**Experiment 10** **Date:12/10/2023**

# Queue 0perations

## Aim:

Porogram to implement queue operations using arrays

## Algorithm:

```
Algorithm Queue Implementation using Array

Data:
  MAX_SIZE: constant integer
  queue: array of integers
  front: integer
  rear: integer

Functions:
  isFull() -> boolean
  isEmpty() -> boolean
  enqueue(value: integer) -> void
  dequeue() -> integer
  display() -> void

Begin

  Function isFull():
    Return rear == MAX_SIZE - 1

  Function isEmpty():
    Return front == -1

  Function enqueue(value):
    If isFull() Then
      Print "Queue is full. Cannot enqueue value."
    Else
      If front == -1 Then
        Set front = 0
      End If
      Increment rear
      queue[rear] = value
      Print value, "enqueued into the queue."

  Function dequeue():
    If isEmpty() Then
      Print "Queue is empty. Cannot dequeue."
      Return -1
    Else
      Set dequeuedValue = queue[front]
      If front == rear Then
        Set front = rear = -1
      Else
        Increment front
```

```
        End If
        Return dequeuedValue

    Function display():
      If isEmpty() Then
        Print "Queue is empty."
      Else
        Print "Queue elements: "
        For i = front To rear
          Print queue[i], " "
        End For
        Print newline

    Main():
      Initialize front = -1, rear = -1
      Loop
        Print "Queue Operations:"
        Print "1. Enqueue"
        Print "2. Dequeue"
        Print "3. Display"
        Print "4. Exit"
        Print "Enter your choice:"
        Input choice
        Switch choice
          Case 1:
            Print "Enter value to enqueue:"
            Input value
            Call enqueue(value)
          Case 2:
            Set value = dequeue()
            If value != -1 Then
              Print value, "dequeued from the queue."
            End If
          Case 3:
            Call display()
          Case 4:
            Print "Exiting the program."
            Exit
          Default:
            Print "Invalid choice. Please try again."
        End Switch
      End Loop

End
```

## Program

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

#define MAX_SIZE 5

int queue[MAX_SIZE];
int front = -1;
int rear = -1;
```

```
bool isFull() {
    return rear == MAX_SIZE - 1;
}

bool isEmpty() {
    return front == -1;
}

void enqueue(int value) {
    if (isFull()) {
        printf("Queue is full. Cannot enqueue %d.\n", value);
    } else {
        if (front == -1) {
            front = 0;
        }
        queue[++rear] = value;
        printf("%d enqueued into the queue.\n", value);
    }
}

int dequeue() {
    if (isEmpty()) {
        printf("Queue is empty. Cannot dequeue.\n");
        return -1;
    } else {
        int dequeuedValue = queue[front];
        if (front == rear) {
            front = rear = -1;
        } else {
            front++;
        }
        return dequeuedValue;
    }
}

void display() {
    if (isEmpty()) {
        printf("Queue is empty.\n");
    } else {
        printf("Queue elements: ");
        for (int i = front; i <= rear; i++) {
            printf("%d ", queue[i]);
        }
        printf("\n");
    }
}

int main() {
    int choice, value;

    while (1) {
        printf("\nQueue Operations:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
```

```
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to enqueue: ");
                scanf("%d", &value);
                enqueue(value);
                break;

            case 2:
                value = dequeue();
                if (value != -1) {
                    printf("%d dequeued from the queue.\n", value);
                }
                break;

            case 3:
                display();
                break;

            case 4:
                printf("Exiting the program.\n");
                return 0;

            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

## **Output**

Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 1

Enter value to enqueue: 3

3 enqueued into the queue.

Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 1

Enter value to enqueue: 4

4 enqueued into the queue.

Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 1

Enter value to enqueue: 6

6 enqueued into the queue.

Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 2

3 dequeued from the queue.

Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 3

Queue elements: 4 6

Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 4

Exiting the program.

**Experiment 11** **Date:12/10/2023**

# Circular Queue

**Aim**

10.Program to implement circular queue using array.

**Algorithm:**

Algorithm: Circular Queue Operations using Arrays

Data:

- MAX_SIZE: Maximum size of the circular queue.
- queue[MAX_SIZE]: Array to store the elements of the queue.
- front: Pointer to the front element of the queue.
- rear: Pointer to the rear element of the queue.

Functions:

1. isFull(): Checks if the queue is full.
2. isEmpty(): Checks if the queue is empty.
3. enqueue(value): Inserts an element into the queue.
4. dequeue(): Removes an element from the queue.
5. display(): Displays the elements of the queue.

Algorithm Steps:

1. Start:
2. Initialize front and rear pointers to -1.
3. Display the menu with options:
    a. Enqueue
    b. Dequeue
    c. Display
    d. Exit
4. Read user choice.
5. Perform the selected operation based on the user choice:

a. If choice is 'Enqueue':

i. Check if the queue is full using isFull().

ii. If not full, read the value from the user and enqueue it using enqueue(value).

b. If choice is 'Dequeue':

i. Check if the queue is empty using isEmpty().

ii. If not empty, dequeue the element and display it.

c. If choice is 'Display':

i. Display the elements of the queue using display().

d. If choice is 'Exit':

i. Exit the program.

e. If choice is invalid:

i. Display an error message.

