Experiment 10 **Date:18/11/2024**

Queue using Arrays

Aim:

Program to implement queue operations using arrays.

Algorithm:

main()

1. Start
2. Initialize variables: inp_arr[SIZE], Rear = -1, Front = -1
3. do
4. Display choices of Queue operations.
5. Read ch
6. switch(ch) {
7. if ch == 1:
8. Call enqueue()
9. if ch == 2:
10. Call dequeue()
11. if ch == 3:
12. Call show()
13. if ch == 4:
14. Exit from switch
15. Default: Print "Incorrect choice"
16. while (true)
17. stop

void enqueue()

1. Start
2. Check if (Rear == SIZE - 1):
3. Print "Overflow"
4. else:
5. If (Front == -1):
6. Set Front=0

7. Read insert_item
8. Increment Rear by 1
9. Set inp_arr[Rear] = insert_item
10. Exit

void dequeue()

1. Start
2. Check if (Front == -1 or Front > Rear):
3. Print "Underflow"
4. Return
5. else:
6. Print "Element deleted from the Queue: inp_arr[Front]"
7. Increment Front by 1
8. Exit

void show()

1. Start
2. Check if (Front == -1):
3. Print "Empty Queue"
4. else:
5. Print "Queue:"
6. for (int i = Front to Rear):
7. Print inp_arr[i]
8. Exit

Program

```
#include <stdio.h>
#include<stdlib.h>
# define SIZE 5
void enqueue();
void dequeue();
void show();
int inp_arr[SIZE];
int Rear = - 1;
int Front = - 1;
```

```
void main()
{
    int ch;
    while (1)
    {
            printf("1.Enqueue Operation\n");
            printf("2.Dequeue Operation\n");
            printf("3.Display the Queue\n");
            printf("4.Exit\n");
            printf("Enter your choice of operations : ");
            scanf("%d", &ch);
            switch (ch)
            {
            case 1:
                    enqueue();
                    break;
            case 2:
                    dequeue();
                    break;
            case 3:
                    show();
                    break;
            case 4:
                    exit(0);
            default:
                    printf("Incorrect choice \n");
            }
    }
}
void enqueue()
{
```

```
    int insert_item;
    if (Rear == SIZE - 1)
        printf("Overflow \n");
    else
    {
        if (Front == - 1)
            Front = 0;
        printf("Element to be inserted in the Queue : ");
        scanf("%d", &insert_item);
        Rear = Rear + 1;
        inp_arr[Rear] = insert_item;
    }
}
void dequeue()
{
    if (Front == - 1 || Front > Rear)
    {
        printf("Underflow \n");
        return ;
    }
    else
    {
        printf("Element deleted from the Queue: %d\n", inp_arr[Front]);
        Front = Front + 1;
    }
}
void show()
{
    if (Front == - 1)
        printf("Empty Queue \n");
    else
```

```
    {
        printf("Queue: \n");
        for (int i = Front; i <= Rear; i++)
            printf("%d ", inp_arr[i]);
        printf("\n");
    }
}
```

Output

1.Enqueue Operation

2.Dequeue Operation

3.Display the Queue

4.Exit

Enter your choice of operations : 1

Element to be inserted in the Queue : 3

1.Enqueue Operation

2.Dequeue Operation

3.Display the Queue

4.Exit

Enter your choice of operations : 1

Element to be inserted in the Queue : 9

1.Enqueue Operation

2.Dequeue Operation

3.Display the Queue

4.Exit

Enter your choice of operations : 1

Element to be inserted in the Queue : 8

1.Enqueue Operation

2.Dequeue Operation

3.Display the Queue

4.Exit

Enter your choice of operations : 1

Element to be inserted in the Queue : 5

1.Enqueue Operation

2.Dequeue Operation

3.Display the Queue

4.Exit

Enter your choice of operations : 1

Element to be inserted in the Queue : 8

1.Enqueue Operation

2.Dequeue Operation

3.Display the Queue

4.Exit

Enter your choice of operations : 1

Overflow

1.Enqueue Operation

2.Dequeue Operation

3.Display the Queue

4.Exit

Enter your choice of operations : 3

Queue:

3 9 8 5 8

1.Enqueue Operation

2.Dequeue Operation

3.Display the Queue

4.Exit

Enter your choice of operations : 2

Element deleted from the Queue: 3

1.Enqueue Operation

2.Dequeue Operation

3.Display the Queue

4.Exit

Enter your choice of operations : 2

Element deleted from the Queue: 9

1.Enqueue Operation

2.Dequeue Operation

3.Display the Queue

4.Exit

Enter your choice of operations : 2

Element deleted from the Queue: 8

1.Enqueue Operation

2.Dequeue Operation

3.Display the Queue

4.Exit

```
Enter your choice of operations : 2
Element deleted from the Queue: 5

1.Enqueue Operation
2.Dequeue Operation
3.Display the Queue
4.Exit
Enter your choice of operations : 2
Element deleted from the Queue: 8

1.Enqueue Operation
2.Dequeue Operation
3.Display the Queue
4.Exit
Enter your choice of operations : 2
Underflow

1.Enqueue Operation
2.Dequeue Operation
3.Display the Queue
4.Exit
Enter your choice of operations : 4
```

Experiment 11 **Date:18/11/2024**

Circular Queue using Arrays

Aim:

Program to implement circular queue using arrays.

Algorithm

main()

1. Start
2. Initialize variables: queue[SIZE], front = -1, rear = -1
3. Repeat until
4. Display menu:

   a) Enqueue
   b) Dequeue
   c) Display
   d) Exit

5. Read choice ch
6. Switch on ch:
   a) If ch == 1:
      1. Call enqueue()
   b) If ch == 2:
      1. Call dequeue()
   c) If ch == 3:
      1. Call display()
   d) If ch == 4:
      1. Exit the program
   e) Else:
      1. Print "Invalid choice"
7. Stop

Void enqueue()

1. Start
2. Check if the queue is full:
3. If (rear + 1) % SIZE == front:
4. Print "Queue Overflow"
5. Else:
6. Read value

7. If front == -1:
8. Set front = 0
9. Update rear = (rear + 1) % SIZE
10. Insert value into queue[rear]
11. Print "value enqueued successfully"
12. Exit

Void dequeue()

1. Start
2. Check if the queue is empty:
3. If front == -1:
4. Print "Queue Underflow"
5. Else:
6. Print "Dequeued: queue[front]"
7. If front == rear:
8. Set front = -1 and rear = -1 (Queue becomes empty)
9. Else:
10. Update front = (front + 1) % SIZE
11. Exit

Void display()

1. Start
2. Check if the queue is empty:
3. If front == -1:
4. Print "Queue is empty"
5. Else:
6. Initialize i = front
7. Repeat until i == rear:
8. Print queue[i]
9. Update i = (i + 1) % SIZE

10. Print queue[i] (for rear)
11. Exit

Program

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 5
void enqueue();
void dequeue();
void display();
int queue[SIZE];
int front = -1, rear = -1;
void main() {
   int ch;
   do {
      printf("\n1) Enqueue\n");
      printf("2) Dequeue\n");
      printf("3) Display\n");
      printf("4) Exit\n");
      printf("Enter a choice: ");
      scanf("%d", &ch);
      switch (ch) {
         case 1:
            enqueue();
            break;
         case 2:
            dequeue();
            break;
         case 3:
```

```
                display();
                break;
            case 4:
                exit(0);
            default:
                printf("Invalid choice entered\n");
        }
    } while (1);
}
void enqueue() {
    int value;
    if ((rear + 1) % SIZE == front) {
        printf("Queue Overflow\n");
        return;
    }
    printf("Enter the value to enqueue: ");
    scanf("%d", &value);
    if (front == -1) {
        front = 0;
    }
    rear = (rear + 1) % SIZE;
    queue[rear] = value;
    printf("%d enqueued successfully\n", value);
}
void dequeue() {
    if (front == -1) {
        printf("Queue Underflow\n");
        return;
```

