

FACULTY OF ENGINEERING AND TECHNOLOGY
BACHELOR OF TECHNOLOGY

DATA STRUCTURES AND ALGORITHMS
(203105205)

SEMESTER III

Information & Technology Department



Laboratory Manual



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

CERTIFICATE

Mr./Ms.....

with enrolment no.....has successfully completed
his/her

laboratory experiments in the DATA

from the department of.....

..... during

the academic year



Date of Submission: Staff In charge:

Head of Department:

Enrollment No :-2303031080131



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

SR no	EXPRIMENT	P NO	Date of perform	Date of Submit	Sign
1	Implement Stack and its operations like (creation push pop traverse peek search) using linear data structure				
2	Implement Infix to Postfix Expression Conversion using Stack				
3	Implement Postfix evaluation using Stack.				
4	Implement Towers of Hanoi using Stack.				
5	Implement queue and its operations like enqueue, dequeue, traverse, search.				
6	Implement Single Linked lists and its operations (creation insertion deletion traversal search reverse)				
7	Implement Double Linked lists and its operations (creation insertion deletion traversal search reverse)				
8	Implement binary search and interpolation search.				
9	Implement Bubble sort, selection sort, Insertion sort, quick sort, merge sort.				
10	Implement Binary search Tree and its operations (creation, insertion, deletion).				
11	Implement Traversals Preorder Inorder Postorder on BST.				
12	Implement Graphs and represent using adjacency list and adjacency matrix and implement basic operations with traversals (BFS and DFS).				



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

EXPERIMENT NO. 1

Objective(s): Implement a stack and perform operations like creation, push, pop, traverse, peek, and search using a linear data structure efficiently

Outcome: Efficient data organization with the ability to quickly add, remove, find, and iterate elements in both forward and backward directions using stack operations.

Problem Statement: Implement Stack and its operations like (creation push pop traverse peek search) using linear data structure

Background Study: A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. This means that the last element added to the stack will be the first one to be removed. Stacks are used in various applications such as expression evaluation, backtracking algorithms, function call management in recursion, and undo mechanisms in text editors. Understanding stacks and their operations is fundamental in computer science and programming.

Basic Operations on Stacks

1. **Creation:** Initializing an empty stack. This can be done using an array or a list in most programming languages.
2. **Push:** Adding an element to the top of the stack. This operation increases the stack's size by one.
3. **Pop:** Removing the top element from the stack and returning it. This operation decreases the stack's size by one.
4. **Traverse:** Iterating through the elements of the stack. This can be done in either forward (from bottom to top) or backward (from top to bottom) direction.
5. **Peek:** Viewing the top element of the stack without removing it.
6. **Search:** Finding the position of a specific element in the stack from the top. If the element is not found, it returns -1.



Applications of Stacks

- **Expression Evaluation:** Stacks are used to evaluate arithmetic expressions, particularly those written in postfix (Reverse Polish Notation) form.
- **Backtracking:** In algorithms like depth-first search, stacks are used to remember the paths that need to be explored.
- **Function Call Management:** Stacks are used to manage function calls and local variables in programming languages that support recursion.
- **Undo Mechanism:** Text editors and other applications use stacks to implement undo functionality, where the most recent operations can be

Algorithm (Student Work Area):

1. Creation

Step 1: Initialize an list `stack` empty Step 2: Return `stack`

2. Push

Step 1: Append `element` to `stack` Step 2: Return `stack`

3. Pop

Step 1: If `stack` is not empty:

- a. Remove and return the last element of
`stack`

Step 2: Else:

- a. Return an error message "Stack is empty"

4. Traverse

Step 1: For each element `e` in `stack`:

- a. Print `e`

5. Peek



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

Step 1: If `stack` is not empty:

a. Return the last element of

`stack` Step 2: Else:

a. Return an error message "Stack is empty"

6. Search

Step 1: If `element` is in `stack`:

a. Return `len(stack) - stack.index(element)` Step 2: Else:

a. Return -1

Code:

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

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

```
#define MAX 100
```

```
typedef struct{ int
```

```
item[MAX]; int top;
```

```
}stack;
```

```
void initialize(stack*s){ s->top=-1;
```

```
}
```

```
int isEmpty(stack*s){ return s->top==-1;
```

```
}
```

```
int isFull(stack*s){ return s->top==MAX-1;
```

```
}
```

```
void push(stack*s,int value){ if(isFull(s)){ printf("Stack
```

```
is overflow"); return;
```

```
}
```



```
s->item[++(s->top)]=value;
printf("Pushed %d to the stack \n",value);
}
int pop(stack*s){ if(isEmpty(s)){ printf("Stack
is underFlow"); return -1;
}
return s->item[(s->top)--];
}
int peek(stack*s){ if(isEmpty(s)){ printf("Stack is Empty"); return -1;
}
printf("Stack elements are \n"); for(int i=s->top;i>=0;i--){ printf("%d\n",s-
>item[i]);
}
}
int search(stack*s,int value) { if (isEmpty(s)) { printf("Stack is empty\n"); return
-1;
}
for (int i = s->top; i >= 0; i--) { if(s->item[i] == value) { return i;
}
}
printf("Element not found\n"); return -1;
}
int main(){ stack s; initialize(&s); push(&s,30);
push(&s,40); push(&s,50);
```

```
printf("Top element is %d\n",peek(&s)); printf("Popped element is  
%d\n",pop(&s)); printf("Top value is %d\n",s.top); int
```

```
n;
```

```
printf("Enter the element to search: \n"); scanf("%d", &n);
```

```
int index = search(&s,n); if (index != -1) { printf("Element  
found at index %d\n", index);
```

```
}
```

```
return 0;
```

```
}:
```

OUTPUT :

Run	Output
^	<pre>/tmp/NtvIEh9KMU.o Pushed 30 to the stack Pushed 40 to the stack Pushed 50 to the stack Stack elements are 50 40 30 Top element is 3 Popped element is 50 Top value is 1 Enter the element to search:</pre>



EXPERIMENT NO. 2

Objective(s):To Implementing Infix to Postfix Expression Conversion using Stack

Outcome:A postfix expression obtained from converting an infix expression using a stack-based algorithm.

Problem Statement:Implement Infix to Postfix Expression Conversion using Stack

Background Study:Converting an infix expression to a postfix (or Reverse Polish Notation, RPN) expression involves rearranging the expression so that each operator follows all of its operands. This is useful for evaluating mathematical expressions efficiently.

1. **Infix Notation:** This is the conventional way of writing mathematical expressions, where operators are placed between operands. For example, $a + b$ or $a * (b + c)$.
2. **Postfix Notation:** Also known as Reverse Polish Notation (RPN), operators are placed after their operands. For example, $a b +$ or $a b c + *$.

Conversion (Using a Stack):To convert an infix expression to postfix, we typically use a stack to keep track of operators and manage the order of operations.

- Initialize an empty stack and an empty output list (or string).
- Scan the infix expression from left to right.
 - If the token is an operand (like a variable or a number), append it to the output list.
 - If the token is an operator, pop operators (if there are any) from the stack to the output list until we find an operator of lower precedence (or with equal precedence, if left-associative).
 - Push the current operator onto the stack.

- When the input expression has been completely scanned, pop all the operators from the stack to the output list

Algorithm (Student Work Area):

Step 1: Initialization

- stack (array of characters)
- top (index of the top of the stack, initialized to -1)
- max_size (maximum size of the stack, input by the user)

Step 2: Define helper functions

- isFull(): Check if the stack is full.
- isEmpty(): Check if the stack is empty.
- push(char value): Push a character onto the stack.
- pop(): Pop and return the top character from the stack.
- peek(): Return the top character from the stack without popping it.
- isOperator(char ch): Check if a character is an operator (+, -, *, /, ^).
- precedence(char ch): Return the precedence of an operator (+, -, *, ^).

Step 3: Function to convert infix to postfix

- Initialize variables i and j for traversing the infix and postfix expressions respectively.
- Allocate memory for postfix array (char* postfix) to store the postfix expression.
- While there are characters in the infix expression:
 - If the character is an operand (digit or letter), add it to the postfix.
 - If the character is (, push it onto the stack. If the character is), pop from the stack to postfix until (is encountered on the stack. Pop and discard (.



- If the character is an operator (+, -, *, /):
 - Pop operators from the stack to postfix until an operator with lower precedence or (is encountered.
 - Push the current operator onto the stack.
 - Pop all remaining operators from the stack to postfix.
 - Terminate postfix with a null character (\0)

Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

#define MAX 100 typedef
struct { int top; char
items[MAX];
} Stack;

void initStack(Stack *s)
{
    s->top = -1;
}

int isEmpty(Stack *s)
{
    return s->top == -1;
}
```



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

```
int isFull(Stack *s) { return
s->top == MAX - 1;
}

void push(Stack *s, char value)
{ if
(isFull(s))
{
printf("Stack Overflow\n");
} else
{
s->items[++(s->top)] = value;
}
}

char pop(Stack *s)
{
if (isEmpty(s))
{
printf("Stack Underflow\n"); return '\0';
} else { return s->items[(s-
>top)--];
}
}
```



```
char peek(Stack *s) { if
(isEmpty(s)) { return '\0'; }
else
{
return s->items[s->top];
}
}

int precedence(char op)
{
switch (op) {
case '+': case
'-':
return 1; case '*':
case '/': return 2;
case '^': return 3;
default:
return 0;
}
}

void infixToPostfix(char* infix, char* postfix)
{ Stack s; initStack(&s); int k =
0; for (int i = 0; infix[i] != '\0';
i++)
{
```



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

```
char ch = infix[i]; if
(isalnum(ch))    {
postfix[k++] = ch; }
else if (ch == '(') {
push(&s, ch);
} else if (ch == ')') {
while (!isEmpty(&s) && peek(&s) != '(')
{
postfix[k++] = pop(&s);
}
pop(&s);
} else
{
while (!isEmpty(&s) && precedence(peek(&s)) >= precedence(ch))
{
postfix[k++] = pop(&s);
}
push(&s, ch);
}
while (!isEmpty(&s))
{
postfix[k++] = pop(&s);
}
```



```
postfix[k] = '\0';  
}  
  
int main() {  
char infix[MAX] = "((A+B)*C)/D"; char  
postfix[MAX];  
printf("Infix expression: %s\n", infix); infixToPostfix(infix,  
postfix);  
printf("Postfix expression: %s\n", postfix); return  
0;  
}
```

OUTPUT:



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd



PATEL SATYAMKUMAR DATTUBHAI

AI

NEW

C

RUN

STDIN

Input for the program (Optional)

Output:

Infix expression: ((A+B)*C)/D
Postfix expression: AB+C*D/

EXPERIMENT NO. 3

Objective(s): To implementing postfix evaluation using a stack is to evaluate arithmetic expressions written in postfix notation efficiently.

Outcome: Utilize a stack data structure to manage operands and intermediate results during evaluation.

Problem Statement: Implement Postfix evaluation using Stack.

Background Study:

1. Postfix Notation (Reverse Polish Notation, RPN)

Postfix notation is a way of writing arithmetic expressions in which every operator follows all of its operands. This eliminates the need for parentheses to denote the order of operations (operator precedence). For example:



- Infix Notation: $(3 + 4) * 5$

- Postfix Notation: $3 4 + 5 *$

2. Why Use Postfix Notation?

Postfix notation is easier to evaluate with a computer program because:

- It removes the need for parentheses and rules about operator precedence.
- It is unambiguous and easier to parse.
- Evaluation can be done using a stack-based algorithm, which is efficient and straightforward.

3. Stack Data Structure

A stack is a Last In First Out (LIFO) data structure, meaning that the last element added to the stack is the first one to be removed. This makes it particularly useful for evaluating postfix expressions, where operands are pushed onto the stack as they appear, and operators cause the operands to be popped, evaluated, and then their result pushed back onto the stack.

4. Evaluation Process

The evaluation process for postfix expressions involves:

- Iterating through each token (operand or operator) in the expression.
- Pushing operands onto the stack.
- When encountering an operator, popping the necessary number of operands from the stack, performing the operation, and pushing the result back onto the stack.
- At the end of the expression, the stack should contain exactly one operand, which is the result of the expression.

5. Operators and Operands

In postfix notation, operands are numbers, and operators are symbols that represent arithmetic operations (e.g., +, -, *, /). Each operator takes a

certain number of operands (e.g., + and - take two operands, while * and / take two operands).

6. Error Handling

It's essential to handle errors such as:

- Invalid tokens in the input expression.
 - Insufficient operands for an operator.
 - Extra operands left after evaluating the expression
- Algorithm (Student Work Area):

Algorithm for Infix to Postfix Conversion:

Input: Infix expression infix[] Output: Postfix expression postfix[]

1. Initialize an empty stack stack[] and an empty string postfix[].
2. Iterate through each character ch in the infix expression infix[]:
 - If ch is an operand (number or variable), append it to postfix[].
 - If ch is an opening parenthesis '(', push it onto stack[].
 - If ch is a closing parenthesis ')':
 - Pop from stack[] to postfix[] until an opening parenthesis '(' is encountered.
 - Pop the opening parenthesis from stack[].
 - If ch is an operator:
 - Pop from stack[] to postfix[] until an operator with lower or equal precedence than ch is at the top of stack[].
 - Push ch onto stack[].
3. Pop all operators from stack[] to postfix[].
4. Return postfix[] as the result.

Algorithm for Evaluating a Postfix Expression:



Input: Postfix expression postfix[] Output: Integer result of the expression

1. Initialize an empty stack stack[].
2. Iterate through each character ch in the postfix expression postfix[]:
 - If ch is an operand (number), push its integer value onto stack[].
 - If ch is an operator:
 - Pop the top two elements b and a from stack[].
 - Compute result as a ch b using the operator ch.
 - Push result onto stack[].
3. The result of the expression is the only element left in stack[].
4. Return the result

Code:

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

