FACULTY OF ENGINEERING AND TECHNOLOGY

Design of Data Structures

(303105202)

3rd SEMESTER

Computer Science & Engineering Department

Lab Mannual

<u>CERTIFICATE</u>

This is to certify that Mr./Ms. **Padarthi Raj Mahesh** with enrolment no.**2203031240959** has completed his/her laboratory experiments in the **DSA (303105201)** from the department of **COMPUTER SCIENCE & ENGINEERING** during the academic year **2023 – 2024.**

Date of Submission...................... Staff In charge...........................

Head Of Department..

TABLE OF CONTENTS

PRACTICAL-1

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

Experiment Objective:

To understand the concept of a stack and implement its operations using a linear data structure.

Experiment Apparatus:

C Programming

Experiment Procedure:

1. Stack Implementation:

- Define the Stack class.
- Initialize necessary variables and data structures.
- Implement the constructor method.
- Define methods for stack operations:
- push(element): Adds an element to the top of the stack.
- pop(): Removes and returns the element from the top of the stack.
- peek(): Returns the element at the top of the stack without removing it.
- search(element): Searches for the given element in the stack.
- traverse(): Displays all elements in the stack.

Experiment Code:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 10
int top = -1;
int stack[MAX];
void push(int value)
{
if (top >= MAX - 1)
{
printf("Stack Overflow\n");
return;
}
stack[++top] = value;
printf("%d pushed into stack\n", value);
}
int pop()
{
if (top < 0)
{
printf("Stack Underflow\n");
return 0;
}
return stack[top--];
}
int peek()
{
if (top < 0)
```

```
{
printf("Stack is Empty\n");
return 0;
}
return stack[top];
}
void traverse()
{
if (top < 0)
{
printf("Stack is Empty\n");
return;
}
printf("Stack elements are: ");
for (int i = top; i >= 0; i--)
{
printf("%d ", stack[i]);
}
printf("\n");
}
int main()
{
push(10);
push(20);
push(30);
push(40);
push(50);
printf("%d popped from stack\n", pop());
printf("Top element is :%d\n", peek());
```

```
traverse();
return 0;
}
```

OUTPUT:

```
10 pushed into stack
20 pushed into stack
30 pushed into stack
40 pushed into stack
50 pushed into stack
50 popped from stack
Top element is :40
Stack elements are: 40 30 20 10
```

PRACTICAL-2

AIM: Implement Infix to Postfix Expression Conversion using Stack

Experiment Objective -

Implement Infix to Postfix Expression Conversion using Stack

Experiment Apparatus –

C programming

Experiment Procedure -

1. **Implementation steps**

- Create an empty stack to store operators temporarily.
- Initialize an empty string to store the postfix expression.
- Scan the infix expression from left to right.
- If the scanned character is an operand, add it to the postfix string.If the scanned character is an operator: 5.1. While the stack is not empty and the precedence of the current operator is less than or equal to the precedence of the top of the stack, pop from the stack and add it to the postfix string. 5.2. Push the current operator onto the stack.
- If the scanned character is '(', push it onto the stack.
- If the scanned character is ')': 7.1. Pop and add operators to the postfix string until a '(' is encountered on the stack. 7.2. Discard the '(' from the stack without adding it to the postfix string.
- Continue scanning the infix expression until all characters are processed.
- Pop and add any remaining operators from the stack to the postfix string.
- The postfix string is the desired output.

Experiment Code

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
// Stack structure definition
typedef struct
{
int top;
unsigned capacity;
char* array;
} Stack;
// Function to create a stack of given capacity
Stack* createStack(unsigned capacity)
{
Stack* stack = (Stack*)malloc(sizeof(Stack));
stack->capacity = capacity;
stack->top = -1;
stack->array = (char*)malloc(stack->capacity * sizeof(char));
return stack;
}
// Stack is full when the top is equal to the last index
bool isFull(Stack* stack)
{
return stack->top == stack->capacity - 1;
}
// Stack is empty when the top is equal to -1
bool isEmpty(Stack* stack)
```

```
{
return stack->top == -1;
}
// Function to add an item to the stack. It increases the top by 1
void push(Stack* stack, char item)
{
if (isFull(stack))
return;
stack->array[++stack->top] = item;
}
// Function to remove an item from the stack. It decreases the top by 1
char pop(Stack* stack)
{
if (isEmpty(stack))
return '\0';
return stack->array[stack->top--];
}
// Function to return the top from stack without removing it
char peek(Stack* stack)
{
if (isEmpty(stack))
return '\0';
return stack->array[stack->top];
}
// Check if the precedence of operator is greater than or equal to the precedence
of the top element in stack
int precedence(char op1, char op2)
{
if (op1 == '(' || op2 == ')')
```

```
return 0;
if (op1 == '+' || op1 == '-')
return 1;
if (op1 == '*' || op1 == '/')
return 2;
return -1;
}
// Function to convert infix expression to postfix expression
void convertInfixToPostfix(char* exp)
{
// Create a stack of characters
Stack* stack = createStack(strlen(exp));
int i, j;
for (i = 0, j = 0; exp[i] != '\0'; i++)
{
// If the scanned character is an operand, add it to output string.
if (exp[i] >= 'a' && exp[i] <= 'z' || exp[i] >= 'A' && exp[i] <= 'Z')
exp[j++] = exp[i];
// If the scanned character is an '(', push it to the stack.
else if (exp[i] == '(')
push(stack, exp[i]);
// If the scanned character is an ')', pop and output from the stack until '(' is
encountered.
else if (exp[i] == ')')
{
while (!isEmpty(stack) && peek(stack) != '(')
exp[j++] = pop(stack);
if (!isEmpty(stack) && peek(stack) != '(')
return; // invalid expression
else
pop(stack);
}
// If an operator is scanned
else {
while (!isEmpty(stack) && precedence(exp[i], peek(stack)) <= 0)
exp[j++] = pop(stack);
push(stack, exp[i]);
}
}
// Once all inital expression characters are traversed,
// adding all left elements from stack to exp
while (!isEmpty(stack))
exp[j++] = pop(stack);
exp[j] = '\0';
printf("%s", exp);
}
// Driver program to test above functions
int main()
{
char expression[] = "((a+(b*c))-d)";
convertInfixToPostfix(expression);
return 0;
}
```

Output:

```
abc*+d-
```

PRACTICAL-3

AIM: Implement Postfix evaluation using Stack.

