sequential file to store and maintain student data. It should allow the 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 student record is found it should display the student details.

.

Aim

Implement sequential file for student Database and perform following operations on it :

i) Create Database ii) Display Database iii) Add a record

iv) Delete a record

Objective :

To Study:

1. The local and physical organization of files.
2. To understand the concept of sequential files.
3. To know about File Operations
4. To understand the use of sequential files.
5. Sequential file handling methods.
6. Creating, reading, writing and deleting records from a variety of file structures.
7. Creating code to carry out the above operations.

Theory:

File : A file is a collection on information, usually stored on a computer's disk. Information can be saved to files and then later reused.

Type of File

- Binary File
  - The binary file consists of binary data
  - It can store text, graphics, sound data in binary format
  - The binary files cannot be read directly
  - Numbers stored efficiently
- Text File
  - The text file contains the plain ASCII characters
  - It contains text data which is marked by 'end of line' at the end of each record
  - This end of record marks help easily to perform operations such as read and write
  - Text file cannot store graphical data.

File Organization :

The proper arrangement of records within a file is called as file organization. The factors that affect file organization are mainly the following:

- Storage device
- Type of query
- Number of keys

- Mode of retrieval/update of record

Different types of File Organizations are as :

- Sequential file
- Direct or random access file
- Indexed sequential file
- Multi-Indexed file

Sequential file : In sequential file, records are stored in the sequential order of their entry. This is the simplest kind of data organization. The order of the records is fixed. Within each block, the records are in sequence . A sequential file stores records in the order they are entered. New records always appear at the end of the file.

Features of Sequential files :

- Records stored in pre-defined order.
- Sequential access to successive records.
- Suited to magnetic tape.
- To maintain the sequential order updating becomes a more complicated and difficult task. Records will usually need to be moved by one place in order to add (slot in) a record in the proper sequential order. Deleting records will usually require that records be shifted back one place to avoid gaps in the sequence.
- Very useful for transaction processing where the hit rate is very high e.g. payroll systems, when the whole file is processed as this is quick and efficient.
- Access times are still too slow (no better on average than serial) to be useful in on-line applications.

Drawbacks of Sequential File Organization

- Insertion and deletion of records in in-between positions huge data movement
- Accessing any record requires a pass through all the preceding records, which is time consuming. Therefore, searching a record also takes more time.
- Needs reorganization of file from time to time. If too many records are deleted logically, then the file must be reorganized to free the space occupied by unwanted records

Primitive Operations on Sequential files

Open—This opens the file and sets the file pointer to immediately before the first record

Read-next—This returns the next record to the user. If no record is present, then EOF condition will be set.

Close—This closes the file and terminates the access to the file

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

EOF—If EOF condition occurs, it returns true, otherwise it returns false

Search—Search for the record with a given key

Update—Current record is written at the same position with updated values

- **Direct or random access file :** Files that have been designed to make direct record retrieval as easy and efficiently as possible is known as directly organized files. Though we search records using key, we still need to know the address of the

record to retrieve it directly. The file organization that supports Files such access is called as direct or random file organization. Direct access files are of great use for immediate access to large amounts of information. They are often used in accessing large databases.

- ***Advantages of Direct Access Files :***
  - Rapid access to records in a direct fashion.
  - It doesn't make use of large index tables and dictionaries and therefore response times are very fast.
- **Indexed sequential file :** Records are stored sequentially but the index file is prepared for accessing the record directly. An index file contains records ordered by a record key. The record key uniquely identifies the record and determines the sequence in which it is accessed with respect to other records.
- A file that is loaded in key sequence but can be accessed directly by use of one or more indices is known as an indexed sequential file. A sequential data file that is indexed is called as indexed sequential file. A solution to improve speed of retrieving target is index sequential file. An indexed file contains records ordered by a record key. Each record contains a field that contains the record key.

- This system organizes the file into sequential order, usually based on a key field, similar in principle to the sequential access file. However, it is also possible to directly access records by using a separate index file. An indexed file system consists of a pair of files: one holding the data and one storing an index to that data. The index file will store the addresses of the records stored on the main file. There may be more than one index created for a data file e.g. a library may have its books stored on computer with indices on author, subject and class mark.

