Vidyavardhini's
College of Engineering & Technology
Vasai Road (W)

Department of
Artificial Intelligence and Data Science

Laboratory Manual

Semester	III	Class	SE
Course No.	CSL303		
Course Name	Data Structure Lab		

Vidyavardhini's College of Engineering & Technology

Vision

To be a premier institution of technical education; aiming at becoming a valuable resource for industry and society.

Mission

We at VCET aim

- To provide technologically inspiring environment for learning.
- To promote creativity, innovation and professional activities.
- To inculcate ethical and moral values.
- To cater personal, professional and societal needs through quality education.

Department Vision:

To evolve as a centre of excellence in the field of Computer Engineering to cater the industrial & societal needs.

Department Mission:

- To provide quality technical education with the aid of modern resources.
- To inculcate creative thinking through innovative ideas and project development.
- To encourage life-long learning, leadership skills, entrepreneur skills with ethical & moral values.

Program Education Objectives (PEOs):

The Program Educational Objectives (PEOs) of Bachelors Degree program in Computer Engineering are:

PEO1: To facilitate learners with a sound foundation in the mathematical, scientific and engineering fundamentals to accomplish professional excellence and succeed in higher studies in computer engineering domain.
PEO2: To enable learners to use modern tools effectively to solve real life problems in the field of computer engineering.
PEO3: To equip learners with extensive education necessary to understand the impact of computer technology in a global and social context.
PEO4: To inculcate professional and ethical attitude, leadership qualities, commitment to societal responsibilities and prepare the learners for life-long learning to built up a successful career in Computer Engineering.

Program Specific Outcomes (PSOs):

The Program Specific Outcomes (PEOs) of Bachelors Degree program in Computer Engineering are:

PSO1: Analyze problems and design applications of database, networking, security, web technology, cloud computing, machine learning using mathematical skills and computational tools

PSO2: Develop computer based systems to provide solutions for organizational, societal problems by working in multidisciplinary teams and pursue a career in IT industry

Program Outcomes (POs):

Engineering Graduates will be able to:

- **PO1. Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
- **PO2. Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
- **PO3. Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
- **PO4. Conduct investigations of complex problems:** 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.
- **PO5. Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
- **PO6. The engineer and society:** 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.
- **PO7. Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
- **PO8. Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
- **PO9. Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
- **PO10. Communication:** Communicate effectively on complex engineering activities with the engineering community and with 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.
- **PO11. Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
- **PO12. Life-long learning:** Recognize 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 Objectives

1	To implement basic data structures such as arrays, linked lists, stacks and queues.
2	Solve problem involving graphs, and trees
3	To develop application using data structure algorithms
4	Compute the complexity of various algorithms.

Course Outcomes

At the end of the course student will be able to:		PO/PSO	Bloom Level
CSL301.1	Implement Linear Data Structure and handle insertion, deletion, traversal operations using array.	Implement	Apply(level 3)
CSL301.2	Apply stack operations to convert and evaluate expression	Apply	Apply(level 3)
CSL301.3	Implement linear, circular or priority queues using arrays	Implement	Apply(level 3)
CSL301.4	Implement Singly, Circular or Doubly Linked list	Implement	Apply(level 3)
CSL301.5	Implement ADT using insertion, deletion and searching operations on Binary tree.	Implement	Apply(level 3)
CSL301.6	Implement Graph Traversal Techniques: BFS and DFS.	Implement	Apply(level 3)

Mapping of Experiments with Course Outcomes

Experiment	Course Outcomes					
	CSL 301.1	CSL 301.2	CSL 301.3	CSL. 301.4	CSL 301.5	CSL 301.6
Implement Stack ADT using array.	3	-	-	-	-	-
Convert an Infix expression to Postfix expression using stack ADT.	-	3	-	-	-	-
Evaluate Postfix Expression using Stack ADT.	-	3	-	-	-	-
Implement Linear Queue ADT using array.	-	-	3	-	-	-
Implement Priority Queue ADT using array.	-	-	3	-	-	-
Implement Singly Linked List ADT.	-	-	-	3	-	-
Implement Circular Linked List ADT.	-	-	-	3	-	-
Implement Binary Search Tree ADT using Linked List.	-	-	-	-	3	-
Implement Graph Traversal techniques (any1) – a) Depth First Search b) Breadth First Search	-	-	-	-	-	3
Implement Binary Search method.	-	-	-	-	-	3
Course Project	3	3	3	3	3	3

Enter correlation level 1, 2 or 3 as defined below
1: Slight (Low) 2: Moderate (Medium) 3: Substatial (High)

If there is no correlation put "—".

INDEX

Sr. No.	Name of Experiment	D.O.P.	D.O.C.	Page No.	Remark
1	Implement Stack ADT using array.				
2	Convert an Infix expression to Postfix expression using stack ADT.				
3	Evaluate Postfix Expression using Stack ADT.				
4	Implement Linear Queue ADT using array.				
5	Implement Priority Queue ADT using array.				
6	Implement Singly Linked List ADT.				
7	Implement Circular Linked List ADT.				
8	Implement Binary Search Tree ADT using Linked List.				
9	Implement Graph Traversal techniques (any1) – a) Depth First Search b) Breadth First Search				
10	Implement Binary Search method.				
11	Course Project				

D.O.P: Date of performance

D.O.C : Date of correction

Experiment No.1
Implement Stack ADT using array.
Date of Performance:
Date of Submission:

Experiment No. 1: To implement stack ADT using arrays

Aim: To implement stack ADT using arrays.

Objective:

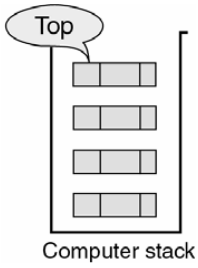
1) Understand the Stack Data Structure and its basic operators.

2) Understand the method of defining stack ADT and implement the basic operators.

3) Learn how to create objects from an ADT and invoke member functions.

Theory:

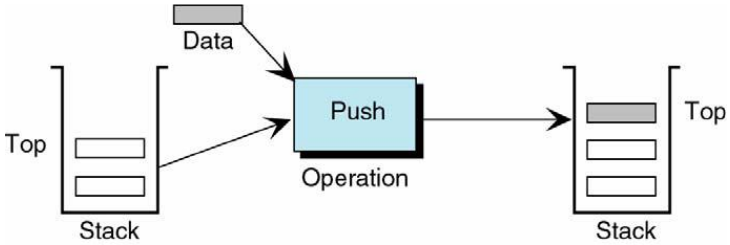
A stack is a list in which all insertions and deletions are made at one end, called the top. It is a collection of contiguous cells, stacked on top of each other. The last element to be inserted into the stack will be the first to be removed. Thus stacks are sometimes referred to as Last In First Out (LIFO) lists.



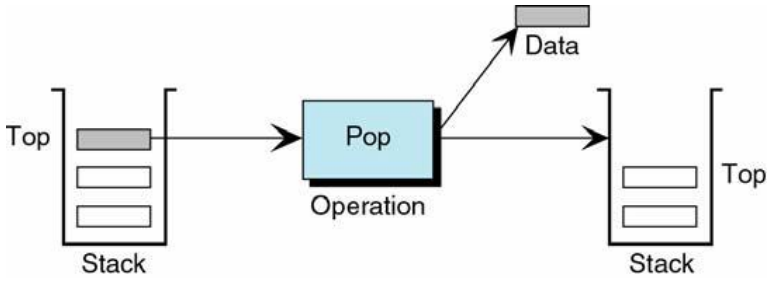
The operations that can be performed on a stack are push, pop which are main operations while auxiliary operations are peek, isEmpty and isFull. Push is to insert an element at the top of the stack. Pop is deleting an element that is at the top most position in the stack. Peek simply examines and returns the top most value in the stack without deleting it.

