BHARATI VIDYAPEETH DEEMED UNIVERSITY
COLLEGE OF ENGINEERING, PUNE - 43

DEPARTMENT OF COMPUTER ENGINEERING

Lab Manual

Principles of Data Structures

B.Tech Computer Sem III

VISION OF THE INSTITUTE

Vision of the Institute

"To be World Class Institute for Social Transformation through Dynamic Education"

MISSION OF THE INSTITUTE

A. To provide quality technical education with advanced equipment, qualified faculty members, infrastructure to meet needs of profession and society.
B. To provide an environment conducive to innovation, creativity, research, and entrepreneurial leadership.
C. To practice and promote professional ethics, transparency and accountability for social community, economic and environmental conditions. VISION OF THE DEPARTMENT
To pursue and excel in the endeavor for creating globally recognized Computer Engineers

VISION OF THE DEPARTMENT

"To pursue and excel in the endeavor for creating globally recognized computer engineers through quality education."

Mission of the Department

- To impart engineering knowledge and skills confirming to a dynamic curriculum.
- To develop professional, entrepreneurial & research competencies encompassing continuous intellectual growth.
- To produce qualified graduates exhibiting societal and ethical responsibilities in working environment

PROGRAM EDUCATIONAL OBJECTIVES(PEOs):

1. Demonstrate technical and professional competencies by applying engineering fundamentals, computing principles and technologies.
2. Learn, Practice, and grow as skilled professionals/ entrepreneur/researchers adapting to the evolving computing landscape.
3. Demonstrate professional attitude, ethics, understanding of social context and interpersonal skills leading to a successful career.

ROGRAM SPECIFIC OUTCOMES(PSO)s:

PSO 1: To design, develop and implement computer programs on hardware towards solving problems.
PSO 2: To employ expertise and ethical practise through continuing intellectual growth and adapting to the working environment.

PROGRAMME OUTCOMES(POs):

Upon completion of the course the graduate engineers will be able to:
1. Apply the knowledge of mathematics, science, engineering fundamentals, and computing for the solution of complex engineering problems.
2. Identify, formulate, research literature, and analyse complex engineering problems reaching substantiated conclusions using computer engineering foundations, principles, and technologies.
3. Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for public health and safety, and cultural, societal, and environmental considerations.
4. Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.
6. Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal, and cultural issues, and the consequent responsibilities relevant to the professional engineering practice.
7. Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and the need for sustainable development.
8. Apply ethical principles while committed to professional responsibilities and norms of the engineering practice.
9. Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings
10. Communicate effectively on complex engineering activities with the engineering community and with the society at large, such as being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. Apply the engineering and management principles to one's work, as a member and

leader in a team.
12.Recognise the need for and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Course Name: -Principles of Data Structures

Hours Per Week: 3 Hrs.

Course Outcomes:

1. Outline basic concepts of data structures and linear data structures
2. Illustrate and exemplify linked list Differentiate between linear and nonlinear data structures and implement tree traversal techniques using recursive and non recursive methods.
3. Design and describe various types of tree structures Describe the characteristics of an algorithm,
4. Comprehend graph data structure and solve sorting problems.
5. Design and analyze various algorithm strategies like divide and conquer, Greedy method and dynamic programming.
6. Distinguish between NP-hard and NP-complete problems.

HOW OUTCOMES ARE ASSESSED?

Course Outcome	Assignment Number	Level	Proficiency evaluated by
Outline basic concepts of data structures and linear data structures	**1,2,3,4,5,7,9**	**3,3,3,3,3,3,3**	Performing Practical and reporting results
Illustrate and exemplify linked list	**6**	**3**	Problem definition &Performing Practical and reporting results
Design and describe various types of tree structures	**6**	**2**	Performing experiments and reporting results
Comprehend graph data structure and solve sorting problems.	**8,9,10**	**3,3,3**	Performing experiments and reporting results
Design and analyse various algorithm strategies like divide and conquer, Greedy Method, Dynamic Programming	**9,10**	**3,3**	Performing experiments and reporting results
Distinguish between NP-hard and NP-complete problems	**8,10**	**3,3**	Performing experiments and reporting results

CO-PO mapping

CO Statements	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO 10	PO 11	PO 12	PSO 1	PSO 2
CO1	3		3					2		3		2	3	
CO2	2	3	3		3			2		3		2	3	
CO3	2		3		3			2		3		2		3
CO4	2	3	3	3	3			2		3		2	2	3

CO5	3	3	3	3	3			2		3		2	1	
CO6	1	3		1	1			2		3		2		2

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

List Of Assignments

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.

2. Implement circular queue using array and perform following operations on it.

i) Add a record ii) Delete a record iii) Checking Empty iv) Checking Underflow v) Checking overflow

3. Implement priority queue as ADT using multiple linked lists ,one list for each priority for servicing patients in an hospital with priorities as i) Serious (top priority) ii) medium illness (medium priority) iii) General (Least priority).

4. Construct and expression tree from postfix/prefix expression and perform recursive and non-recursive Inorder , preorder and postorder traversals.

5. Implement binary search tree as an ADT

6. Construct an inorder threaded binary tree from inorder/postorder expression and traverse it in inorder and preorder

7. Represent any real world graph using adjacency list/adjacency matrix find minimum spanning tree using Prim's or Kruskal's algorithm.

8. Represent a given graph using adjacency matrix/adjacency list and find the shortest path using Dijkstra's algorithm.

9. Implementation of Hash table using array and handle collisions using Linear probing, chaining without replacement and Chaining with replacement

10. Implement Heap sort by constructing max or min heap .

11. Implement an index sequential file for any Database and perform following operations on it i) Create Database ii) Display Database iii) Add a record iv) Delete a record v)Modify a record

i) **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 means 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 node
- **Pop** removes a node

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:

1. **Add**: Pushes an element to the stack. It takes an integer element as argument. If the stack is full then error is returned.
2. **Delete**: Pop an element from the stack. If the stack is empty then error is returned. Th element is deleted from the top of the stack.
3. **Search**: This function takes an integer element as an argument and returns the location on the element. If number is not found then 0 is returned.
4. **Replace**: This function takes two integers as arguments, first number is to find and second is to replace with. It first performs the search operation the replaces the integers.