Characteristics of Indexed Sequential File

- Records are stored sequentially but the index file is prepared for accessing the record directly
- Records can be accessed randomly
- File has records and also the index
- Magnetic tape is not suitable for index sequential storage
- Index is the address of physical storage of a record
- When randomly very few are required/accessed, then index sequential is better
- Faster access method
- Addition overhead is to maintain index
- Index sequential files are popularly used in many applications like digital library

Primitive operations on Index Sequential files (IS)

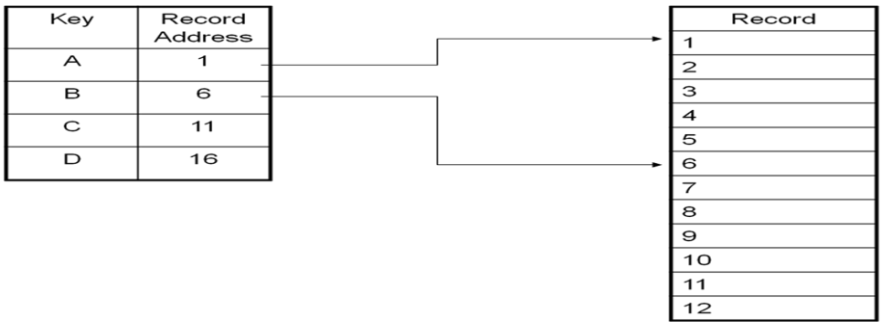
- ***Write (add, store) :*** User provides a new key and record, IS file inserts the new record and key.
- ***Sequential Access (read next) :*** IS file returns the next record (in key order)
- ***Random access (random read, fetch) :*** User provides key, IS file returns the record or "not there"
- ***Rewrite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

Types of Indexed Files : There are two types of indexed files:

- Fully Indexed
- Indexed Sequential

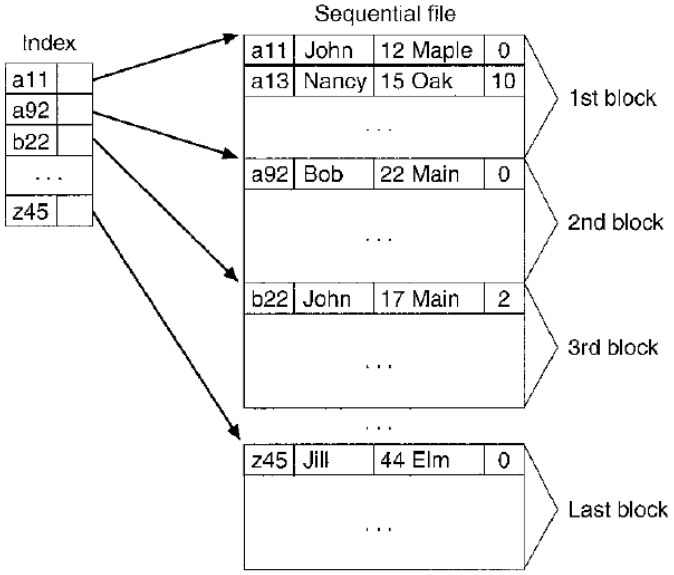
Fully Indexed Files :

An index to a fully indexed file will contain an entry for every single record stored on the main file. The records will be indexed on some key e.g. student number. Very large files will have correspondingly large indices. The index to a (large) file may be split into different index levels. When records are added to such a file, the index (or indices) must also be updated to include their relative position and change the relative position of any other records involved.



Indexed Sequential Files :

This is basically a mixture of sequential and indexed file organisation techniques. Records are held in sequential order and can be accessed randomly through an index. Thus, these files share the merits of both systems enabling sequential or direct access to the data. The index to these files operates by storing the highest record key in given cylinders and tracks. Note how this organisation gives the index a tree structure. Obviously this type of file organisation will require a direct access device, such as a hard disk. Indexed sequential file organisation is very useful where records are often retrieved randomly and are also processed in (sequential) key order. Banks may use this organisation for their auto-bank machines i.e. customers randomly access their accounts throughout the day and at the end of the day the banks can update the whole file sequentially.



Advantages of Indexed Sequential Files

1. Allows records to be accessed directly or sequentially.
2. Direct access ability provides vastly superior (average) access times.

Disadvantages of Indexed Sequential Files

1. The fact that several tables must be stored for the index makes for a considerable storage overhead.
2. As the items are stored in a sequential fashion this adds complexity to the addition/deletion of records. Because frequent updating can be very inefficient, especially for large files, batch updates are often performed.