Experiment Procedure:

1. Stack Implementation:

 - Define the Stack class.
 - Initialize necessary variables and data structures.
 - Implement the constructor method.
 - Define methods for stack operations:
 - push(element): Adds an element to the top of the stack.
 - pop(): Removes and returns the element from the top of the stack.
 - peek(): Returns the element at the top of the stack without removing it.
 - isEmpty(): Checks if the stack is empty.

2. Postfix Evaluation Algorithm:

 - Describe the algorithm for evaluating postfix expressions using a stack.
 - Explain the steps involved in evaluating an expression.

Experiment Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX 100
typedef struct
{
int data[MAX];
int top;
} Stack;
void initStack(Stack* stack)
{
stack->top = -1;
}
int isEmpty(Stack* stack)
{
return stack->top == -1;
}
int isFull(Stack* stack)
{
return stack->top == MAX - 1;
}
void push(Stack* stack, int value)
{
if (isFull(stack))
{
printf("Stack Overflow\n");
return;
}
```

```
stack->data[++stack->top] = value;
}
int pop(Stack* stack)
{
if (isEmpty(stack))
{
printf("Stack Underflow\n");
return 0;
}
return stack->data[stack->top--];
}
int evaluatePostfix(char* postfix)
{
Stack stack;
initStack(&stack);
int i, operand1, operand2, result;
for (i = 0; i < strlen(postfix); i++)
{
if (postfix[i] == ' ') continue;
if (postfix[i] >= '0' && postfix[i] <= '9')
{
push(&stack, postfix[i] - '0');
}
else
{
operand2 = pop(&stack);
operand1 = pop(&stack);
switch (postfix[i])
{
```

```
case '+':
result = operand1 + operand2;
break;
case '-':
result = operand1 - operand2;
break;
case '*':
result = operand1 * operand2;
break;
case '/':
result = operand1 / operand2;
break;
}
push(&stack, result);
}
}
return pop(&stack);
}
int main()
{
char postfix[] = "23+45*";
int result = evaluatePostfix(postfix);
printf("Result: %d\n", result);
return 0;
}
```

OUTPUT:

```
Result: 20
```

PRACTICAL-4

AIM-Implement Towers of Hanoi using Stack.

Experiment Procedure -

1. **Implementation steps**
   - Create a stack data structure to represent each peg in the Tower of Hanoi problem.
   - Initialize three stacks to represent the three pegs.
   - Push the disks onto the first peg in the correct order (largest disk at the bottom and smallest disk at the top).
   - Define a function to move a disk from one peg to another peg using the stack data structure. This function should follow the rules of the Tower of Hanoi problem.
   - Define a recursive function to solve the Tower of Hanoi problem. This function should take the following arguments: the number of disks, the source peg, the destination peg, and the intermediate peg.
   - In the recursive function, first move n-1 disks from the source peg to the intermediate peg using the helper function.
   - Then, move the largest disk (the nth disk) from the source peg to the destination peg using the helper function.
   - Finally, move the n-1 disks from the intermediate peg to the destination peg using the helper function.
   - Call the recursive function with the initial values to solve the Tower of Hanoi problem.

Example:

```
#include <stdio.h>
#include <stdlib.h>
// Structure to represent a stack node
struct StackNode
{
int data;
struct StackNode* next;
};
// Function to create a new stack node
struct StackNode* createNode(int data)
{
struct StackNode* newNode = (struct StackNode*)malloc(sizeof(struct
StackNode));
newNode->data = data;
newNode->next = NULL;
return newNode;
}
// Function to push an element onto the stack
void push(struct StackNode** top, int data)
{
struct StackNode* newNode = createNode(data);
newNode->next = *top;
*top = newNode;
}
// Function to pop an element from the stack
int pop(struct StackNode** top)
{
if (*top == NULL)
```

```
{
printf("Stack underflow!\n");
return -1;
}
struct StackNode* temp = *top;
*top = (*top)->next;
int popped = temp->data;
free(temp);
return popped;
}
// Function to move a disk from source rod to destination rod
void moveDisk(char source, char destination, int disk)
{
printf("Move disk %d from rod %c to rod %c\n", disk, source, destination);
}
// Function to implement Tower of Hanoi
void towerOfHanoi(int numDisks, struct StackNode* source, struct StackNode*
aux, struct StackNode* dest)
{
if (numDisks == 1)
{
int disk = pop(&source);
push(&dest, disk);
moveDisk('A', 'C', disk);
return;
}
towerOfHanoi(numDisks - 1, source, dest, aux);
int disk = pop(&source);
push(&dest, disk);
```

```
moveDisk('A', 'C', disk);

towerOfHanoi(numDisks - 1, aux, source, dest);

}

int main() // Main function

{

int numDisks = 3; // Number of disks

struct StackNode* source = NULL;

struct StackNode* aux = NULL;

struct StackNode* dest = NULL;

for (int i = numDisks; i >= 1; i--) // Push disks onto source rod

{

push(&source, i);