4) With an example explain how an infix expression can be converted to either prefix or postfix form with the stack status shown tabular for every character of the expression scanned.

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(Top,Node)
Step 1: Node->Next = *Top;
Step 2: *Top = Node;
Step 3: Stop.

Pop Operation:
Top & Temp pointer of structure Stack.
Pop(Top)
Step 1:If Top != NULL
Then
i) Temp = *Top;
ii) *Top = (*Top)->Next;
Step 2: Else Stack is Empty.
Step 3: return Temp;

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"

Infix expression: a+b

OUTPUT:

Prefix expression: +ab

Postfix expression: ab+

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?

PRACTISE ASSIGNMENTS / EXERCISE / MODIFICATIONS:

1. Write a program to implement stack as an abstract data type using linked list and use this ADT for the following operations.

   Expression conversion

   a) Prefix to Infix

   b) Prefix to Postfix

2. Write a program to implement stack as an abstract data type using linked list and use this ADT for the following operations.

   Expression conversion

   a) Postfix to Infix

   b) Postfix to Prefix

ii) Implementation of circular queue

AIM: Implementation of circular queue using array and perform following operations on it.

i) Add a record ii) Delete a record iii) Checking Empty iv) Checking Underflow v) Checking overflow

OBJECTIVE:

1. To understand the concept of Queue.
2. To understand the concept of Circular Queue.
3. 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. init - intialize queue Q, reset all variables.

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 Circular Queue ? Explain with example.

In a standard queue data structure re-buffering problem occurs for each dequeue operation. To solve this problem by joining the front and rear ends of a queue to make the queue as a circular queue. Circular queue is a linear data structure. It follows FIFO principle.

- In circular queue the last node is connected back to the first node to make a circle.
- Circular linked list fallow the First In First Out principle
- Elements are added at the rear end and the elements are deleted at front end of the queue
- Both the front and the rear pointers points to the beginning of the array.
- It is also called as "Ring buffer".
- Items can inserted and deleted from a queue in O(1) time.

Circular Queue can be created in three ways they are

· Using single linked list

· Using double linked list

· Using arrays

Using single linked list:

It is an extension for the basic single linked list. In circular linked list Instead of storing a Null value in the last node of a single linked list, store the address of the 1st node (root) forms a circular linked list. Using circular linked list it is possible to directly traverse to the first node after reaching the last node.

Using double linked list

In double linked list the right side pointer points to the next node address or the address of first node and left side pointer points to the previous node address or the address of last node of a list. Hence the above list is known as circular double linked list.

Using array

In arrays the range of a subscript is 0 to n-1 where n is the maximum size. To make the array as a circular array by making the subscript 0 as the next address of the subscript n-1 by using the formula subscript = (subscript +1) % maximum size. In circular queue the front and rear pointer are updated by using the above formula.

ALGORITHM:

Enqueue operation using array

Step 1. start

Step 2. if (front == (rear+1)%max)

Print error “circular queue overflow “

Step 3. else

{ rear = (rear+1)%max

Q[rear] = element;

If (front == -1) f = 0;

}

Step 4. stop

Dequeue operation using array

Step 1. start

Step 2. if ((front == rear) && (rear == -1))

Print error “circular queue underflow “

Step 3. else

{ element = Q[front]

If (front == rear) front=rear = -1

Else

Front = (front + 1) % max

}

Step 4. stop

INPUT:

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

OUTPUT:

Display elements of Circular queue.

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

PRACTISE ASSIGNMENTS / EXERCISE / MODIFICATIONS:

1. WAP to implement Queue.
2. WAP to implement Circular queue using array to perform the following Operations.

i) Addition of the element in queue.

ii) Deletion of the element from queue.

iii) Display front element of queue.

iv) Display rear element of queue.

3. Implementation of Priority Queue

AIM: Implement priority queue as ADT using multiple linked lists ,one list for each priority 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 Queue.

2) How data structures Queue is represented as an ADT.

THEORY:

4) 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. init - intialize queue Q, reset all variables.

5) 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.

6) 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 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.

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).

OUTPUT:

Priority queue cater services to the patients based on priorities.

FAQ:

8. What are the types of data structure?
9. What are the operations can implement on queue?
10. What is circular queue?
11. What is Multiqueue?
12. Explain importance of stack in recursion
13. Explain Implicit & Explicit Stack.
14. Applications of stack and Queue as a data structure

PRACTISE ASSIGNMENTS / EXERCISE / MODIFICATIONS:

3. WAP to implement Queue.
4. WAP to implement Circular queue using array to perform the following Operations.

i) Addition of the element in queue.

ii) Deletion of the element from queue.

iii) Display front element of queue.

iv) Display rear element of queue.

4. Expression tree Traversals

AIM: Write a program to Construct and expression tree from postfix/prefix expression and perform recursive and non-recursive Inorder , preorder and postorder 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.

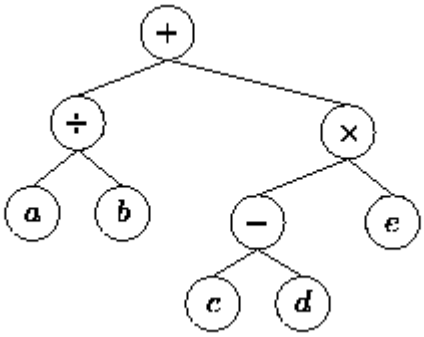


Figure: Tree representing the expression a/b+(c-d)e.

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.

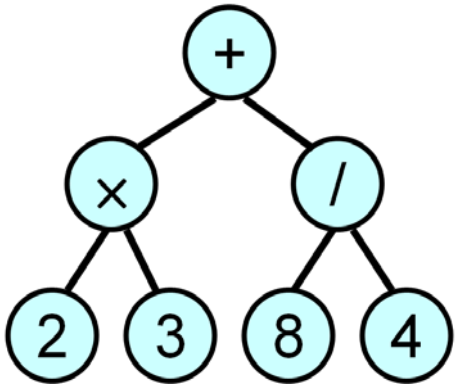
To traverse a non-empty binary tree in **inorder**:

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 Binary 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;
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->Right;
Step 7: Temp = Stack[Top --]; //Pop Element
Step 8: Temp = Temp->Left;

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;

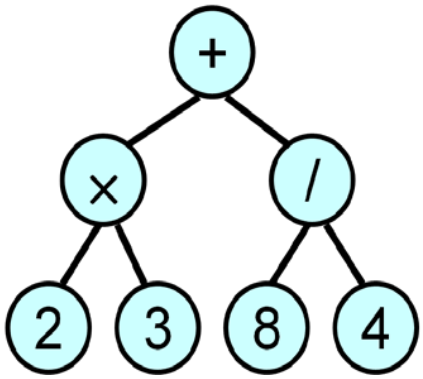
INPUT:

Postfix Expression: **2 3 × 8 4 / +**

OUTPUT:

Display result of each operation with error checking.