Multi-Indexed file : In multi-indexed file, the data file is associated with one or more logically separated index files. Inverted files and multilist files are examples of multiindexed files

Algorithms :

1. Algorithm for main function

MAIN FUNCTION()

S1: Read the two filenames from user master and temporary.

S2: Read the operations to be performed from the keyboard

S3: If the operation specified is create go to the create function, if the operation specified is display go to the display function, if the operation specified is add go to the add function , if the operation specified is delete go to delete function, if the operation specified is display particular record go to the search function, if the operation specified is exit go to step 4.

S4: Stop

2. Algorithm for create function

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

S2: Read the no: of records N to be inserted to the file .

S3: Repeat the step4 N number of times .

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

S5: Close the file .

S6: Return to the main function

3. Algorithm for displaying all records

S1: Open the specified file in read mode.

S2: If the file is unable to open or not found then display error message and go to step 4 else go to Step 3

S3: Scan all the student details one by one from file and display the same at the console until end of file is reached.

S4: Close the file

S5: Return to the main function

4. Algorithm for add a record

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

S2: Scan all the student details one by one from file until end of file is reached.

S3: Read the details of the from the keyboard and write the same to the file

S4: Close the file .

S5: Return to the main function

5. Algorithm for deleting a record

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

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

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

S4:Close both files

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

6. Algorithm for displaying particular record(search)

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

S2: Read the roll number of the student whose details need to be displayed.

S3: Read each student record from the file.

S4: Compare the students roll number scanned from file with roll number specified by the user.

S5: If they are equal then display the details of that record else display required roll number not found message and go to step6.

S6: Close the file.

S7: Return to the main function.

Test Conditions:

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

Sample Input Output

MENU

Choice 1 : Create
Choice 2 : Display
Choice 3 : add
Choice 4 : Delete
Choice 5 : Display particular record
Choice 6 : Exit

Create

Accept the records and write into the file
File created successfully

Display

Display all records present in Master file.

Add

Accept the record to be add into the file at the end of the file.
Record inserted successfully

Delete

Accept the record to be deleted.
Record deleted successfully

Search

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

FAQ:

1. Define File? What are the factors affecting the file organization?
2. Compare Text and Binary File?
3. Explain the different File opening modes in C++?
4. What is indexed sequential file ? Explain the primitive operations on indexed sequential file.
5. Write a note on Direct Access File?
6. What are the advantages and disadvantages of indexed-sequential file organization?
7. What are the advantages of Sequential File Organization?
8. What are serial and sequential files and how are they used in organization?
9. Distinguish between logical and physical deletion of records and illustrate it with suitable examples.
10. Compare and contrast sequential file and random access file organization?
11. Explain the different types of external storage devices?
12. With the prototype and example, explain following functions:
    i. i) seekg() ii)tellp() iii) seekp() iv) tellp()

Conclusion:

Thus we have implemented sequential file and performed all the primitive operations on it.

12. Implementation of Direct Access File

Aim: Implement direct access file using hashing (chaining without replacement) and perform following operations on it

a. Create Database
b. Display Database
c. Add a record
d. Search a record
e. Modify a record

Objectives:

1. To learn the concept of file handling
2. To learn the concept of hashing
2. To create & maintain a Direct Access File using Chaining without replacement

Theory:

1. What is Direct Access File Organization?
2. What is a Hash Function?
3. Which are the primitive operations performed on direct access file?
4. How direct access file organization is different from sequential file organization?
5. How direct access file organization is different from index sequential file organization?
6. What is the time required to access any record from direct access file?
7. List the advantages and disadvantages of direct access file organization.

Input :

- create an empty direct access file of size max (say 10)
- add new student records to the direct access file
- modify a student record from direct access file for a given roll-no
- search a student record from direct access file for a given roll-no

Output :

- display all student records from direct access file
- search and display the student record from direct access file for a given roll-no
- display modified / updated student's record from from direct access file for a given roll-no (i.e. modified / updated).