```
char stack[100];
```

```
int top = -1; void
```

```
push(char x)
```

```
{
```

```
stack[++top] = x;
```

```
}
```

```
char pop()
```

```
{
```



```
if(top == -1)
return -1; else
return stack[top--];
}

int priority(char x)
{ if(x == '(') return
0; if(x == '+' | x ==
'-') return 1; if(x
== '*' | x == '/')
return 2; return 0;
}

int main()
{
char exp[100]; char *e, x;
printf("Enter the expression : ");
scanf("%s",exp); printf("\n"); e
= exp; while(*e != '\0')
{
if(isalnum(*e))
printf("%c ",*e);
else if(*e == '(')
push(*e); else
if(*e == ')')
```



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

```
{
while((x = pop()) != '(')
printf("%c ", x);
}
else
{
while(priority(stack[top]) >= priority(*e))
printf("%c ",pop()); push(*e);
}
e++;
}
while(top != -1)
{
printf("%c ",pop());
}
return 0;
}
```

OUTPUT :

```
Output
^ /tmp/J4l8xnyAiq.o
Enter the expression : (A*B)/(C+D-E)

A B * C D + E - /

=== Code Execution Successful ===
```

EXPERIMENT NO. 4

Objective(s): The objective is to move a stack of disks from one rod to another.

Outcome: The outcome is to solve the puzzle using the minimum number of moves. The minimum number of moves required to solve a Towers of Hanoi puzzle with n disks is $2^n - 1$.

Problem Statement: Implement Towers of Hanoi using Stack.

Background Study:

1. Origin and History:

- The Towers of Hanoi puzzle was invented by the French mathematician Édouard Lucas in

1883. Lucas named the puzzle after the Tower of Hanoi, a shrine in Vietnam where the legend of the puzzle originates.

- It was first introduced to the public in a recreational mathematics column in the French newspaper L'Echo de Paris.

2. Problem Description:

- The Towers of Hanoi puzzle consists of three rods and a number of disks of different sizes that can slide onto any rod.
- The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, making a conical shape.

3. Objective:

- The objective is to move the entire stack to another rod, obeying the following simple rules:
 - Only one disk can be moved at a time.
 - Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
 - No larger disk may be placed on top of a smaller disk.

4. Minimum Moves:

- The minimum number of moves required to solve the puzzle with n disks is $2^n - 1$. This exponential growth makes the puzzle interesting as n increases.

5. Mathematical and Computational Aspects:

- Recursive Solution: The problem can be solved recursively. The recursive solution is elegant and can be described as:
 - Move $n-1$ disks from the source rod to the auxiliary rod.
 - Move the n th disk (the largest one) directly to the destination rod.
 - Move $n-1$ disks from the auxiliary rod to the destination rod.

- This recursive approach demonstrates the power and elegance of recursion in solving problems that can be broken down into smaller, similar problems Algorithm (Student Work Area):

Function t_o_h:

- This function is defined to solve the Towers of Hanoi problem recursively.
- Parameters:
 - n: Number of disks to be moved.
 - s, a, d: Characters representing the three rods ('A', 'B', 'C' in this case).
 - s: Source rod.
 - a: Auxiliary rod.
 - d: Destination rod.

Base Case ($n == 1$):

- If there is only one disk ($n == 1$):
 - Move the disk from the source rod (s) to the destination rod (d).
 - Print: Move disk 1 from rod <source> to rod <destination>.
 - Return from the function.

Recursive Case ($n > 1$):

- Move $n-1$ disks from the source rod (s) to the auxiliary rod (a), using the destination rod (d) as auxiliary:
 - Call t_o_h($n - 1$, s, d, a);.
- Move the nth disk (the largest one) from the source rod (s) to the destination rod (d):



- Print: Move disk <n> from rod <source> to rod <destination>.
- Move n-1 disks from the auxiliary rod (a) to the destination rod (d), using the source rod (s) as auxiliary:
- Call t_o_h(n - 1, a, s, d);

Code:

```
#include <stdio.h>

#include <stdlib.h> #define
MAX 100

int src[MAX], aux[MAX], dest[MAX]; int
topSrc = -1, topAux = -1, topDest = -1;

// Check if a stack is empty
int isEmpty(int top) {
return top == -1;
}

// Push an item onto a stack void
push(int stack[], int *top, int item) {
stack[++(*top)] = item;
}

// Pop an item from a stack int
pop(int stack[], int *top) {
return stack[(*top)--];
}
```

```
// Move a disk between two pegs and print the move void
moveDisk(int src[], int *topSrc, int dest[], int *topDest, char s,
char d) {

    int srcTopDisk = isEmpty(*topSrc) ? -1 : pop(src, topSrc);    int
destTopDisk = isEmpty(*topDest) ? -1 : pop(dest, topDest); if
(srcTopDisk == -1) {

        push(src, topSrc, destTopDisk);

        printf("Move disk %d from %c to %c\n", destTopDisk, d, s);

    } else if (destTopDisk == -1) {
push(dest, topDest, srcTopDisk);

        printf("Move disk %d from %c to %c\n", srcTopDisk, s, d);

    } else if (srcTopDisk > destTopDisk) {
push(src,    topSrc,    srcTopDisk);
push(src, topSrc, destTopDisk);

        printf("Move disk %d from %c to %c\n", destTopDisk, d, s);

    } else {

        push(dest, topDest, destTopDisk);
push(dest, topDest, srcTopDisk);

        printf("Move disk %d from %c to %c\n", srcTopDisk, s, d);

    }
}

void towerOfHanoi(int num_of_disks) {

    char s = 'A', d = 'C', a = 'B';
```

```
int total_num_of_moves = (1 << num_of_disks) - 1;
for (int i = num_of_disks; i >= 1; i--) {    push(src,
&topSrc, i);
}
for (int i = 1; i <= total_num_of_moves; i++) {
    if (i % 3 == 1) {
        moveDisk(src, &topSrc, dest, &topDest, s, d);
    } else if (i % 3 == 2) {
        moveDisk(src, &topSrc, aux, &topAux, s, a);
    } else if (i % 3 == 0) {
        moveDisk(aux, &topAux, dest, &topDest, a, d);
    }
}
}
// Main function

int main() {    int num_of_disks;
printf("Enter the number of disks: ");
scanf("%d", &num_of_disks);
towerOfHanoi(num_of_disks);
return 0;
}
```



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

OUTPUT :

AI NEW C RUN

4

Output:
Enter the number of disks: Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
Move disk 4 from A to B
Move disk 1 from C to B
Move disk 2 from C to A
Move disk 1 from B to A



EXPERIMENT NO. 5

Objective(s): To implement a queue is to understand the basic operations of a queue data structure,

Outcome: Gain practical understanding and experience in working with a queue data structure, including its basic operations of enqueue, dequeue, traverse, and search.

Problem Statement: Implement queue and its operations like enqueue, dequeue, traverse, search.

Background Study:

1. Definition:

- Queue is a linear data structure that follows the FIFO (First In, First Out) principle.
- Elements are inserted at the rear (enqueue) and removed from the front (dequeue).

2. Operations:

- Enqueue: Adds an element to the rear of the queue.
- Dequeue: Removes and returns the element from the front of the queue.
- Traverse: Iterates through all elements in the queue.
- Search: Checks if a specific element is present in the queue.

3. Key Concepts:

- Front and Rear: Pointers indicating the position of the first and last elements in the queue.

- Empty Queue: A condition where no elements are present in the queue (front= NULL).
- Full Queue: A condition where the queue cannot accept more elements due to memory constraints (rare in linked list-based implementations).

Algorithm (Student Work Area):

1. Create Queue
 - Input: Maximum size of the queue max_size.
 - Output: Pointer to the created queue.
 - Algorithm:
 - Allocate memory for the queue structure (Queue).
 - Allocate memory for the array (array) inside the queue to hold elements.
 - Initialize front and rear to -1.
 - Return the pointer to the created queue.
2. Check if Queue is Full (isFull(Queue *q))
 - Input: Pointer to the queue q.
 - Output: true if the queue is full; otherwise, false.
 - Algorithm:
 - Return (q->rear == q->max_size - 1).
3. Check if Queue is Empty (isEmpty(Queue *q))
 - Input: Pointer to the queue q.
 - Output: true if the queue is empty; otherwise, false.
 - Algorithm
 - Return (q->front == -1 && q->rear == -1).
4. Enqueue (enqueue(Queue *q, int value))
 - Input: Pointer to the queue q and element value to be added.



- Output: None.
- Algorithm:
 - Check if the queue is full (isFull(q)).
 - If isEmpty(q), set front to 0.
 - Increment rear.
 - Add value to q->array[q->rear].

5. Dequeue (dequeue(Queue *q)) ●

Input: Pointer to the queue q.

- Output: Element removed from the front of the queue, or -1 if the queue is empty.
- Algorithm:
 - Check if the queue is empty (isEmpty(q)).
 - Get the element at q->array[q->front].

CODE:

```
#include <stdio.h>

#include <stdlib.h>

#define MAX 100

typedef struct
{
    int items[MAX];
    int front; int
    rear; } Queue;

void initializeQueue(Queue *q)
```



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

```
{
    q->front = -1; q->rear
    = -1;
}

int isFull(Queue *q)
{
    return q->rear == MAX - 1;
}

int isEmpty(Queue *q)
{
    return q->front == -1 || q->front > q->rear;
}

void enqueue(Queue *q, int value)
{ if
(isFull(q))
{
    printf("Queue is full!\n");
} else
{
    if (q->front == -1) q->front = 0;
    q->rear++; q->items[q->rear]
    = value; printf("Inserted
    %d\n", value);
}
```




```
}  
  
int dequeue(Queue *q)  
{  
    int item; if (isEmpty(q)) {  
        printf("Queue is empty!\n");  
        item = -1 } else  
        {  
            item = q->items[q->front]; q->front++;  
            if (q->front > q->rear)  
            {  
                q->front = q->rear = -1;  
            }  
        }  
    return item;  
}  
  
int search(Queue*q,int value)  
{  
    if(isEmpty(q))  
    {  
        printf("Queue is Empty\n"); return  
        -1;  
    }  
    for(int i=q->rear;i>=0;i--)  
    {
```



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

```
if(q->items[i]==value)
{
    return i;
}

void display(Queue *q)
{
    if (isEmpty(q))
    {
        printf("Queue is empty!\n");
    } else
    {
        printf("Queue elements are: \n"); for
        (int i = q->front; i <= q->rear; i++)
        {
            printf("%d ", q->items[i]);
        }
        printf("\n");
    }
}

int main()
{
    Queue q; initializeQueue(&q);
    enqueue(&q, 10);
```



```
enqueue(&q, 20);
enqueue(&q, 30);
enqueue(&q, 40);
printf("Dequeued element: %d\n", dequeue(&q));
printf("Dequeued element: %d\n", dequeue(&q)); int
n;
printf("Enter the elemnt to search:\n");
scanf("%d",&n); int
index=search(&q,n); if(index!=-1)
{
printf("Elemnt found at index %d\n",index);
}
display(&q); return
0;
}
```

OUTPUT:



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

STDIN

30

Output:

```
Inserted 10
Inserted 20
Inserted 30
Inserted 40
Dequeued element: 10
Dequeued element: 20
Enter the elemnt to search:
Elemnt found at index 2
Queue elements are:
30 40
```

EXPERIMENT NO. 6

Objective(s): To implement a single linked list data structure and perform operations such as creation, insertion, deletion, traversal, search, and reversal.

Enrollment No :-2303031080131



Outcome: Understanding how to implement and manipulate a single linked list, gaining proficiency in basic operations of linked lists.

Problem Statement: Implement Single Linked lists and its operations (creation insertion deletion traversal search reverse).

Background Study:

Definition:

- A linked list is a linear data structure where each element is a separate object called a node.
- Each node contains two parts: the data and a reference (link) to the next node in the sequence.

Types of Linked Lists:

- Single Linked List: Each node points to the next node in the sequence.
- Double Linked List: Each node has two references, one to the next node and another to the previous node.
- Circular Linked List: Last node points back to the first node.

Operations on Single Linked Lists:

1. Creation of a Linked List:

- Create an empty linked list by setting the head pointer to NULL.

2. Insertion:

- Insert at the Beginning: Create a new node, set its next to the current head, and update the head to point to the new node.
- Insert at the End: Traverse to the end of the list, create a new node, and set the next of the last node to the new node.
- Insert at a Position: Traverse to the desired position, adjust the next pointers to insert the new node.



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

3. Deletion:

- Delete at the Beginning: Update the head to point to the second node and free the memory of the first node.
- Delete at the End: Traverse to the second last node, update its next to NULL, and free the memory of the last node.
- Delete at a Position: Traverse to the node before the position, adjust the next pointers to skip the node to be deleted, and free its memory.

4. Traversal:

- Start from the head and move to the next node until the end (NULL) is reached, printing or processing each node.

5. Search:

- Traverse the list, comparing each node's data with the target value until the target is found or the end of the list is reached.

6. Reverse:

- Reverse the order of nodes in the linked list by adjusting the next pointers to point in the opposite direction.

Advantages:

- Dynamic size: Easily grow and shrink in size during runtime.
- Efficient Insertions and Deletions: Insertions and deletions can be done in constant time, $O(1)$, when performed at the beginning of the list.

Disadvantages:

- More memory overhead than arrays because of the storage used by pointers.
- Sequential access is slow compared to arrays.

Applications:

- Implementation of stacks, queues, and hash tables.



- Undo functionality in software applications.
- Used in adjacency list representation of graphs.

Algorithm (Student Work Area):

1. Creation

- Step 1: Initialize the head pointer to null or `None`.

2. Insertion

a. At the beginning:

- i. Step 1: Create a new node.
- ii. Step 2: Set the new node's next pointer to the current head.
- iii.

Step 3: Update the head pointer to the new node.

b. At the end:

- i. Step 1: Create a new node.
- ii. Step 2: If the list is empty, set the head pointer to the new node.
- iii.

Step 3: Otherwise, traverse to the last node. iv. Step 4: Set the last node's next pointer to the new node.

c. At a given position:

- i. Step 1: Create a new node.
- ii. Step 2: Traverse to the node currently at the given position.
- iii. Step 3: Adjust the new node's next pointer to point to the node at the given position.
- iv. Step 4: Adjust the previous node's next pointer to include the new node.

3. Deletion

a. From the beginning:



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

- i. Step 1: If the list is empty, return.
- ii. Step 2: Update the head pointer to the next node.
- b. From the end:
 - i. Step 1: If the list is empty, return.
 - ii. Step 2: Traverse to the second-to-last node. iii. Step 3: Update the second-to-last node's next pointer to null.
- c. From a given position:
 - i. Step 1: Traverse to the node just before the given position.
 - ii. Step 2: Adjust the previous node's next pointer to exclude the node to be deleted.
4. Traversal
 - a. Forward Traversal:
 - i. Step 1: Start from the head.
 - ii. Step 2: While the current node is not null, process the current node and move to the next node.
5. Search
 - I. Step 1: Start from the head.
 - II. Step 2: While the current node is not null, compare its value with the target value.
 - III. Step 3: If a match is found, return the node.
 - IV. Step 4: Move to the next node.
 - V. Step 5: If the end of the list is reached without finding the value, return null or `None`.
6. Reverse
 - I. Step 1: Start from the head.



- II. Step 2: Initialize a temporary variable to null to store the previous node.
- III. Step 3: While the current node is not null: A. Step 3a: Store the next node.
 - B. Step 3b: Set the current node's next pointer to the previous node.
 - C. Step 3c: Move the temporary variable to the current node.
 - D. Step 3d: Move the current node to the next node (stored in step 3a).
- IV. Step 4: After the loop, update the head pointer to the last processed node (stored in the temporary variable).

Code:

```
#include <stdio.h>

#include<stdlib.h>

typedef struct node
{
    int data; struct
    node* next;
}node;

node* createNode(int data)
{
    node *newNode = (node *)malloc(sizeof(node));
    newNode->data=data; newNode->next=NULL;
    return newNode;
}

void addElementAtLast(int data,node **headPtr)
{
```



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

```
node *newNode = createNode(data);  
if(*headPtr== NULL)  
{  
    *headPtr = newNode;  
    return;  
}  
node* last = *headPtr; while(last-  
>next != NULL)  
{  
    last = last->next;  
}  
last->next = newNode;  
return;  
}  
void deleteElementAtLast(node** headPtr)  
{  
    if(*headPtr==NULL)  
    {  
        printf("\nLinked List is Empty");  
        return;  
    }  
    node* temp = *headPtr;  
    if(temp->next == NULL)  
    {
```



```
*headPtr = NULL;

}

while(temp->next->next != NULL)

{

    temp = temp->next;

}

temp->next = NULL;

//return;

}

void display(node *head)

{

    if(head== NULL)

    {

        printf("NO ELEMENTS TO DISPLAY\n");

    }

    return;

}

node *temp = head;

while(temp!= NULL)

{

    printf("%d\n",temp->data);    temp=temp-

>next;

}

return;

}
```



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

```
void search(node* head,int key)
{
    node* temp = head;
    while(temp != NULL)
    {
        if(temp->data==key)
        {
            printf("Element with value %d Found\n",key);
            /* break;(for this we have to maintain    flagg as
            another variable for elemnt found)
            */
            return;
        }
        temp=temp->next;
    }
    printf("ELEMENT WITH VALUE %d  NOT FOUND \n",key);
    return;
}

void reverse(node **headPtr)
{
    if(*headPtr == NULL)
    {
        printf("LINKED LIST is Empty\ n");
    }
}
```



```
node *prev=NULL;

node* crr= *headPtr;
node *next; while(crr!=
NULL)
{
    next = crr->next;
    crr->next=prev;
    prev=crr;    crr =
    next;
}
*headPtr = prev;
return;
}

int main()
{
    node* head = NULL;    //add
    element 12
    addElementAtLast(12,&head);
    addElementAtLast(13,&head);
    addElementAtLast(14,&head);
    addElementAtLast(16,&head);
    addElementAtLast(17,&head);
    printf("LINKED LIST BEFORE
```



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

```
REVERSE.....\n");
display(head);
//deleteElementAtLast(&head);
//display(head);
search(head,15);
search(head,13);
printf("LINKED LIST AFTER REVERSE.....\n");
reverse(&head);  display(head); return 0;
}
```

OUTPUT :

```
STDIN
Input for the program ( Optional )
Output:
LINKED LIST BEFORE REVERSE.....
12
13
14
16
17
ELEMENT WITH VALUE 15  NOT FOUND
Element with value 13 Found
LINKED LIST AFTER REVERSE.....
17
16
14
```

EXPERIMENT NO. 7

Objective(s): Design and implement a doubly linked list data structure supporting creation, insertion, deletion, traversal, search, and reversal operations.

Enrollment No :-2303031080131



Outcome: Efficient data organization with ability to add, remove, find, and iterate elements in both forward and backward directions.

Problem Statement: Implement Double Linked lists and its operations (creation insertion deletion traversal search reverse) **Background Study:**

Doubly Linked Lists

A doubly linked list is a linear data structure where each element (node) contains data and references to two other nodes:

- Next: Points to the next node in the list.
- Previous: Points to the previous node in the list.

This enables traversal and modification in both forward and backward directions, unlike singly linked lists which only allow forward traversal.

Theory:

- Nodes: Each node consists of three parts:
 - Data: The actual information stored in the node (integer, string, etc.).
 - Next: A pointer to the next node in the list.
 - Previous: A pointer to the previous node in the list.
- Head and Tail:
 - Head: A pointer to the first node in the list. If the list is empty, the head points to null.
 - Tail: A pointer to the last node in the list. If the list is empty, the tail points to null.
- Operations:
 - Creation: Allocate memory for a new node, initialize its data, and set its next and previous pointers to null. The head and tail are also set to this new node if the list is empty.



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

■ Insertion:

■ At the beginning: Update the new node's next pointer to point to the current head, update the current head's previous pointer to point to the new node, and finally update the head to point to the new node.

■ At the end: Update the new node's previous pointer to point to the current tail, update the current tail's next pointer to point to the new node, and finally update the tail to point to the new node.

■ In the middle: Traverse to the desired position, update pointers of the new node, its neighbors, and the surrounding nodes.

■ Deletion:

■ First node: Update the head to point to the second node (if it exists) and set the second node's previous pointer to null.

■ Last node: Update the tail to point to the second-last node (if it exists) and set the second-last node's next pointer to null.

■ Middle node: Traverse to the node, update pointers of its neighbors to bypass it.

■ Traversal: Start from the head and follow the next pointers until a null pointer is encountered. Print the data of each node during traversal.

■ Search: Start from the head and traverse the list, comparing the data of each node with the search key. Return the node's pointer if found, otherwise return null.

■ Reversal: Reverse the direction of the next and previous pointers for each node in the list. The head becomes the tail and vice versa.

Benefits of Doubly Linked Lists:

- Efficient insertion and deletion at any position compared to arrays.
- No need to shift elements during insertion/deletion in the middle.
- Support for forward and backward traversal.



Drawbacks of Doubly Linked Lists:

- More memory overhead compared to arrays due to the extra pointer (previous) in each node.
- Random access (accessing a specific element by index) is inefficient as it requires traversal from the beginning.

Use Cases

- Navigation Systems: Where you need to go back and forth through the list of locations or steps.
- Undo/Redo Functionality: In applications like text editors, where you can move back and forth between states.
- Complex Data Structures: Forms the basis for more advanced data structures like balanced trees and certain types of heaps.

Algorithm (Student Work Area):

Algorithms for Doubly Linked List Operations

1. Creation

Objective: Initialize an empty doubly linked list.

- Step 1: Initialize the head pointer to null or None.

2. Insertion

Objective: Add a new node to the list at a specified position.

- a. At the beginning:
 - Step 1: Create a new node.
 - Step 2: Set the new node's next pointer to the current head.
 - Step 3: If the list is not empty, set the current head's previous pointer to the new node.
 - Step 4: Update the head pointer to the new node.
- b. At the end:

- Step 1: Create a new node.
- Step 2: If the list is empty, set the head pointer to the new node.
- Step 3: Otherwise, traverse to the last node.
- Step 4: Set the last node's next pointer to the new node.
- Step 5: Set the new node's previous pointer to the last node.
- c. At a given position:
 - Step 1: Create a new node.
 - Step 2: Traverse to the node currently at the given position.
 - Step 3: Adjust the new node's next and previous pointers to point to the surrounding nodes.
 - Step 4: Adjust the surrounding nodes' pointers to include the new node.

3. Deletion

Objective: Remove a node from the list.

- a. From the beginning:
 - Step 1: If the list is empty, return.
 - Step 2: Update the head pointer to the next node.
 - Step 3: If the new head is not null, set its previous pointer to null.
- b. From the end:
 - Step 1: If the list is empty, return.
 - Step 2: Traverse to the last node.
 - Step 3: Update the second-to-last node's next pointer to null.
- c. From a given position:
 - Step 1: Traverse to the node at the given position.
 - Step 2: Adjust the surrounding nodes' pointers to exclude the node to be deleted.



4. Traversal

Objective: Visit each node in the list.

- a. Forward Traversal:
 - Step 1: Start from the head.
 - Step 2: While the current node is not null, process the current node and move to the next node.
- b. Backward Traversal:
 - Step 1: Start from the tail (requires maintaining a tail pointer or traversing to the end).
 - Step 2: While the current node is not null, process the current node and move to the previous node.

5. Search

Objective: Find a node with a specified value.

- Step 1: Start from the head.
- Step 2: While the current node is not null, compare its value with the target value.
- Step 3: If a match is found, return the node.
- Step 4: Move to the next node.
- Step 5: If the end of the list is reached without finding the value, return null or None.

6. Reverse

Objective: Reverse the order of nodes in the list.

- Step 1: Start from the head.
- Step 2: Initialize a temporary variable to null to store the previous node.
- Step 3: While the current node is not null:
 - Step 3a: Swap the current node's next and previous pointers.

- Step 3b: Move the temporary variable to the current node.
- Step 3c: Move the current node to its new next node (previously the previous node).
- Step 4: After the loop, update the head pointer to the last processed node (stored in the temporary variable).

Code:

```
#include <stdio.h>

#include <stdlib.h>

// Define the structure of a node in a doubly linked list
typedef struct Node {    int data;    struct Node* prev;
    struct Node* next;
} Node;

// Function to create a new node
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;    newNode->prev = NULL;

    newNode->next = NULL;
    return newNode;
}

// Function to insert a node at the beginning
void insertAtBeginning(Node** head, int data)
{
    Node* newNode = createNode(data);    if
(*head == NULL) {
        *head = newNode;
```



```
} else {  
    newNode->next = *head;  
    (*head)->prev = newNode;  
    *head = newNode;  
}  
}  
  
// Function to insert a node at the end  
void insertAtEnd(Node** head, int data) {  
    Node* newNode = createNode(data);    if  
    (*head == NULL) {  
        *head = newNode;  
    } else {  
        Node* temp = *head;  
        while (temp->next != NULL) {  
            temp = temp->next;  
        }  
        temp->next = newNode;    newNode->  
        prev = temp;  
    }  
}  
  
// Function to insert a node at a given position void  
insertAtPosition(Node** head, int data, int position) {    if  
    (position == 1) {        insertAtBeginning(head, data);  
    return;
```



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

```
}  
  
Node* newNode = createNode(data);  
  
Node* temp = *head;  
  
for (int i = 1; i < position - 1 && temp != NULL; i++) {  
  
temp = temp->next;  
  
}  
  
if (temp == NULL) {  
printf("Position out of range\n");  
free(newNode);    return;  
}  
  
newNode->next = temp->next;  
if (temp->next != NULL) {  
temp->next->prev = newNode;  
}  
  
temp->next = newNode;    newNode->  
>prev = temp;  
}  
  
// Function to delete a node from the beginning  
void deleteFromBeginning(Node** head) {    if  
(*head == NULL) {        printf("List is empty\n");  
return;  
}
```



```
Node* temp = *head;
*head = (*head)->next;
if (*head != NULL) {
    (*head)->prev = NULL;
}
free(temp);
}

// Function to delete a node from the end
void deleteFromEnd(Node** head) {    if
(*head == NULL) {        printf("List is
empty\n");        return;
}
Node* temp = *head; while
(temp->next != NULL) {
    temp = temp->next;
}
if (temp->prev != NULL) {        temp->prev-
>next = NULL;
} else {
    *head = NULL;
}
free(temp);
}
```



```
// Function to delete a node from a given position
void deleteFromPosition(Node** head, int position)
{   if (*head == NULL) {       printf("List is empty\n");
return;
    }
    if (position == 1) {
deleteFromBeginning(head);
        return;
    }
    Node* temp = *head;
    for (int i = 1; i < position && temp != NULL; i++) {
temp = temp->next;
    }
    if (temp == NULL) {
        printf("Position out of range\n");
return;
    }
    if (temp->next != NULL) {        temp->next-
>prev = temp->prev;
    }
    if (temp->prev != NULL) {        temp->prev-
>next = temp->next;
    }
    free(temp);
```




```
}

// Function to traverse the list in forward direction
void traverseForward(Node* head) {   Node*
temp = head;   while (temp != NULL) {
printf("%d ", temp->data);       temp = temp-
>next;
}
printf("\n");
}