Expression tree



FAQS:

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

PRACTISE ASSIGNMENTS / EXERCISE / MODIFICATIONS:

1. Write a program to construct a binary tree and perform recursive and non-recursive traversals.
2. Write a program to accept a prefix expression and construct a Expression tree and perform recursive and non-recursive traversals.

5. Operations on Binary search tree

AIM: Implement binary search tree as an ADT. Perform following operations: a) Insert, b) delete, c) depth of the tree, d) search a node, e) Find its mirror image f) Print original g) mirror image 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?

PRACTISE ASSIGNMENTS / EXERCISE / MODIFICATIONS:

1. Create a binary search tree of mnemonics from assembly language(e.g. add, mult, div, sub etc.) and perform following operations:

    a. Find parent and children of a given node.
    b. Search for a node and display the path from root to the node.
    c. Count leaf nodes.

2. Write a program to construct a binary threaded tree and perform recursive and non-recursive traversals.

6. Threaded Binary tree Traversals

AIM: Write a program to Construct an inorder threaded binary tree from inorder/postorder expression and traverse it in inorder and preorder

OBJECTIVE:

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

THEORY:

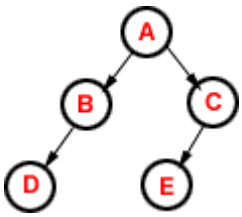
Explain an inorder threaded binary tree.

A Threaded Binary Tree is a binary tree in which every node that does not have a right child has a THREAD (in actual sense, a link) to its INORDER successor. By doing this threading we avoid the recursive method of traversing a Tree, which makes use of stacks and consumes a lot of memory and time.

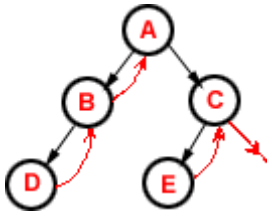
The node structure for a threaded binary tree varies a bit and its like this --

```
struct NODE
{
 struct NODE *leftchild;
 int node_value;
 struct NODE *rightchild;
 struct NODE *thread;
}
```

Let's make the Threaded Binary tree out of a normal binary tree...



The INORDER traversal for the above tree is -- D B A E C. So, the respective Threaded Binary tree will be –



B has no right child and its inorder successor is A and so a thread has been made in between them. Similarly, for D and E. C has no right child but it has no inorder successor even, so it has a hanging thread.

A **threaded binary tree** defined as follows:

"A binary tree is *threaded* by making all right child pointers that would normally be null point to the inorder successor of the node, and all left child pointers that would normally be null point to the inorder predecessor of the node."[1]

A threaded binary tree makes it possible to traverse the values in the binary tree via a linear traversal that is more rapid than a recursive in-order traversal. It is also possible to discover the parent of a node from a threaded binary tree, without explicit use of parent pointers or a stack, albeit slowly. This can be useful where stack space is limited, or where a stack of parent pointers is unavailable (for finding the parent pointer via DFS).

To see how this is possible, consider a node *k* that has a right child *r*. Then the left pointer of *r* must be either a child or a thread back to *k*. In the case that *r* has a left child, that left child must in turn have either a left child of its own or a thread back to *k*, and so on for all successive left children. So by following the chain of left pointers from *r*, we will eventually find a thread pointing back to *k*. The situation is symmetrically similar when *q* is the left child of *p*—we can follow *q*'s right children to a thread pointing ahead to *p*.

Non recursive Inorder traversal for a Threaded Binary Tree

As this is a non-recursive method for traversal, it has to be an iterative procedure; meaning, all the steps for the traversal of a node have to be under a loop so that the same can be applied to all the nodes in the tree. I'll consider the INORDER traversal again. Here, for every node, we'll visit the left sub-tree (if it exists) first (if and only if we haven't visited it earlier); then we visit (i.e print its value, in our case) the node itself and then the right sub-tree (if it exists). If the right sub-tree is not there, we check for the threaded link and make the threaded node the current node in consideration.

Show the different type of traversals with example

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

Visit the root.

Traverse the left subtree.

Traverse the right subtree.

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

Traverse the left subtree.

Visit the root.

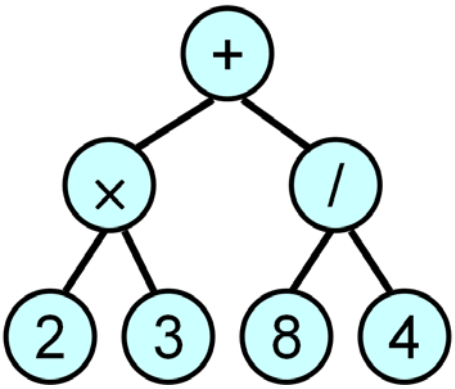
Traverse the right subtree.

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

Traverse the left subtree.

Traverse the right subtree.

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:

Step-1: For the current node check whether it has a left child which is not there in the visited list. If it has then go to step-2 or else step-3.
Step-2: Put that left child in the list of visited nodes and make it your current node in consideration. Go to step-6.
Step-3: For the current node check whether it has a right child. If it has then go to step-4 else go to step-5
Step-4: Make that right child as your current node in consideration. Go to step-6.
Step-5: Check for the threaded node and if its there make it your current node.
Step-6: Go to step-1 if all the nodes are not over otherwise quit

INPUT:

Normal binary tree

OUTPUT:

Display result of each operation with error checking.

FAQS:

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

PRACTISE ASSIGNMENTS / EXERCISE / MODIFICATIONS:

1. Write a program to construct a binary tree and perform recursive and non-recursive traversals.
2. Write a program to accept a prefix expression and construct a Expression tree and perform recursive and non-recursive traversals.