Sample Input / Output :

********** INPUT **********

Roll-No	Name of Student	Sub1-marks	Sub-2-marks	Sub-3-marks	Add More?
1000	Rajiv Goyal	80	90	85	Y
1001	Vandana Patil	78	91	86	Y
1004	Sachin Choudhari	70	65	75	Y
1081	Varsha Kanade	80	83	87	Y
1065	Arun Deshmukh	60	80	90	Y
1025	Shweta Kulkarni	51	21	41	Y

1036	Parag Mahajan	31	55	19	Y
1016	Sarika Shinde	77	56	49	N

Chaining without replacement

********** OUTPUT **********

Display all records from direct access file

Roll-No	Name of Student	Sub1-marks	Sub-2-marks	Sub-3-marks	Chain
1000	Rajiv Goyal	80	90	85	0
1001	Vandana Patil	78	91	86	2
1081	Varsha Kanade	80	83	87	0
-	-	-	-	-	-
1004	Sachin Choudhari	70	65	75	0
1065	Arun Deshmukh	60	80	**90**	**6**
1025	Shweta Kulkarni	51	21	41	0
1036	Parag Mahajan	31	55	19	0
1016	Sarika Shinde	77	56	49	0
-	-	-	-	-	-

Chaining with replacement

********** OUTPUT **********

Display all records from direct access file

Roll-No	Name of Student	Sub1-marks	Sub-2-marks	Sub-3-marks	Chain
1000	Rajiv Goyal	80	90	85	0
1001	Vandana Patil	78	91	86	2
1081	Varsha Kanade	80	83	87	0
-	-	-	-	-	-
1004	Sachin Choudhari	70	65	75	0
1065	Arun Deshmukh	60	80	**90**	**7**
1036	Parag Mahajan	31	55	19	0
1025	Shweta Kulkarni	51	21	41	0
1016	Sarika Shinde	77	56	49	0
-	-	-	-	-	-

Testing : - Test your program for the following test cases.

- For each test case,
  (a) Display the **total number of records** read from file for each option
  (b) Display the **results** as given in sample output above.
- **Comment** on usage of direct access files.

ALGORITHM:

DIRECT ACCESS FILE OPERATIONS

(1) **Create an empty Direct Access File**
(2) /* procedure create_df */
(3) Chaining without replacement

a. Add a new record to Direct Access File // procedure add_df_wor
b. Modify existing record from DAF //procedure modify_df
c. Search a record from DAF // procedure search_df
d. Display all records from DAF // procedure display_all_df

procedure create_df( )
This procedure creates direct access file of size max – roll-no of each record is initialized to if the file exists with this name then it will be truncated

1) ofstream fp;
2) fp.open ("datafile.dat", ios::binary) /* data file opened in write mode */
3) for (i=0, i < max; i++)
   {
   temp_data.roll_no = -1;
   temp_data.chain = 0;
   fp.write ((char *) &temp_data, sizeof(temp_data));
   }
4) fp.close ();

end procedure create_df

procedure hash_key (int key)

```
{
        return (key%max);
}
```

end procedure hash_key

procedure add_df_wor( )

/* this procedure adds a new record in direct access file – using chaining without */
/* replacement */

1) int i, cnt, addr, addr1=0, prev_syn_pos;
   fstream fp;
2) fp.open ("datafile.dat", ios::in | ios::out | ios::binary)
3) accept temp_data.roll_no,

```
4)  addr = hash_key (temp_data.roll_no);
    fp.seekg(addr*sizeof(temp_data),ios::beg);
    fp.read( (char *) &temp1_data, sizeof(temp1_data));
5)  if   (temp1_data.roll_no = -1)          /* first insertion at natural address */
    {
        accept  temp-data.name, temp-data.marks1, temp-data.marks2,
                temp-data.marks3;
        temp-data.chain = NULL;
        fp.seekp(addr*sizeof(temp_data),ios::beg);
        fp.write((char *) &temp_data, sizeof(temp_data));
        go to step 9
    }
     else addr1 = hash_key (temp1_data.roll_no);
6)  if   (addr1 = addr)                     /* to insert a record in existing natural */
        i = addr;                           /* address chain                          */
        call append_chain;
        go to step 9
7)   i = addr; cnt = 0;
    while       (temp1_data.roll_no > 0) &&          /* search for free slot or */
                (hash-key(temp1_data.roll_no) != addr) && /* synonym serially */
       (cnt != max)
    {
        i = (i + 1) % max;
        fp.seekg(i*sizeof(temp1_data),ios::beg);
        fp.read((char *)&temp1-data, sizeof(temp1_data));
        cnt = cnt + 1;
    }
8)   if   (hash-key(temp1_data.roll_no) = addr)   /* to insert a record in unnatural */
    {                                       /* address chain  of synonym          */
        call append_chain                   /* no empty slot file              .*/

        go to step 9
    }
    else
    if   (temp1_data.roll_no <= 0)
    {
        accept  temp-data.name, temp-data.marks1, temp-data.marks2,
                temp-data.marks3;
        temp-data.chain = NULL;
        fp.seekp(i*sizeof(temp_data, ios::beg);
        fp.write((char *)&temp-data, sizeof(temp_data));
    }
    else
    {
        display ("table overflow")                  /* cnt = max */
    }
9) accept reply                                     /* reply for "any more"       */
10) if   (reply = 'Y' OR 'y')
                go to step 3
```

```
    11) fp.close();
        stop
    end procedure add_df_wor
```