// Function to traverse the list in backward direction
void traverseBackward(Node* head) {   if (head ==
NULL) {       return;

}

Node* temp = head;   while
(temp->next != NULL) {
temp = temp->next;
}

while (temp != NULL) {
printf("%d ", temp->data);
temp = temp->prev;
}

printf("\n");
}

// Function to search for a node with a given value
```



```
Node* search(Node* head, int data) {  
    Node* temp = head;    while (temp  
    != NULL) {        if (temp->data ==  
    data) {            return temp;  
        }  
        temp = temp->next;  
    }  
    return NULL;  
}
```

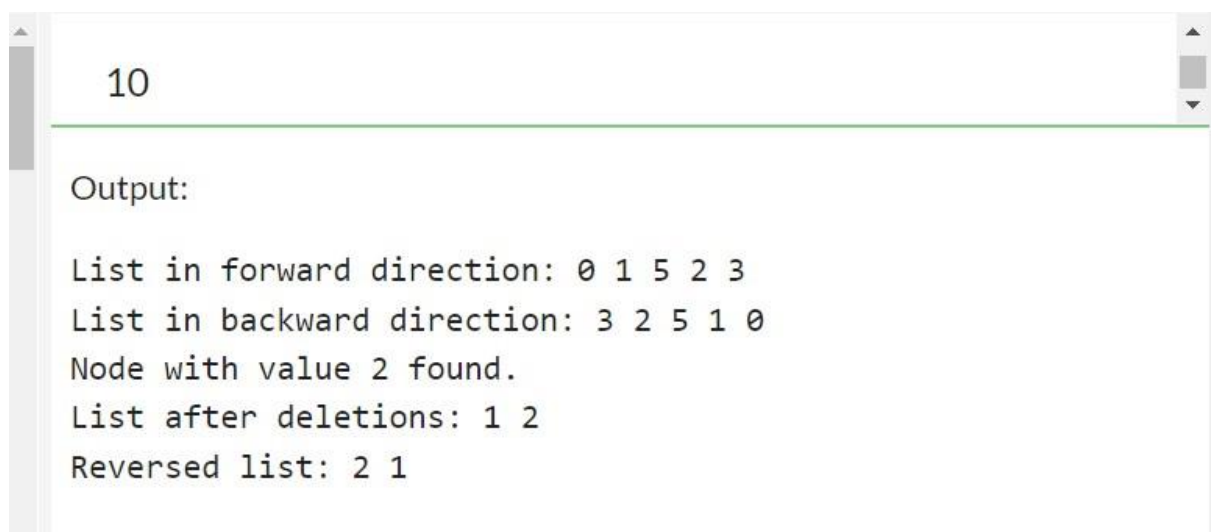
```
// Function to reverse the list  
void reverseList(Node** head) {  
    if (*head == NULL) {        return;  
    }  
    Node* current = *head;    Node*  
    temp = NULL;    while (current !=  
    NULL) {        temp = current->prev;  
    current->prev    =    current->next;  
    current->next = temp;        current  
    = current->prev;  
    }  
    if (temp != NULL) {  
        *head = temp->prev;  
    }  
}
```



```
}  
  
// Main function to demonstrate the operations int  
main() {  
    Node* head = NULL;    //  
  
    Insert nodes  
  
    insertAtEnd(&head, 1);  
    insertAtEnd(&head, 2);  
    insertAtEnd(&head, 3);  
    insertAtBeginning(&head, 0);  
    insertAtPosition(&head, 5, 3);  
  
    // Traverse the list  
    printf("List in forward direction: ");  
    traverseForward(head);    printf("List  
in backward direction: ");  
    traverseBackward(head);    //  
  
    Search for a node    int searchValue =  
    2;  
  
    Node* searchResult = search(head, searchValue);  
    if (searchResult != NULL)  
    {  
        printf("Node with value %d found.\n", searchValue);  
    } else  
    {  
        printf("Node with value %d not found.\n", searchValue);  
    }  
}
```

```
}  
  
// Delete nodes  
deleteFromPosition(&head, 3);  
deleteFromEnd(&head);  
deleteFromBeginning(&head); //  
Traverse the list again  printf("List  
after deletions: ");  
traverseForward(head);  
  
// Reverse the list  
reverseList(&head);  
printf("Reversed list: ");  
traverseForward(head);  
return 0;  
}
```

OUTPUT :



```
10  
  
Output:  
  
List in forward direction: 0 1 5 2 3  
List in backward direction: 3 2 5 1 0  
Node with value 2 found.  
List after deletions: 1 2  
Reversed list: 2 1
```



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

EXPERIMENT NO. 8

Objective(s): To develop efficient search algorithms that locate the position of a target value within a sorted array using binary search and interpolation search techniques.

Outcome: Implementation of binary search and interpolation search algorithms that can quickly find the index of a target element in a sorted array, improving search efficiency compared to linear search.

Problem Statement: Implement binary search and interpolation search.

Background Study:

1. Binary Search

Objective: To locate the position of a target value within a sorted array efficiently by repeatedly dividing the search interval in half.

How Binary Search Works

1. Initial Setup:

■ Start with two pointers: low at the beginning (index 0) and high at the end (index n-1) of the array.

2. Midpoint Calculation:

- Calculate the midpoint (mid) using the formula: $\text{mid} = \text{low} + (\text{high} - \text{low}) / 2$.
3. Comparison:
 - Compare the target value with the element at the midpoint ($\text{array}[\text{mid}]$).
 - If the target is equal to $\text{array}[\text{mid}]$, return the index mid.
 - If the target is less than $\text{array}[\text{mid}]$, narrow the search to the left half by setting $\text{high} = \text{mid} - 1$.
 - If the target is greater than $\text{array}[\text{mid}]$, narrow the search to the right half by setting $\text{low} = \text{mid} + 1$.
4. Repeat:
 - Repeat the process until low exceeds high or the target is found.



- Time Complexity
 1. Best Case: $O(1)$, when the target is at the midpoint on the first check.

2. Average and Worst Case: $\{O(\log n)\}$, because the search interval is halved in each step.

- Space Complexity

1. Iterative Version: $\{O(1)\}$, as it uses a constant amount of space.

2. Recursive Version: $\{O(\log n)\}$, due to the recursion stack.

- Use Cases

1. Binary search is effective for searching in large sorted datasets where the cost of sorting the data is justified by the number of searches performed.

2. Interpolation Search

Objective: To improve the efficiency of search operations for uniformly distributed sorted arrays by estimating the position of the target value.

How Interpolation Search Work

1. Initial Setup:

- Similar to binary search, start with two pointers: low at the beginning (index 0) and high at the end (index n-1) of the array.

2. Position Calculation:

- Calculate the estimated position (pos) of the target value using the formula:

- $$pos = low + \frac{(target - array[low]) \times (high - low)}{array[high] - array[low]}$$

3. Comparison:

- Compare the target value with the element at the estimated position (array[pos]).

- If the target is equal to array[pos], return the index pos.

- If the target is less than array[pos], narrow the search to the left side by setting high = pos - 1.

- If the target is greater than array[pos], narrow the search to the right side by setting low = pos + 1.
- 4. Repeat:
- Repeat the process until low exceeds high or the target is found.

Interpolation Search

$$\text{mid} = \text{low} + ((\text{target} - \text{arr}[\text{low}]) * \frac{(\text{high} - \text{low})}{(\text{arr}[\text{high}] - \text{arr}[\text{low}])})$$

															target = 18
1st Iteration	10	12	13	16	18	19	20	21	22	23	24	33	35	42	47
	low			mid			arr[mid] < 18							high	
2nd Iteration	10	12	13	16	18	19	20	21	22	23	24	33	35	42	47
	low			mid			arr[mid] == 18							high	

q.opengenus.org

- Time Complexity
 1. Best Case: $O(1)$, when the target is at the estimated position on the first check.
 2. Average Case: $O(\log \log n)$ for uniformly distributed data.
 3. Worst Case: $O(n)$, when the distribution of elements is skewed.
 - Space Complexity
 1. Iterative Version: $O(1)$, as it uses a constant amount of space.
 2. Recursive Version: $O(\log \log n)$, due to the recursion stack.
 - Use Cases
 1. Interpolation search is particularly effective for large, uniformly distributed datasets where the values are spread evenly.
- Comparison: Binary Search vs. Interpolation Search
1. Binary Search:
 - Works well with any sorted dataset.



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

- Time complexity of $O(\log n)$.
 - Simple to implement.
2. Interpolation Search:

- Optimized for uniformly distributed data.
- Can achieve $O(\log n)$ time complexity in the best case.
- Slightly more complex to implement due to the estimation formula.
- Practical Implementation Considerations
 1. Data Distribution:
 - Use binary search for general-purpose searching in sorted arrays.
 - Use interpolation search for specific cases where data is uniformly distributed.
 2. Array Size:
 - For small to moderately sized arrays, the difference in performance might be negligible.
 - For very large arrays, interpolation search can provide significant performance improvements if the data distribution is suitable.
 3. Dynamic Data:
 - Both searches require the array to be sorted. Any dynamic updates (insertions or deletions) may necessitate re-sorting, impacting overall performance.
 - Understanding these search algorithms and their appropriate use cases ensures that you can select the most efficient algorithm for a given problem, optimizing search operations in various scenarios.

Algorithm (Student Work Area):

Algorithms for Binary Search and Interpolation Search

1. Binary Search

Objective: Efficiently locate the position of a target value within a sorted array.

- Algorithm:
 1. Initialization:

- Set low to the starting index of the array (0).
- Set high to the ending index of the array (n-1).
- 2. Loop:
 - While low is less than or equal to high:
 1. Calculate the midpoint: $mid = low + (high - low) / 2$.
 2. Compare the target value with array[mid]:
 - If the target equals array[mid], return mid.
 - If the target is less than array[mid], set high = mid - 1.
 - If the target is greater than array[mid], set low = mid + 1.
- 3. Completion:
 - If the loop ends without finding the target, return an indication that the target is not present (e.g., -1 or null).

2. Interpolation Search

Objective: Efficiently locate the position of a target value within a uniformly distributed sorted array.

- Algorithm:
- 4. Initialization:
 - Set low to the starting index of the array (0).
 - Set high to the ending index of the array (n-1).
- 5. Loop:
 - While low is less than or equal to high and the target is within the range of array[low] to array[high]:
 1. Estimate the position: $pos = low + ((target - array[low]) * (high - low)) / (array[high] - array[low])$.
 2. Compare the target value with array[pos]:

- If the target equals array[pos], return pos.
 - If the target is less than array[pos], set high = pos - 1.
 - If the target is greater than array[pos], set low = pos + 1.
6. Completion:
- If the loop ends without finding the target, return an indication that the target is not present (e.g., -1 or null).
 - These algorithms provide a structured approach to efficiently search for a target value in a sorted array, with binary search being more universally applicable and interpolation search offering potential performance benefits for uniformly distributed data.

Code:

Binary Search Implementation

```
#include <stdio.h>

// Function to perform binary search on a sorted array
int binarySearch(int array[], int size, int target) {    int
low = 0;    int high = size - 1; while (low <= high) {
    int mid = low + (high - low) / 2;
// Check if target is present at mid
if (array[mid] == target) {        return
mid;
    }
// If target is greater, ignore left half
```



```
        else if (array[mid] < target) {  
            low = mid + 1;  
        }  
        // If target is smaller, ignore right half  
    else {        high = mid - 1;  
    }  
}  
// Target not found  
return -1;  
}  
// Example usage int  
main() {  
    int array[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};  
    int size = sizeof(array) / sizeof(array[0]);    int  
    target = 23;  
    int result = binarySearch(array, size, target);  
    if (result != -1) {  
        printf("Element found at index %d\n", result);  
    } else {  
        printf("Element not found\n");  
    }  
    return 0;  
}
```



Interpolation Search Implementation

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

```
// Function to perform interpolation search on a sorted array
```

```
int interpolationSearch(int array[], int size, int target) {    int
```

```
low = 0;    int high = size - 1;
```

```
    while (low <= high && target >= array[low] && target <= array[high]) {
```

```
        // Estimate the position
```

```
        int pos = low + ((double)(high - low) / (array[high] - array[low])) * (target -  
array[low]);
```