7. Implementation of Prim's & Kruskal's algorithms

AIM: Represent any real world graph using adjacency list/adjacency matrix find minimum spanning tree using Prim's or 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 $G = (V,E, \varphi)$ 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 $G = (V,E, \varphi)$ be a graph and $v \in V$ a vertex.

   Define the degree of v, d(v) to be the number of $e \in E$ such that $v \in \varphi(e)$; i.e., e is Incident on v.

   Directed graph: A directed graph (or digraph) is a triple $D = (V,E, \varphi)$

   where V and E are finite sets and φ is a function with domain E and codomain $V \times V$. We call E the set of edges of the digraph D and call V the set of vertices of D.

   Path: Let $G = (V,E, \varphi)$ be a graph.

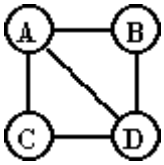
   Let $e1, e2, \ldots, en-1$ be a sequence of elements of E (edges of G) for which there is a sequence $a1, a2, \ldots,$ an of distinct elements of V (vertices of G) such that $\varphi(ei) = \{ai, ai+1\}$ for $i = 1, 2, \ldots, n - 1$. The sequence of edges $e1, e2, \ldots, en-1$ is called a path in G. The sequence of vertices $a1, a2, \ldots,$ an is called the vertex sequence of the path.

   Circuit and Cycle: Let $G = (V,E, \varphi)$ be a graph and let $e1, \ldots,$ en be a trail with vertex sequence $a1, \ldots,$ an, a1. (It returns to its starting point.) The subgraph G′ of G induced by the set of edges $\{e1, \ldots, en\}$ is called a circuit of G. The length of the circuit is n.

2. Different representations of graph.

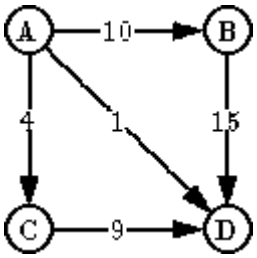
Adjacency matrix:

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

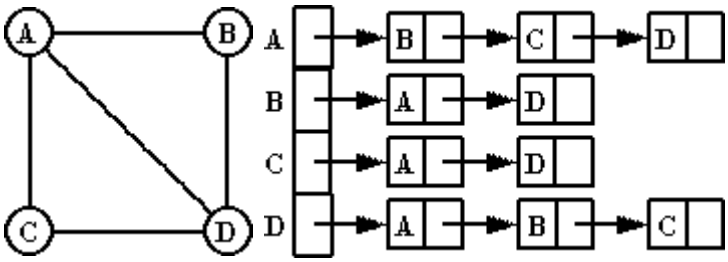
In case of labeled graphs, the labels themselves may be introduced into the entries.



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

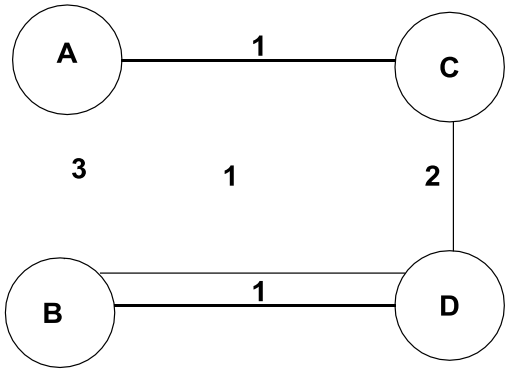
Adjacency List:

A representation of the graph consisting of a list of nodes, with each node containing a list of its neighboring nodes.



3. Explain Prim's & Kruskal's Algorithm:

In computer science, Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected weighted undirected 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. Other algorithms for this problem include Kruskal's algorithm and Borůvka's algorithm. However, these other algorithms can also find minimum spanning forests of disconnected graphs, while Prim's algorithm requires the graph to be connected. Kruskal's Algorithm may also be used to find a minimum spanning tree, but this considers the weights themselves rather than the connecting points



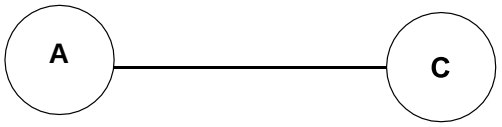
Node and Edge List:

Node A, edges to: B(3) C(1)
Node B, edges to: A(3) C(1) D(1)
Node C, edges to: A(1) B(1) D(2)
Node D, edges to: B(1) C(2)

MinHeap: AC(1) | AB(3)
C off: AB(3) — then enter CB(1), CD(2)

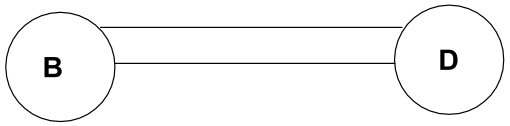
MinHeap: CB(1) | AB(3) CD(2)
B off: CD(2) | AB(3) — then add BD(1)

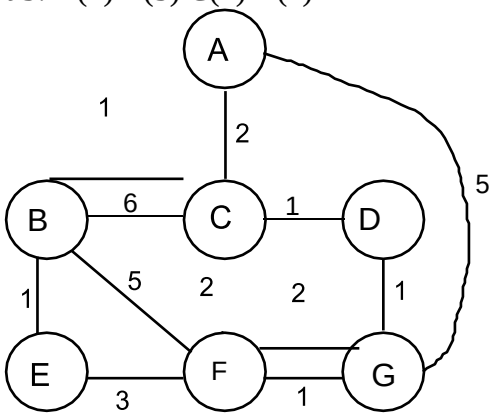
MinHeap: BD(1) | AB(3) CD(2)
D off: CD(2) | AB(3)



Drop D : AB(3)
Drop B : <empty>

Prim's min.spanning tree
A AC CB BD
Weight 3: A(1) B(3) C(2) D(4)





Note: This graph has a shape *somewhat* similar to the one for assignment 9, but the different edges and different edge weights will make Prim's Algorithm behave very differently on this graph than it does on the homework assignment.