}

printf("Steps to solve Tower of Hanoi with %d disks:\n", numDisks);

towerOfHanoi(numDisks, source, aux, dest);

return 0;

}
```

OUTPUT:

```
Steps to solve Tower of Hanoi with 3 disks:
Move disk 1 from rod A to rod C
Move disk 1 from rod A to rod C
Stack underflow!
Move disk -1 from rod A to rod C
Move disk 1 from rod A to rod C
Stack underflow!
Move disk -1 from rod A to rod C
Stack underflow!
Move disk -1 from rod A to rod C
Stack underflow!
Move disk -1 from rod A to rod C
```

PRACTICAL-5

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

Experiment Procedure:

1. Queue Implementation:
   - Define the Queue class.
   - Initialize necessary variables and data structures.
   - Implement the constructor method.
   - Define methods for queue operations:
     - enqueue(element): Adds an element to the rear of the queue.
     - dequeue(): Removes and returns the element from the front of the queue.
     - traverse(): Displays all elements in the queue.
     - search(element): Searches for the given element in the queue.

Experiment Code:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
typedef struct
{
int data[MAX];
int front;
int rear;
} Queue;
void initQueue(Queue* queue)
{
queue->front = 0;
queue->rear = -1;
}
int isEmpty(Queue* queue)
{
return queue->front > queue->rear;
}
int isFull(Queue* queue)
{
return (queue->rear + 1) % MAX == queue->front;
}
void enqueue(Queue* queue, int value)
{
if (isFull(queue))
{
printf("Queue Overflow\n");
return;
}
queue->rear = (queue->rear + 1) % MAX;
queue->data[queue->rear] = value;
}
int dequeue(Queue* queue)
{
if (isEmpty(queue))
{
printf("Queue Underflow\n");
return 0;
}
int value = queue->data[queue->front];
queue->front = (queue->front + 1) % MAX;
return value;
}
void traverse(Queue* queue)
{
if (isEmpty(queue))
{
printf("Queue is Empty\n");
return;
}
printf("Queue elements are: ");
for (int i = queue->front; i != queue->rear; i = (i + 1) % MAX)
{
printf("%d ", queue->data[i]);
}
printf("%d\n", queue->data[queue->rear]);
```

```
}
int search(Queue* queue, int value)
{
if (isEmpty(queue))
{
printf("Queue is Empty\n");
return -1;
}
for (int i = queue->front; i != queue->rear; i = (i + 1) % MAX)
{
if (queue->data[i] == value)
{
return i;
}
}
if (queue->data[queue->rear] == value)
{
return queue->rear;
}
return -1;
}
int main()
{
Queue queue;
initQueue(&queue);
enqueue(&queue, 10);
enqueue(&queue, 20);
enqueue(&queue, 30);
printf("%d dequeued from queue\n", dequeue(&queue));
```

```
traverse(&queue);
printf("Element found at index %d\n", search(&queue, 20));
return 0;
}
```

OUTPUT:

```
Queue Overflow
Queue Overflow
Queue Overflow
Queue Underflow
0 dequeued from queue
Queue is Empty
Queue is Empty
Element found at index -1
```

PRACTICAL-6

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

Experiment Procedure -

1. **Implementation steps**

- Create a Node struct with an integer data field and a next pointer to the next node in the list.
- Implement a function to create a new node with the given data.
- Implement a function to insert a new node at the beginning of the list.
- Implement a function to insert a new node after a given node.
- Implement a function to insert a new node at the end of the list.
- Implement a function to delete a node with the given key.
- Implement a function to search for a node with the given key.
- Implement a function to reverse the linked list.
- Implement a function to print the linked list.
- Finally, implement a main() function to demonstrate the usage of these functions.

Implementation Code

```
#include <stdio.h>
#include <stdlib.h>
// Node structure definition
typedef struct Node
{
int data;
struct Node* next;
} Node;

// Function to create a new node with given data
Node* newNode(int data)
{
Node* node = (Node*)malloc(sizeof(Node));
node->data = data;
node->next = NULL;
return node;
}
// Function to insert a new node at the beginning of the list
void push(Node** head, int data)
{
Node* node = newNode(data);
node->next = *head;
*head = node;
}
// Function to insert a new node at the end of the list
void append(Node** head, int data)
{
```

```
Node* node = newNode(data);
if (*head == NULL)
{
*head = node;
return;
}
Node* last = *head;
while (last->next != NULL)
last = last->next;
last->next = node;
}
// Function to delete the first occurrence of a key in the list
void deleteNode(Node** head, int key)
{
if (*head == NULL)
return;
Node* temp = *head;
if (temp->data == key)
{
*head = temp->next;
free(temp);
return;
}
while (temp->next != NULL && temp->next->data != key)
temp = temp->next;
if (temp->next == NULL)
return;
Node* toDelete = temp->next;
temp->next = temp->next->next;
```

```
free(toDelete);
}
// Function to print the list
void printList(Node* head)
{
while (head != NULL)
{
printf("%d -> ", head->data);
head = head->next;
}
printf("NULL\n");
}
// Function to search for a key in the list
Node* search(Node* head, int key)
{
while (head != NULL)
{
if (head->data == key)
return head;
head = head->next;
}
return NULL;
}
// Function to reverse the list
void reverse(Node** head)
{
Node* prev = NULL;
Node* current = *head;
Node* next;
```

```
while (current != NULL)
{
next = current->next;
current->next = prev;
prev = current;
current = next;
}
*head = prev;
}
// Driver program to test above functions
int main()
{
Node* head = NULL;
push(&head, 10);
append(&head, 20);
append(&head, 30);
push(&head, 40);
printList(head); // 40 -> 10 -> 20 -> 30 -> NULL
deleteNode(&head, 10);
printList(head); // 40 -> 20 -> 30 -> NULL
Node* found = search(head, 20);
if (found != NULL)
printf("Found %d\n", found->data);
reverse(&head);
printList(head); // 30 -> 20 -> 40 -> NULL
return 0;
}
```