```
    }
    printf("Dequeued: %d\n", queue[front]);
    if (front == rear) {
        front = rear = -1;
    } else {
        front = (front + 1) % SIZE;
    }
}
void display() {
    if (front == -1) {
        printf("Queue is empty\n");
        return;
    }
    printf("Queue elements are: ");
    int i = front;
    while (1) {
        printf("%d ", queue[i]);
        if (i == rear) {
            break;
        }
        i = (i + 1) % SIZE;
    }
    printf("\n");
}
```

Output

1) Enqueue

2) Dequeue

3) Display

4) Exit

Enter a choice: 1

Enter the value to enqueue: 4

4 enqueued successfully

1) Enqueue

2) Dequeue

3) Display

4) Exit

Enter a choice: 1

Enter the value to enqueue: 6

6 enqueued successfully

1) Enqueue

2) Dequeue

3) Display

4) Exit

Enter a choice: 1

Enter the value to enqueue: 8

8 enqueued successfully

1) Enqueue

2) Dequeue

3) Display

4) Exit

Enter a choice: 1

Enter the value to enqueue: 9

9 enqueued successfully

1) Enqueue

2) Dequeue

3) Display

4) Exit

Enter a choice: 1

Enter the value to enqueue: 2

2 enqueued successfully

1) Enqueue

2) Dequeue

3) Display

4) Exit

Enter a choice: 1

Queue Overflow

1) Enqueue

2) Dequeue

3) Display

4) Exit

Enter a choice: 3

Queue elements are: 4 6 8 9 2

1) Enqueue

2) Dequeue

3) Display

4) Exit

Enter a choice: 2

Dequeued: 4

1) Enqueue

2) Dequeue

3) Display

4) Exit

Enter a choice: 2

Dequeued: 6

1) Enqueue

2) Dequeue

3) Display

4) Exit

Enter a choice: 2

Dequeued: 8

1) Enqueue

2) Dequeue

3) Display

4) Exit

Enter a choice: 2

Dequeued: 9

1) Enqueue

2) Dequeue

3) Display

4) Exit

Enter a choice: 2

Dequeued: 2

1) Enqueue

2) Dequeue

3) Display

4) Exit

Enter a choice: 2

Queue Underflow

1) Enqueue

2) Dequeue

3) Display

4) Exit

Enter a choice: 4

Experiment 12 **Date:20/11/2024**

Singly Linked List

Aim:

Program to implement the following operations on a singly linked list

a. Creation,
b. Insert a new node at front
c. Insert an element after a particular node
d. Insert a new node at end
e. Searching
f. Traversal

Algorithm

main()

1. Start
2. Initialize variables: struct Node* head = NULL
3. Repeat until user exits:
4. Display menu:
   a) Creation
   b) Insert at Front
   c) Insert After a Particular Node
   d) Insert at End
   e) Search for an Element
   f) Traversal
   g) Exit
5. Read choice
6. Switch on choice:
7. If ch == 1:
8. Read data
9. Call create_list(data)
10. If ch == 2:
11. Read data
12. Call insert_at_front(data)
13. If ch == 3:
14. Read key and data
15. Call insert_after_node(key, data)
16. If ch == 4:
17. Read data
18. Call insert_at_end(data)
19. If ch == 5:
20. Read key

21. Call search(key)
22. If ch == 6:
23. Call traverse()
24. If ch == 7:
25. Exit from switch
26. Stop

Void create_list(data)

1. Start
2. Check if head == NULL:
3. Allocate memory for a new node
4. Set head->data = data and head->next = NULL
5. Print "Node created with data: data"
6. Else:
7. Print "List already exists. Use insert operations to add nodes."
8. Exit

Void insert_at_front(data)

1. Start
2. Allocate memory for a new node
3. Check if memory allocation is successful:
4. If false:
5. Print "Memory allocation failed."
6. Set newNode->data = data and newNode->next = head
7. Update head = newNode
8. Print "Node inserted at front with data: data"
9. Exit

Void insert_after_node(key, data)

1. Start
2. Initialize temp = head
3. Traverse the list until temp == NULL or temp->data == key:
4. If temp == NULL:

5. Print "Key key not found in the list."
6. Allocate memory for a new node
7. Check if memory allocation is successful:
8. If false:
9. Print "Memory allocation failed."
10. Set newNode->data = data and newNode->next = temp->next
11. Update temp->next = newNode
12. Print "Node inserted after key with data: data"
13. Exit

Void insert_at_end(data)

1. Start
2. Allocate memory for a new node
3. Check if memory allocation is successful:
4. If false:
5. Print "Memory allocation failed."
6. Set newNode->data = data and newNode->next = NULL
7. Check if head == NULL:
8. If true:
9. Set head = newNode
10. Print "Node inserted at end with data: data"
11. Initialize temp = head
12. Traverse the list until temp->next == NULL
13. Update temp->next = newNode
14. Print "Node inserted at end with data: data"
15. Exit

Void search(key)

1. Start
2. Initialize temp = head and pos = 1
3. Traverse the list until temp == NULL:

4. If temp->data == key:
5. Print "Element key found at position pos"
6. Update temp = temp->next and pos = pos + 1
7. Print "Element key not found in the list"
8. Exit

Void traverse()

1. Start
2. Check if head == NULL:
3. If true:
4. Print "List is empty"
5. Initialize temp = head
6. Print "List elements: "
7. Traverse the list:
8. Print temp->data followed by ->
9. Update temp = temp->next
10. Print “NULL”
11. Exit

Program

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
   int data;
   struct Node* next;
};
struct Node* head = NULL;
void create_list(int data);
void insert_at_front(int data);
void insert_after_node(int key, int data);
void insert_at_end(int data);
void search(int key);
```

```
void traverse();
int main() {
   int ch, data, key;
   do {
      printf("1) Creation\n");
      printf("2) Insert at Front\n");
      printf("3) Insert at Particular Node\n");
      printf("4) Insert at End\n");
      printf("5) Search  an Element\n");
      printf("6) Traversal\n");
      printf("7) Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &ch);
      switch (ch) {
         case 1:
            printf("Enter data for the node: ");
            scanf("%d", &data);
            create_list(data);
            break;
         case 2:
            printf("Enter data to insert at front: ");
            scanf("%d", &data);
            insert_at_front(data);
            break;
         case 3:
            printf("Enter the key after which to insert: ");
            scanf("%d", &key);
            printf("Enter data to insert: ");
            scanf("%d", &data);
            insert_after_node(key, data);
            break;
```

```
            case 4:
                printf("Enter data to insert at end: ");
                scanf("%d", &data);
                insert_at_end(data);
                break;
            case 5:
                printf("Enter the element to search for: ");
                scanf("%d", &key);
                search(key);
                break;
            case 6:
                traverse();
                break;
            case 7:
                exit(0);
            default:
                printf("Invalid choice! Try again.\n");
        }
    } while (ch != 7);

    return 0;
}
void create_list(int data) {
    if (head != NULL) {
        printf("List already exists. Use insert operations to add nodes.\n");
        return;
    }
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode) {
        printf("Memory allocation failed.\n");
        return;
```

```
    }
    newNode->data = data;
    newNode->next = NULL;
    head = newNode;
    printf("Node created with data: %d\n", data);
}
void insert_at_front(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode) {
        printf("Memory allocation failed.\n");
        return;
    }
    newNode->data = data;
    newNode->next = head;
    head = newNode;
    printf("Node inserted at front with data: %d\n", data);
}
void insert_after_node(int key, int data) {
    struct Node* temp = head;
    while (temp != NULL && temp->data != key) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Key %d not found in the list.\n", key);
        return;
    }
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode) {
        printf("Memory allocation failed.\n");
        return;
    }
```

```
    newNode->data = data;
    newNode->next = temp->next;
    temp->next = newNode;
    printf("Node inserted after %d with data: %d\n", key, data);
}
void insert_at_end(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode) {
        printf("Memory allocation failed.\n");
        return;
    }
    newNode->data = data;
    newNode->next = NULL;
    if (head == NULL) {
        head = newNode;
        printf("Node inserted at end with data: %d\n", data);
        return;
    }
    struct Node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
    printf("Node inserted at end with data: %d\n", data);
}
void search(int key) {
    struct Node* temp = head;
    int pos = 1;
    while (temp != NULL) {
        if (temp->data == key) {
            printf("Element %d found at position %d.\n", key, pos);
```

```
            return;
        }
        temp = temp->next;
        pos++;
    }
    printf("Element %d not found in the list.\n", key);
}
void traverse() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    struct Node* temp = head;
    printf("List elements: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
```