procedure append_chain( )

```
/* this procedure append the new record from temp-data to the end of chain starting from */
/* temp1-data and updates the chain pointer of  last record of old chain                  */
    1)    i = addr;
          while  (temp1_data.roll-no != temp_data.roll-no) &&    /* traversing chain */
                 (temp1_data.chain != NULL)
          {
                 i =  temp1_data.chain;
                 fp.seekg(i*sizeof(temp1_data), ios::beg);
                 fp.read( (char *) &temp_data, sizeof(temp_data));
          }
          if     (temp1-data.roll-no = temp-data.roll-no)  /* duplicate roll-no    */
          {
                 display ("record exists in direct access file")
                 go to step 2
          }
          prev-syn-pos = i;
          while  (temp1-data.roll-no > 0)                  /* to find next empty slot*/
          {
                 i =  (i + 1)%max;
                 fp.seekg(i*sizeof(temp1_data),ios::beg);
                 fp.read((char *)&temp1-data, sizeof(temp1_data));
          }
          accept  temp-data.name, temp-data.marks1, temp-data.marks2,
          temp-data.marks3;
          temp-data.chain = NULL;
          fseek(fp, i*sizeof(stud),0);
          fwrite(&temp-data, sizeof(stud), 1, fp);
          fseek(fp, prev-syn-pos*sizeof(stud),0);
          fread(&temp1-data, sizeof(stud), 1, fp);
          temp1-data.chain = i;                            /* change prev record link*/
          fseek(fp, prev-syn-pos*sizeof(stud),0);
          fwrite(&temp1-data, sizeof(stud), 1, fp);
    2)    return
end procedure append_chain
```

procedure modify_df ( )

```
/* this procedure modifies the existing record from direct access file – using   chaining  */
/* without  replacement                                                                    */
```

1) int i, cnt, addr, addr1=0, prev-syn-pos; fstream fp;
2) fp.open("datafile.dat", ios::in | ios::out | ios::binary)
3) accept temp-data.roll-no,
4) addr = hash_key (temp-data.roll-no);

```
            fp.seekg(addr*sizeof(temp1_data),ios::beg);
            fp.read((char *)&temp1-data, sizeof(temp1_data));
        5)  if   (temp1-data.roll-no =  temp-data.roll-no)   /* modification  at natural address
*/
            {
                display temp1-data.name, temp1-data.marks1, temp1-data.marks2,
                        temp1-data.marks3;
                accept  temp-data.name, temp-data.marks1, temp-data.marks2,
                        temp-data.marks3;
                temp-data.chain = temp1-data.chain;
                fp.seekp(addr*sizeof(temp_data),ios::beg);
                fp.write((char *)&temp-data, sizeof(temp_data));
                go to step 10
            }
             else addr1 = hash_key (temp1-data.roll-no);
        6)  if   (addr1 = addr)                    /* to search  a record in existing natural */
                i = addr;                          /* address chain                          */
                call update_chain;
                go to step 10
        7)   i = addr; cnt = 0;
            while       (temp1-data.roll-no > 0) &&
                        (hash-key(temp1-data.roll-no) != addr) &&
                        (cnt != max)
             {
                i = (i + 1) % max;
                fp.seekp(i*sizeof(temp1_data),ios::beg);
                fp.read((char *)&temp1-data, sizeof(temp1_data));
                cnt = cnt + 1;
             }
        8)   if   (hash-key(temp1-data.roll-no) = addr)  /* to search a record in an unnatural */
                call update_chain;                       /* address chain of synonym      */
                go to step 10
        9)   display (“record does not exist in direct access file”)        /* no record found.*/
        10)accept reply                                  /* reply for “any more”          */
        11)if   (reply = ‘Y’ OR ‘y’)
                 go to step 3
        12)fp.close();
        13)stop
end procedure modify_df
```