6. Repeat steps 3-5 until the user chooses to exit.

7. End.

## Program

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

#define MAX_SIZE 5

int queue[MAX_SIZE];
int front = -1;
int rear = -1;

bool isFull() {
    return (front == 0 && rear == MAX_SIZE - 1) || (rear == (front - 1) % (MAX_SIZE - 1));
}

bool isEmpty() {
    return front == -1;
}
```

```
void enqueue(int value) {
    if (isFull()) {
        printf("Queue is full. Cannot enqueue %d.\n", value);
    } else {
        if (front == -1) {
            front = rear = 0;
        } else {
            rear = (rear + 1) % MAX_SIZE;
        }
        queue[rear] = value;
        printf("%d enqueued into the queue.\n", value);
    }
}

int dequeue() {
    if (isEmpty()) {
        printf("Queue is empty. Cannot dequeue.\n");
        return -1;
    } else {
        int dequeuedValue = queue[front];
        if (front == rear) {
            front = rear = -1;
        } else {
            front = (front + 1) % MAX_SIZE;
        }
        return dequeuedValue;
    }
}

void display() {
    if (isEmpty()) {
        printf("Queue is empty.\n");
```

```
    } else {
        printf("Queue elements: ");
        int i = front;
        do {
            printf("%d ", queue[i]);
            i = (i + 1) % MAX_SIZE;
        } while (i != (rear + 1) % MAX_SIZE);
        printf("\n");
    }
}

int main() {
    int choice, value;

    while (1) {
        printf("\nCircular Queue Operations:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Display\n");
        printf("4. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to enqueue: ");
                scanf("%d", &value);
                enqueue(value);
                break;

            case 2:
                value = dequeue();
                if (value != -1) {
```

```
                printf("%d dequeued from the queue.\n", value);
            }
            break;

        case 3:
            display();
            break;

        case 4:
            printf("Exiting the program.\n");
            return 0;

        default:
            printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

## Output

Circular Queue Operations:

1. Enqueue
2. Dequeue
3. Display
4. Exit

Enter your choice: 1

Enter value to enqueue: 2

2 enqueued into the queue.

Circular Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 1

Enter value to enqueue: 2

2 enqueued into the queue.

Circular Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 1

Enter value to enqueue: 3

3 enqueued into the queue.

Circular Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 2

2 dequeued from the queue.

Circular Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice:

3

Queue elements: 2 3

Circular Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 4

Exiting the program.

**Experiment 12** **Date:19/10/2023**

# Singly Linkedlist

## Aim

To implement the following operations on a singly

linked list

i. Creation,

ii. Insert a new node at front

iii. Insert an element after a particular

iv. Deletion from beginning

v. Deletion from the end

vi. Searching

vii. Traversal.

## Algorithm

Algorithm: Singly Linked List Operations

Data:

- struct Node: Defines the structure of a node in the linked list.
  - int data: Data value stored in the node.
  - struct Node* next: Pointer to the next node.
- head: Pointer to the first node of the linked list.

Functions:

1. createNode(data): Creates a new node with the given data and returns its pointer.
2. insertAtFront(data): Inserts a node with the given data at the beginning of the linked list.
3. insertAfterNode(data, key): Inserts a node with the given data after the node with the given key.
4. deleteFromBeginning(): Deletes the first node from the linked list.
5. deleteFromEnd(): Deletes the last node from the linked list.
6. search(key): Searches for a node with the given key in the linked list.
7. traverse(): Displays all the nodes in the linked list.

Algorithm Steps:

1. Start:
2. Initialize head pointer to NULL.
3. Display the menu with options:
   a. Insert at the front
   b. Insert after a node
   c. Delete from the beginning
   d. Delete from the end
   e. Search
   f. Traverse
   g. Exit
4. Read user choice.
5. Perform the selected operation based on the user choice:
   a. If choice is 'Insert at the front':
      i. Read data from the user.
      ii. Call insertAtFront(data).
   b. If choice is 'Insert after a node':
      i. Read data and key from the user.
      ii. Call insertAfterNode(data, key).
   c. If choice is 'Delete from the beginning':
      i. Call deleteFromBeginning().
   d. If choice is 'Delete from the end':
      i. Call deleteFromEnd().
   e. If choice is 'Search':
      i. Read key from the user.
      ii. Call search(key).
   f. If choice is 'Traverse':
      i. Call traverse().
   g. If choice is 'Exit':
      i. Exit the program.

h. If choice is invalid:

i. Display an error message.