Push on an already filled stack and pop on an empty stack results in serious errors so isEmpty and isFull function checks for stack empty and stack full respectively. Before any insertion, the value of the variable top is initialized to -1.

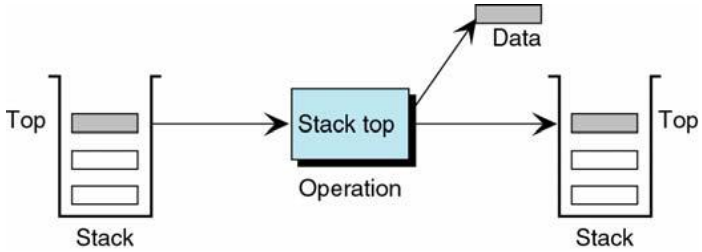
Push Operation



Pop Operation



Peek Operation



Algorithm:

PUSH(item)

1. If (stack is full)

   Print "overflow"

2. top = top + 1
3. stack[top] = item

   Return

POP()

1. If (stack is empty)

   Print "underflow"

2. Item = stack[top]
3. top = top – 1
4. Return item

PEEK()

1. If (stack is empty)

   Print "underflow"

2. Item = stack[top]

3. Return item

ISEMPTY()

1. If(top = -1)then

   return 1

2. return 0

ISFULL()

1. If(top = max)then

   return 1

2. return 0

Code:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100
struct Stack {
    int arr[MAX_SIZE];
    int top;
};
void initialize(struct Stack* stack) {
    stack->top = -1;
}
int isEmpty(struct Stack* stack) {
    return stack->top == -1;
}
int isFull(struct Stack* stack) {
    return stack->top == MAX_SIZE - 1;
}
void push(struct Stack* stack, int value) {
    if (isFull(stack)) {
        printf("Stack overflow!\n");
```

```
        return;
    }
    stack->arr[++stack->top] = value;
}
int pop(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack underflow!\n");
        return -1;
    }
    return stack->arr[stack->top--];
}
int peek(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty!\n");
        return -1;
    }
    return stack->arr[stack->top];
}
void display(struct Stack* stack) {
    if (isEmpty(stack)) {
        printf("Stack is empty!\n");
        return;
    }
    printf("Stack elements: ");
    for (int i = 0; i <= stack->top; ++i) {
        printf("%d ", stack->arr[i]);
    }
    printf("\n");
}
int main() {
    struct Stack myStack;
    initialize(&myStack);
    push(&myStack, 10);
    push(&myStack, 20);
    push(&myStack, 30);
    display(&myStack);
    printf("Top element: %d\n", peek(&myStack));
    printf("Popped element: %d\n", pop(&myStack));
    display(&myStack);
    return 0;
}
```

Output:

```
PS F:\AIDS 3SEM\AIDS_ANKIT_BARI> cd "f:\AIDS 3SEM\AIDS_ANKIT_BARI\" ; if ($?) {
gcc stack_arr.c -o stack_arr } ; if ($?) { .\stack_arr }
Stack elements: 10 20 30
Top element: 30
Popped element: 30
Stack elements: 10 20
PS F:\AIDS 3SEM\AIDS_ANKIT_BARI>
```

Conclusion:

Implementing a stack using arrays in C is a versatile and efficient solution for managing data in a LIFO manner, with applications ranging from function call management to algorithmic problem-solving. Understanding the time complexities of various operations is crucial for assessing the efficiency of the stack implementation in different scenarios.

Time Complexity:

- **Push Operation (Insertion):** O(1)
- **Pop Operation (Deletion):** O(1)
- **Peek Operation (Accessing the Top Element):** O(1)
- **Display Operation:** O(n)

Applications :

- **Function Call Management**
- **Expression Evaluation**
- **Undo Mechanism in Text Editors**
- **Backtracking in Algorithms**

Experiment No.2
Convert an Infix expression to Postfix expression using stack ADT.
Date of Performance:
Date of Submission:

Experiment No. 2: Conversion of Infix to postfix expression using stack ADT

Aim: To convert infix expression to postfix expression using stack ADT.

Objective:

1) Understand the use of Stack.

2) Understand how to import an ADT in an application program.

3) Understand the instantiation of Stack ADT in an application program.

4) Understand how the member functions of an ADT are accessed in an application program.

Theory:

Postfix notation is a way of writing algebraic expressions without the use of parentheses or rules of operator precedence. The expression (A+B)/(C–D) would be written as AB+CD-/ in postfix notation. An expression is scanned from user in infix form; it is converted into postfix form and then evaluated without considering the parenthesis and priority of the operators.

An arithmetic expression consists of operands and operators. For a given expression in an postfix form, stack can be used to evaluate the expression. The rule is whenever an operands comes into the string push it on to the stack and when an operator is found then last two elements from the stack are poped and computed and the result is pushed back on to the stack. One by one whole string of postfix expression is parsed and final result is obtained at an end of computation that remains in the stack.

Conversion of infix to postfix expression

Expression	Stack	Output
2	Empty	2
*	*	2
3	*	23
/	/	23*
(	/(	23*
2	/(	23*2
-	/(-	23*2
1	/(-	23*21
)	/	23*21-
+	+	23*21-/
5	+	23*21-/5
*	+*	23*21-/53
3	+*	23*21-/53
	Empty	23*21-/53*+

So, the Postfix Expression is 23*21-/53*+

Algorithm :

Conversion

1. Read the symbol one at a time from the input expression.
2. If it is operand, output it.
3. If it is opening parenthesis, push it on stack.
4. If it is an operator, then check its incoming priority
5. If stack is empty, push operator on stack.
6. If the top of stack is opening parenthesis, push operator on stack
7. If it has higher priority operator than the top of stack, push operator on stack.
8. Else pop the operator from the stack and output it, repeat step 4
9. If it is a closing parenthesis, pop operators from stack and output them until an opening parenthesis is encountered. pop and discard the opening parenthesis.
10. If there is more input go to step 1
11. If there is no more input, pop the remaining operators to output.

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct Stack {
    char* array;
    int top;
    unsigned capacity;
};
struct Stack* createStack(unsigned capacity) {
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (char*)malloc(stack->capacity * sizeof(char));
    return stack;
}
int isEmpty(struct Stack* stack) {
    return stack->top == -1;
}
```

```
void push(struct Stack* stack, char item) {
    stack->array[++stack->top] = item;
}
char pop(struct Stack* stack) {
    if (!isEmpty(stack))
        return stack->array[stack->top--];
    return '\0';
}
char peek(struct Stack* stack) {
    if (!isEmpty(stack))
        return stack->array[stack->top];
    return '\0';
}
int isOperand(char ch) {
    return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z') || (ch >= '0'
&& ch <= '9');
}
int getPrecedence(char op) {
    switch (op) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        case '^':
            return 3;
    }
    return -1;
}
void infixToPostfix(char* infixExpression) {
    struct Stack* stack = createStack(strlen(infixExpression));
    int i, j;
    for (i = 0, j = -1; infixExpression[i]; ++i) {
        if (isOperand(infixExpression[i]))
            infixExpression[++j] = infixExpression[i];
        else if (infixExpression[i] == '(')
            push(stack, infixExpression[i]);
        else if (infixExpression[i] == ')') {
            while (!isEmpty(stack) && peek(stack) != '(')
                infixExpression[++j] = pop(stack);
            if (!isEmpty(stack) && peek(stack) != '(')
                return;
            else
                pop(stack);
        } else {
            while (!isEmpty(stack) && getPrecedence(infixExpression[i]) <=
getPrecedence(peek(stack)))
```

```
                infixExpression[++j] = pop(stack);
            push(stack, infixExpression[i]);
        }
    }
    while (!isEmpty(stack))
        infixExpression[++j] = pop(stack);

    infixExpression[++j] = '\0';
}
int main() {
    char infixExpression[100];
    printf("Enter infix expression: ");
    scanf("%s", infixExpression);
    infixToPostfix(infixExpression);
    printf("Postfix expression: %s\n", infixExpression);
    return 0;
}
```