procedure update_chain ( )

```
/* this procedure search a record for accepted temp-data.roll-no in the chain of it’s
synonym */
/* and if found updates that record – temp1-data contains synonym
*/
        1)      while  (temp1-data.roll-no != temp-data.roll-no) &&
                       (temp1-data.chain != NULL)
```

```
            {
                    i =  temp1-data.chain;
                    fp.seekg(i*sizeof(temp1_data),ios::beg);
                    fp.read((char *)&temp1-data, sizeof(temp1_data));
            }
            if      (temp1-data.roll-no = temp-data.roll-no)  /* record found        */
            {
                    display temp1-data.name, temp1-data.marks1, temp1-data.marks2,
                    temp1-data.marks3;
                    accept  temp-data.name, temp-data.marks1, temp-data.marks2,
                    temp-data.marks3;
                    temp-data.chain = temp1-data.chain;
                    fp.seekp(i*sizeof(temp_data),ios::beg);
                    fp.write((char *)&temp-data, sizeof(temp_data));
            }
            else                                                /* no record found    */{
            {
                    display ("record does not exist in direct access file")
            }
    2)  return
end procedure update_chain

procedure display_all_df
/* this procedure displays all records  from direct access file – using chaining              */
1) int   i ;fstream fp;
2) fp.open("datafile.dat", ios::in| ios::binary
3) i = 0;
            fp.seekg(i*sizeof(temp1_data,ios::beg);
            fp.read((char *)&temp1_data, sizeof(temp1_data));
          for (i = 0; i < max; i++)
          {
            if      (temp-data.roll-no >=0)
            {       display temp-data.roll-no, temp-data.name,
                            temp-data.marks1, temp-data.marks2,
                            temp-data.marks3;
            }
            i = (i + 1);
            fp.seekg(i*sizeof(temp1_data,ios::beg);
            fp.read((char *)&temp1_data, sizeof(temp1_data));
          }
    4)  stop
end procedure display_all_df

 procedure search_df  ( )
/* this procedure displays a record of given roll-no from direct access file    –          using
chaining without replacement*/
    1)  int  i, cnt, addr, addr1=0, prev-syn-pos;
    2)  fp.open("datafile.dat", ios::in |ios::binary)
```

```
3)  accept  temp-data.roll-no,
4)  addr = hash_key (temp-data.roll-no);
    fp.seekg(addr*sizeof(temp1_data),ios::beg);
    fp.read((char *)&temp1-data, sizeof(temp1_data));
5)  if   (temp1-data.roll-no =  temp-data.roll-no)   /* located at natural address */
    {
        display temp1-data.name, temp1-data.marks1, temp1-data.marks2,
                temp1-data.marks3;
        go to step 10
     }
     else addr1 = hash_key (temp1-data.roll-no);
6)  if   (addr1 = addr)                    /* to search  a record in existing natural */
        i = addr;                          /* address chain                          */
        call display_chain;
        go to step 10
7)  i = addr; cnt = 0;
    while       (temp1-data.roll-no > 0) &&
                (hash-key(temp1-data.roll-no) != addr) &&
                (cnt != max)
    {
        i = (i + 1) % max;
    fp.seekg(i*sizeof(temp1_data),ios::beg);
    fp.read((char *)&temp1-data, sizeof(temp1_data));
        cnt = cnt + 1;
     }
8)  if   (hash-key(temp1-data.roll-no) = addr)  /* to search a record in an unnatural */
        call display_chain;                    /* address chain of synonym      */
        go to step 10
9)  display ("record does not exist in direct access file")/* no record found     .*/
10)accept reply                                /* reply for "any more"        */
11)if   (reply = 'Y' OR 'y')
        go to step 3
12) stop
    end procedure search_df
```

procedure display_chain ( )

```
/* this procedure searches a record for accepted temp-data.roll-no in a chain of it's
synonym */
/* and if found displays that record – temp1-data contains synonym       */
    1)      while  (temp1-data.roll-no != temp-data.roll-no) &&
                   (temp1-data.chain != NULL)
            {
                   i =  temp1-data.chain;
                   fp.seekg(i*sizeof(temp1_data,ios::beg);
                   fp.read((char *)&temp1-data, sizeof(temp1_data));
            }
            if     (temp1-data.roll-no = temp-data.roll-no)  /* record found        */
            {
                   display temp1-data.name, temp1-data.marks1, temp1-data.marks2,
```

```
                temp1-data.marks3;
            }
            else                                        /* no record found    */{
            {
                display ("record does not exist in direct access file")
            }
        2)  return
end procedure display_chain
```