6. Repeat steps 3-5 until the user chooses to exit.

7. End

## Program.

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

struct Node {
    int data;
    struct Node* next;
};
struct Node* head = NULL;
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}
void insertAtFront(int data) {
    struct Node* newNode = createNode(data);
    if (head == NULL) {
        head = newNode;
    } else {
        newNode->next = head;
        head = newNode;
    }
    printf("Node inserted at the front.\n");
}
void insertAfterNode(int data,int key)
{
struct Node* newNode=createNode(data);
```

```
struct Node* current = head;
    while (current != NULL) {
        if (current->data == key) {
            newNode->next = current->next;
            current->next = newNode;
            printf("Node inserted after %d.\n", key);
            return;
        }
        current = current->next;
    }
    printf("Node with key %d not found.\n", key);
}

void deleteFromBeginning() {
    if (head == NULL) {
        printf("List is empty. Nothing to delete.\n");
    } else {
        struct Node* temp = head;
        head = head->next;
        free(temp);
        printf("Node deleted from the beginning.\n");
    }
}

void deleteFromEnd() {
    if (head == NULL) {
        printf("List is empty. Nothing to delete.\n");
    } else if (head->next == NULL) {
        free(head);
        head = NULL;
        printf("Node deleted from the end.\n");
    } else {
        struct Node* current = head;
```

```
        while (current->next->next != NULL) {
            current = current->next;
        }
        free(current->next);
        current->next = NULL;
        printf("Node deleted from the end.\n");
    }
}

void search(int key) {
    struct Node* current = head;
    while (current != NULL) {
        if (current->data == key) {
            printf("Node with key %d found.\n", key);
            return;
        }
        current = current->next;
    }
    printf("Node with key %d not found.\n", key);
}

void traverse() {
    struct Node* current = head;
    printf("Linked List: ");
    while (current != NULL) {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}

int main() {
    int choice, data, key;
```

```
    while (1) {
        printf("\nMenu:\n");
        printf("1. Insert at the front\n");
        printf("2. Insert after a node\n");
        printf("3. Delete from the beginning\n");
        printf("4. Delete from the end\n");
        printf("5. Search\n");
        printf("6. Traverse\n");
        printf("7. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter data to insert: ");
                scanf("%d", &data);
                insertAtFront(data);
                break;
            case 2:
                printf("Enter data to insert: ");
                scanf("%d", &data);
                printf("Enter the key after which to insert: ");
                scanf("%d", &key);
                insertAfterNode(data, key);
                break;
            case 3:
                deleteFromBeginning();
                break;
            case 4:
                deleteFromEnd();
                break;
            case 5:
```

```
            printf("Enter the key to search: ");
            scanf("%d", &key);
            search(key);
            break;
        case 6:
            traverse();
            break;
        case 7:
            exit(0);
        default:
            printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

## Output

Menu:

1. Insert at the front

2. Insert after a node

3. Delete from the beginning

4. Delete from the end

5. Search

6. Traverse

7. Exit

Enter your choice: 1

Enter data to insert: 2

Node inserted at the front.

Menu:

1. Insert at the front

2. Insert after a node

3. Delete from the beginning

4. Delete from the end

5. Search

6. Traverse

7. Exit

Enter your choice: 2

Enter data to insert: 3

Enter the key after which to insert: 2

Node inserted after 2.

Menu:

1. Insert at the front

2. Insert after a node

3. Delete from the beginning

4. Delete from the end

5. Search

6. Traverse

7. Exit

Enter your choice:

3

Node deleted from the beginning.

Menu:

1. Insert at the front

2. Insert after a node

3. Delete from the beginning

4. Delete from the end

5. Search

6. Traverse

7. Exit

Enter your choice: 4

Node deleted from the end.

Menu:

1. Insert at the front
2. Insert after a node
3. Delete from the beginning
4. Delete from the end
5. Search
6. Traverse
7. Exit

Enter your choice: 5

Enter the key to search: 3

Node with key 3 not found.

Menu:

1. Insert at the front
2. Insert after a node
3. Delete from the beginning
4. Delete from the end
5. Search
6. Traverse
7. Exit

Enter your choice: 6

Linked List: NULL

Menu:

1. Insert at the front
2. Insert after a node
3. Delete from the beginning
4. Delete from the end
5. Search

6. Traverse

7. Exit

Enter your choice: 7

Exiting the program

**Experiment 13** **Date:20/10/2023**

# Doubly Linkedlist

## Aim

To implement the following operations on a Doubly

linked list.

i. creation

ii. Count the number of nodes

iii. Insert a node at first position

iv. Insert a node at l

v. Delete a node from the first position

vi. Delete a node from last

vii. Searching

viii. Traversal

## Algorithm

Algorithm: Doubly Linked List Operations

Data:

- struct Node: Defines the structure of a node in the doubly linked list.
  - int data: Data value stored in the node.
  - struct Node* prev: Pointer to the previous node.
  - struct Node* next: Pointer to the next node.
- head: Pointer to the first node of the doubly linked list.