Output:

```
PS F:\AIDS 3SEM\AIDS_ANKIT_BARI> cd "f:\AIDS 3SEM\AIDS_ANKIT_BARI\" ; if ($?) {
gcc infix_to_postfix.c -o infix_to_postfix } ; if ($?) { .\infix_to_postfix }
Enter infix expression: (a+b)*(a-b)
Postfix expression: ab+ab-*
PS F:\AIDS 3SEM\AIDS_ANKIT_BARI>
```

Conclusion:

The implementation of infix to postfix conversion using a stack ADT in C provides a clear and efficient way to transform mathematical expressions, making it suitable for various applications in computing and education. The simplicity of the algorithm and its relevance to real-world scenarios make it a useful tool in software development and algorithmic understanding.

Applications:

- **Infix to postfix conversion is a fundamental step in the evaluation of mathematical expressions, making it applicable in calculators, compilers, and mathematical software.**

Experiment No.3
Evaluate Postfix Expression using Stack ADT.
Date of Performance:
Date of Submission:

Experiment No. 3: Evaluation of Postfix Expression using stack ADT

Aim : Implementation of Evaluation of Postfix Expression using stack ADT

Objective:

1) Understand the use of Stack.
2) Understand importing an ADT in an application program.
3) Understand the instantiation of Stack ADT in an application program.
4) Understand how the member functions of an ADT are accessed in an application program

Theory:

An arithmetic expression consists of operands and operators. For a given expression in an postfix form, stack can be used to evaluate the expression. The rule is when ever an operands comes into the string push it on to the stack and when an operator is found then last two elements from the stack are poped and computed and the result is pushed back on to the stack. One by one whole string of postfix expression is parsed and final result is obtained at an end of computation that remains in the stack.

Algorithm

Algorithm : EVAL_POSTFIX

Input : E is an expression in an postfix form.

Output : Result after computing the expression.

Data Structure : An array representation of stack is used with top as a pointer to the top most element.

1. Append # as an delimiter at an end of expression.
2. item = READ_SYMBOL()
3. while item != '#' do

```
if item =operand then
        PUSH(item)
else
        op=item
        y=POP()
        x=POP()
        t=x op y
        PUSH(t)
end if
```

item = READ_SYMBOL()

end while

4. value = POP()

5. stop

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
struct Stack {
    int* array;
    int top;
    unsigned capacity;
};
struct Stack* createStack(unsigned capacity) {
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
    stack->capacity = capacity;
    stack->top = -1;
    stack->array = (int*)malloc(stack->capacity * sizeof(int));
    return stack;
}
int isEmpty(struct Stack* stack) {
    return stack->top == -1;
}
void push(struct Stack* stack, int item) {
    stack->array[++stack->top] = item;
}
int pop(struct Stack* stack) {
    if (!isEmpty(stack))
        return stack->array[stack->top--];
    return -1;
}
int evaluatePostfix(char* postfixExpression) {
    struct Stack* stack = createStack(strlen(postfixExpression));
    int i;
    for (i = 0; postfixExpression[i]; ++i) {
        if (isdigit(postfixExpression[i]))
            push(stack, postfixExpression[i] - '0');
        else {
            int operand2 = pop(stack);
            int operand1 = pop(stack);

            switch (postfixExpression[i]) {
                case '+':
```

```
                    push(stack, operand1 + operand2);
                    break;
                case '-':
                    push(stack, operand1 - operand2);
                    break;
                case '*':
                    push(stack, operand1 * operand2);
                    break;
                case '/':
                    push(stack, operand1 / operand2);
                    break;
            }
        }
    }
    return pop(stack);
}
int main() {
    char postfixExpression[100];
    printf("Enter postfix expression: ");
    scanf("%s", postfixExpression);
    int result = evaluatePostfix(postfixExpression);
    printf("Result of the postfix expression: %d\n", result);
    return 0;
}
```

Output:

```
PS F:\AIDS 3SEM\AIDS_ANKIT_BARI> cd "f:\AIDS 3SEM\AIDS_ANKIT_BARI\" ; if ($?) {
gcc postfix_eval.c -o postfix_eval } ; if ($?) { .\postfix_eval }
Enter postfix expression: ab+ab*
Result of the postfix expression: 1
PS F:\AIDS 3SEM\AIDS_ANKIT_BARI>
```

Conclusion:

The implementation of postfix expression evaluation using a stack ADT in C provides a clear, efficient, and versatile approach for calculating the result of mathematical expressions. The simplicity of the algorithm, combined with its practical applications, makes it a valuable tool for both educational purposes and real-world software development scenarios.

- **Postfix expression evaluation is widely used in calculators, compilers, and other systems where efficient arithmetic calculations are required. It is particularly useful in situations where parentheses are not needed for operator precedence.**

Experiment No.4
Implement Linear Queue ADT using Array
Date of Performance:
Date of Submission:

Experiment No. 4: Implement of Linear Queue ADT using Array

Aim: To implement a Queue using arrays.

Objective:

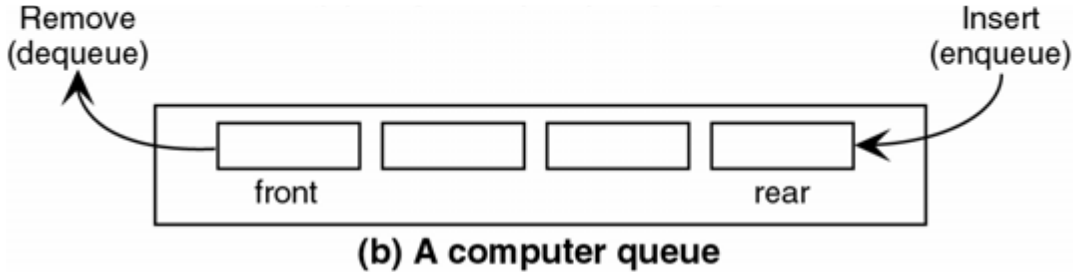
1 Understand the Queue data structure and its basis operations.

2. Understand the method of defining Queue ADT and its basic operations.

3. Learn how to create objects from an ADT and member function are invoked.

Theory:

A Queue is an ordered collection of items from which items may be deleted at one end (called the *front* of the queue) and into which items may be inserted at the other end (the *rear* of the queue). Queues remember things in first-in-first-out (FIFO) order. The basic operations in a queue are: Enqueue - Adds an item to the end of queue. Dequeue - Removes an item from the front



(b) A computer queue

A queue is implemented using a one dimensional array. FRONT is an integer value, which contains the array index of the front element of the array. REAR is an integer value, which contains the array index of the rear element of the array. When an element is deleted from the queue, the value of front is increased by one. When an element is inserted into the queue, the value of rear is increased by one.

Algorithm:

ENQUEUE(item)

1. If (queue is full)

   Print "overflow"

2. if (First node insertion)

   Front++

3. rear++

Queue[rear]=value

DEQUEUE()

1. If (queue is empty)

   Print "underflow"

2. if(front=rear)

   Front=-1 and rear=-1

3. t = queue[front]

4. front++

5. Return t

ISEMPTY()

1. If(front = -1)then

   return 1

2. return 0

ISFULL()

1. If(rear = max)then

   return 1

2. return 0

Code :