Node and Edge List:
Node A, edges to: B(1) C(2) G(5)
Node B, edges to: A(1) C(6) E(1) F(5)
Node C, edges to: A(2) B(6) D(1) F(2)
Node D, edges to: C(1) F(2) G(1)
Node E, edges to: B(1) F(3)
Node F, edges to: B(5) C(2) D(2) E(3) G(1)
Node G, edges to: A(5) D(1) F(1)

MinHeap: AB(1) | AC(2) AG(5)
B off: AC(2) | AG(5) — then enter BC(6), BE(1), BF(5)

MinHeap: BE(1) | AC(2) BC(6) | AG(5) BF(5)
E off: AC(2) | BF(5) BC(6) | AG(5) — then enter EF(3)

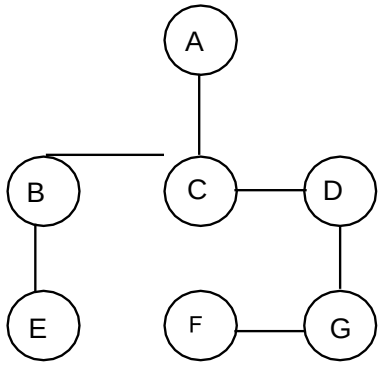
MinHeap: AC(2) | EF(3) BC(6) | AG(5) BF(5)
C off: EF(3) | BF(5) BC(6) | AG(5) — then enter CD(1), CF(2)

MinHeap: CD(1) | EF(3) CF(2) | AG(5) BF(5) BC(6)
D off: CF(2) | EF(3) BC(6) | AG(5) BF(5) — then enter DF(2), DG(1)

MinHeap: DG(1) | EF(3) CF(2) | AG(5) BF(5) BC(6) DF(2) |
G off: DF(2) | EF(3) CF(2) | AG(5) BF(5) BC(6) — then enter GF(1)

MinHeap: GF(1) | EF(3) DF(2) | AG(5) BF(5) BC(6) CF(2)
F off: CF(2) | EF(3) DF(2) | AG(5) BF(5) BC(6)

At this point, all vertices are in the MST, so that successive removals will be discarded:



Prim's min.spanning tree
A AB BE AC CD DG GF
Weight 7: A(1) B(2) C(4) D(5) E(3) F(7) G(6)

A subgraph T of a undirected graph G is a spanning tree of _ if it is a tree and contains every vertex of G. A Minimum Spanning Tree in an undirected connected weighted graph is a spanning tree of minimum weight (among all spanning trees). The *Prim's* algorithm makes a nature choice of the cut in each iteration – it grows a single tree and adds a light edge in each iteration.

Algorithm:

Prim's Algorithm:

Step 1: Select any node to be the first node of T.
Step 2: Consider the arcs which connect nodes in T to nodes outside T. Pick the one with minimum weight. Add this arc and the extra node to T. (If there are two or more arcs of minimum weight, choose any one of them.)
Step 3: Repeat Step 2 until T contains every node of the graph.

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.

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?

PRACTISE ASSIGNMENTS / EXERCISE / MODIFICATIONS:

1. WAP to implement BFS & DFS Graph Traversal techniques.

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.

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.

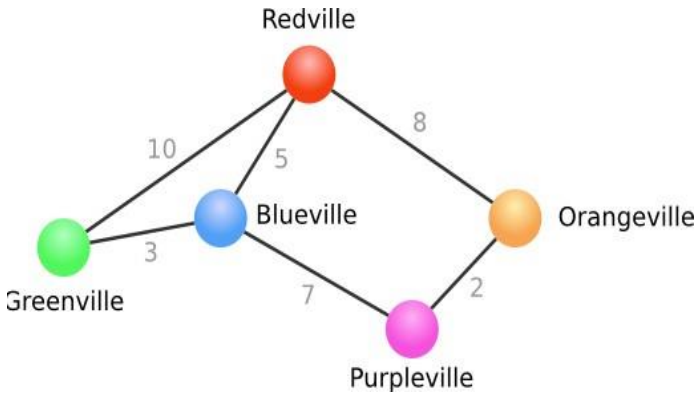
3. Dijkstra's algorithm finds the shortest path from x to y in order of increasing distance from x. That is, it chooses the first minimum edge, stores this value and adds the next minimum value from the next edge it selects.

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.



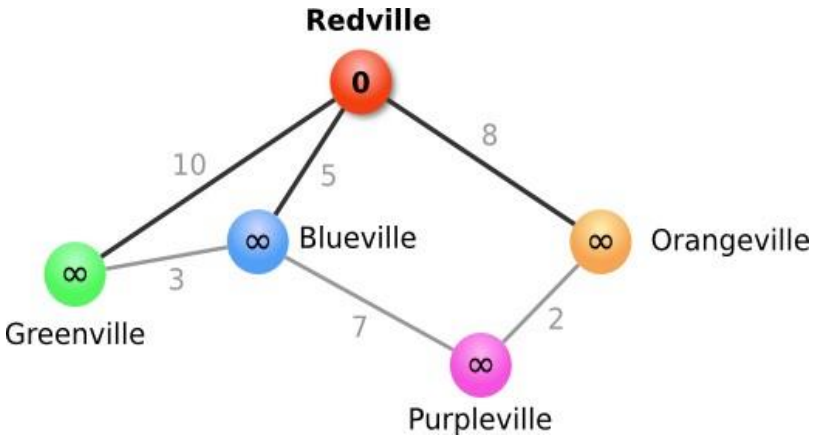
Imagine you live in Redville, and would like to know the shortest way to get to the surrounding towns: Greenville, Blueville, Orangeville, and Purpleville. You would be confronted with problems like: Is it faster to go through Orangeville or Blueville to get to Purpleville? Is it faster to take a direct 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.

1. 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.
2. 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.
3. 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.

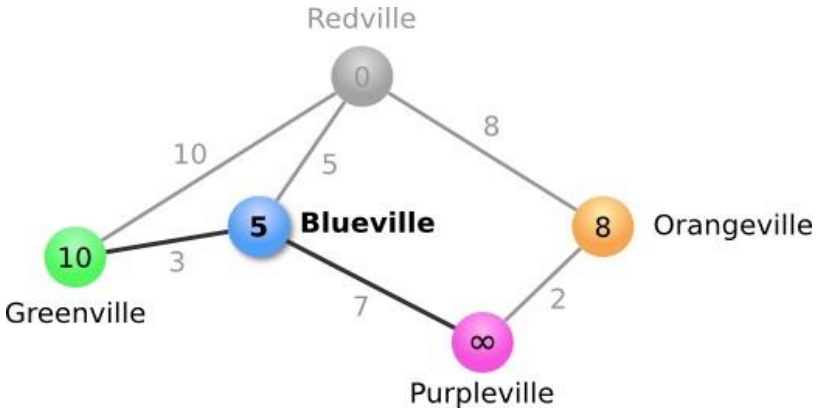
Step I:

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.