Functions:

1. createNode(data): Creates a new node with the given data and returns its pointer.
2. countNodes(): Counts the number of nodes in the list.
3. insertAtFirst(data): Inserts a node with the given data at the beginning of the list.
4. insertAtPosition(data, pos): Inserts a node with the given data at the specified position.
5. deleteFromFirst(): Deletes the first node from the list.
6. deleteFromLast(): Deletes the last node from the list.
7. search(data): Searches for a node with the given data in the list.
8. traverse(): Displays all the nodes in the list.

Algorithm Steps:

1. Start:
2. Initialize head pointer to NULL.
3. Display the menu with options:
   a. Insert at the first position
   b. Insert at a specific position
   c. Delete from the first position
   d. Delete from the last position
   e. Search
   f. Traverse
   g. Exit
4. Read user choice.
5. Perform the selected operation based on the user choice:
   a. If choice is 'Insert at the first position':
      i. Read data from the user.
      ii. Call insertAtFirst(data).
   b. If choice is 'Insert at a specific position':
      i. Read data and position from the user.
      ii. Call insertAtPosition(data, pos).
   c. If choice is 'Delete from the first position':
      i. Call deleteFromFirst().
   d. If choice is 'Delete from the last position':
      i. Call deleteFromLast().
   e. If choice is 'Search':
      i. Read data from the user.
      ii. Call search(data).
   f. If choice is 'Traverse':
      i. Call traverse().
   g. If choice is 'Exit':
      i. Exit the program.
   h. If choice is invalid:

i. Display an error message.

6. Repeat steps 3-5 until the user chooses to exit.

7. End.

## Program

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

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

struct Node* head = NULL;

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

// Function to count the number of nodes in the list
int countNodes() {
    struct Node* current = head;
    int count = 0;
    while (current != NULL) {
        count++;
        current = current->next;
```

```
    }
    return count;
}

// Function to insert a node at the first position
void insertAtFirst(int data) {
    struct Node* newNode = createNode(data);
    if (head == NULL) {
        head = newNode;
    } else {
        newNode->next = head;
        head->prev = newNode;
        head = newNode;
    }
    printf("Node inserted at the first position.\n");
}

// Function to insert a node at position 'pos'
void insertAtPosition(int data, int pos) {
    if (pos < 1 || pos > countNodes() + 1) {
        printf("Invalid position.\n");
        return;
    }
    if (pos == 1) {
        insertAtFirst(data);
        return;
    }
    struct Node* newNode = createNode(data);
    struct Node* current = head;
    for (int i = 1; i < pos - 1; i++) {
        current = current->next;
    }
    newNode->next = current->next;
```

```
    if (current->next != NULL) {
        current->next->prev = newNode;
    }
    current->next = newNode;
    newNode->prev = current;
    printf("Node inserted at position %d.\n", pos);
}

// Function to delete a node from the first position
void deleteFromFirst() {
    if (head == NULL) {
        printf("List is empty. Nothing to delete.\n");
        return;
    }
    struct Node* temp = head;
    head = head->next;
    if (head != NULL) {
        head->prev = NULL;
    }
    free(temp);
    printf("Node deleted from the first position.\n");
}

// Function to delete a node from the last position
void deleteFromLast() {
    if (head == NULL) {
        printf("List is empty. Nothing to delete.\n");
        return;
    }
    if (head->next == NULL) {
        free(head);
        head = NULL;
        printf("Node deleted from the last position.\n");
```

```
        return;
    }
    struct Node* current = head;
    while (current->next != NULL) {
        current = current->next;
    }
    current->prev->next = NULL;
    free(current);
    printf("Node deleted from the last position.\n");
}

// Function to search for a node with given data
void search(int data) {
    struct Node* current = head;
    int pos = 1;
    while (current != NULL) {
        if (current->data == data) {
            printf("Node with data %d found at position %d.\n", data, pos);
            return;
        }
        current = current->next;
        pos++;
    }
    printf("Node with data %d not found.\n", data);
}