```c
#include <stdio.h>
#define MAX 100
typedef struct {
  int arr[MAX];
  int front;
  int rear;
} Queue;
void initQueue(Queue *q) {
  q->front = -1;
  q->rear = -1;
}
int isEmptyQueue(Queue *q) {
  return (q->front == q->rear);
}
int isFullQueue(Queue *q) {
  return (q->rear == MAX - 1);
```

```
}
void enqueue(Queue *q, int data) {
  if (isFullQueue(q)) {
    printf("Queue overflow!\n");
    return;
  }
  q->rear++;
  q->arr[q->rear] = data;
}
int dequeue(Queue *q) {
  if (isEmptyQueue(q)) {
    printf("Queue underflow!\n");
    return -1;
  }
  q->front++;
  int data = q->arr[q->front];
  return data;
}
void displayQueue(Queue *q) {
  if (isEmptyQueue(q)) {
    printf("Queue is empty!\n");
    return;
  }
  printf("Queue elements are: ");
  for (int i = q->front + 1; i <= q->rear; i++) {
    printf("%d ", q->arr[i]);
  }
  printf("\n");
}
int main() {
  Queue q;
  initQueue(&q);
  enqueue(&q, 10);
  enqueue(&q, 20);
  enqueue(&q, 30);
  displayQueue(&q);
  int data = dequeue(&q);
  printf("Dequeued element: %d\n", data);
  displayQueue(&q);
  return 0;
}
```

Output :

```
PS F:\AIDS 3SEM\AIDS_ANKIT_BARI> cd "f:\AIDS 3SEM\AIDS_ANKIT_BARI\" ; if ($?) {
gcc linear_queue.c -o linear_queue } ; if ($?) { .\linear_queue }
Queue elements are: 10 20 30
Dequeued element: 10
Queue elements are: 20 30
PS F:\AIDS 3SEM\AIDS_ANKIT_BARI>
```

Conclusion:

The implementation of a linear queue ADT using an array is a simple and efficient way to implement a queue. It is easy to understand and implement, and it has good performance characteristics.

However, there are some limitations to this implementation. One limitation is that the size of the queue is fixed at compile time. This can be a problem if the queue needs to be able to store a large number of elements. Another limitation is that this implementation can be inefficient if the queue is frequently enqueued and dequeued. This is because the front and rear indices of the queue need to be updated every time an element is enqueued or dequeued.

Overall, the implementation of a linear queue ADT using an array is a good choice for applications where a simple and efficient queue implementation is needed. However, it is important to be aware of the limitations of this implementation before using it in a production application.

Experiment No.5
Implement Priority Queue ADT using array
Date of Performance:
Date of Submission:

Experiment No. 5: Priority Queue

Aim: To Implement Priority Queue ADT using array

Objective:

Circular Queues offer a quick and clean way to store FIFO data with a maximum size

Theory:

A priority queue is a type of queue that arranges elements based on their priority values. Elements with higher priority values are typically retrieved before elements with lower priority values.

In a priority queue, each element has a priority value associated with it. When you add an element to the queue, it is inserted in a position based on its priority value. For example, if you add an element with a high priority value to a priority queue, it may be inserted near the front of the queue, while an element with a low priority value may be inserted near the back..

Operations of a Priority Queue:

A typical priority queue supports the following operations:

1) Insertion in a Priority Queue

When a new element is inserted in a priority queue, it moves to the empty slot from top to bottom and left to right. However, if the element is not in the correct place then it will be compared with the parent node. If the element is not in the correct order, the elements are swapped. The swapping process continues until all the elements are placed in the correct position.

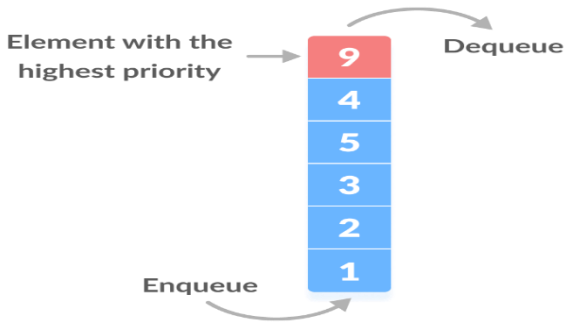
2) Deletion in a Priority Queue

As you know that in a max heap, the maximum element is the root node. And it will remove the element which has maximum priority first. Thus, you remove the root node from the queue. This removal creates an empty slot, which will be further filled with new insertion. Then, it compares the newly inserted element with all the elements inside the queue to maintain the heap invariant.

3) Peek in a Priority Queue

This operation helps to return the maximum element from Max Heap or the minimum element from Min Heap without deleting the node from the priority queue.

Priority Queue



Algorithm

Algorithm : PUSH(HEAD, DATA, PRIORITY):

Step 1: Create new node with DATA and PRIORITY

Step 2: Check if HEAD has lower priority. If true follow Steps 3-4 and end. Else goto Step 5.

Step 3: NEW -> NEXT = HEAD

Step 4: HEAD = NEW

Step 5: Set TEMP to head of the list

Step 6: While TEMP -> NEXT != NULL and TEMP -> NEXT -> PRIORITY > PRIORITY

Step 7: TEMP = TEMP -> NEXT

[END OF LOOP]

Step 8: NEW -> NEXT = TEMP -> NEXT

Step 9: TEMP -> NEXT = NEW

Step 10: End

POP(HEAD):

Step 1: Set the head of the list to the next node in the list. HEAD = HEAD -> NEXT.

Step 2: Free the node at the head of the list

Step 3: End

PEEK(HEAD):

Step 1: Return HEAD -> DATA

Step 2: End

Code:

```
#include <stdio.h>
#define MAX 100
typedef struct {
  int priority;
  int data;
} PQElement;
typedef struct {
  PQElement arr[MAX];
  int size;
} PriorityQueue;
void initPriorityQueue(PriorityQueue *pq) {
  pq->size = 0;
}
int isEmptyPriorityQueue(PriorityQueue *pq) {
```

```
  return (pq->size == 0);
}
int isFullPriorityQueue(PriorityQueue *pq) {
  return (pq->size == MAX);
}
void enqueuePriorityQueue(PriorityQueue *pq, int priority, int data) {
  if (isFullPriorityQueue(pq)) {
    printf("Priority queue overflow!\n");
    return;
  }
  int i;
  for (i = pq->size - 1; i >= 0; i--) {
    if (pq->arr[i].priority < priority) {
      break;
    }
    pq->arr[i + 1] = pq->arr[i];
  }
  pq->arr[i + 1].priority = priority;
  pq->arr[i + 1].data = data;
  pq->size++;
}
int dequeuePriorityQueue(PriorityQueue *pq) {
  if (isEmptyPriorityQueue(pq)) {
    printf("Priority queue underflow!\n");
    return -1;
  }
  int data = pq->arr[0].data;
  for (int i = 0; i < pq->size - 1; i++) {
    pq->arr[i] = pq->arr[i + 1];
  }
  pq->size--;
  return data;
}
void displayPriorityQueue(PriorityQueue *pq) {
  if (isEmptyPriorityQueue(pq)) {
    printf("Priority queue is empty!\n");
    return;
  }
  printf("Priority queue elements are: ");
  for (int i = 0; i < pq->size; i++) {
    printf("%d (%d) ", pq->arr[i].data, pq->arr[i].priority);
  }
  printf("\n");
}
int main() {
  PriorityQueue pq;
  initPriorityQueue(&pq);
  enqueuePriorityQueue(&pq, 10, 100);
```