Output

1) Creation
2) Insert at Front
3) Insert at Particular Node
4) Insert at End
5) Search an Element
6) Traversal
7) Exit

Enter your choice: 1

Enter data for the node: 3

Node created with data: 3

1) Creation
2) Insert at Front
3) Insert at Particular Node
4) Insert at End
5) Search an Element
6) Traversal
7) Exit
Enter your choice: 2
Enter data to insert at front: 4
Node inserted at front with data: 4

1) Creation
2) Insert at Front
3) Insert at Particular Node
4) Insert at End
5) Search an Element
6) Traversal
7) Exit
Enter your choice: 3
Enter the key after which to insert: 3
Enter data to insert: 6
Node inserted after 3 with data: 6

1) Creation
2) Insert at Front
3) Insert at Particular Node
4) Insert at End
5) Search an Element
6) Traversal
7) Exit

```
Enter your choice: 6
List elements: 4 -> 3 -> 6 -> NULL

1) Creation
2) Insert at Front
3) Insert at Particular Node
4) Insert at End
5) Search  an Element
6) Traversal
7) Exit
Enter your choice: 4
Enter data to insert at end: 8
Node inserted at end with data: 8

1) Creation
2) Insert at Front
3) Insert at Particular Node
4) Insert at End
5) Search  an Element
6) Traversal
7) Exit
Enter your choice: 6
List elements: 4 -> 3 -> 6 -> 8 -> NULL

1) Creation
2) Insert at Front
3) Insert at Particular Node
4) Insert at End
5) Search  an Element
6) Traversal
7) Exit
```

Enter your choice: 5

Enter the element to search for: 6

Element 6 found at position 3.

1) Creation

2) Insert at Front

3) Insert at Particular Node

4) Insert at End

5) Search an Element

6) Traversal

7) Exit

Enter your choice: 7

Experiment 13 **Date:02/12/2024**

Singly Linked List

Aim: Program to implement the following operations on a singly linked list

a. Creation,
b. Deletion from beginning
c. Deletion from the end
d. Deletion from particular location
e. Traversal.

Algorithm

main()

1. Start
2. Initialize struct Node* head = NULL
3. Repeat until the user exits:
4. Display menu:
    a. Creation
    b. Deletion from Beginning
    c. Deletion from End
    d. Deletion from a Particular Location
    e. Traversal
    f. Exit
5. Read choice
6. Switch on choice:
7. If ch == 1:
8. Call create_list()
9. If ch == 2:
10. Call delete_from_beginning()
11. If ch == 3:
12. Call delete_from_end()
13. If ch == 4:
14. Call delete_from_location()
15. If ch == 5:
16. Call traverse()
17. If choice == 6:
18. Exit the switch
19. Stop

Void create_list()

1. Start
2. Allocate memory for a new node

3. Check if memory allocation is successful:
4. If false:
5. Print "Memory allocation failed."
6. Read data
7. If head == NULL:
8. Set head->data = data and head->next = NULL
9. Else:
10. Traverse to the end of the list
11. Set last_node->next = newNode
12. Print "Node created"
13. Exit

Void delete_from_beginning()

1. Start
2. Check if head == NULL:
3. Print "List is empty"
4. Initialize temp = head
5. Update head = head->next
6. Print "Deleted: temp->data"
7. Free temp
8. Exit

Void delete_from_end()

1. Start
2. Check if head == NULL:
3. Print "List is empty"
4. If head->next == NULL:
5. Print "Deleted: head->data"
6. Free head and set head = NULL
7. Initialize temp = head
8. Traverse the list until temp->next->next == NULL
9. Print "Deleted: temp->next->data"
10. Free temp->next and set temp->next = NULL
11. Exit

Void delete_from_location()

1. Start
2. Read key
3. Check if head == NULL:
4. Print "List is empty"
5. If head->data == key:
6. Call delete_from_beginning()
7. Initialize temp = head