// Function to traverse and display the doubly linked list
void traverse() {
    struct Node* current = head;
    printf("Doubly Linked List: ");
    while (current != NULL) {
        printf("%d <-> ", current->data);
        current = current->next;
```

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

int main() {
    int choice, data, pos;

    while (1) {
        printf("\nMenu:\n");
        printf("1. Insert at the first position\n");
        printf("2. Insert at a specific position\n");
        printf("3. Delete from the first position\n");
        printf("4. Delete from the last position\n");
        printf("5. Search\n");
        printf("6. Traverse\n");
        printf("7. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter data to insert: ");
                scanf("%d", &data);
                insertAtFirst(data);
                break;
            case 2:
                printf("Enter data to insert: ");
                scanf("%d", &data);
                printf("Enter position to insert: ");
                scanf("%d", &pos);
                insertAtPosition(data, pos);
                break;
            case 3:
```

```
            deleteFromFirst();
            break;
        case 4:
            deleteFromLast();
            break;
        case 5:
            printf("Enter data to search: ");
            scanf("%d", &data);
            search(data);
            break;
        case 6:
            traverse();
            break;
        case 7:
            exit(0);
        default:
            printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

## Output

Menu:

1. Insert at the first position

2. Insert at a specific position

3. Delete from the first position

4. Delete from the last position

5. Search

6. Traverse

7. Exit

Enter your choice: 1

Enter data to insert: 3

Node inserted at the first position.

Menu:

1. Insert at the first position
2. Insert at a specific position
3. Delete from the first position
4. Delete from the last position
5. Search
6. Traverse
7. Exit

Enter your choice: 1

Enter data to insert: 4

Node inserted at the first position.

Menu:

1. Insert at the first position
2. Insert at a specific position
3. Delete from the first position
4. Delete from the last position
5. Search
6. Traverse
7. Exit

Enter your choice: 1

Enter data to insert: 6

Node inserted at the first position.

Menu:

1. Insert at the first position
2. Insert at a specific position
3. Delete from the first position
4. Delete from the last position
5. Search

6. Traverse

7. Exit

Enter your choice: 2

Enter data to insert: 2

Enter position to insert: 3

Node inserted at position 3.

Menu:

1. Insert at the first position

2. Insert at a specific position

3. Delete from the first position

4. Delete from the last position

5. Search

6. Traverse

7. Exit

Enter your choice: 3

Node deleted from the first position.

Menu:

1. Insert at the first position

2. Insert at a specific position

3. Delete from the first position

4. Delete from the last position

5. Search

6. Traverse

7. Exit

Enter your choice: 4

Node deleted from the last position.

Menu:

1. Insert at the first position

2. Insert at a specific position

3. Delete from the first position

4. Delete from the last position

5. Search

6. Traverse

7. Exit

Enter your choice: 5

Enter data to search: 4

Node with data 4 found at position 1.

Menu:

1. Insert at the first position

2. Insert at a specific position

3. Delete from the first position

4. Delete from the last position

5. Search

6. Traverse

7. Exit

Enter your choice: 6

Doubly Linked List: 4 <-> 2 <-> NULL

Menu:

1. Insert at the first position

2. Insert at a specific position

3. Delete from the first position

4. Delete from the last position

5. Search

6. Traverse

7. Exit

Enter your choice: 7

Exiting

**Experiment 14** **Date:27/10/2023**

# Stack Using Linkedlist

## Aim

To implement a menu driven program to perform

following stack operations using linked list

i. push

ii. pop

iii.Traversal

## Algorithm

Algorithm: Stack Operations using Singly Linked List

Data:

- struct Node: Defines the structure of a node in the stack.
  - int data: Data value stored in the node.
  - struct Node* next: Pointer to the next node.
- top: Pointer to the top node of the stack.

Functions:

1. push(&top, data): Pushes a node with the given data onto the stack.
2. pop(&top): Pops the top node from the stack and returns its data.
3. traverse(top): Displays all the nodes in the stack from top to bottom.

Algorithm Steps:

1. Start:
2. Initialize top pointer to NULL.
3. Display the menu with options:
   a. Push
   b. Pop
   c. Traverse
   d. Exit
4. Read user choice.

5. Perform the selected operation based on the user choice:
   a. If choice is 'Push':
      i. Read data from the user.
      ii. Call push(&top, data).
   b. If choice is 'Pop':
      i. Check if the stack is not empty.
      ii. Call pop(&top) and display the popped element.
   c. If choice is 'Traverse':
      i. Call traverse(top) to display all elements in the stack.
   d. If choice is 'Exit':
      i. Exit the program.
   e. If choice is invalid:
      i. Display an error message.
6. Repeat steps 3-5 until the user chooses to exit.
7. End.

## Program

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


struct Node {
    int data;
    struct Node* next;
};


void push(struct Node** top, int data);
int pop(struct Node** top);
void traverse(struct Node* top);

int main() {
    struct Node* top = NULL;
```

```
int choice, data;

do {

    printf("\nMenu:\n");
    printf("1. Push\n");
    printf("2. Pop\n");
    printf("3. Traverse\n");
    printf("4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
        case 1:
            printf("Enter data to push: ");
            scanf("%d", &data);
            push(&top, data);
            break;
        case 2:
            if (top != NULL) {
                printf("Popped element: %d\n", pop(&top));
            } else {
                printf("Stack is empty\n");
            }
            break;
        case 3:
            printf("Stack elements:\n");
            traverse(top);
            break;
        case 4:
            printf("Exiting the program.\n");
            break;
```

```
        default:
          printf("Invalid choice. Please enter a valid option.\n");
      }
    } while (choice != 4);

    return 0;
}


void push(struct Node** top, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation error\n");
        exit(EXIT_FAILURE);
    }

    newNode->data = data;
    newNode->next = *top;
    *top = newNode;

    printf("%d pushed onto the stack.\n", data);
}


int pop(struct Node** top) {
    if (*top == NULL) {
        printf("Stack underflow\n");
        exit(EXIT_FAILURE);
    }

    struct Node* temp = *top;
    int data = temp->data;
    *top = temp->next;
```

```
    free(temp);

    return data;
}



void traverse(struct Node* top) {
    if (top == NULL) {
        printf("Stack is empty\n");
        return;
    }

    while (top != NULL) {
        printf("%d ", top->data);
        top = top->next;
    }
    printf("\n");
}
```

## Output

Menu:

1. Push

2. Pop

3. Traverse

4. Exit

Enter your choice: 1

Enter data to push: 3

3 pushed onto the stack.