OUTPUT:

```
40 -> 10 -> 20 -> 30 -> NULL

40 -> 20 -> 30 -> NULL

Found 20

30 -> 20 -> 40 -> NULL
```

PRACTICAL-7

AIM: Implement Double Linked lists and its operations(creation insertion deletion traversal search reverse)

Experiment Procedure:

1. Doubly Linked List Implementation:
   - Define the Node class to represent individual nodes of the doubly linked list.
   - Define the DoublyLinkedList class.
   - Initialize necessary variables and data structures.
   - Implement the constructor method to create an empty doubly linked list.

2. Operations:

   a. Creation:

   - Define methods to create a doubly linked list:
     - insert_at_beginning(data): Inserts a new node at the beginning of the list.
     - insert_at_end(data): Inserts a new node at the end of the list.
     - insert_after_node(data, node): Inserts a new node after a specified node.

   b. Insertion:

   - Define methods to insert nodes into the doubly linked list.

   c. Deletion:

   - Define methods to delete nodes from the doubly linked list:
     - delete_at_beginning(): Deletes the node at the beginning of the list.
     - delete_at_end(): Deletes the node at the end of the list.
     - delete_by_value(data): Deletes the node with the specified data.

d. Traversal:

- Define a method to traverse the doubly linked list and display its elements.

e. Search:

- Define a method to search for a specific element in the doubly linked list.

f. Reverse:

- Define a method to reverse the doubly linked list.

Experiment Code:

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

typedef struct Node
{
int data;
struct Node* next;
struct Node* prev;
} Node;
Node* createNode(int data)
{
Node* newNode = (Node*)malloc(sizeof(Node));
newNode->data = data;
newNode->next = NULL;
newNode->prev = NULL;
return newNode;
}
void insertAtBeginning(Node** head, int data)
{
Node* newNode = createNode(data);
if (*head != NULL)
{
(*head)->prev = newNode;
newNode->next = *head;
}
*head = newNode;
}
void insertAtEnd(Node** head, int data)
```

```
{
Node* newNode = createNode(data);
if (*head == NULL)
{
*head = newNode;
return;
}
Node* last = *head;
while (last->next != NULL)
{
last = last->next;
}
last->next = newNode;
newNode->prev = last;
}
void deleteNode(Node** head, Node* node)
{
if (node->prev != NULL)
{
node->prev->next = node->next;
}
else
{
*head = node->next;
}
if (node->next != NULL)
{
node->next->prev = node->prev;
}
```

```
free(node);
}
void traverse(Node* head)
{
Node* current = head;
while (current != NULL)
{
printf("%d -> ", current->data);
current = current->next;
}
printf("NULL\n");
}
int search(Node* head, int data)
{
Node* current = head;
int index = 0;
while (current != NULL)
{
if (current->data == data)
{
return index;
}
current = current->next;
index++;
}
return -1;
}
void reverse(Node** head)
{
```

```
Node* current = *head;
Node* prev = NULL;
Node* next = NULL;
while (current != NULL)
{
next = current->next;
current->next = prev;
current->prev = next;
prev = current;
current = next;
}
*head = prev;
}
int main()
{
Node* head = NULL;
insertAtBeginning(&head, 10);
insertAtBeginning(&head, 20);
insertAtEnd(&head, 30);
insertAtEnd(&head, 40);
printf("Original List: ");
traverse(head);
printf("Search for 30: %d\n", search(head, 30));
deleteNode(&head, head->next);
printf("After deleting first node: ");
traverse(head);
reverse(&head);
printf("Reversed List: ");
traverse(head);
```

```
return 0;
}
```

OUTPUT:

```
Original List: 20 -> 10 -> 30 -> 40 -> NULL

Search for 30: 2

After deleting first node:

 20 -> 30 -> 40 -> NULL

Reversed List:

 40 -> 30 -> 20 -> NULL
```

PRACTICAL-8

AIM: Implement binary search and interpolation search.

Experiment Procedure:

Algorithm

The rest of the Interpolation algorithm is the same except for the above partition logic.

- **Step1:** In a loop, calculate the value of "pos" using the probe position formula.
- **Step2:** If it is a match, return the index of the item, and exit.
- **Step3:** If the item is less than arr[pos], calculate the probe position of the left sub-array. Otherwise, calculate the same in the right sub-array.
- **Step4:** Repeat until a match is found or the sub-array reduces to zero.

Experiment Code