8. Traverse until temp->next != NULL and temp->next->data != key:
9. If temp->next == NULL:
10. Print "Node not found"
11. Initialize to_delete = temp->next
12. Update temp->next = to_delete->next
13. Print "Deleted: to_delete->data"
14. Free to_delete
15. Exit

Void traverse()

1. Start
2. Check if head == NULL:
3. Print "List is empty"
4. Initialize temp = head
5. Print "List elements: "
6. Traverse the list:
7. Print temp->data
8. Update temp = temp->next
9. Print "NULL"
10. Exit

Program

```
#include <stdio.h>

#include <stdlib.h>

struct Node {

   int data;

   struct Node* next;

};

struct Node* head = NULL;

void create_list();

void delete_from_beginning();

void delete_from_end();

void delete_from_location();

void traverse();

void main() {
```

```
    int ch;
    do {
        printf("\n1) Creation\n");
        printf("2) Deletion from Beginning\n");
        printf("3) Deletion from End\n");
        printf("4) Deletion from a Particular Location\n");
        printf("5) Traversal\n");
        printf("6) Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &ch);
        switch (ch) {
            case 1:
                create_list();
                break;
            case 2:
                delete_from_beginning();
                break;
            case 3:
                delete_from_end();
                break;
            case 4:
                delete_from_location();
                break;
            case 5:
                traverse();
                break;
            case 6:
```

```
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice.\n");
        }
    } while (ch != 6);
}
void create_list() {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        return;
    }
    printf("Enter data: ");
    scanf("%d", &newNode->data);
    newNode->next = NULL;
    if (head == NULL) {
        head = newNode;
    } else {
        struct Node* temp = head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
    printf("Node created.\n");
}
```

```
void delete_from_beginning() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    struct Node* temp = head;
    head = head->next;
    printf("Deleted: %d\n", temp->data);
    free(temp);
}
void delete_from_end() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    if (head->next == NULL) {
        printf("Deleted: %d\n", head->data);
        free(head);
        head = NULL;
        return;
    }
    struct Node* temp = head;
    while (temp->next->next != NULL) {
        temp = temp->next;
    }
    printf("Deleted: %d\n", temp->next->data);
    free(temp->next);
```

```
    temp->next = NULL;
}
void delete_from_location() {
    int key;
    printf("Enter the element to delete: ");
    scanf("%d", &key);
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    if (head->data == key) {
        delete_from_beginning();
        return;
    }
    struct Node* temp = head;
    while (temp->next != NULL && temp->next->data != key) {
        temp = temp->next;
    }
    if (temp->next == NULL) {
        printf("Node not found.\n");
        return;
    }
    struct Node* to_delete = temp->next;
    temp->next = to_delete->next;
    printf("Deleted: %d\n", to_delete->data);
    free(to_delete);
}
```

```
void traverse() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    struct Node* temp = head;
    printf("List elements: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
```

Output

1) Creation

2) Deletion from Beginning

3) Deletion from End

4) Deletion from a Particular Location

5) Traversal

6) Exit

Enter your choice: 1

Enter data: 4

Node created.

1) Creation

2) Deletion from Beginning

3) Deletion from End

4) Deletion from a Particular Location

5) Traversal

6) Exit

Enter your choice: 1

Enter data: 6

Node created.

1) Creation

2) Deletion from Beginning

3) Deletion from End

4) Deletion from a Particular Location

5) Traversal

6) Exit

Enter your choice: 1

Enter data: 5

Node created.

1) Creation

2) Deletion from Beginning

3) Deletion from End

4) Deletion from a Particular Location

5) Traversal

6) Exit

Enter your choice: 1

Enter data: 7

Node created.

1) Creation

2) Deletion from Beginning

3) Deletion from End

4) Deletion from a Particular Location

5) Traversal

6) Exit

Enter your choice: 1

Enter data: 8

Node created.

1) Creation

2) Deletion from Beginning

3) Deletion from End

4) Deletion from a Particular Location

5) Traversal

6) Exit

Enter your choice: 1

Enter data: 9

Node created.