Menu:

1. Push

2. Pop

3. Traverse

4. Exit

Enter your choice: 1

Enter data to push: 4

4 pushed onto the stack.

Menu:

1. Push

2. Pop

3. Traverse

4. Exit

Enter your choice: 1

Enter data to push: 6

6 pushed onto the stack.

Menu:

1. Push

2. Pop

3. Traverse

4. Exit

Enter your choice: 2

Popped element: 6

Menu:

1. Push

2. Pop

3. Traverse

4. Exit

Enter your choice: 4

Exiting the program.

**Experiment 15** **Date:27/10/2023**

# Queue Using Linkedlist

## Aim

To implement a menu driven program to perform

following queue operations using linked list

1. enqueue
2. dequeue
3. Traversal

## Algorithm

Algorithm: Queue Operations using Singly Linked List

Data:

- struct Node: Defines the structure of a node in the queue.
  - int data: Data value stored in the node.
  - struct Node* next: Pointer to the next node.
- struct Queue: Defines the structure of a queue.
  - struct Node* front: Pointer to the front node of the queue.
  - struct Node* rear: Pointer to the rear node of the queue.

Functions:

1. initializeQueue(q): Initializes the queue by setting front and rear pointers to NULL.
2. isEmpty(q): Checks if the queue is empty.
3. enqueue(q, value): Enqueues a node with the given value into the queue.
4. dequeue(q): Dequeues a node from the queue.
5. traverse(q): Displays all the nodes in the queue.

Algorithm Steps:

1. Start:
2. Initialize an empty queue using initializeQueue(&q).
3. Display the menu with options:
   a. Enqueue

b. Dequeue

c. Traverse

d. Exit

4. Read user choice.

5. Perform the selected operation based on the user choice:

   a. If choice is 'Enqueue':

      i. Read value from the user.

      ii. Call enqueue(&q, value).

   b. If choice is 'Dequeue':

      i. Call dequeue(&q).

   c. If choice is 'Traverse':

      i. Call traverse(&q) to display all elements in the queue.

   d. If choice is 'Exit':

      i. Exit the program.

   e. If choice is invalid:

      i. Display an error message.

6. Repeat steps 3-5 until the user chooses to exit.

7. End.

## Program

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

struct Node {
    int data;
    struct Node* next;
};

struct Queue {
    struct Node* front;
    struct Node* rear;
};
```

```
void initializeQueue(struct Queue* q) {
    q->front = q->rear = NULL;
}

int isEmpty(struct Queue* q) {
    return (q->front == NULL);
}

void enqueue(struct Queue* q, int value) {

    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;

    if (isEmpty(q)) {
        q->front = q->rear = newNode;
    } else {

        q->rear->next = newNode;
        q->rear = newNode;
    }

    printf("Enqueued %d\n", value);
}

void dequeue(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty, cannot dequeue\n");
        return;
    }

    struct Node* temp = q->front;
```

```
    q->front = q->front->next;

    if (q->front == NULL) {
        q->rear = NULL;
    }

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

    free(temp);
}

void traverse(struct Queue* q) {
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        return;
    }

    printf("Queue elements: ");
    struct Node* current = q->front;
    while (current != NULL) {
        printf("%d ", current->data);
        current = current->next;
    }
    printf("\n");
}

int main() {
    struct Queue q;
    initializeQueue(&q);

    int choice, value;

    do {
```

```
        printf("\nQueue Operations:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Traverse\n");
        printf("4. Exit\n");



        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:

                printf("Enter the value to enqueue: ");
                scanf("%d", &value);
                enqueue(&q, value);
                break;
            case 2:

                dequeue(&q);
                break;
            case 3:

                traverse(&q);
                break;
            case 4:

                printf("Exiting program\n");
                break;
            default:
                printf("Invalid choice! Please enter a valid option.\n");
        }
```

```
    } while (choice != 4);


    return 0;
}
```

## Output

```
/tmp/NCSQ6a4T6q.o
Queue Operations:
1. Enqueue
2. Dequeue
3. Traverse
4. Exit
Enter your choice: 1
Enter the value to enqueue: 3
Enqueued 3

Queue Operations:
1. Enqueue
2. Dequeue
3. Traverse
4. Exit
Enter your choice: 1
Enter the value to enqueue: 4
Enqueued 4