```
#include <stdio.h>
// Binary search function
int binarySearch(int arr[], int l, int r, int x)
{
if (r >= l)
{
int mid = l + (r - l) / 2;
if (arr[mid] == x)
return mid;
if (arr[mid] < x)
return binarySearch(arr, mid + 1, r, x);
return binarySearch(arr, l, mid - 1, x);
}
return -1;
}
// Interpolation search function
int interpolationSearch(int arr[], int l, int r, int x)
{
if (l <= r && x >= arr[l] && x <= arr[r])
{
int mid = l + ((double)(r - l) / (arr[r] - arr[l])) * (x - arr[l]);
if (arr[mid] == x)
return mid;
if (arr[mid] < x)
return interpolationSearch(arr, mid + 1, r, x);
return interpolationSearch(arr, l, mid - 1, x);
}
return -1;
}
// Driver program to test above functions
int main()
{
int arr[] = {2, 3, 4, 10, 40};
int n = sizeof(arr) / sizeof(arr[0]);
int x = 10;
printf("Found %d at index %d\n", x, binarySearch(arr, 0, n - 1, x));
printf("Found %d at index %d\n", x, interpolationSearch(arr, 0, n - 1, x));
x = 60;
printf("Found %d at index %d\n", x, binarySearch(arr, 0, n - 1, x)); // Not found
printf("Found %d at index %d\n", x, interpolationSearch(arr, 0, n - 1, x)); // Not
found
    return 0;
}
```

OUTPUT:

```
Found 10 at index 3

Found 10 at index 3

Found 60 at index -1

Found 60 at index -1
```

PACTICAL-9

AIM: Implement Bubble sort, selection sort, Insertion sort, quick sort , merge sort.

Experiment Procedure:

1. Bubble Sort:
   a. Algorithm:
   - Explain the bubble sort algorithm.
   b. Implementation:
   - Write code to implement the bubble sort algorithm.

2. Selection Sort:
   a. Algorithm:
   - Explain the selection sort algorithm.
   b. Implementation:
   - Write code to implement the selection sort algorithm.

3. Insertion Sort:
   a. Algorithm:
   - Explain the insertion sort algorithm.
   b. Implementation:
   - Write code to implement the insertion sort algorithm.

4. Quick Sort:
   a. Algorithm:
   - Explain the quick sort algorithm.
   b. Implementation:
   - Write code to implement the quick sort algorithm.

5. Merge Sort:

a. Algorithm:

- Explain the merge sort algorithm.

b. Implementation:

- Write code to implement the merge sort algorithm.

{

Experiment Code:

Bubble Sort

```
#include <stdio.h>
void bubbleSort(int arr[], int n)
{
for (int i = 0; i < n - 1; i++)
{
for (int j = 0; j < n - i - 1; j++)
{
if (arr[j] > arr[j + 1])
{
int temp = arr[j];
arr[j] = arr[j + 1];
arr[j + 1] = temp;
}
}
}
}
int main()
{
int arr[] = {64, 34, 25, 12, 22, 11, 90};
int n = sizeof(arr) / sizeof(arr[0]);
bubbleSort(arr, n);
printf("Sorted array: \n");
for (int i = 0; i < n; i++)
{
printf("%d ", arr[i]);
}
printf("\n");
return 0;
}
```

OUTPUT:

```
Sorted array:

11 12 22 25 34 64 90
```

Selection Sort

```
#include <stdio.h>
void selectionSort(int arr[], int n)
{
for (int i = 0; i < n - 1; i++)
{
int minIndex = i;
for (int j = i + 1; j < n; j++)
{
if (arr[j] < arr[minIndex])
{
minIndex = j;
}
}
int temp = arr[minIndex];
arr[minIndex] = arr[i];
arr[i] = temp;
}
}

int main()
{
int arr[] = {64, 34, 25, 12, 22, 11, 90};
int n = sizeof(arr) / sizeof(arr[0]);
selectionSort(arr, n);
printf("Sorted array: \n");
for (int i = 0; i < n; i++)
{
```

```
printf("%d ", arr[i]);
}
printf("\n");
return 0;
}
```

OUTPUT:

```
Sorted array:
11 12 22 25 34 64 90
```

Insertion Sort

```
#include <stdio.h>
void insertionSort(int arr[], int n)
{
for (int i = 1; i < n; i++)
{
int key = arr[i];
int j = i - 1;
while (j >= 0 && arr[j] > key)
{
arr[j + 1] = arr[j];
j--;
}
arr[j + 1] = key;
}
}
int main()
{
int arr[] = {65, 35, 25, 11, 21, 12, 90};
int n = sizeof(arr) / sizeof(arr[0]);
insertionSort(arr, n);
printf("Sorted array: \n");
for (int i = 0; i < n; i++)
{
printf("%d ", arr[i]);
}
printf("\n");
return 0;
}
```

OUTPUT:

```
Sorted array:

11 12 21 25 35 65 90
```

Quick Sort

```
#include <stdio.h>
// Function to swap two elements
void swap(int* a, int* b)
{
int temp = *a;
*a = *b;
*b = temp;
}
// Function to partition the array and return the pivot index
int partition(int arr[], int low, int high)
{
int pivot = arr[high]; // Choose the pivot element (last element)
int i = (low - 1); // Index of smaller element
for (int j = low; j < high; j++)
{
// If current element is smaller than or equal to pivot
if (arr[j] <= pivot)
{
i++; // Increment index of smaller element
swap(&arr[i], &arr[j]);
}
}
swap(&arr[i + 1], &arr[high]);
return (i + 1); // Return the partitioning index
}
// Function to implement Quick Sort algorithm
void quickSort(int arr[], int low, int high)
```

```
{
if (low < high)
{
// Partition the array
int pi = partition(arr, low, high);
// Recursive call on the left and right sub-arrays
quickSort(arr, low, pi - 1);
quickSort(arr, pi + 1, high);
}
}
// Function to print an array
void printArray(int arr[], int size)
{
for (int i = 0; i < size; i++)
{
printf("%d ", arr[i]);
}
printf("\n");
}
// Main function
int main()
{
int arr[] = {10, 7, 8, 9, 1, 5};
int n = sizeof(arr) / sizeof(arr[0]);
printf("Original array: \n");
printArray(arr, n);
// Perform Quick Sort
quickSort(arr, 0, n - 1);
printf("Sorted array: \n");
printArray(arr, n);
return 0;
}
```

OUTPUT:

```
Original array:

10 7 8 9 1 5

Sorted array:

1 5 7 8 9 10
```

Merge Sort:

```
#include <stdio.h>
// Function to merge two halves arr[l..m] and arr[m+1..r] of array arr[]
void merge(int arr[], int l, int m, int r)
{
int i, j, k;
int n1 = m - l + 1;
int n2 = r - m;
// Create temporary arrays
int L[n1], R[n2];
// Copy data to temporary arrays L[] and R[]
for (i = 0; i < n1; i++)
L[i] = arr[l + i];
for (j = 0; j < n2; j++)
R[j] = arr[m + 1 + j];
// Merge the temporary arrays back into arr[l..r]
i = 0; // Initial index of first subarray
j = 0; // Initial index of second subarray
k = l; // Initial index of merged subarray
while (i < n1 && j < n2)
{
if (L[i] <= R[j])
{
arr[k] = L[i];
i++;
}
else
{
```

```
arr[k] = R[j];
j++;
}
k++;
}
// Copy the remaining elements of L[], if any
while (i < n1)
{
arr[k] = L[i];
i++;
k++;
}
// Copy the remaining elements of R[], if any
while (j < n2)
{
arr[k] = R[j];
j++;
k++;
}
}
// Function to implement Merge Sort recursively
void mergeSort(int arr[], int l, int r)
{
if (l < r)
{
// Find the middle point
int m = l + (r - l) / 2;
// Sort first and second halves
mergeSort(arr, l, m);
mergeSort(arr, m + 1, r);
// Merge the sorted halves
merge(arr, l, m, r);
}
}
// Utility function to print an array
void printArray(int arr[], int size)
{
for (int i = 0; i < size; i++)
printf("%d ", arr[i]);
printf("\n");
}
// Driver program to test above functions
int main()
{
int arr[] = {12, 11, 13, 5, 6, 7};
int arr_size = sizeof(arr) / sizeof(arr[0]);
printf("Given array is \n");
printArray(arr, arr_size);
mergeSort(arr, 0, arr_size - 1);
printf("\nSorted array is \n");
printArray(arr, arr_size);
return 0;
}
```

Output:

```
Given array is
12 11 13 5 6 7
Sorted array is
5 6 7 11 12 13
```

PRACTICAL-10

AIM Implement Binary search Tree and its operations (creation, insertion, deletion)

Properties of Binary Search Tree

Following are some main properties of the binary search tree in C:

- All nodes of the left subtree are less than the root node and nodes of the right subtree are greater than the root node.
- The In-order traversal of binary search trees gives the values in ascending order.
- All the subtrees of BST hold the same properties

- **Binary**
- **key**: It will be the data stored.
- **left**: Pointer to the left child.
- **right:** Pointer to the right child

Operations on BST C

- Search in BST
- Insertion in BST
- Deletion in BST

1. Search Operation on BST in C

The algorithm for the search operation is:

1. *If the root is NULL or the key of the root matches the target:*
    1. *Return the current root node.*
2. *If the target is greater than the key of the current root:*

    1. *Recursively call searchNode with the right subtree of the current root and the target.*
    2. *Return the result of the recursive call.*
3. *If the target is smaller than the key of the current root:*
    1. *Recursively call searchNode with the left subtree of the current root and the target.*
    2. *Return the result of the recursive call.*
4. *If the target is not found in the current subtree, return NULL.*

where the target is the key to search for in the tree.

Time Complexity: O(log n)

2. Insertion in BST

The insertNode function inserts a new node with a specific value into a Binary Search Tree while maintaining the binary search tree property.

Algorithm

1. *If the current node is NULL*
    1. *Return a new node created with the specified value.*
2. *If the value is less than the key of the current node:*
    1. *Recursively call insertNode with the left subtree of the current node and the value.*
    2. *Update the left child of the current node with the result of the recursive call.*
3. *If the value is greater than the key of the current node:*
    1. *Recursively call insertNode with the right subtree of the current node and the value.*
    2. *Update the right child of the current node with the result of the recursive call.*

4. *Return the current node (the root of the modified tree).*

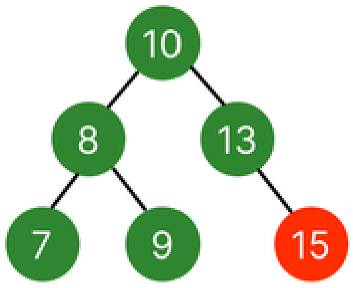
Time Complexity: O(log n)

4. Deletion in BST

There are few cases at the time of deleting a node in BST

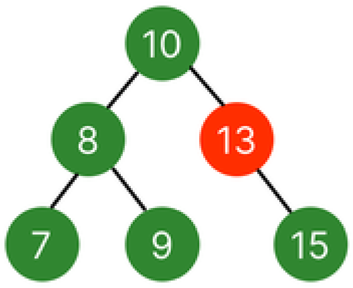
a. Deleted node is the leaf node

In this case, simply delete the node from the tree.



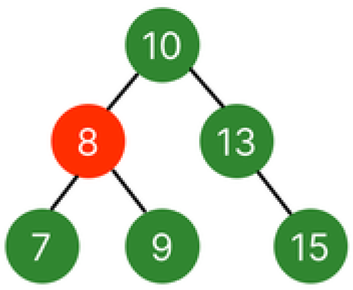
b. Deleted node has a single child node

In this case, replace the target node with its child (single child) and then delete that child node.



c. Deleted node has two children

In this case, at first, we will find the inorder successor of that node. And then replace the node with the inorder successor. Then remove the inorder successor from its original position.



Algorithm

1. *If the root is NULL:*
    1. *Return NULL (base case for recursion).*
2. *If the value to be deleted (x) is greater than the key of the current root:*
    1. *Recursively call delete with the right subtree of the current root and the value x.*
    2. *Update the right child of the current root with the result of the recursive call.*
3. *If the value to be deleted (x) is smaller than the key of the current root:*
    1. *Recursively call delete with the left subtree of the current root and the value x.*
    2. *Update the left child of the current root with the result of the recursive call.*
4. *If the value to be deleted (x) is equal to the key of the current root:*
    1. *If the current node has no child:*
        1. *Free the current node.*
        2. *Return NULL.*
    2. *If the current node has one child:*

Experiment code:

```
#include <stdio.h>
#include <stdlib.h>
// Define the structure of a BST node
struct Node
{
int data;
struct Node* left;
struct Node* right;
};
// Function to create a new BST node
struct Node* createNode(int data)
{
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = data;
newNode->left = newNode->right = NULL;
return newNode;
}
// Function to insert a new node into BST
struct Node* insert(struct Node* root, int data)
{
if (root == NULL)
{
return createNode(data);
}
if (data < root->data)
{
root->left = insert(root->left, data);
}
```

```
}
else if (data > root->data)
{
root->right = insert(root->right, data);
}
return root;
}
// Function to find the minimum value node in BST
struct Node* minValueNode(struct Node* node)
{
struct Node* current = node;
while (current && current->left != NULL)
{
current = current->left;
}
return current;
}
// Function to delete a node from BST
struct Node* deleteNode(struct Node* root, int key)
{
if (root == NULL)
{
return root;
}
if (key < root->data)
{
root->left = deleteNode(root->left, key);
}
else if (key > root->data)
{
```

```
root->right = deleteNode(root->right, key);
}
else
{
if (root->left == NULL)
{
struct Node* temp = root->right;
free(root);
return temp;
}
else if (root->right == NULL)
{
struct Node* temp = root->left;
free(root);
return temp;
}
struct Node* temp = minValueNode(root->right);
root->data = temp->data;
root->right = deleteNode(root->right, temp->data);
}
return root;
}
// Function to perform Inorder traversal of BST
void inorderTraversal(struct Node* root)
{
if (root != NULL)
{
inorderTraversal(root->left);
printf("%d ", root->data);
```

```
inorderTraversal(root->right);
}
}
// Main function
int main()
{
struct Node* root = NULL;
// Insert elements into BST
root = insert(root, 50);
root = insert(root, 30);
root = insert(root, 20);
root = insert(root, 40);
root = insert(root, 70);
root = insert(root, 60);
root = insert(root, 80);
printf("Inorder Traversal before deletion: ");
inorderTraversal(root);
printf("\n");
// Delete node with key 20
root = deleteNode(root, 20);
printf("Inorder Traversal after deletion of 20: ");
inorderTraversal(root);
printf("\n");
return 0;
}
```

OUTPUT:

```
Inorder Traversal before deletion:
 20 30 40 50 60 70 80
Inorder Traversal after deletion of 20:
 30 40 50 60 70 80
```

PRACTICAL-11

AIM: Implement Traversals Preorder Inorder Postorder on BST.

Experiment Procedure:

1. Binary Search Tree (BST) Implementation:
   - Define the Node structure to represent nodes in the BST.
   - Define the BinarySearchTree class.
   - Implement methods for BST operations (insertion, deletion, searching) if required.

2. Traversal Implementations:

   a. Preorder Traversal:
   - Explain the Preorder traversal algorithm.
   - Implement the Preorder traversal method.

   b. Inorder Traversal:
   - Explain the Inorder traversal algorithm.
   - Implement the Inorder traversal method.

   c. Postorder Traversal:
   - Explain the Postorder traversal algorithm.
   - Implement the Postorder traversal method.

Experiment Code:

```
#include <stdio.h>
#include <stdlib.h>
// Define the structure of a BST node
struct Node
{
int data;
struct Node* left;
struct Node* right;
};
// Function to create a new BST node
struct Node* createNode(int data)
{
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = data;
newNode->left = newNode->right = NULL;
return newNode;
}
// Function to insert a new node into BST
struct Node* insert(struct Node* root, int data)
{
if (root == NULL)
{
return createNode(data);
}
if (data < root->data)
{
root->left = insert(root->left, data);
}
```

```
else if (data > root->data)
{
root->right = insert(root->right, data);
}
return root;
}
// Function to perform Preorder traversal of BST
void preorderTraversal(struct Node* root)
{
if (root != NULL)
{
printf("%d ", root->data);
preorderTraversal(root->left);
preorderTraversal(root->right);
}
}
// Function to perform Inorder traversal of BST
void inorderTraversal(struct Node* root)
{
if (root != NULL)
{
inorderTraversal(root->left);
printf("%d ", root->data);
inorderTraversal(root->right);
}
}
// Function to perform Postorder traversal of BST
void postorderTraversal(struct Node* root)
{
```

```
if (root != NULL)
{
postorderTraversal(root->left);
postorderTraversal(root->right);
printf("%d ", root->data);
}
}
// Main function
int main()
{
struct Node* root = NULL;
// Insert elements into BST
root = insert(root, 50);
root = insert(root, 30);
root = insert(root, 20);
root = insert(root, 40);
root = insert(root, 70);
root = insert(root, 60);
root = insert(root, 80);
// Perform traversals
printf("Preorder Traversal: ");
preorderTraversal(root);
printf("\n");
printf("Inorder Traversal: ");
inorderTraversal(root);
printf("\n");
printf("Postorder Traversal: ");
postorderTraversal(root);
printf("\n");
```