```
        // Check if target is present at pos
```

```
if (array[pos] == target) {        return
```

```
pos;
```

```
    }
```

```
    // If target is greater, ignore left half
```

```
else if (array[pos] < target) {        low
```

```
= pos + 1;
```

```
    }
```

```
    // If target is smaller, ignore right half
```

```
else {
```

```
    high = pos - 1;
```

```
    }
```

```
}
```



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

```
// Target not found
return -1;
}
// Example usage int
main() {
    int array[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};
    int size = sizeof(array) / sizeof(array[0]);    int
    target = 23;
    int result = interpolationSearch(array, size, target);
    if (result != -1) {
        printf("Element found at index %d\n", result);
    } else {
        printf("Element not found\n");
    }
    return 0;
}
```

OUTPUT :

Enrollment No :-2303031080073



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

12

Output:

Element found at index 5



EXPERIMENT NO. 9

Objective(s): Implement and analyze various sorting algorithms to efficiently arrange elements in ascending or descending order.

Outcome: Develop functions for Bubble sort, Selection sort, Insertion sort, Quick sort, and Merge sort that accurately sort input arrays, demonstrating their respective time complexities and performance characteristics in sorting operations.

Problem Statement: Implement Bubble sort, selection sort, Insertion sort, quick sort, merge sort.

Background Study:

1. Bubble Sort

Objective: Sort an array of elements by repeatedly swapping adjacent elements if they are in the wrong order.

How Bubble Sort Works

1. Iteration through Array:

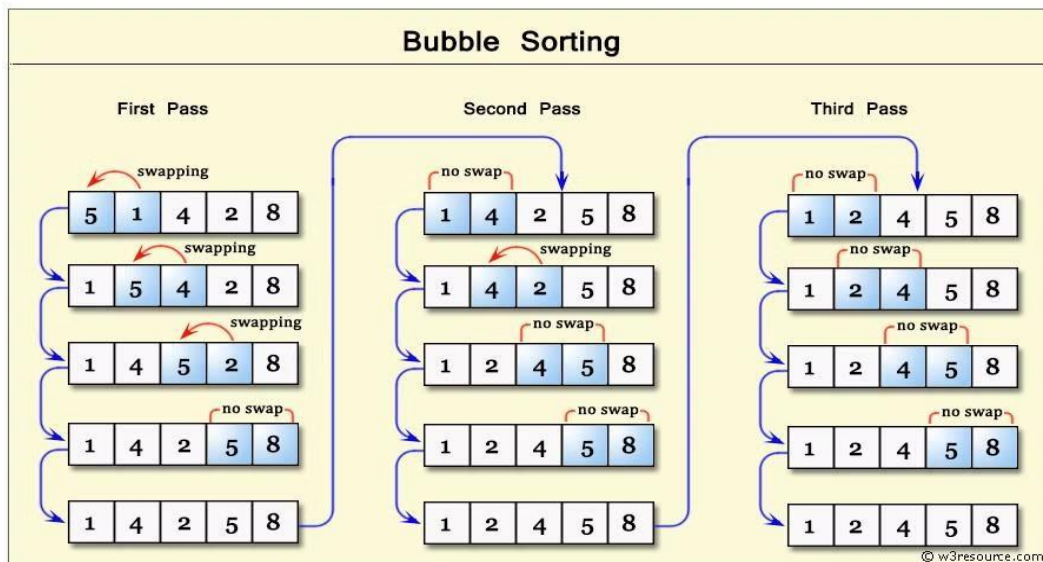
- Pass through the array multiple times.
- Compare each pair of adjacent elements.
- Swap them if they are in the wrong order (smaller follows larger).

2. Passes:

- After each pass, the largest unsorted element reaches its correct position at the end of the array.
- Reduce the number of comparisons in subsequent passes.

3. Complexity:

- Time Complexity: $O(n^2)$ in the worst and average cases.
- Space Complexity: $O(1)$ (in-place sorting).



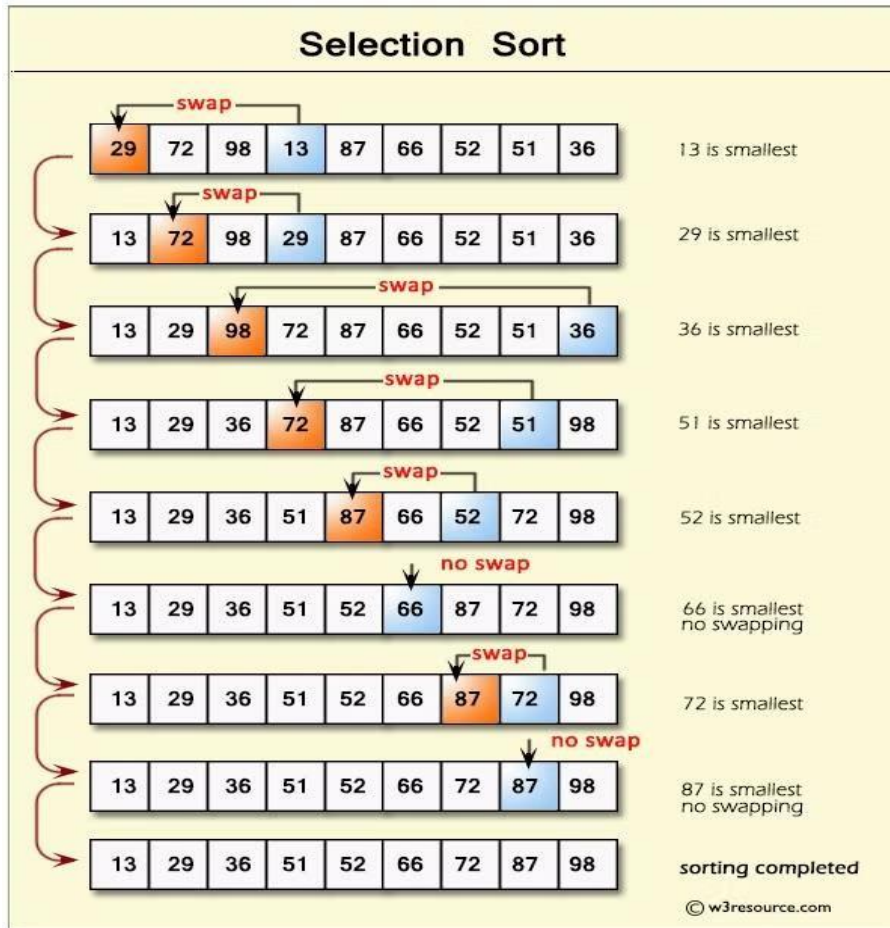
2. Selection Sort

Objective: Sort an array by repeatedly finding the minimum element (ascending order) and swapping it with the first unsorted element.

How Selection Sort Works

1. Iteration through Array:
 - Divide the array into two subarrays: sorted and unsorted.
 - Find the smallest element from the unsorted subarray.
 - Swap it with the first element of the unsorted subarray.
2. Repeat:
 - Expand the sorted subarray by one element.
 - Continue until the entire array is sorted.
3. Complexity:
 - Time Complexity: $O(n^2)$ in all cases (worst, average, and best).

- Space Complexity: $O(1)$ (in-place sorting).



3. Insertion Sort

Objective: Sort an array by inserting each element into its correct position in a growing sorted subarray.

How Insertion Sort Works

1. Iteration through Array:

- Divide the array into sorted and unsorted subarrays.
- Pick an element from the unsorted subarray and insert it into its correct position in the sorted subarray.

2. Shifting Elements:

○

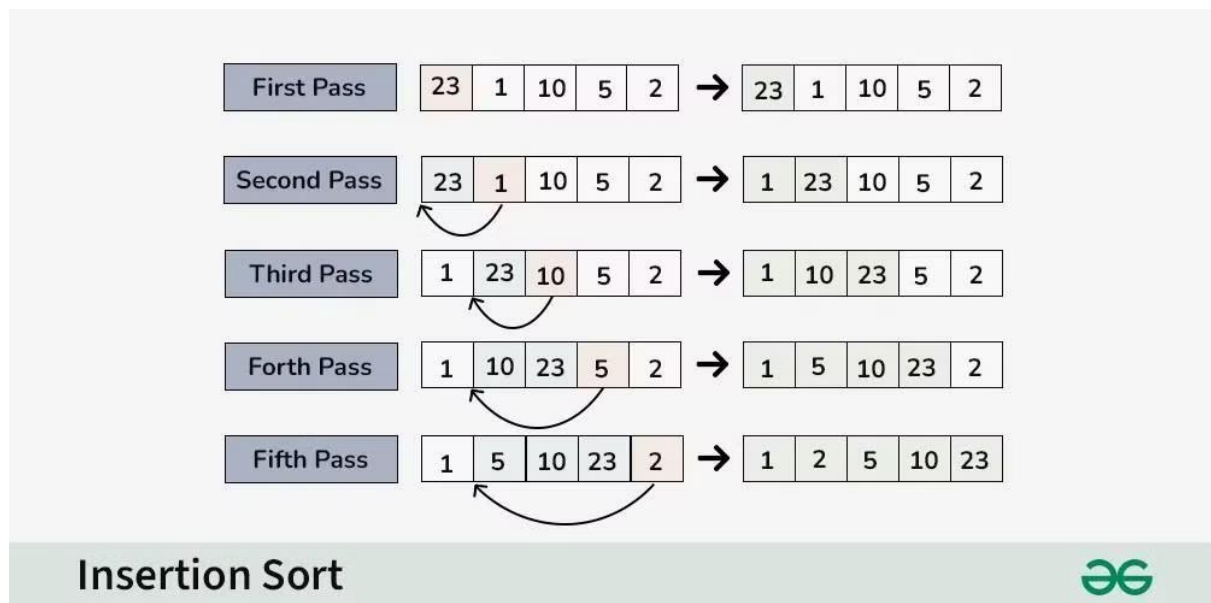
Shift larger elements one position to the right to make space for the inserted element.

○ Continue until all elements are sorted.

3. Complexity:

○ Time Complexity: $(O(n^2))$ in the worst and average cases, $(O(n))$ in the best case (already sorted).

○ Space Complexity: $(O(1))$ (in-place sorting).



4. Quick Sort

Objective: Sort an array using a divide-and-conquer approach by recursively partitioning around a pivot element.

How Quick Sort Works

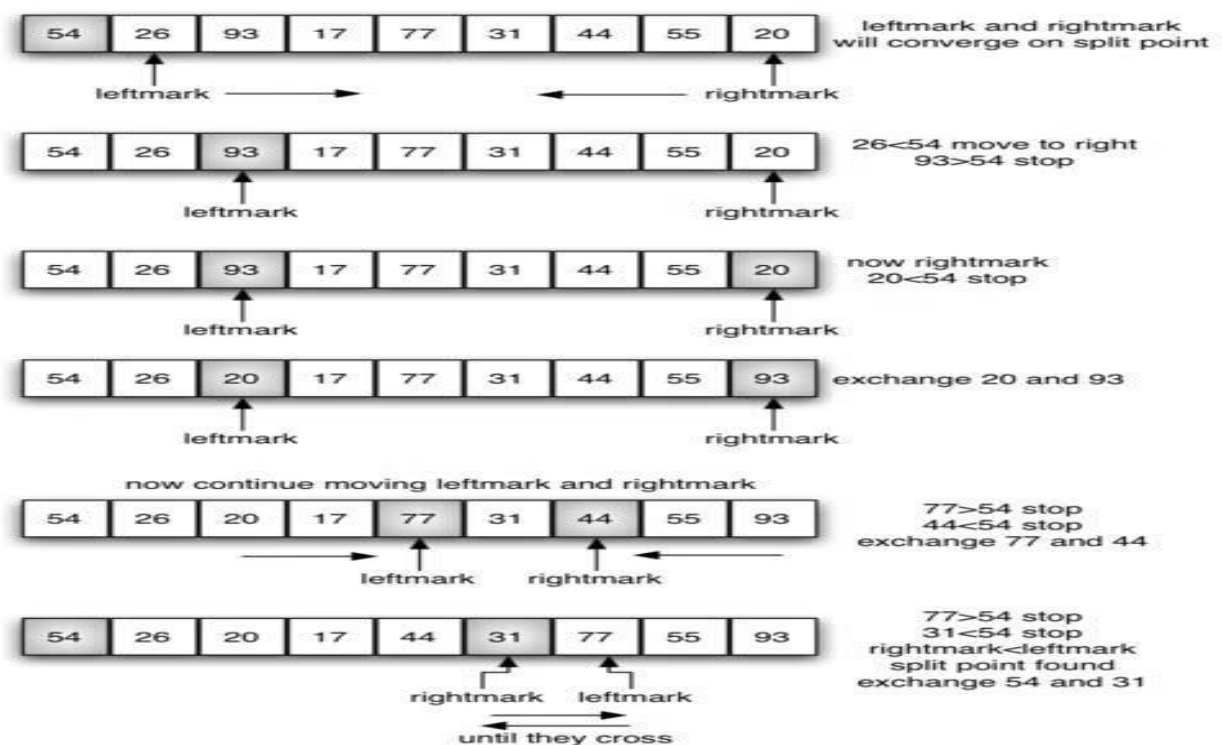
1. Partitioning:

○ Choose a pivot element (typically the last element).

○ Reorder the array such that all elements less than the pivot are before it, and all greater elements are after it.

2. Recursion:

- Recursively apply the same partitioning to the subarrays formed by the pivot.
- Sort the subarrays until the entire array is sorted.
- 3. Complexity:
 - Time Complexity: $\mathcal{O}(n \log n)$ on average, $\mathcal{O}(n^2)$ in the worst case (unbalanced partitions).
 - Space Complexity: $\mathcal{O}(\log n)$ to $\mathcal{O}(n)$ depending on the implementation (typically $\mathcal{O}(\log n)$ due to recursion stack).



5. Merge Sort

Objective: Sort an array using a divide-and-conquer approach by recursively splitting the array into halves, sorting each half, and merging them back together.

How Merge Sort Works

o

1. Divide:

Divide the array into two halves recursively until each subarray contains one element (base case).

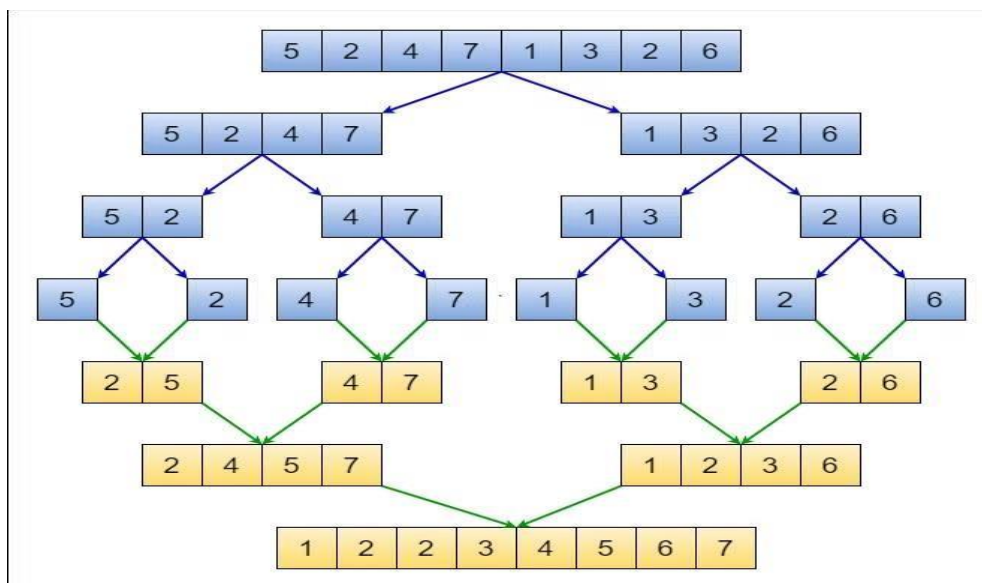
2. Merge:

o Merge two sorted subarrays into a single sorted array by comparing elements one by one.

3. Complexity:

o Time Complexity: $\mathcal{O}(n \log n)$ in all cases (worst, average, and

best). o Space Complexity: $\mathcal{O}(n)$ due to the auxiliary array used for merging.



Comparison of Sorting Algorithms

- Bubble Sort, Selection Sort, and Insertion Sort have $\mathcal{O}(n^2)$ time complexity, making them inefficient for large datasets compared to $\mathcal{O}(n \log n)$ algorithms like Quick Sort and Merge Sort.
- Quick Sort is typically faster than Merge Sort due to better cache locality and in-place partitioning, but can degrade to $\mathcal{O}(n^2)$ if not implemented carefully.



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

○

- Merge Sort is stable and guarantees $O(n \log n)$ performance but requires additional space for merging.

Algorithm (Student Work Area):



1. Bubble Sort

Objective: Sort an array of elements by repeatedly swapping adjacent elements if they are in the wrong order.

Algorithm:

1. Start from the beginning of the array.
2. Compare each pair of adjacent elements.
3. If the elements are in the wrong order (larger followed by smaller for ascending order), swap them.
4. Repeat steps 1-3 for each pair of adjacent elements until no more swaps are needed in a pass.

2. Selection Sort

Objective: Sort an array by repeatedly finding the minimum element and swapping it with the first unsorted element.

Algorithm:

1. Iterate through the array from the beginning.
2. Assume the current element is the minimum.
3. Compare it with each subsequent element to find the actual minimum.
4. Swap the current element with the minimum element found.
5. Repeat steps 1-4, expanding the sorted portion of the array by one element each time, until the entire array is sorted.

3. Insertion Sort

Objective: Sort an array by inserting each element into its correct position in a growing sorted subarray.

Algorithm:

1. Start from the second element of the array.

2. Compare the current element with elements in the sorted subarray (to its left).
3. Shift all larger elements one position to the right.
4. Insert the current element into its correct position in the sorted subarray.
5. Repeat steps 1-4 for each subsequent element until the entire array is sorted.

4. Quick Sort

Objective: Sort an array using a divide-and-conquer approach by recursively partitioning around a pivot element.

Algorithm:

1. Choose a pivot element from the array (usually the last element).
2. Partition the array into two subarrays:
 - Elements less than the pivot (left subarray).
 - Elements greater than or equal to the pivot (right subarray).
3. Recursively apply the same partitioning process to the left and right subarrays.
4. Concatenate the sorted left subarray, pivot, and sorted right subarray to get the sorted array.

5. Merge Sort

Objective: Sort an array using a divide-and-conquer approach by recursively splitting the array into halves, sorting each half, and merging them back together.

Algorithm:

1. Divide the array into two halves.
2. Recursively sort each half by applying merge sort.



3. Merge the two sorted halves into a single sorted array:
 - Compare elements from the two halves.
 - Place the smaller element in the merged array.
 - Move to the next element in the respective subarray.
4. Repeat step 3 until all elements are merged into a single sorted array.

Code:

Bubble Sort #include

<stdio.h>