Queue Operations:
1. Enqueue
2. Dequeue
3. Traverse
4. Exit
Enter your choice: 1
Enter the value to enqueue: 5
```

Enqueued 5

Queue Operations:

1. Enqueue

2. Dequeue

3. Traverse

4. Exit

Enter your choice: 2

Dequeued 3

Queue Operations:

1. Enqueue

2. Dequeue

3. Traverse

4. Exit

Enter your choice: 3

Queue elements: 4 5

Queue Operations:

1. Enqueue

2. Dequeue

3. Traverse

4. Exit

Enter your choice: 4

Exiting program

**Experiment 16** **Date:2/11/2023**

# Binary Search Tree

## Aim

Menu Driven program to implement Binary Search Tree

Operations- Insertion of node, Deletion of a node,inorder-traversal,

Pre-order traversal and post-order

## Algorithm

Algorithm: Binary Search Tree (BST) Operations

Data:

- struct Node: Defines the structure of a node in the BST.
  - int data: Data value stored in the node.
  - struct Node* left: Pointer to the left child node.
  - struct Node* right: Pointer to the right child node.

Functions:

1. createNode(value): Creates a new node with the given value.
2. findMin(root): Finds the node with the minimum value in the subtree rooted at the given node.
3. insert(root, value): Inserts a node with the given value into the BST.
4. deleteNode(root, value): Deletes a node with the given value from the BST.
5. inorder(root): Performs in-order traversal of the BST.
6. preOrderTraversal(root): Performs pre-order traversal of the BST.
7. postOrderTraversal(root): Performs post-order traversal of the BST.
8. displayMenu(): Displays the menu of operations for the BST.

Algorithm Steps:

1. Start:
2. Initialize an empty BST with root = NULL.
3. Display the menu of operations using displayMenu().

4. Read user choice.

5. Perform the selected operation based on the user choice:

   a. If choice is 'Insert Node':

      i. Read value from the user.

      ii. Call insert(root, value) to insert the value into the BST.

   b. If choice is 'Delete Node':

      i. Read value from the user.

      ii. Call deleteNode(root, value) to delete the node with the given value from the BST.

   c. If choice is 'In-order Traversal':

      i. Call inorder(root) to perform in-order traversal and display the nodes.

   d. If choice is 'Pre-order Traversal':

      i. Call preOrderTraversal(root) to perform pre-order traversal and display the nodes.

   e. If choice is 'Post-order Traversal':

      i. Call postOrderTraversal(root) to perform post-order traversal and display the nodes.

   f. If choice is 'Exit':

      i. Exit the program.

   g. If choice is invalid:

      i. Display an error message.

6. Repeat steps 3-5 until the user chooses to exit.

7. End.

## **Program**

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

struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
```

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

struct Node* findMin(struct Node* root) {
    while (root->left != NULL) {
        root = root->left;
    }
    return root;
}

struct Node* insert(struct Node* root, int value) {
    if (root == NULL) {
        return createNode(value);
    }

    if (value < root->data) {
        root->left = insert(root->left, value);
    } else if (value > root->data) {
        root->right = insert(root->right, value);
    }

    return root;
}

struct Node* deleteNode(struct Node* root, int value) {
    if (root == NULL) {
        return root;
    }

    if (value < root->data) {
        root->left = deleteNode(root->left, value);
    } else if (value > root->data) {
```

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

    return root;
}

void inorder(struct Node* root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

void preOrderTraversal(struct Node* root) {
    if (root != NULL) {
        printf("%d ", root->data);
        preOrderTraversal(root->left);
```

```
        preOrderTraversal(root->right);
    }
}

void postOrderTraversal(struct Node* root) {
    if (root != NULL) {
        postOrderTraversal(root->left);
        postOrderTraversal(root->right);
        printf("%d ", root->data);
    }
}

void displayMenu() {
    printf("\nBinary Search Tree Operations:\n");
    printf("1. Insert Node\n");
    printf("2. Delete Node\n");
    printf("3. In-order traversal\n");
    printf("4. Pre-order Traversal\n");
    printf("5. Post-order Traversal\n");
    printf("6. Exit\n");
}

int main() {
    struct Node* root = NULL;
    int choice, value;

    do {
        displayMenu();
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
```

```
            printf("Enter the value to insert: ");
            scanf("%d", &value);
            root = insert(root, value);
            break;
        case 2:
            printf("Enter the value to delete: ");
            scanf("%d", &value);
            root = deleteNode(root, value);
            break;
        case 3:
            printf("In-order Traversal: ");
            inorder(root);
            printf("\n");
            break;
        case 4:
            printf("Pre-order Traversal: ");
            preOrderTraversal(root);
            printf("\n");
            break;
        case 5:
            printf("Post-order Traversal: ");
            postOrderTraversal(root);
            printf("\n");
            break;
        case 6:
            printf("Exiting program\n");
            break;
        default:
            printf("Invalid choice! Please enter a valid option.\n");
    }

  } while (choice != 6);
  return 0;
```

}

## Output

/tmp/NCSQ6a4T6q.o

Binary Search Tree Operations:

1. Insert Node
2. Delete Node
3. In-order traversal
4. Pre-order Traversal
5. Post-order Traversal
6. Exit

Enter your choice: 1

Enter the value to insert: 2

Binary Search Tree Operations:

1. Insert Node
2. Delete Node
3. In-order traversal
4. Pre-order Traversal
5. Post-order Traversal
6. Exit

Enter your choice: 1

3Enter the value to insert:

1

Binary Search Tree Operations:

1. Insert Node
2. Delete Node
3. In-order traversal
4. Pre-order Traversal
5. Post-order Traversal
6. Exit

Enter your choice: 1

Enter the value to insert: 5

Binary Search Tree Operations:

1. Insert Node

2. Delete Node

3. In-order traversal

4. Pre-order Traversal

5. Post-order Traversal

6. Exit

Enter your choice: 1

Enter the value to insert: 6

Binary Search Tree Operations:

1. Insert Node

2. Delete Node

3. In-order traversal

4. Pre-order Traversal

5. Post-order Traversal

6. Exit

Enter your choice: 2

Enter the value to delete: 3

Binary Search Tree Operations:

1. Insert Node

2. Delete Node

3. In-order traversal

4. Pre-order Traversal

5. Post-order Traversal

6. Exit

Enter your choice: 3

In-order Traversal: 1 2 5 6

Binary Search Tree Operations:

1. Insert Node

2. Delete Node

3. In-order traversal

4. Pre-order Traversal

5. Post-order Traversal

6. Exit

Enter your choice: 4

Pre-order Traversal: 2 1 5 6

Binary Search Tree Operations:

1. Insert Node
2. Delete Node
3. In-order traversal
4. Pre-order Traversal
5. Post-order Traversal
6. Exit

Enter your choice: 5

Post-order Traversal: 1 6 5 2

Binary Search Tree Operations:

1. Insert Node
2. Delete Node
3. In-order traversal
4. Pre-order Traversal
5. Post-order Traversal
6. Exit

Enter your choice: 6

Exiting program

**Experiment 17** **Date:9/11/2023**

# Bit String

## Aim

Program to implement set operations using Bit String

## Algoithm

Algorithm: Set Operations using Bitstrings

Data:

- int a[11]: Bitstring representation for set A.
- int b[11]: Bitstring representation for set B.
- int u[11]: Bitstring representation for the union of sets A and B.
- int us[11]: Universal set containing numbers 1 to 11.

Functions:

1. seta(): Accepts input for set A and generates its bitstring representation.
2. setb(): Accepts input for set B and generates its bitstring representation.
3. Union(): Computes the union of sets A and B and generates its bitstring representation.

Algorithm Steps:

1. Start:
2. Initialize arrays a, b, u with zeros.
3. Initialize the universal set us = {1, 2, 3, ..., 11}.
4. Call seta() to input elements for set A and generate its bitstring representation.
   a. Read the size of set A, s1.
   b. For each element d1 in set A, set a[d1] = 1.
   c. Display the bitstring representation of set A.
5. Call setb() to input elements for set B and generate its bitstring representation.
   a. Read the size of set B, s2.
   b. For each element d2 in set B, set b[d2] = 1.
   c. Display the bitstring representation of set B.
6. Call Union() to compute the union of sets A and B and generate its bitstring representation.

a. For each index i from 1 to 10:

- If a[i] or b[i] is 1, set u[i] = 1; otherwise, set u[i] = 0.

b. Display the bitstring representation of the union.

7. End.

## Program

```
#include<stdio.h>
#include<stdlib.h>
int a[11],b[11],u[11],i;
int us[11]={1,2,3,4,5,6,7,8,9,10,11};
void seta()
{
int s1,d1;
printf("Enter the size of first set\n");
scanf("%d",&s1);
printf("Enter elements\n");
for(i=0;i<s1;i++)
{
scanf("%d",&d1);
a[d1]=1;
}
printf("Bitstring of A:\n");
for(i=1;i<11;i++)
{
printf("%d\t",a[i]);
}
printf("\n");
}
void setb()
{
int s2,d2;
printf("Enter the size of second set\n");
```

```
scanf("%d",&s2);
printf("Enter elements\n");
for(i=0;i<s2;i++)

{
scanf("%d",&d2);
b[d2]=1;
}
printf("Bitstring of B: \n");
for(i=1;i<11;i++)
{ printf("%d \t",b[i]); }
printf(" \n"); }
void Union()
{
for(i=1;i<11;i++)
{
if(a[i]==1 || b[i]==1)
{ u[i]=1; }
else { u[i]=0; }
}
printf("Union: \n");
for(i=1;i<11;i++)
{ printf("%d \t",u[i]); }
}
void main()
{
seta();
setb();
Union();
}
```

## **Output**

Enter the size of first set

2

Enter elements

3 5

Bitstring of A:

0 0 1 0 1 0 0 0 0 0

Enter the size of second set

3

Enter elements

4 5 6

4 5 6

Bitstring of B:

0 0 0 1 1 1 0 0 0 0

Union:

0 0 1 1 1 1 0 0 0 0