```
return 0;
}
```

OUTPUT:

```
Preorder Traversal:
50 30 20 40 70 60 80
In order Traversal:
20 30 40 50 60 70 80
Post order Traversal:
20 40 30 60 80 70 50
```

PRACTICAL-12

AIM Implement Graphs and represent using adjaceny list and adjacency matrix and implement basic operations with traversals (BFS and DFS)

Experiment Procedure:

Sure, here are the definitions for Breadth First Search (BFS) and Depth First Search (DFS) algorithms:

1. **Breadth First Search (BFS)**:
   - BFS is a graph traversal algorithm that explores a graph level by level.
   - Starting from a given source vertex, BFS explores all the vertices at the current level before moving to the vertices at the next level.
   - It uses a queue data structure to keep track of the vertices to be visited next.
   - BFS is typically used to find the shortest path between two vertices in an unweighted graph, to check if a graph is bipartite, or to visit all the vertices of a graph in a systematic way.

2. **Depth First Search (DFS)**:
   - DFS is a graph traversal algorithm that explores a graph by going as deep as possible along each branch before backtracking.
   - Starting from a given source vertex, DFS explores as far as possible along each branch before backtracking to explore other branches.
   - It uses a stack data structure (or recursion) to keep track of the vertices to be visited next.
   - DFS is typically used to find connected components in a graph, to detect

cycles in a graph, or to perform topological sorting.

In summary, BFS explores the graph level by level, while DFS explores the graph by going as deep as possible along each branch. Each algorithm has its own use cases and advantages depending on the problem being solved.

Experiment 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;
};
// Structure to represent a graph
struct Graph
{
int V; // Number of vertices
struct AdjList* array;
};
// Function to create a new adjacency list node
struct AdjListNode* newAdjListNode(int dest)
{
struct AdjListNode* newNode = (struct AdjListNode*)malloc(sizeof(struct
AdjListNode));
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));
graph->V = V;
// Create an array of adjacency lists. Size of the array is V.
graph->array = (struct AdjList*)malloc(V * sizeof(struct AdjList));
// 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* pCrawl = graph->array[v].head;
printf("\n Adjacency list of vertex %d\n head ", v);
while (pCrawl)
{
printf("-> %d", pCrawl->dest);
pCrawl = pCrawl->next;
}
printf("\n");
}
}
// Function to perform Breadth First Traversal (BFS) of the graph
void BFS(struct Graph* graph, int startVertex)
{
// Create a boolean array to mark visited vertices
int* visited = (int*)malloc(graph->V * sizeof(int));
for (int i = 0; i < graph->V; i++)
{
visited[i] = 0;
}
// Create a queue for BFS
int* queue = (int*)malloc(graph->V * sizeof(int));
int front = 0, rear = 0;
// Mark the current node as visited and enqueue it
visited[startVertex] = 1;
queue[rear++] = startVertex;
printf("Breadth First Traversal starting from vertex %d: ", startVertex);
```

```
while (front != rear)
{
// Dequeue a vertex from queue and print it
int vertex = queue[front++];
printf("%d ", vertex);
// Get all adjacent vertices of the dequeued vertex
struct AdjListNode* pCrawl = graph->array[vertex].head;
while (pCrawl)
{
int adjVertex = pCrawl->dest;
// If an adjacent vertex has not been visited, mark it visited and enqueue it
if (!visited[adjVertex])
{
visited[adjVertex] = 1;
queue[rear++] = adjVertex;
}
pCrawl = pCrawl->next;
}
}
printf("\n");
free(visited);
free(queue);
}
// Function to perform Depth First Traversal (DFS) of the graph
void DFSUtil(struct Graph* graph, int vertex, int* visited)
{
// Mark the current node as visited and print it
visited[vertex] = 1;
printf("%d ", vertex);
```

```
// Recur for all the vertices adjacent to this vertex
struct AdjListNode* pCrawl = graph->array[vertex].head;
while (pCrawl)
{
int adjVertex = pCrawl->dest;
if (!visited[adjVertex])
{
DFSUtil(graph, adjVertex, visited);
}
pCrawl = pCrawl->next;
}
}
void DFS(struct Graph* graph, int startVertex)
{
// Create a boolean array to mark visited vertices
int* visited = (int*)malloc(graph->V * sizeof(int));
for (int i = 0; i < graph->V; i++)
{
visited[i] = 0;
}
printf("Depth First Traversal starting from vertex %d: ", startVertex);
DFSUtil(graph, startVertex, visited);
printf("\n");
free(visited);
}
```

```
int main()
{
int V = 5;
struct Graph* graph = createGraph(V);
addEdge(graph, 0, 1);
addEdge(graph, 0, 2);
addEdge(graph, 1, 2);
addEdge(graph, 1, 3);
addEdge(graph, 2, 3);
addEdge(graph, 3, 4);
printf("Adjacency list representation of the graph:\n");
printGraph(graph);
BFS(graph, 0);
DFS(graph, 0);
return 0;
}
```

OUTPUT:

```
Adjacency list representation of the graph:
Adjacency list of vertex 0
head -> 2-> 1
Adjacency list of vertex 1
head -> 3-> 2-> 0

Adjacency list of vertex 2
head -> 3-> 1-> 0

Adjacency list of vertex 3
head -> 4-> 2-> 1

Adjacency list of vertex 4
head -> 3
Breadth First Traversal starting from vertex 0: 0 2 1 3 4
Depth First Traversal starting from vertex 0: 0 2 3 4 1
```