```
void bubbleSort(int array[], int size) {
```

```
    for (int i = 0; i < size - 1; ++i) {
```

```
        // Flag to optimize by stopping if no swap is made
```

```
        int swapped = 0;
```

```
        for (int j = 0; j < size - 1 - i; ++j) {
```

```
            // Swap if current element is greater than next element
```

```
            if (array[j] > array[j + 1]) {                int temp = array[j];
```

```
                array[j] = array[j + 1];                array[j + 1] = temp;
```

```
                swapped = 1;
```

```
            }
```

```
        }
```

```
    // If no elements were swapped, array is sorted
```

```
    if (swapped == 0) {
```

```
        break;
```



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

```
    }  
}  
}  
// Example usage int  
main() {  
    int array[] = {64, 25, 12, 22, 11};    int  
size = sizeof(array) / sizeof(array[0]);  
bubbleSort(array, size); printf("Sorted  
array: ");    for (int i = 0; i < size; ++i) {  
printf("%d ", array[i]);  
    } printf("\\n"); return  
0;  
}
```

Selection Sort #include

<stdio.h>

```
void selectionSort(int array[], int size) {
```

```
for (int i = 0; i < size - 1; ++i) {    int
```

```
minIndex = i;
```

```
    // Find the index of the minimum element in the remaining unsorted array
```

```
for (int j = i + 1; j < size; ++j) {    if (array[j] < array[minIndex]) {
```

```
minIndex = j;
```

```
    }
```

```
}
```



// Swap the found minimum element with the first element of the unsorted array

```
    int temp = array[minIndex];  
    array[minIndex] = array[i];  
    array[i] = temp;  
}  
}
```

// Example usage int

```
main() {  
    int array[] = {64, 25, 12, 22, 11};    int  
    size = sizeof(array) / sizeof(array[0]);  
    selectionSort(array, size); printf("Sorted  
array: ");    for (int i = 0; i < size; ++i) {  
        printf("%d ", array[i]);  
    }  
    printf("\n");  
    return 0;  
}
```

Insertion Sort #include

<stdio.h>

```
void insertionSort(int array[], int size) {  
    for (int i = 1; i < size; ++i) {  
        int key = array[i];    int j =  
        i - 1;
```



```
// Move elements of array[0..i-1], that are greater than key, to one position  
ahead of their current position      while (j >= 0 && array[j] > key) {  
array[j + 1] = array[j];
```

```
    j--;
```

```
}
```

```
array[j + 1] = key;
```

```
}
```

```
}
```

```
// Example usage int
```

```
main() {
```

```
    int array[] = {64, 25, 12, 22, 11};    int
```

```
size = sizeof(array) / sizeof(array[0]);
```

```
insertionSort(array, size); printf("Sorted
```

```
array: ");    for (int i = 0; i < size; ++i) {
```

```
printf("%d ", array[i]);
```

```
}
```

```
printf("\n");
```

```
return 0;
```

```
}
```

```
Quick Sort #include
```

```
<stdio.h> void swap(int*
```

```
a, int* b) {    int temp =
```

```
*a;    *a = *b;
```

```
*b = temp;
```



```
}  
  
int partition(int array[], int low, int high) {  
    int pivot = array[high]; // pivot  
    int i = (low  
- 1); // Index of smaller element  
    for (int j =  
low; j <= high - 1; j++) {  
        // If current element is smaller than or equal to pivot  
        if (array[j] <= pivot) {  
            i++; // increment index of smaller element  
            swap(&array[i], &array[j]);  
        }  
    }  
    swap(&array[i + 1], &array[high]);  
    return (i + 1);  
}  
  
void quickSort(int array[], int low, int high) {  
    if (low < high) {  
        // pi is partitioning index, array[p] is now at right place  
        int pi = partition(array, low, high);  
  
        // Separately sort elements before partition and after partition  
        quickSort(array, low, pi - 1);    quickSort(array, pi + 1, high);  
    }  
}  
  
// Example usage  
int  
main() {
```



```
int array[] = {64, 25, 12, 22, 11};    int
size = sizeof(array) / sizeof(array[0]);
quickSort(array, 0, size - 1);
printf("Sorted array: ");    for (int i = 0; i <
size; ++i) {        printf("%d ", array[i]);
    }
    printf("\n");
return 0;
}
```

Merge Sort

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

```
void merge(int array[], int left, int mid, int right) {
int i, j, k;    int n1 = mid - left + 1;    int n2 = right
- mid;
```

```
    // Create temporary arrays
```

```
int L[n1], R[n2];
```

```
    // Copy data to temporary arrays L[] and R[]
```

```
for (i = 0; i < n1; i++)        L[i] = array[left + i];
```

```
for (j = 0; j < n2; j++)
```

```
    R[j] = array[mid + 1 + j];
```

```
    // Merge the temporary arrays back into array[left..right]
```

```
i = 0; // Initial index of first subarray    j = 0; // Initial
```

```
index of second subarray    k = left; // Initial index of
```

```
merged subarray while (i < n1 && j < n2) {    if (L[i] <=
R[j]) {        array[k] = L[i];        i++;    } else {
array[k] = R[j];        j++;
    }
k++;
}
// Copy the remaining elements of L[], if any
while (i < n1) {    array[k] = L[i];

    i++;
k++;
}
// Copy the remaining elements of R[], if any
while (j < n2) {    array[k] = R[j];    j++;
k++;
}
}
void mergeSort(int array[], int left, int right) {
if (left < right) {
    // Same as (left + right) / 2, but avoids overflow for large left and right
    int mid = left + (right - left) / 2; // Sort first and second halves
    mergeSort(array, left, mid);    mergeSort(array, mid + 1, right);    //
    Merge the sorted halves    merge(array, left, mid, right);
}
}
```




Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

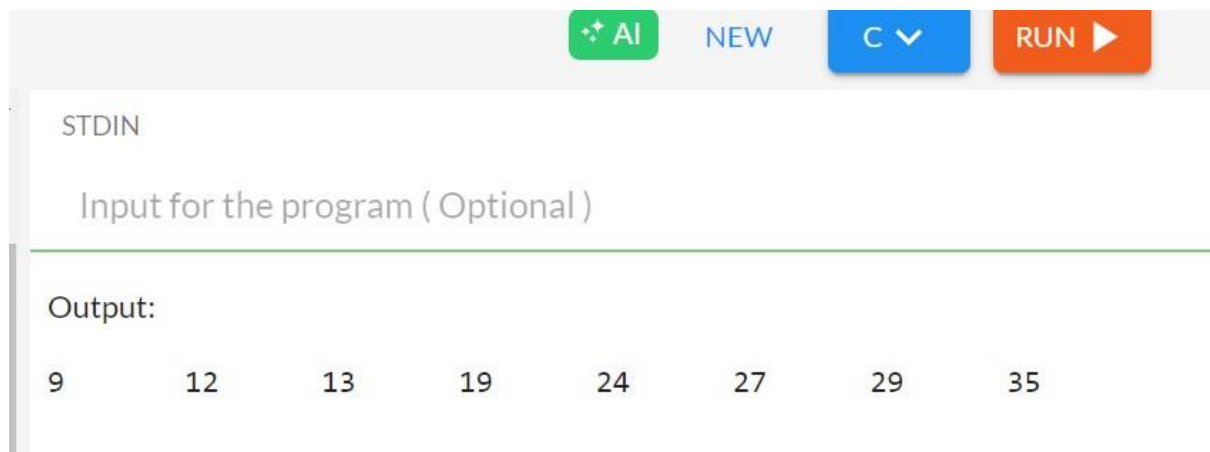
Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

// Example usage int

```
main() {  
    int array[] = {64, 25, 12, 22, 11};  
    int size = sizeof(array) / sizeof(array[0]);  
    mergeSort(array, 0, size - 1);  
    printf("Sorted array: ");    for (int i = 0; i <  
size; ++i) {        printf("%d ", array[i]);  
        }  
    printf("\n"); return  
0;  
}
```

OUTPUT :





Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

EXPERIMENT NO. 10

Objective(s): Implement a Binary Search Tree (BST) data structure to efficiently store and manage elements while preserving the BST properties.

Outcome: Develop functions for creating a BST, inserting nodes respecting the BST property, and deleting nodes while maintaining the BST structure and properties, ensuring efficient searching, insertion, and deletion operations.

Problem Statement: Implement Binary search Tree and its operations (creation, insertion, deletion).

Background Study:

Binary Search Tree (BST) is a hierarchical data structure that organizes elements in a way that allows for fast search, insertion, and deletion operations. It follows the properties:

1. Binary Tree Structure:
 - Each node has at most two children: left and right.
2. Binary Search Property:
 - For every node n , all values in its left subtree are less than n , and all values in its right subtree are greater than n .

Key Operations

1. Creation:
 - Start with an empty tree or initialize with a root node.
2. Insertion:
 - Place new elements by comparing with the current node and moving left or right based on the BST properties until an appropriate position is found.
3. Deletion:



- Remove nodes while maintaining the BST properties:
 - Case 1: Node has no children - Remove it directly.
 - Case 2: Node has one child - Replace it with its child.
 - Case 3: Node has two children - Find the successor (or predecessor), replace the node with it, and delete the successor (or predecessor).
- 4. Traversal:
 - Visit nodes in a specific order:
 - Inorder: Left subtree, current node, right subtree (sorted order).
 - Preorder: Current node, left subtree, right subtree (used to create a copy of the tree).
 - Postorder: Left subtree, right subtree, current node (used in deleting a tree).
- 5. Search:
 - Find a specific node based on its value using the BST properties (left for smaller, right for larger).

Balancing

1. Balanced vs Unbalanced:
 - Balanced BST: Ensures that the height of the tree remains logarithmic ($O(\log n)$), optimizing search, insertion, and deletion operations.
 - Unbalanced BST: May degrade to linear time ($O(n)$) operations in the worst case, losing the efficiency benefits.
2. Self-Balancing Trees:
 - AVL Tree: Maintains a balance factor for each node to ensure the tree remains balanced after insertions and deletions ($O(\log n)$ operations).
 - Red-Black Tree: Ensures balanced conditions using color attributes on nodes, offering efficient operations ($O(\log n)$).



- Splay Tree: Reorganizes itself based on access patterns to bring frequently accessed nodes closer to the root.

Applications

1. Database Systems:

- Efficient indexing and searching of records.

2. Symbol Tables:

- Storing identifiers in compilers and symbol tables in interpreters.

3. File Systems:

- Representing directory structures for efficient file retrieval.

4. Networking:

- Routing tables in routers for fast packet forwarding.

Advantages and Disadvantages ●

Advantages:

- Efficient average-case performance ($O(\log n)$ operations).
- Simple implementation of operations.

● Disadvantages:

- Degradation to $O(n)$ in the worst case (unbalanced tree).
- Extra space for pointers.

Algorithm (Student Work Area):

1. Creation

● Algorithm:

- Start with an empty tree or initialize with a root node.
- If inserting nodes dynamically, allocate memory for each new node and assign values.



2. Insertion

- Algorithm:
 - Begin at the root.
 - Compare the value of the node to be inserted with the current node's value.
 - If smaller, move to the left child; if larger or equal, move to the right child.
 - Repeat until a suitable empty spot (leaf) is found.
 - Insert the new node as a left or right child based on the comparison.

3. Deletion

- Algorithm:
 - Find the node to delete.
 - If the node has no children (leaf node), simply remove it.
 - If the node has one child, replace it with its child.
 - If the node has two children:
 - Find the inorder successor (smallest node in the right subtree) or predecessor (largest node in the left subtree).
 - Replace the node to be deleted with the successor or predecessor.
 - Recursively delete the successor or predecessor from the subtree.

Traversal

- Inorder Traversal:
 - Visit left subtree.
 - Visit the current node.
 - Visit right subtree.
 - Used to print elements in non-decreasing order.



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

- Preorder Traversal:
 - Visit the current node.
 - Visit left subtree.
 - Visit right subtree.
 - Used to create a copy of the tree or to prefix expression evaluation.
- Postorder Traversal:
 - Visit left subtree.
 - Visit right subtree.
 - Visit the current node.
 - Used in deleting a tree or for postfix expression evaluation.

5. Searching

- Algorithm:
 - Start from the root.
 - Compare the value of the node to be searched with the current node's value.
 - If smaller, move to the left child; if larger, move to the right child.
 - Repeat until the node is found or a null pointer is encountered.

Code:

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

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

Definition of a BST node