```
  enqueuePriorityQueue(&pq, 20, 200);
  enqueuePriorityQueue(&pq, 30, 300);
  displayPriorityQueue(&pq);
  int data = dequeuePriorityQueue(&pq);
  printf("Dequeued element: %d (%d)\n", data, pq.arr[0].priority);
  displayPriorityQueue(&pq);
  return 0;
}
```

Output:

```
PS F:\AIDS 3SEM\AIDS_ANKIT_BARI> cd "f:\AIDS 3SEM\AIDS_ANKIT_BARI\" ; if ($?) { gcc prio
rity_queue.c -o priority_queue } ; if ($?) { .\priority_queue }
Priority queue elements are: 100 (10) 200 (20) 300 (30)
Dequeued element: 100 (20)
Priority queue elements are: 200 (20) 300 (30)
PS F:\AIDS 3SEM\AIDS_ANKIT_BARI>
```

Conclusion:

The implementation of a priority queue ADT using an array is a simple and efficient way to implement a priority queue. It is easy to understand and implement, and it has good performance characteristics.

However, there are some limitations to this implementation. One limitation is that the size of the queue is fixed at compile time. This can be a problem if the queue needs to be able to store a large number of elements. Another limitation is that this implementation can be inefficient if the queue is frequently enqueued and dequeued. This is because the elements of the array need to be shifted every time an element is enqueued or dequeued.

Overall, the implementation of a priority queue ADT using an array is a good choice for applications where a simple and efficient priority queue implementation is needed. However, it is important to be aware of the limitations of this implementation before using it in a production application.

Experiment No.6
Implement Singly Linked List ADT
Date of Performance:
Date of Submission:

Experiment No. 6: Singly Linked List Operations

Aim: Implementation of Singly Linked List

Objective:

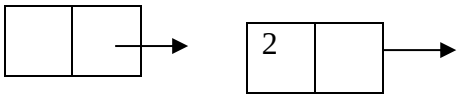
It is used to implement stacks and queues which are like fundamental needs throughout computer science. To prevent the collision between the data in the hash map, we use a singly linked list.

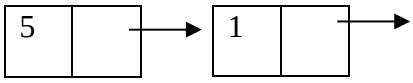
Theory :

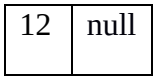
A linked list is a ordered collection of finite, homogenous elements referred as a node. Each node consist of two fields: one field for data which is referred as information and other field is an address field to store the address of next element in the list.

The address field of last node contains null value to indicate the end of list. The elements of linked list are not stored in continuous memory location but they are scattered, and still bounded to each other by an explicit link

The structure of linked list is as shown below







Header is a node containing null in its information field and an next address field contains the address of first data node in the list. Various operations can be performed on singly linked list like insertion at front, end and at specified position , deletion at front, end and at specified position, traversal, copying and merging.

Algorithm

Algorithm : INSERT_SPECIFIED(Header, X, Key)

Input : Header is a pointer to header node. X is a data of node to be inserted and Key is data of node after which insertion is to be done.

Output :A singly linked list enriched with newly inserted node.

Data Structure : A singly linked list whose address of starting node is in Header. And two fields info and next to point to data field and address field respectively. ptr is used to an address of current node

1. new = GETNODE()
2. if new = NULL then
   print "Insufficient Memory"
   Exit
3. else
   ptr = HEADER
   while info(ptr)!= Key AND next(ptr)!= NULL do
   ptr = next(ptr)
   end while
3. if next(ptr) = NULL then
   print "Key not found"
   exit
4. else
   next(new) = next(ptr)
   info(new) = x
   next(ptr) = new
   end if
   end if
5. stop

Algorithm : DELETE_SPECIFIED(Header, Key)

Input : Header is a pointer to header node. Key is data of node after which is to be deleted.

Output :A singly linked list with removed node.

Data Structure : A singly linked list whose address of starting node is in Header. And two

fields info and next to point to data field and address field respectively. ptr is used to an address of current node

1. ptr1 = HEADER
   ptr = next(ptr1)
2. while ptr!= NULL do
   if info(ptr) != Key
      ptr1 = ptr
      ptr = next(ptr)
   else
      next(ptr1) = next(ptr)
      print(info(ptr))
   FREENODE(ptr)
   End if
   End while
3. if ptr = NULL then
   print " key not found"
   end if
4. stop

Algorithm : TRAVERSAL(Header)

Input : Header is a pointer to header node.

Output :A singly linked list is traversed and its data value is printed.

Data Structure : A singly linked list whose address of starting node is in Header. And two fields info and next to point to data field and address field respectively. ptr is used to an address of current node

1. ptr = next(HEADER)
2. while ptr!= NULL do
   print (Info(ptr))
   End while
3. stop

Code:

```
#include <stdio.h>
#include <stdlib.h>
```

```c
struct Node {
    int data;
    struct Node* next;
};
struct Node* createNode(int newData) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = newData;
    newNode->next = NULL;
    return newNode;
}
void insertAtBeginning(struct Node** head, int newData) {
    struct Node* newNode = createNode(newData);
    newNode->next = *head;
    *head = newNode;
}
void insertAtEnd(struct Node** head, int newData) {
    struct Node* newNode = createNode(newData);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    struct Node* current = *head;
    while (current->next != NULL) {
        current = current->next;
    }
    current->next = newNode;
}
void deleteNode(struct Node** head, int key) {
    struct Node *current = *head, *prev = NULL;
    if (current != NULL && current->data == key) {
        *head = current->next;
        free(current);
        return;
    }
    while (current != NULL && current->data != key) {
        prev = current;
        current = current->next;
    }
    if (current == NULL) {
        printf("Key not found in the list\n");
        return;
    }
    prev->next = current->next;
    free(current);
```

```
}
void printList(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}
void freeList(struct Node** head) {
    struct Node* current = *head;
    struct Node* next;

    while (current != NULL) {
        next = current->next;
        free(current);
        current = next;
    }
    *head = NULL;
}
int main() {
    struct Node* head = NULL;
    insertAtBeginning(&head, 3);
    insertAtBeginning(&head, 7);
    insertAtBeginning(&head, 9);
    insertAtEnd(&head, 11);
    printf("Linked List: ");
    printList(head);
    deleteNode(&head, 7);
    printf("Linked List after deletion: ");
    printList(head);
    freeList(&head);
    return 0;
}
```

Output:

```
PS F:\AIDS 3SEM\AIDS_ANKIT_BARI> cd "f:\AIDS 3SEM\AIDS_ANKIT_BARI\" ; if ($?) { gcc sing
ly_linkedlist.c -o singly_linkedlist } ; if ($?) { .\singly_linkedlist }
Linked List: 9 -> 7 -> 3 -> 11 -> NULL
Linked List after deletion: 9 -> 3 -> 11 -> NULL
PS F:\AIDS 3SEM\AIDS_ANKIT_BARI>
```

Conclusion:

The implementation of a singly linked list in C is a simple and efficient way to implement a linked list. It is easy to understand and implement, and it has good performance characteristics.

However, there are some limitations to this implementation. One limitation is that the list can only be traversed in one direction, from head to tail. This can make it difficult to perform operations such as searching for a node in the list or deleting a node from the middle of the list.

Another limitation of this implementation is that it can be inefficient if the list is frequently modified. This is because every time a node is added or removed from the list, the pointers of the other nodes in the list need to be updated.

Overall, the implementation of a singly linked list in C is a good choice for applications where a simple and efficient linked list implementation is needed. However, it is important to be aware of the limitations of this implementation before using it in a production application.

Experiment No.7
Implement Circular Linked List ADT
Date of Performance:
Date of Submission:

Experiment No. 7: Circular Linked List Operations

Aim: Implementation of Circular Linked List ADT

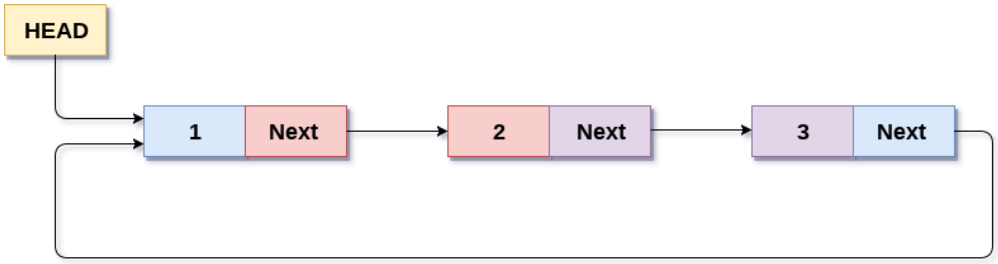
Objective: Circular Linked Lists can be used to manage the computing resources of the computer. Data structures such as stacks and queues are implemented with the help of the circular linked lists

Theory :

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.



Circular Singly Linked List

Algorithm

Algorithm :

- **Step 1:** IF PTR = NULL

  Write OVERFLOW
  Go to Step 11
  [END OF IF]

- **Step 2:** SET NEW_NODE = PTR
- **Step 3:** SET PTR = PTR -> NEXT
- **Step 4:** SET NEW_NODE -> DATA = VAL
- **Step 5:** SET TEMP = HEAD
- **Step 6:** Repeat Step 8 while TEMP -> NEXT != HEAD
- **Step 7:** SET TEMP = TEMP -> NEXT

  [END OF LOOP]

- **Step 8:** SET NEW_NODE -> NEXT = HEAD
- **Step 9:** SET TEMP → NEXT = NEW_NODE
- **Step 10:** SET HEAD = NEW_NODE
- **Step 11:** EXIT

Code:

```c
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
struct Node* createNode(int newData) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = newData;
    newNode->next = NULL;
    return newNode;
}
void insertAtBeginning(struct Node** head, int newData) {
    struct Node* newNode = createNode(newData);
    if (*head == NULL) {
        newNode->next = newNode;
        *head = newNode;
    } else {
        struct Node* last = *head;
        while (last->next != *head) {
            last = last->next;
```

```
        }
        newNode->next = *head;
        last->next = newNode;
        *head = newNode;
    }
}
void insertAtEnd(struct Node** head, int newData) {
    struct Node* newNode = createNode(newData);
    if (*head == NULL) {
        newNode->next = newNode;
        *head = newNode;
    } else {
        struct Node* last = *head;
        while (last->next != *head) {
            last = last->next;
        }
        last->next = newNode;
        newNode->next = *head;
    }
}
void deleteNode(struct Node** head, int key) {
    if (*head == NULL) {
        printf("List is empty. Cannot delete.\n");
        return;
    }
    struct Node *current = *head, *prev = NULL;
    if (current->data == key && current->next == *head) {
        free(current);
        *head = NULL;
        return;
    }
    if (current->data == key) {
        while (current->next != *head) {
            current = current->next;
        }
        current->next = (*head)->next;
        free(*head);
        *head = current->next;
    } else {
        while (current->next != *head && current->data != key) {
            prev = current;
            current = current->next;
        }
        if (current->data != key) {
            printf("Key not found in the list\n");
            return;
        }
        prev->next = current->next;
```

```
        free(current);
    }
}
void printList(struct Node* head) {
    if (head == NULL) {
        printf("List is empty\n");
        return;
    }
    struct Node* current = head;
    do {
        printf("%d -> ", current->data);
        current = current->next;
    } while (current != head);
    printf("(head)\n");
}
void freeList(struct Node** head) {
    if (*head == NULL) {
        return;
    }
    struct Node* current = *head;
    struct Node* next;
    do {
        next = current->next;
        free(current);
        current = next;
    } while (current != *head);
    *head = NULL;
}
int main() {
    struct Node* head = NULL;
    insertAtBeginning(&head, 3);
    insertAtBeginning(&head, 7);
    insertAtBeginning(&head, 9);
    insertAtEnd(&head, 11);
    printf("Circular Linked List: ");
    printList(head);
    deleteNode(&head, 7);
    printf("Circular Linked List after deletion: ");
    printList(head);
    freeList(&head);
    return 0;
}
```

Output:

```
PS F:\AIDS 3SEM\AIDS_ANKIT_BARI> cd "f:\AIDS 3SEM\AIDS_ANKIT_BARI\" ; if ($?) { gcc circ
ular_linkedlist.c -o circular_linkedlist } ; if ($?) { .\circular_linkedlist }
Circular Linked List: 9 -> 7 -> 3 -> 11 -> (head)
Circular Linked List after deletion: 9 -> 3 -> 11 -> (head)
PS F:\AIDS 3SEM\AIDS_ANKIT_BARI>
```

Conclusion:

The implementation of the Circular Linked List ADT in C provides a robust and efficient data structure for managing a collection of elements in a circular fashion. The key functionalities, including insertion at the beginning and end, printing, and memory management, have been implemented.

1. **Advantages of Circular Linked List:**
    - **Efficient Insertion and Deletion: Insertion and deletion operations are more efficient than arrays since there is no need to shift elements.**
2. **Memory Management:**
    - **Dynamic Memory Allocation: The createNode function ensures dynamic memory allocation for each node, preventing memory leaks and enabling efficient use of memory resources.**
3. **Insertion Operations:**
    - **Insertion at the End: The insertAtEnd function appends elements to the end of the circular list, ensuring that the last node points back to the head, maintaining the circular linkage.**
4. **Traversal and Printing:**
    - **Circular Traversal: The printList function traverses the Circular Linked List, accounting for its circular nature, and prints the elements.**

Experiment No.8
Implement Binary Search Tree ADT using Linked List.
Date of Performance:
Date of Submission:

Experiment No. 8: Binary Search Tree Operations

Aim : Implementation of Binary Search Tree ADT using Linked List.

Objective:

1) Understand how to implement a BST using a predefined BST ADT.

2) Understand the method of counting the number of nodes of a binary tree.

Theory:

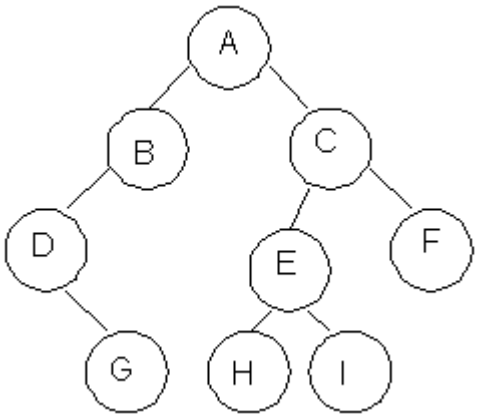
A binary tree is a finite set of elements that is either empty or partitioned into disjoint subsets. In other words node in a binary tree has at most two children and each child node is referred as left or right child.

Traversals in tree can be in one of the three ways : preorder, postorder, inorder.

Preorder Traversal

Here the following strategy is followed in sequence

1. Visit the root node R
2. Traverse the left subtree of R
3. Traverse the right sub tree of R



Description	**Output**
Visit Root	A
Traverse left sub tree – step to B then D	ABD
Traverse right sub tree – step to G	ABDG
As left subtree is over. Visit root , which is already visited so go for right subtree	ABDGC

Traverse the left subtree	ABDGCEH
Traverse the right sub tree	ABDGCEHIF

Inorder Traversal

Here the following strategy is followed in sequence

1. Traverse the left subtree of R
2. Visit the root node R
3. Traverse the right sub tree of R

Description	**Output**
Start with root and traverse left sub tree from A-B-D	D
As D doesn't have left child visit D and go for right subtree of D which is G so visit this.	DG
Backtrack to D and then to B and visit it.	DGB
Backtract to A and visit it	DGBA
Start with right sub tree from C-E-H and visit H	DGBAH
Now traverse through parent of H which is E and then I	DGBAHEI
Backtrack to C and visit it and then right subtree of E which is F	DGBAHEICF

Postorder Traversal

Here the following strategy is followed in sequence

1. Traverse the left subtree of R
2. Traverse the right sub tree of R
3. Visit the root node R

Description	**Output**
Start with left sub tree from A-B-D and then traverse right sub tree to get G	G
Now Backtrack to D and visit it then to B and visit it.	GD
Now as the left sub tree is over go for right sub tree	GDB
In right sub tree start with leftmost child to visit H followed by I	GDBHI

Visit its root as E and then go for right sibling of C as F	GDBHIEF
Traverse its root as C	GDBHIEFC
Finally a root of tree as A	GDBHIEFCA

Algorithm

Algorithm: PREORDER(ROOT)

Input : Root is a pointer to root node of binary tree

Output : Visiting all the nodes in preorder fashion.

Description : Linked structure of binary tree

1. ptr=ROOT
2. if ptr!=NULL then
   visit(ptr)
   PREORDER(LSON(ptr))\
   PREORDER(RSON(ptr))
   End if
3. Stop

Algorithm: INORDER(ROOT)

Input : Root is a pointer to root node of binary tree

Output : Visiting all the nodes in inorder fashion.

Description : Linked structure of binary tree

1. ptr=ROOT
2. if ptr!=NULL then
   INORDER (LSON(ptr))
   visit(ptr)
   INORDER (RSON(ptr))
   End if
3. Stop

Algorithm: POSTORDER(ROOT)

Input : Root is a pointer to root node of binary tree

Output : Visiting all the nodes in postorder fashion.

Description : Linked structure of binary tree

1. ptr=ROOT
2. if ptr!=NULL then
   PREORDER(LSON(ptr))
   PREORDER(RSON(ptr))
   visit(ptr)
   End if
3. Stop

Code:

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};
struct Node* createNode(int newData) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = newData;
    newNode->left = newNode->right = NULL;
    return newNode;
}
struct Node* insert(struct Node* root, int newData) {
    if (root == NULL) {
        return createNode(newData);
    }
    if (newData < root->data) {
        root->left = insert(root->left, newData);
    } else if (newData > root->data) {
        root->right = insert(root->right, newData);
    }
    return root;
}
struct Node* minValueNode(struct Node* node) {
    struct Node* current = node;
    while (current->left != NULL) {
```

```
        current = current->left;
    }
    return current;
}
struct Node* deleteNode(struct Node* root, int key) {
    if (root == NULL) {
        return root;
    }
    if (key < root->data) {
        root->left = deleteNode(root->left, key);
    } else if (key > root->data) {
        root->right = deleteNode(root->right, key);
    } else {
        if (root->left == NULL) {
            struct Node* temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct Node* temp = root->left;
            free(root);
            return temp;
        }
        struct Node* temp = minValueNode(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}
void inOrderTraversal(struct Node* root) {
    if (root != NULL) {
        inOrderTraversal(root->left);
        printf("%d ", root->data);
        inOrderTraversal(root->right);
    }
}
int main() {
    struct Node* root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);
    printf("In-order Traversal: ");
    inOrderTraversal(root);
    printf("\n");
    root = deleteNode(root, 30);
```

```
    printf("In-order Traversal after deletion: ");
    inOrderTraversal(root);
    printf("\n");
    free(root);
    return 0;
}
```

Output:

```
PS F:\AIDS 3SEM\AIDS_ANKIT_BARI> cd "f:\AIDS 3SEM\AIDS_ANKIT_BARI\" ; if ($?) { gcc bst.
c -o bst } ; if ($?) { .\bst }
In-order Traversal: 20 30 40 50 60 70 80
In-order Traversal after deletion: 20 40 50 60 70 80
PS F:\AIDS 3SEM\AIDS_ANKIT_BARI>
```

Conclusion:

The Binary Search Tree (BST) Abstract Data Type (ADT) implementation using linked lists offers an organized and efficient data structure for storing and retrieving elements. The following points summarize the key aspects of the implementation:

1. **The Binary Search Tree ADT using linked lists provides an effective and versatile solution for storing and managing ordered data. The implementation adheres to principles of efficient searching, ordered storage, and proper memory management, forming a solid foundation for applications requiring fast and organized retrieval of element Versatility and Extensibility:**
    - **The modular design allows for easy extension with additional operations tailored to specific use cases.**
2. **Ordered Storage:**
    - **The ordered nature of BSTs makes them suitable for applications requiring sorted data, such as dictionaries, databases, and symbol tables.**
3. **Search Tree Property Enforcement:**
    - **The implementation ensures that the binary search property is maintained during insertions and deletions, preserving the efficiency of search operations.**

Experiment No.9
Implementation of Graph traversal techniques - Depth First Search, Breadth First Search
Date of Performance:
Date of Submission:

Experiment No. 9: Depth First Search and Breath First Search

Aim : Implementation of DFS and BFS traversal of graph.

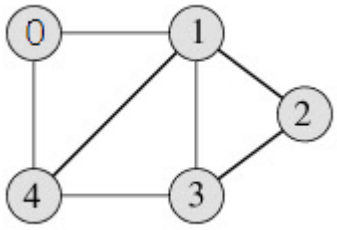
Objective:

1. Understand the Graph data structure and its basic operations.
2. Understand the method of representing a graph.
3. Understand the method of constructing the Graph ADT and defining its operations

Theory:

A graph is a collection of nodes or vertex, connected in pairs by lines referred as edges. A graph can be directed or undirected graph.

One method of traversing through nodes is depth first search. Here we traverse from starting node and proceeds from top to bottom. At a moment we reach a dead end from where the further movement is not possible and we backtrack and then proceed according to left right order. A stack is used to keep track of a visited node which helps in backtracking.



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

DFS Traversal –0 1 2 3 4

Algorithm

Algorithm: DFS_LL(V)

Input: V is a starting vertex

Output : A list VISIT giving order of visited vertices during traversal.

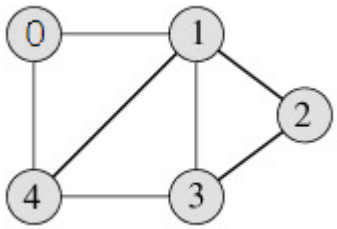
Description: linked structure of graph with gptr as pointer

1. if gptr = NULL then
   print "Graph is empty" exit
2. u=v
3. OPEN.PUSH(u)
4. while OPEN.TOP !=NULL do

```
u=OPEN.POP()
if search(VISIT,u) = FALSE then
        INSERT_END(VISIT,u)
        Ptr = gptr(u)
        While ptr.LINK != NULL do
                Vptr = ptr.LINK
                OPEN.PUSH(vptr.LABEL)
        End while
    End if
  End while
```

5. Return VISIT
6. Stop

BFS Traversal



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

BFS Traversal – 0 1 4 2 3

Algorithm

Algorithm: DFS()

```
i=0
count=1
visited[i]=1
print("Visited vertex  i")

repeat this till queue is empty or all nodes visited
repeat this for all nodes from first till last
if(g[i][j]!=0&&visited[j]!=1)
```