1) Creation

2) Deletion from Beginning

3) Deletion from End

4) Deletion from a Particular Location

5) Traversal

6) Exit

Enter your choice: 5

List elements: 4 -> 6 -> 5 -> 7 -> 8 -> 9 -> NULL

1) Creation

2) Deletion from Beginning

3) Deletion from End

4) Deletion from a Particular Location

5) Traversal

6) Exit

Enter your choice: 2

Deleted: 4

1) Creation

2) Deletion from Beginning

3) Deletion from End

4) Deletion from a Particular Location

5) Traversal

6) Exit

Enter your choice: 3

Deleted: 9

1) Creation

2) Deletion from Beginning

3) Deletion from End

4) Deletion from a Particular Location

5) Traversal

6) Exit

Enter your choice: 4

Enter the element to delete: 7

Deleted: 7

1) Creation

2) Deletion from Beginning

3) Deletion from End

4) Deletion from a Particular Location

5) Traversal

6) Exit

Enter your choice: 5

List elements: 6 -> 5 -> 8 -> NULL

1) Creation

2) Deletion from Beginning

3) Deletion from End

4) Deletion from a Particular Location

5) Traversal

6) Exit

Enter your choice: 6

Exiting program.

Experiment 14 **Date:02/12/2024**

Stack using Linked List

Aim: To implement a menu driven program to perform following stack operations using linked list

a) push
b) pop
c) Traversal

Algorithm

Main()

1. Start.
2. Initialize top = NULL.
3. Do the following:
4. Display the menu:
    a) Push
    b) Pop
    c) Traversal
    d) Exit
5. Read choice
6. Switch
    - Case 1: Call push().
    - Case 2: Call pop().
    - Case 3: Call traverse().
    - Case 4: Exit switch
    - Default: Print "Invalid choice."
7. Stop.

void push()

1. Start.
2. Create a new node using malloc.
3. If memory allocation fails, print "Memory allocation failed." and exit.

4. Read the value
5. Set newNode->data = value and newNode->next = top.
6. top = newNode.
7. Print "Pushed: value".
8. Exit.

void pop()

1. Start.
2. If top == NULL, print "Stack is empty. Underflow!" and exit.
3. Set temp = top.
4. Print "Popped: temp->data".
5. top = top->next.
6. Free temp.
7. Exit.

void traverse()

1. Start.
2. If top == NULL, print "Stack is empty." and exit.
3. Set temp = top.
4. Print "Stack elements:".
5. While temp != NULL:
6. Print temp->data.
7. temp = temp->next.
8. Print "NULL".
9. Exit.

Program

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
```

```
    struct Node* next;
};
struct Node* top = NULL;
void push();
void pop();
void traverse();
void main() {
    int ch;
    do {
        printf("\n1) Push\n");
        printf("2) Pop\n");
        printf("3) Traversal\n");
        printf("4) Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &ch);
        switch (ch) {
            case 1:
                push();
                break;
            case 2:
                pop();
                break;
            case 3:
                traverse();
                break;
            case 4:
                printf("Exiting program.\n");
                break;
```

```
            default:
                printf("Invalid choice.\n");
        }
    } while (ch != 4);
}
void push() {
    int value;
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        return;
    }
    printf("Enter the value to push: ");
    scanf("%d", &value);
    newNode->data = value;
    newNode->next = top;
    top = newNode;
    printf("Pushed: %d\n", value);
}
void pop() {
    if (top == NULL) {
        printf("Stack is empty. Underflow!\n");
        return;
    }
    struct Node* temp = top;
    printf("Popped: %d\n", temp->data);
    top = top->next;
    free(temp);
```

```
}
void traverse() {
   if (top == NULL) {
      printf("Stack is empty.\n");
      return;
   }
   struct Node* temp = top;
   printf("Stack elements: ");
   while (temp != NULL) {
      printf("%d -> ", temp->data);
      temp = temp->next;
   }
   printf("NULL\n");
}
```

Output

```
1) Push
2) Pop
3) Traversal
4) Exit
Enter your choice: 1
Enter the value to push: 9
Pushed: 9