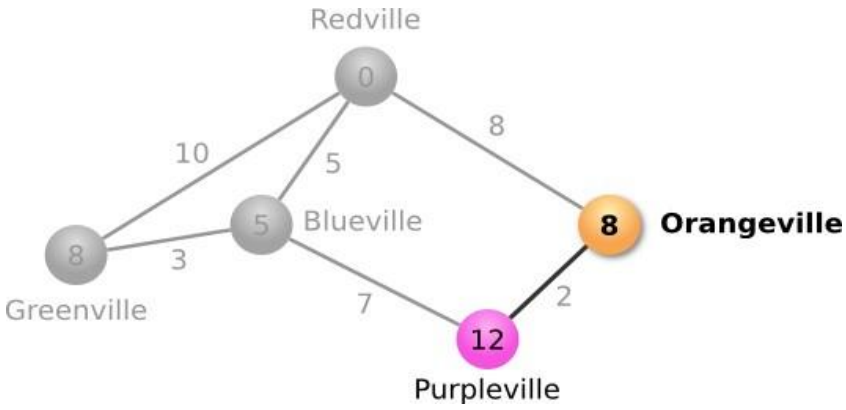
Step II:

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.



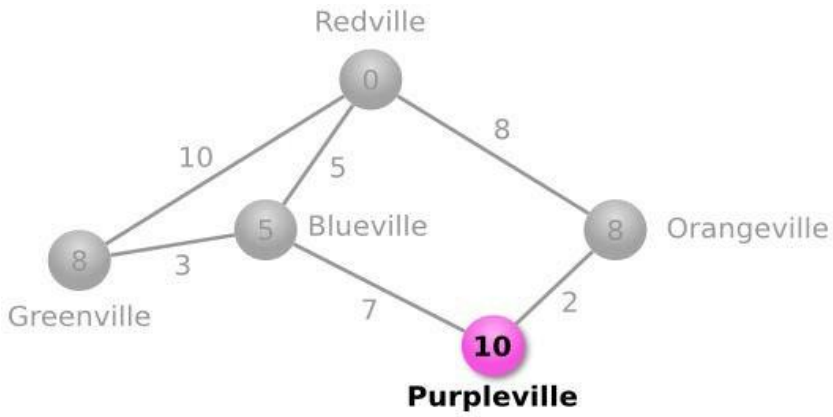
Step III:

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

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?

PRACTISE ASSIGNMENTS / EXERCISE / MODIFICATIONS:

1. WAP to implement Travelling salesman problem using
   Nearest neighbor method

9. Implementation of Hash Table

AIM: Implementation of Hash table using array and handle collisions using Linear probing, chaining without replacement and Chaining with replacement.

OBJECTIVE:

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

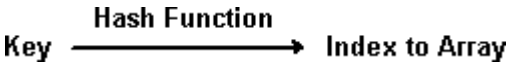
THEORY:

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 transfer 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

2. Characteristics of a good hash function.

There are four main characteristics of a good hash function:

1) The hash value is fully determined by the data being hashed.

2) The hash function uses all the input data.

3) The hash function "uniformly" distributes the data across the entire set of possible

hash values.

4) The hash function generates very different hash values for similar strings.

3. 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 Figur). 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.

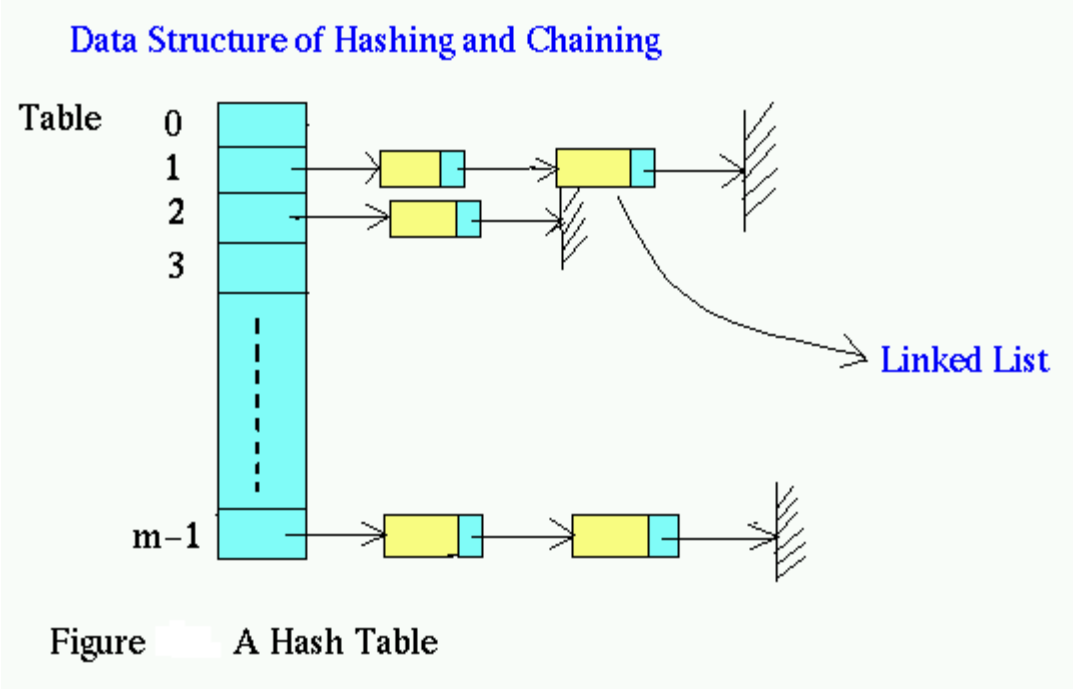


Figure A Hash Table

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.

4. 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 $\mathbf{k>1}$. 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.