```
struct TreeNode {    int
```

```
data;
```

```
    struct TreeNode *left;
```

```
    struct TreeNode *right;
```

Enrollment No :-2303031080073



```
};
```

```
// Function to create a new node
```

```
struct TreeNode* createNode(int data) {
```

```
    struct TreeNode* newNode = (struct TreeNode*) malloc(sizeof(struct  
    TreeNode));    newNode->data = data;    newNode->left = NULL;  
    newNode->right = NULL;    return newNode;  
}
```

```
// Function to insert a new node in BST
```

```
struct TreeNode* insertNode(struct TreeNode* root, int data) {
```

```
    // If the tree is empty, return a new node  
    if (root == NULL) {        return  
    createNode(data);  
    }
```

```
    // Otherwise, recur down the tree  
    if (data < root->data) {  
        root->left = insertNode(root->left, data);  
    } else if (data > root->data) {  
        root->right = insertNode(root->right, data);  
    }
```

```
    // return the (unchanged) node pointer  
    return root;  
}
```

```
// Function to find the inorder successor in BST
```



```
struct TreeNode* minValueNode(struct TreeNode* node) {  
    struct TreeNode* current = node;    // Loop down to find  
    the leftmost leaf    while (current && current->left !=  
    NULL) {        current = current->left;  
    }  
    return current;  
}  
  
// Function to delete a node from BST  
struct TreeNode* deleteNode(struct TreeNode* root, int data) {  
    // Base case: If the tree is empty  
    if (root == NULL) { return root;  
    }  
    // Recur down the tree  
    if (data < root->data) {  
        root->left = deleteNode(root->left, data);  
    } else if (data > root->data) {  
        root->right = deleteNode(root->right, data);  
    } else {  
        // Node found with the data  
        // Case 1: Node with only one child or no child  
        if (root->left == NULL) {  
            struct TreeNode* temp = root->right;  
            free(root);  
            return temp;  
        }
```




```
    }

    else if (root->right == NULL) {
        struct TreeNode* temp = root->left;
        free(root);    return temp;
    }

    // Case 2: Node with two children
        struct TreeNode* temp = minValueNode(root->right);
    // Copy the inorder successor's content to this node
    root->data = temp->data; // Delete the inorder
    successor
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}

// Function to perform inorder traversal of BST
void inorderTraversal(struct TreeNode* root) {
    if (root != NULL) {    inorderTraversal(root-
>left);    printf("%d ", root->data);
    inorderTraversal(root->right);
    }
}

// Function to search for a node in BST
```



```
struct TreeNode* search(struct TreeNode* root, int data) {  
    // Base Cases: root is null or data is present at root    if  
    (root == NULL || root->data == data) {        return root;  
    }  
    // data is greater than root's data  
    if (root->data < data) {        return  
    search(root->right, data);  
    }  
    // data is smaller than root's data  
    return search(root->left, data);  
}  
// Example usage int  
main() {  
    struct TreeNode* root = NULL;  
    root = insertNode(root, 50);  
    insertNode(root, 30);  
    insertNode(root, 20);  
    insertNode(root, 40);  
    insertNode(root, 70);  
    insertNode(root, 60);  
    insertNode(root, 80);  
    printf("Inorder traversal of the BST:  
    ");    inorderTraversal(root);
```



```
printf("\n"); // Search for a node
int key = 40;
struct TreeNode* result = search(root, key);
if (result) {
    printf("Node %d found in the BST.\n", key);
} else {
    printf("Node %d not found in the BST.\n", key);
}
// Delete a node root = deleteNode(root,
20); printf("Inorder traversal after deleting
20: "); inorderTraversal(root);
printf("\n"); return 0;
}
```

OUTPUT :

Input for the program (Optional)

Output:

Inorder traversal of the BST: 20 30 40 50 60 70 80

Node 40 found in the BST.

Inorder traversal after deleting 20: 30 40 50 60 70 80



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

EXPERIMENT NO. 11

Objective(s): Implement traversal algorithms (Preorder, Inorder, Postorder) on a Binary Search Tree (BST) to visit nodes in specific orders.

Outcome: Develop functions to traverse a BST and print nodes in Preorder (Root-Left-Right), Inorder (Left-Root-Right), and Postorder (Left-Right-Root) sequences, facilitating different ways to process or display tree elements.

Problem Statement: Implement Traversals Preorder Inorder Postorder on BST.

Background Study:



- Binary Search Tree (BST) is a hierarchical data structure that organizes nodes in a binary tree format. Each node has at most two children, referred to as the left child and the right child. The structure follows a specific property:
 - Binary Search Property: For every node n , all values in its left subtree are less than n , and all values in its right subtree are greater than n .

- Key Operations ○

Creation:

- Start with an empty tree or initialize with a root node.
- Insert nodes ensuring the binary search property is maintained.
- Traversal:
 - Visit nodes in specific orders to access or process data.
- Search:
 - Find a specific node based on its value using the BST properties.
- Insertion:
 - Add new nodes while maintaining the BST properties.
- Deletion:
 - Remove nodes while ensuring the BST properties are preserved.
- Traversal Algorithms

Traversing a BST involves visiting nodes in a specific order. The three main traversal algorithms are:

- Inorder Traversal (Left-Root-Right):
 - Visit the left subtree recursively.
 - Visit the root node.
 - Visit the right subtree recursively.



- In a BST, this traversal visits nodes in sorted order (ascending).
- Preorder Traversal (Root-Left-Right):
 - Visit the root node.
 - Visit the left subtree recursively.
 - Visit the right subtree recursively.
- Preorder traversal is useful for creating a copy of the tree or evaluating prefix expressions.
- Postorder Traversal (Left-Right-Root):
 - Visit the left subtree recursively.
 - Visit the right subtree recursively.
 - Visit the root node.
- Postorder traversal is useful for deleting a tree or evaluating postfix expressions.

Algorithm:

Traversal

- Inorder Traversal:
 - Traverse the left subtree recursively.
 - Visit the current node.
 - Traverse the right subtree recursively.
 - Used to print elements in non-decreasing order.
- Preorder Traversal:
 - Visit the current node.
 - Traverse the left subtree recursively.
 - Traverse the right subtree recursively.
 - Used for creating a copy of the tree or prefix expression evaluation.



- Postorder Traversal:
 - Traverse the left subtree recursively.
 - Traverse the right subtree recursively.
 - Visit the current node.
 - Used in deleting a tree or postfix expression evaluation.

Code:

```
#include <stdio.h>

#include <stdlib.h> //
Definition of a BST node
struct TreeNode {    int
data;    struct TreeNode*
left;    struct TreeNode*
right;
};

// Function to create a new node
struct TreeNode* createNode(int data) {
    struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct
TreeNode));    if (newNode == NULL) {
        fprintf(stderr, "Memory allocation failed\\n");
        exit(EXIT_FAILURE);
    }
    newNode->data = data;
    newNode->left = NULL;
```



```
newNode->right = NULL;    return
```

```
newNode;
```

```
}
```

```
// Function to insert a new node in BST
```

```
struct TreeNode* insertNode(struct TreeNode* root, int data) {
```

```
    // If the tree is empty, return a new node
```

```
    if (root == NULL) {          return
```

```
        createNode(data);
```

```
    }
```

```
    // Otherwise, recur down the tree
```

```
    if (data < root->data) {
```

```
        root->left = insertNode(root->left, data);
```

```
    } else if (data > root->data) {
```

```
        root->right = insertNode(root->right, data);
```

```
    }
```

```
    // return the (unchanged) node pointer
```

```
    return root;
```

```
}
```

```
// Function to find the inorder successor in BST struct
```

```
TreeNode* minValueNode(struct TreeNode* node) {
```

```
    struct TreeNode* current = node;    // Loop down to find
```

```
    the leftmost leaf    while (current && current->left !=
```

```
    NULL) {        current = current->left;
```

```
    }
```




```
    return current;
}

// Function to delete a node from BST
struct TreeNode* deleteNode(struct TreeNode* root, int data) {
    // Base case: If the tree is empty
    if (root == NULL) {    return root;
    }

    // Recur down the tree
    if (data < root->data) {
        root->left = deleteNode(root->left, data);
    } else if (data > root->data) {
        root->right = deleteNode(root->right, data);
    } else
    {
        // Node found with the data
        // Case 1: Node with only one child or no child
        if (root->left == NULL) {    struct TreeNode*
temp = root->right;    free(root);
return temp;

        } else if (root->right == NULL) {
struct TreeNode* temp = root->left;
free(root);    return temp;
        }

        // Case 2: Node with two children
```

```
    struct TreeNode* temp = minValueNode(root->right);

// Copy the inorder successor's content to this node
root->data = temp->data;    // Delete the inorder
successor

    root->right = deleteNode(root->right, temp->data);
}

return root;
}

// Function to perform inorder traversal of BST void
inorderTraversal(struct TreeNode* root) {
    if (root != NULL) {
        inorderTraversal(root->left);
        printf("%d ", root->data);
        inorderTraversal(root->right);
    }
}

// Function to perform preorder traversal of BST
void preorderTraversal(struct TreeNode* root) {
    if (root != NULL) {        printf("%d ", root->data);
        preorderTraversal(root->left);
        preorderTraversal(root->right);
    }
}
```



```
// Function to perform postorder traversal of BST
void postorderTraversal(struct TreeNode* root) {
    if (root != NULL) {        postorderTraversal(root-
>left);        postorderTraversal(root->right);
    printf("%d ", root->data);
    }
}

// Function to search for a node in BST
struct TreeNode* search(struct TreeNode* root, int data) {
    // Base Cases: root is null or data is present at root
    if (root == NULL || root->data == data) {
        return root;
    }
    // data is greater than root's data
    if (root->data < data) {        return
search(root->right, data);
    }
    // data is smaller than root's data
    return search(root->left, data);
}

// Example usage int main() {    struct
TreeNode* root = NULL;    root =
insertNode(root, 50);
insertNode(root, 30);
```