{
push(j)
}
i=pop()
print("Visited vertex i")
visited[i]=1
count++
Algorithm: BFS()
i=0
count=1
visited[i]=1
print("Visited vertex i")

repeat this till queue is empty or all nodes visited
repeat this for all nodes from first till last
if(g[i][j]!=0&&visited[j]!=1)
{
enqueue(j)
}

i=dequeue()
print("Visited vertex i")
visited[i]=1
count++

Code:

```c
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
struct Graph {
    int numVertices;
    struct Node** adjacencyList;
```

```
};
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}
struct Graph* createGraph(int numVertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = numVertices;
    graph->adjacencyList = (struct Node**)malloc(numVertices * sizeof(struct
Node*));

    for (int i = 0; i < numVertices; ++i) {
        graph->adjacencyList[i] = NULL;
    }
    return graph;
}
void addEdge(struct Graph* graph, int src, int dest) {
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjacencyList[src];
    graph->adjacencyList[src] = newNode;
    newNode = createNode(src);
    newNode->next = graph->adjacencyList[dest];
    graph->adjacencyList[dest] = newNode;
}
void DFS(struct Graph* graph, int vertex, int* visited) {
    visited[vertex] = 1;
    printf("%d ", vertex);
    struct Node* adjList = graph->adjacencyList[vertex];
    while (adjList != NULL) {
        int connectedVertex = adjList->data;
        if (!visited[connectedVertex]) {
            DFS(graph, connectedVertex, visited);
        }
        adjList = adjList->next;
    }
}
void BFS(struct Graph* graph, int startVertex) {
    int* visited = (int*)malloc(graph->numVertices * sizeof(int));
    for (int i = 0; i < graph->numVertices; ++i) {
        visited[i] = 0;
    }
    int* queue = (int*)malloc(graph->numVertices * sizeof(int));
    int front = -1, rear = -1;
    queue[++rear] = startVertex;
    visited[startVertex] = 1;
    while (front != rear) {
```

```
        int currentVertex = queue[++front];
        printf("%d ", currentVertex);
        struct Node* adjList = graph->adjacencyList[currentVertex];
        while (adjList != NULL) {
            int connectedVertex = adjList->data;
            if (!visited[connectedVertex]) {
                queue[++rear] = connectedVertex;
                visited[connectedVertex] = 1;
            }
            adjList = adjList->next;
        }
    }
    free(visited);
    free(queue);
}
void printGraph(struct Graph* graph) {
    for (int i = 0; i < graph->numVertices; ++i) {
        struct Node* current = graph->adjacencyList[i];
        printf("Vertex %d: ", i);
        while (current != NULL) {
            printf("%d ", current->data);
            current = current->next;
        }
        printf("\n");
    }
}
int main() {
    struct Graph* graph = createGraph(5);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    printf("Adjacency List Representation:\n");
    printGraph(graph);
    printf("\nDFS Traversal starting from vertex 0: ");
    int* visitedDFS = (int*)malloc(graph->numVertices * sizeof(int));
    for (int i = 0; i < graph->numVertices; ++i) {
        visitedDFS[i] = 0;
    }
    DFS(graph, 0, visitedDFS);
    free(visitedDFS);
    printf("\nBFS Traversal starting from vertex 0: ");
    BFS(graph, 0);
    free(graph->adjacencyList);
    free(graph);
    return 0;
}
```

Output:

```
PS F:\AIDS 3SEM\AIDS_ANKIT_BARI> cd "f:\AIDS 3SEM\AIDS_ANKIT_BARI\" ; if ($?) { gcc dfs_
bfs.c -o dfs_bfs } ; if ($?) { .\dfs_bfs }
Adjacency List Representation:
Vertex 0: 2 1
Vertex 1: 4 3 0
Vertex 2: 0
Vertex 3: 1
Vertex 4: 1

DFS Traversal starting from vertex 0: 0 2 1 4 3
BFS Traversal starting from vertex 0: 0 2 1 4 3
PS F:\AIDS 3SEM\AIDS_ANKIT_BARI>
```

Conclusion:

The Depth-First Search (DFS) and Breadth-First Search (BFS) traversal implementations for a graph offer versatile methods for exploring and analyzing graph structures. The following points summarize the key aspects and conclusions of these implementations:

Depth-First Search (DFS):

1. **Exploration Order:**
    - **DFS explores as deeply as possible along each branch before backtracking. This results in a traversal that goes as deep as possible before exploring siblings.**
2. **Stack-Based Implementation:**
    - **The recursive or stack-based approach for DFS is well-suited to explore the depth of a graph. It allows for an elegant and concise implementation.**
3. **Applications:**
    - **DFS is particularly useful for tasks such as detecting cycles, topological sorting, and solving problems that require exploring paths deep into the graph.**
4. **Memory Usage:**

- **The recursive implementation uses the call stack, which can lead to a large stack depth for deep graphs. This should be considered in terms of memory usage.**

Breadth-First Search (BFS):

1. **Exploration Order:**
    - **BFS explores the graph level by level, visiting all neighbors of a node before moving on to the next level. This results in a breadth-first traversal.**
2. **Queue-Based Implementation:**
    - **BFS is often implemented using a queue. This ensures that nodes at the same level are visited before moving on to the next level.**
3. **Applications:**
    - **BFS is useful for finding the shortest path in an unweighted graph, connected components, and solving problems that require exploring all neighbors before moving to the next level.**
4. **Memory Usage:**
    - **BFS typically requires more memory than DFS due to the need for a queue. However, it guarantees the shortest path in unweighted graphs.**

Experiment No.10
Implementation of Binary Search Method
Date of Performance:
Date of Submission:

Experiment No. 10: Binary Search Method

Aim : Implementation of Binary Search Method

Objective: 1) Understand how to implement Binary Search algorithm.

Theory:

The improvement to searching method to reduce the amount of work can be done using binary searching. Binary searching is more efficient than linear searching if an array to be searched is in sorted manner.

Here an key item to be searched is compared with the item at middle of array. If they are equal search is completed. If the middle element is greater than key item searching proceeds with left sub array. Similarly, if middle element is less than key item than searching proceeds with right sub array and so on till the element is found.

For large arrays, this method is superior to sequential searching.

Algorithm

Algorithm : FIND(arr, x, first, last)

```
if (first > last)then
  return -1
End if
mid = (first + last) / 2
if (arr[mid] = x)
  return mid
End if
if (arr[mid] < x)
  return find(arr, x, mid+1, last)
End if
return find(arr, x, first, mid-1)
```

Code:

```c
#include <stdio.h>
int binarySearch(int arr[], int low, int high, int key) {
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == key)
            return mid;
```

```c
        else if (arr[mid] > key)
            high = mid - 1;
        else
            low = mid + 1;
    }
    return -1;
}
int main() {
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int n = sizeof(arr) / sizeof(arr[0]);
    int key = 7;
    int result = binarySearch(arr, 0, n - 1, key);
    if (result != -1)
        printf("Element %d found at index %d\n", key, result);
    else
        printf("Element %d not found in the array\n", key);
    return 0;
}
```

Output:

```
PS F:\AIDS 3SEM\AIDS_ANKIT_BARI> cd "f:\AIDS 3SEM\AIDS_ANKIT_BARI\" ; if ($?) { gcc bina
ry_search.c -o binary_search } ; if ($?) { .\binary_search }
Element 7 found at index 6
PS F:\AIDS 3SEM\AIDS_ANKIT_BARI>
```

Conclusion:

Binary Search algorithm stands out as a highly efficient method for finding a specific element in a sorted dataset. Its logarithmic time complexity makes it a preferred choice for large datasets, and its adaptability to different data structures enhances its versatility. However, it's crucial to ensure that the data is sorted before applying Binary Search, and careful consideration should be given to handling edge cases. The implementation style, whether recursive or iterative, can be chosen based on programming preferences and environmental constraints. Overall, Binary Search is a fundamental algorithm with wide applications in computer science and information retrieval.