Linear Probing With Replacement :

WO_Chaing_W_Replacement(HashTable,KeyValue)
HashTable is one dimensional integer array, which store primary key.
KeyValue is a variable of type integer,it's primary key. Position is integer variable, it is Empty position in Hash Table.
Step 1: Location = Hash(KeyValue);
Step 2: If Hash_Table[Location] is Empty
Then insert KeyValue at Locetion;
Step 3: Else If Location = = Hash(Hash_Table[Location]
Then Insert KeyValue to Empty Location

Else Move HashTable[Location] Key to Empty Location

and insert KeyValue at Location
Step 4: Stop.

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

INSTRUCTIONS:

1. Use of user-defined functions for every operation in mandatory.
2. Proper documentation of the program is a must.

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. Implementation of Heap sort

AIM: Implement Heap sort by constructing max or min heap.

OBJECTIVE:

1. Understand the concept and applications of heap sort.

THEORY:

What is a heap sort?

Heapsort is a comparison-based sorting algorithm to create a sorted array (or list), and is part of the selection sort family. Although somewhat slower in practice on most machines than a well- implemented quicksort, it has the advantage of a more favorable worst-case O(n log n) runtime. Heapsort is an in-place algorithm, but it is not a stable sort.

The heapsort algorithm can be divided into two parts. In the first step, a heap is built out of the data. In the second step, a sorted array is created by repeatedly removing the largest element from the heap, and inserting it into the array. The heap is reconstructed after each removal. Once all objects have been removed from the heap, we have a sorted array. The direction of the sorted elements can be varied by choosing a min-heap or max-heap in step one. Heapsort can be performed in place. The array can be split into two parts, the sorted array and the heap.

The (Binary) heap data structure is an array object that can be viewed as a nearly complete binary tree. A binary tree with n nodes and depth k is complete iff its nodes correspond to the nodes numbered from 1 to n in the full binary tree of depth k

Heap properties

There are two kind of binary heaps: max-heaps and min-heaps.

In a max-heap, the max-heap property is that for every node I other than the root, A[PARENT(i)

] ≥ A[i] .

the largest element in a max-heap is stored at the root

the subtree rooted at a node contains values no larger than that contained at the node itself

In a min-heap, the min-heap property is that for every node I other than the root, A[PARENT(i)

] ≤ A[i] .

the smallest element in a min-heap is at the root

the subtree rooted at a node contains values no smaller than that contained at the node itself

Comparison with other sorts

Heapsort primarily competes with quicksort, another very efficient general purpose nearly-in-place comparison-based sort algorithm. Quicksort is typically somewhat faster due to some factors, but the worst-case running time for quicksort is $O(n^2)$, which is unacceptable for large data sets and can be deliberately triggered given enough knowledge of the implementation, creating a security risk. See quicksort for a detailed discussion of this problem and possible solutions. Thus, because of the $O(n \log n)$ upper bound on heapsort's running time and constant upper bound on its auxiliary storage, embedded systems with real-time constraints or systems concerned with security often use heapsort.

Heapsort also competes with merge sort, which has the same time bounds. Merge sort requires $\Omega(n)$ auxiliary space, but heapsort requires only a constant amount. Heapsort typically runs faster in practice on machines with small or slow data caches. On the other hand, merge sort has several advantages over heapsort:

- Merge sort on arrays has considerably better data cache performance, often outperforming heapsort on modern desktop computers because merge sort frequently accesses contiguous memory locations (good locality of reference); heapsort references are spread throughout the heap.
- Heapsort is not a stable sort; merge sort is stable.
- Merge sort parallelizes well and can achieve close to linear speedup with a trivial implementation; heapsort is not an obvious candidate for a parallel algorithm.
- Merge sort can be adapted to operate on linked lists with O(1) extra space.Heapsort can be adapted to operate on doubly linked lists with only O(1) extra space overhead.
- Merge sort is used in external sorting; heapsort is not. Locality of reference is the issue.

Introsort is an interesting alternative to heapsort that combines quicksort and heapsort to retain advantages of both: worst case speed of heapsort and average speed of quicksort.

The heapify function can be thought of as building a heap from the bottom up, successively shifting downward to establish the heap property. An alternative version (shown below) that builds the heap top-down and sifts upward may be conceptually simpler to grasp. This "siftUp" version can be visualized as starting with an empty heap and successively inserting elements, whereas the "siftDown" version given above treats the entire input array as a full, "broken" heap and "repairs" it starting from the last non-trivial sub-heap (that is, the last parent node).

Difference in time complexity between the "siftDown" version and the "siftUp" version.

Also, the "siftDown" version of heapify has $O(n)$ time complexity, while the "siftUp" version given below has $O(n \log n)$ time complexity due to its equivalence with inserting each element, one at a time, into an empty heap.[8] This may seem counter-intuitive since, at a glance, it is apparent that the former only makes half as many calls to its logarithmic-time sifting function as the latter; i.e., they seem to differ only by a constant factor, which never has an impact on asymptotic analysis.

To grasp the intuition behind this difference in complexity, note that the number of swaps that may occur during any one siftUp call *increases* with the depth of the node on which the call is made. The crux is that there are many (exponentially many) more "deep" nodes than there are "shallow" nodes in a heap, so that siftUp may have its full logarithmic running-time on the approximately linear number of calls made on the nodes at or near the "bottom" of the heap. On the other hand, the number of swaps that may occur during any one siftDown call *decreases* as the depth of the node on which the call is made increases. Thus, when the "siftDown" heapify begins and is calling siftDown on the bottom and most numerous node-layers, each sifting call will incur, at most, a number of swaps equal to the "height" (from the bottom of the heap) of the node on which the sifting call is made. In other words, about half the calls to siftDown will have at most only one swap, then about a quarter of the calls will have at most two swaps, etc.

ALGORITHM:

Heapsort(*a*)

1. Build-max-heap(*a*)

2. **for** *i* ← *length*[*a*] **downto** 2

3. **do** exchange a[1] a[*i*]

4. *heap-size*[a] ← *heap-size*[a] −1

5. max-heapify(*a*, 1)

6. step 4: stop.

heapify(A,i)

1. if (i ≠ l)
2. exchange A[i] with A[l]
3. Heapify(A, l)

Build-max-heap(A)

1. for i = n/2 downto 1
2. Heapify(A, i)

INPUT:

Sequence of elements will be passed as input.

OUTPUT:

Sorted list of elements.