```
insertNode(root, 20);
insertNode(root, 40);
insertNode(root, 70);
insertNode(root, 60);
insertNode(root, 80);  printf("Inorder
traversal of the BST: ");
inorderTraversal(root);  printf("\n");
    printf("Preorder traversal of the BST: ");
    preorderTraversal(root);  printf("\n");
    printf("Postorder traversal of the BST: ");
postorderTraversal(root);  printf("\n");
// Search for a node    int key = 40;
    struct TreeNode* result = search(root, key);
    if (result) {
        printf("Node %d found in the BST.\n", key);
    } else {
        printf("Node %d not found in the BST.\n", key);
    }
// Delete a node    root = deleteNode(root,
20);  printf("Inorder traversal after deleting
20: ");  inorderTraversal(root);
printf("\n");  return 0;
}
```



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

OUTPUT :

Input for the program (Optional)

Output:

Inorder traversal of the BST: 20 30 40 50 60 70 80

Preorder traversal of the BST: 50 30 20 40 70 60 80

Postorder traversal of the BST: 20 40 30 60 80 70 50

Node 40 found in the BST.

Inorder traversal after deleting 20: 30 40 50 60 70 80



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

EXPERIMENT NO. 12

Objective(s): Implement Graph data structure using adjacency list and adjacency matrix representations, and perform basic operations including BFS and DFS traversals.

Outcome: Develop methods to efficiently store graph data, allowing traversal algorithms (BFS and DFS) to explore nodes and edges, facilitating various graph-related computations and analyses.

Problem Statement: Implement Graphs and represent using adjacency list and adjacency matrix and implement basic operations with traversals

Background Study:

Graphs

A graph is a non-linear data structure that consists of a set of vertices (also called nodes) and a set of edges that connect pairs of vertices. Graphs are widely used to represent networks of interconnected objects, such as social networks, road networks, computer networks, etc.

Types of Graphs

1. Undirected Graph:

- A graph where edges have no direction. If there is an edge between vertex A and vertex B, it can be traversed in both directions.

2. Directed Graph (Digraph):

- A graph where edges have a direction. If there is a directed edge from vertex A to vertex B, it can only be traversed from A to B, not from B to A unless there is a separate edge in the opposite direction.

3. Weighted Graph:

- A graph where edges are assigned weights or costs. These weights can represent distances, capacities, costs, etc.

4. Unweighted Graph:

- A graph where all edges have the same weight, often represented simply by their existence.

Graph Representation

Graphs can be represented in two primary ways:

1. Adjacency Matrix:

- A 2D array `graph[V][V]` where `V` is the number of vertices.
- `graph[i][j]` indicates the presence of an edge between vertex `i` and vertex `j`.
- For unweighted graphs, `graph[i][j]` is typically 1 if there is an edge, and 0 otherwise.
- For weighted graphs, `graph[i][j]` may contain the weight of the edge, or a special value (like `INT_MAX`) to denote absence of an edge.

2. Adjacency List:

- An array of lists, where each element of the array represents a vertex.
- Each vertex `v` has a list of vertices adjacent to it (neighbors).
- This representation is more space-efficient for sparse graphs (where there are few edges compared to the number of vertices).
- Allows efficient insertion and removal of edges.

Basic Operations on Graphs

1. Traversal:

○ Depth-First Search (DFS):

- Start from a vertex and explore as far as possible along each branch before backtracking.
- Uses a stack (or recursion) to maintain the sequence of visited vertices.
- Useful for finding connected components, topological sorting, and detecting cycles in graphs.

○ Breadth-First Search (BFS):

- Start from a vertex and explore all its neighbors at the present depth level before moving on to vertices at the next depth level.
- Uses a queue to maintain the sequence of visited vertices.
- Useful for finding shortest paths in unweighted graphs, level-order traversal, and network broadcasting.

2. Adding and Removing Vertices and Edges:

- Vertices are typically added or removed straightforwardly by adjusting the adjacency matrix or list.
- Adding or removing edges involves updating the appropriate entry in the matrix or adjusting the adjacency list.

3. Finding Shortest Paths:

- Algorithms like Dijkstra's algorithm (for non-negative weights) or Bellman-Ford algorithm (for negative weights) are used to find the shortest path between two vertices in a weighted graph.

4. Cycle Detection:

- Techniques such as DFS can be used to detect cycles in directed and undirected graphs by maintaining a list of visited vertices and checking for back edges.

Applications of Graphs

- Routing and Network Design: Used in computer networks and telecommunications for routing packets and designing optimal network topologies.
- Social Network Analysis: Analyzing relationships and connections between individuals or entities in social networks.
- Recommendation Systems: Understanding user preferences and relationships to recommend items or content.

- Transportation and Logistics: Optimizing routes for vehicles, scheduling deliveries, and managing traffic flow.

Algorithm :

1. Graph Representation ●

Adjacency Matrix:

- Create a 2D array $graph[V][V]$ where V is the number of vertices.
- Initialize all elements to 0 or INF (infinity) depending on whether the graph is weighted or unweighted.
- Set $graph[i][j]$ to 1 or to the weight of the edge (i, j) for an unweighted or weighted graph respectively.
- Adjust entries based on adding or removing edges.

● Adjacency List:

- Create an array of linked lists (or dynamic arrays) where each array index represents a vertex.
- For each vertex v , maintain a list of vertices adjacent to v .
- Insertion and deletion of edges involve appending or removing elements from these lists.

2. Basic Operations

● Traversal:

1. Depth-First Search (DFS):

- Start from a vertex v .
- Mark v as visited.
- Recursively visit all adjacent vertices not yet visited.
- Use a stack (or recursion) to maintain the traversal path.

2. Breadth-First Search (BFS):



- Start from a vertex v.
- Mark v as visited.
- Use a queue to store vertices at the current level.
- Dequeue a vertex, visit its neighbors, and enqueue them.
- Repeat until the queue is empty.
- Adding and Removing Vertices and Edges:
 1. Adding a Vertex:
 - Increase the size of the adjacency list or matrix and initialize its entries.
 2. Removing a Vertex:

Remove the vertex from the adjacency list or matrix,

 - along with its edges.
 3. Adding an Edge:
 - Update the corresponding entries in the adjacency list or matrix.
 4. Removing an Edge:
 - Set the corresponding entries in the adjacency list or matrix to 0 or INF.
- Finding Shortest Paths:
 1. Use algorithms like Dijkstra's (for non-negative weights) or Bellman-Ford (for negative weights) to find the shortest path between two vertices.
- Algorithm Steps:
 1. Initialization:
 - Create a distance array dist[] where dist[s] = 0 and dist[v] = INF for all vertices $v \neq s$, indicating the shortest known distance from the source vertex s to each vertex.
 - Maintain a set of vertices S whose shortest path from the source vertex s has been found.



2. Process:

- While S does not include all vertices:
- Select the vertex u not in S with the smallest distance $\text{dist}[u]$.
- Add u to S.
- Update the distances of all neighboring vertices v of u if $\text{dist}[u] + \text{weight}(u, v) < \text{dist}[v]$.
- Continue until all vertices have been added to S.

3. Output:

- The array $\text{dist}[]$ now contains the shortest path distances from the source vertex s to every other vertex in the graph.

Code:

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

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

```
// Structure to represent a node in adjacency list
```

```
struct AdjListNode {    int dest;
```

```
    struct AdjListNode* next;
```

```
};
```

```
// Structure to represent an adjacency list struct
```

```
AdjList {
```

```
    struct AdjListNode* head; // Pointer to head node of the list
```

```
};
```



```
// Structure to represent a graph struct
```

```
Graph {
```

```
    int V;          // Number of vertices
```

```
    struct AdjList* array; // Array of adjacency lists
```

```
};
```

```
// Function to create a new adjacency list node struct
```

```
AdjListNode* newAdjListNode(int dest) {
```

```
    struct AdjListNode* newNode = (struct AdjListNode*)malloc(sizeof(struct
```

```
AdjListNode));    if (newNode == NULL) {
```

```
        fprintf(stderr, "Memory allocation failed\n");
```

```
exit(EXIT_FAILURE);
```

```
    }
```

```
    newNode->dest = dest;
```

```
    newNode->next = NULL;    return
```

```
    newNode;
```

```
}
```

```
// Function to create a graph with V vertices struct
```

```
Graph* createGraph(int V) {
```

```
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
```

```
if (graph == NULL) {
```

```
    fprintf(stderr, "Memory allocation failed\n");
```

```
exit(EXIT_FAILURE);
```

```
}
```

```
graph->V = V;
```

```
// Create an array of adjacency lists. Size of array will be V graph-
>array = (struct AdjList*)malloc(V * sizeof(struct AdjList)); if (graph-
>array == NULL) {

    fprintf(stderr, "Memory allocation failed\n");
    exit(EXIT_FAILURE);
}

// Initialize each adjacency list as empty by making head as NULL
for (int i = 0; i < V; ++i) {    graph->array[i].head = NULL;
}

return graph;
}

// Function to add an edge to an undirected graph void
addEdge(struct Graph* graph, int src, int dest) {

    // Add an edge from src to dest
    struct AdjListNode* newNode = newAdjListNode(dest);
    newNode->next = graph->array[src].head;    graph-
>array[src].head = newNode;

    // Since graph is undirected, add an edge from dest to src also
    newNode = newAdjListNode(src);    newNode->next = graph-
>array[dest].head;    graph->array[dest].head = newNode;
}

// Function to print the adjacency list representation of the graph void
printGraph(struct Graph* graph) {
```



```
for (int v = 0; v < graph->V; ++v) {  
    struct AdjListNode* temp = graph->array[v].head;  
    printf("Adjacency list of vertex %d\n head", v);    while  
(temp) {  
        printf(" -> %d", temp->dest);  
        temp = temp->next;  
    }  
    printf("\n");  
}  
  
// Structure for stack  
struct Stack {    int  
    top;    unsigned  
    capacity;    int*  
    array;  
};  
  
// Function to create a stack of given capacity struct  
Stack* createStack(unsigned capacity) {  
    struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));  
    if (stack == NULL) {  
        fprintf(stderr, "Memory allocation failed\n");  
        exit(EXIT_FAILURE);  
    }  
    stack->capacity = capacity;
```



```
stack->top = -1;

stack->array = (int*)malloc(stack->capacity * sizeof(int));

if (stack->array == NULL) {
    fprintf(stderr, "Memory allocation failed\n");
    exit(EXIT_FAILURE);
}

return stack;
}

// Stack functions

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; // Stack is empty
}

// Function to perform Depth First Search (DFS) traversal void
DFS(struct Graph* graph, int start) {
    // Create a stack for DFS
```



```
struct Stack* stack = createStack(graph->V); //
Array to keep track of visited vertices  int* visited
= (int*)malloc(graph->V * sizeof(int));  if (visited
== NULL) {
    fprintf(stderr, "Memory allocation failed\n");
    exit(EXIT_FAILURE);
}
for (int i = 0; i < graph->V; ++i) {
    visited[i] = 0;
}
// Push the start vertex onto the stack and mark it as visited
push(stack, start);  visited[start] = 1;
// Traverse while stack is not empty
while (!isEmpty(stack)) {    // Pop a
vertex from stack and print it      int
current = pop(stack);      printf("%d ",
current);
    // Get all adjacent vertices of the popped vertex current
    // If an adjacent has not been visited, then mark it visited
    // and push it to the stack
    struct AdjListNode* temp = graph->array[current].head;
    while (temp) {        if (!visited[temp->dest]) {
        push(stack, temp->dest);
        visited[temp->dest] = 1;
```




Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

```
    }

    temp = temp->next;

}

}

// Free allocated memory
free(stack->array);
free(stack); free(visited);
}

// Structure for queue struct
Queue {
    int front, rear, size;
    unsigned capacity; int*
    array;
};

// Function to create a queue of given capacity struct
Queue* createQueue(unsigned capacity) {
    struct Queue* queue = (struct Queue*)malloc(sizeof(struct Queue));
    if (queue == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        exit(EXIT_FAILURE);
    }
    queue->capacity = capacity;
    queue->front = queue->size = 0;
```



```
queue->rear = capacity - 1; // This is important, see the enqueue
queue->array = (int*)malloc(queue->capacity * sizeof(int));  if
(queue->array == NULL) {
    fprintf(stderr, "Memory allocation failed\n");
    exit(EXIT_FAILURE);
}
return queue;
}

// Queue functions
int isEmpty(struct Queue* queue) {
    return (queue->size == 0);
}

int isQueueFull(struct Queue* queue) {
    return (queue->size == queue->capacity);
}

void enqueue(struct Queue* queue, int item) {
    if (isQueueFull(queue))    return;

    queue->rear = (queue->rear + 1) % queue->capacity;
    queue->array[queue->rear] = item;    queue->size =
queue->size + 1;
}

int dequeue(struct Queue* queue) {
    if (isEmpty(queue))

        return -1;
```



```
int item = queue->array[queue->front]; queue->front
= (queue->front + 1) % queue->capacity; queue->size =
queue->size - 1; return item;
}

// Function to perform Breadth First Search (BFS) traversal
void BFS(struct Graph* graph, int start) { // Create a
queue for BFS

    struct Queue* queue = createQueue(graph->V);
// Array to keep track of visited vertices int*
visited = (int*)malloc(graph->V * sizeof(int)); if
(visited == NULL) {
    fprintf(stderr, "Memory allocation failed\n");
    exit(EXIT_FAILURE);
}

    for (int i = 0; i < graph->V; ++i) {
visited[i] = 0;
    }

    // Enqueue the start vertex and mark it as visited
enqueue(queue, start);

    visited[start] = 1;

    // Traverse while queue is not empty
while (!isEmpty(queue)) {
```



```
// Dequeue a vertex from queue and print it
int current = dequeue(queue);    printf("%d ",
current);

// Get all adjacent vertices of the dequeued vertex current
// If an adjacent has not been visited, then mark it visited
// and enqueue it
struct AdjListNode* temp = graph->array[current].head;
while (temp) {    if (!visited[temp->dest]) {
enqueue(queue, temp->dest);    visited[temp->dest]
= 1;
    }
    temp = temp->next;
}
}

// Free allocated memory
free(queue->array);
free(queue);    free(visited);
}

// Driver program to test above functions int
main() {

// Create a graph with 5 vertices
int V = 5;

struct Graph* graph = createGraph(V);
```



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

```
// Add edges
addEdge(graph, 0, 1);
addEdge(graph, 0, 4);
addEdge(graph, 1, 2);
addEdge(graph, 1, 3);
addEdge(graph, 1, 4);
addEdge(graph, 2, 3);
addEdge(graph, 3, 4);

// Print the adjacency list representation of the graph
printf("Graph represented using adjacency list:\n");
printGraph(graph);  printf("\n");

// Perform BFS traversal starting from vertex 0
printf("BFS traversal starting from vertex 0:\n");
BFS(graph, 0);  printf("\n");
}
```

OUTPUT :



Faculty of Engineering & Technology

Subject Name: Object Oriented Programming with java

Subject Code: 303105206

B.Tech. IT Year: 2nd / Semester: 3rd

Output:

Graph represented using adjacency list:

Adjacency list of vertex 0

head -> 4 -> 1

Adjacency list of vertex 1

head -> 4 -> 3 -> 2 -> 0

Adjacency list of vertex 2

head -> 3 -> 1

Adjacency list of vertex 3

head -> 4 -> 2 -> 1

Adjacency list of vertex 4

head -> 3 -> 1 -> 0