Specifications :

Data Structures to be used :

a) Student Record :

```
struct  data-rec
{
        int     roll-no;
        char    name[30];
        int     marks1;
        int     marks2;
        int     marks3;
        int     chain;          /* location in file where next synonym */
}                               /* is stored. '0' means it is a last entry  */
typedef  struct  data-rec    student-rec;
```

b) Class Direct_File

ANALYSIS OF ALGORITHM :

1) **Time Complexity :**
   - Chaining with replacement gives better results in searching or locating a record but insertion / deletion of a record will take more time.
   - With low loading density (less collision) for all the methods time complexity is **O(1).**
   - With high loading density (more collision) time complexity for chaining with or without replacement method will be more than **O(1)** .
2) **Space Complexity:**
   - Additional space required to store link address in record in case of chaining with or without replacement.

Conclusion: In Direct file organization Records can be added randomly and access to any record in a file requires constant time. It is Useful for interactive & on-line applications.

Contributors

Sr. No	Title	Name of the Faculty	College
1	Implement stack as an abstract data type using linked list and use this ADT for conversion of infix expression to postfix, prefix and evaluation of postfix/prefix expression.	N. P. Kulkarni / S S Koul	SKNCOE, Pune
2	Implement priority queue as ADT using single linked list for servicing patients in an hospital with priorities as i) Serious (top priority) ii) medium illness (medium priority) iii) General (Least priority)	Shruti Choudhari	DYPIET, Ambi
3	Create Binary tree and perform following operations: a. Insert b. Display c. Depth of a tree d. Display leaf-nodes e. Create a copy of a tree	N. P. Kulkarni / S S Koul	SKNCOE, Pune
4	Construct and expression tree from postfix/prefix expression and perform recursive and non-recursive In-order, pre-order and post-order traversals.	N. P. Kulkarni / S S Koul	SKNCOE, Pune
5	Implement binary search tree and perform following operations: a. Insert b. Delete c. Search d. Mirror image e. Display f. Display level wise	Amarnath Chadchankar	Zeal College of Engg and Research, Pune
6	Consider a friends' network on face book social web site. Model it as a graph to represent each node as a user and a link to represent the fried relationship between them. Store data such as date of birth, number of comments for each user. 1. Find who is having maximum friends 2. Find who has post maximum and minimum comments 3. Find users having birthday in this month Hint: (Use adjacency list representation and	Snehal Bamare	NDMVP, Nashik

	perform DFS and BFS traversals)		
7	Represent any real world graph using adjacency list /adjacency matrix find minimum spanning tree using Kruskal's algorithm. .	N. P. Kulkarni / S S Koul	SKNCOE, Pune
8	Represent a given graph using adjacency matrix /adjacency list and find the shortest path using Dijkstra's algorithm (single source all destination).	N. P. Kulkarni / S S Koul	SKNCOE, Pune
9	Store data of students with telephone no and name in the structure using hashing function for telephone number and implement chaining with and without replacement.	N. P. Kulkarni / S S Koul	SKNCOE, Pune
10	A business house has several offices in different countries; they want to lease phone lines to connect them with each other and the phone company charges different rent to connect different pairs of cities. Business house want to connect all its offices with a minimum total cost. Solve the problem by suggesting appropriate data structures	Pranjali Kuche	MMCOE, Pune
11	Department maintains a student information. The file contains roll number, name, division and address. Write a program to create a sequential file to store and maintain student data. It should allow the 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 student record is found it should display the student details.	Sonali Kulkarni	NMIT
12	Implement direct access file using hashing (chaining without replacement) perform following operations on it a. Create Database b. Display Database c. Add a record d. Search a record e. Modify a record	Tanuja Sali	PCCOE, Pune