INSTRUCTIONS:

Use of user-defined functions for every operation in mandatory.

Proper documentation of the program is a must.

FAQS:

What is max heap?

What is min heap?

What are the advantages & disadvantages of heap sort?

11. Implementation of index sequential file

AIM: Implement an index sequential file for any Database and perform following operations on it i) Create Database ii) Display Database iii) Add a record iv) Delete a record v)Modify a record.

OBJECTIVE:

1. To understand the implementation of index sequential files.

THEORY:

1. What is file?

   A file is a collection of data stored in one unit, or resource for storing information identified by a filename, which is available to a computer program and is usually based on some kind of durable storage. A file is durable in the sense that it remains available for programs to use after the current program has finished. It can be a document, picture, audio or video stream, data library, application, or other collection of data

2. What is a Sequential file and different operations on it?

   Consider example of a tape where the songs are stored sequentially . whenever we play any song from it , we read it sequentially. Storing data sequentially is the simplest form of file. It is the most common type of file.

   Sequential file is organized by writing the records in the serial order. Thus the first record in the file is the oldest entry & last record corresponds to most recent entry. The record contained in sequential file may be either of fixed or of varying size.

   Operations on sequential file:

   1) Open: It sets the file pointer to immediately before the first records and allow to access the file.

   2) Read : Set the file pointer to the next record in the file and return that record to the user. If there is no record present, then an 'end of file' condition is set.

   3) Write/Update: Write backs the current record in the same location where it was read from , with entries modified. The file pointer is unchanged.

   4) Append: Position the file pointer to the next of last record and there copies the new record.

   5) Close: Terminate the access to the file.
   6) EOF: Returns a value of true if the file pointer is beyond the last record of the File, otherwise it is false.

3. Advantages/Disadvantages of Sequential files.
   Advantages:

1) Simplest organization
2) Minimum overload on disk
3) Most versatile format
4) Allow back spanning

Disadvantages:

1) To get a particular record, it must access all records before it
2) No random access by key.

Index Sequential file:

An *index* is an auxiliary structure for a file that consists of an ordered sequence of value-pointer pairs, or, more generally, an ordered sequence of value-tuple/pointer-list pairs. A value-pointer pair, for example, might be a customer-ID value and a disk-address pointer that points to the block that includes the customer-ID value. A sequence of these pairs ordered by customer ID would be an index. More generally, the value can be a value-tuple because we may be searching on a composite key such as *Name, Address* or we may be searching on several attribute values such as *Customer ID* and *Date* to find an order. Since a value-tuple is also a value, we need not and do not distinguish these two cases in our discussion. The pointer part is more generally a pointer list because there may be more than one record in more than one block that contains the value of interest. As a specific example, we could have an index on *Customer ID* for our *Order* relation, which we discussed in Figure 2.10 in the last chapter. This would help us quickly find all orders placed by a customer even though these orders may be in several different blocks. The general idea of an index is simple, but there are a number of factors to consider. One is whether the index itself is stored on disk or in main memory. If it is on disk, we must consider the number of disk accesses required to read the index into main memory.

The number of disk accesses depends on how large the index is. If it is particularly large, we may organize it as a two or three-level, or in general as an *n*-level, index tree so that we need not read all the blocks of the index to find the index value we are seeking. Pointers in the upper-level blocks of an index tree point to index blocks, and pointers in the leaf-level blocks point to file blocks. When the values in the index pairs are primary keys, we call an index a *primary-key index* or sometimes just a *primary index*. When the values are not primary keys, we call an index a *secondary-key index* or often just a *secondary index*. (Note that secondary keys may not be keys in the sense that they uniquely identify a tuple, but instead are merely keys used for accessing records.) If the file is sorted, it is almost always sorted on its primary key. Indeed, if it is sorted on any other key, for all intents and purposes we may as well consider it to be the primary key. Files sorted on a nonkey value are rare. If a file is sorted on a key, the index can be *sparse*, which means that it has a value-pointer pair for only one record in each block of the file. An index that has pointers for every record is called a *dense index*. A sequential file sorted on its primary key, along with a sparse index on the primary key, is called an *indexed sequential file*.

ALGORITHM:

Declare Data structure for given Data(Employee/Student)

Insert:

Insert(File Name)

Step 1: Open File

Step 2: Accept Data

Step 3: Write Data in a Database File & Index File.

Step 4: Close File.

Step 5: Stop.

Search:

Search(File Name, KeyValue)

Step 1: Open Index File in read mode

Step 2: Read offset value.

Step 3: Read Data from Database File.

Step 4: If Database Key = = KeyValue

Then Record Present ,Display Record,Exit.

Step 5: Do Step 2 & 3 Till End of File

Step 6: Close File.

Step 7: Stop.

Delete:

Delete(File Name,KeyValue)//Logical Delete

Step 1: Open index File in read & write mode

Step 2: Read offset value.

Step 3: Read Data from Database File.

Step 4: If Database Key = = KeyValue

Then Record Present ,Seek File pointer at Record Location,

Set Status to -1, write Record in a File ,Exit.

Step 5: Do Step 2 & 3 till end of file

Step 6: Close File.

Step 7: Stop.

Modify:

Modify(File Name,KeyValue)

Step 1: Open Index File in read & write mode

Step 2: Read offset value.

Step 3: Read Data from Database File.

Step 4: If Database Key = = KeyValue

Then Record Present ,Seek File pointer at Record Location,

Accept New Data , write Record in a File ,Exit.

Step 5: Do Step 2 & 3 Till End of File

Step 6: Close File.

Step 7: Stop.

INPUT:

Test Cases	O/P
1. Enter File name	
File Error	Display Message "File Opening Error"
2. Emp no <= 0	Display Message "Invalid Emp. No."
3. Salary <=0	Display Message "Invalid Salary."

OUTPUT:

Display result of each operation with error checking.

INSTRUCTIONS:

1. Use of user-defined functions for every operation in mandatory.
2. Proper documentation of the program is a must.

FAQS:

What is file? What type of data stored in file?

What are the different file opening modes?

What are different file accessing methods?

Compare sequential ,index sequential & random access file.

PRACTISE ASSIGNMENTS / EXERCISE / MODIFICATIONS:

WAP to read & write a character in a file.

WAP to read & write & search a string in a file