1) Push
2) Pop
3) Traversal
4) Exit
Enter your choice: 1
```

Enter the value to push: 4

Pushed: 4

1) Push

2) Pop

3) Traversal

4) Exit

Enter your choice: 1

Enter the value to push: 6

Pushed: 6

1) Push

2) Pop

3) Traversal

4) Exit

Enter your choice: 1

Enter the value to push: 5

Pushed: 5

1) Push

2) Pop

3) Traversal

4) Exit

Enter your choice: 1

Enter the value to push: 7

Pushed: 7

1) Push

2) Pop

3) Traversal

4) Exit

Enter your choice: 1

Enter the value to push: 8

Pushed: 8

1) Push

2) Pop

3) Traversal

4) Exit

Enter your choice: 3

Stack elements: 8 -> 7 -> 5 -> 6 -> 4 -> 9 -> NULL

1) Push

2) Pop

3) Traversal

4) Exit

Enter your choice: 2

Popped: 8

1) Push

2) Pop

3) Traversal

4) Exit

Enter your choice: 2

Popped: 7

1) Push

2) Pop

3) Traversal

4) Exit

Enter your choice: 3

Stack elements: 5 -> 6 -> 4 -> 9 -> NULL

1) Push

2) Pop

3) Traversal

4) Exit

Enter your choice: 4

Exiting program.

Experiment 15 **Date:02/12/2024**

Queue using Linked List

Aim: To implement a menu driven program to perform following queue operations using linked list

a) enqueue
b) dequeue
c) Traversal

Algorithm

Main()

1. Start.
2. Initialize front = NULL and rear = NULL.
3. Display menu:
    a) Enqueue
    b) Dequeue
    c) Traversal
    d) Exit
4. Read choice
5. Switch case:
    a) Case 1: Call enqueue().
    b) Case 2: Call dequeue().
    c) Case 3: Call traverse().
    d) Case 4: Exit the program.
    e) Default: Print "Invalid choice."
6. Stop.

void enqueue()

1. Start.
2. Create a new node using malloc.
3. If memory allocation fails, print "Memory allocation failed." and exit.
4. Read the value

5. Set newNode->data = value and newNode->next = NULL.
6. If rear == NULL (queue is empty):
7. Set front = rear = newNode.
8. Else:
9. Set rear->next = newNode.
10. Update rear = newNode.
11. Print "Enqueued: value".
12. Exit.

void dequeue()

1. Start.
2. If front == NULL, print "Queue is empty. Underflow!" and exit.
3. Set temp = front.
4. Print "Dequeued: temp->data".
5. front = front->next.
6. If front == NULL:
7. Set rear = NULL (queue is now empty).
8. Free temp.
9. Exit.

void traverse()

1. Start.
2. If front == NULL, print "Queue is empty." and exit.
3. Set temp = front.
4. While temp != NULL:
5. Print temp->data.
6. Move temp = temp->next.

7. Print "NULL".
8. Exit.

Program

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
struct Node* top = NULL;
void push();
void pop();
void traverse();
void main() {
    int ch;
    do {
        printf("\n1) Push\n");
        printf("2) Pop\n");
        printf("3) Traversal\n");
        printf("4) Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &ch);
        switch (ch) {
            case 1:
                push();
                break;
            case 2:
                pop();
```

```
            break;
        case 3:
            traverse();
            break;
        case 4:
            printf("Exiting program.\n");
            break;
        default:
            printf("Invalid choice.\n");
      }
   } while (ch != 4);
}
void push() {
   int value;
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   if (newNode == NULL) {
      printf("Memory allocation failed.\n");
      return;
   }
   printf("Enter the value to push: ");
   scanf("%d", &value);
   newNode->data = value;
   newNode->next = top;
   top = newNode;
   printf("Pushed: %d\n", value);
}
void pop() {
   if (top == NULL) {
```

```
        printf("Stack is empty. Underflow!\n");
        return;
    }
    struct Node* temp = top;
    printf("Popped: %d\n", temp->data);
    top = top->next;
    free(temp);
}
void traverse() {
    if (top == NULL) {
        printf("Stack is empty.\n");
        return;
    }
    struct Node* temp = top;
    printf("Stack elements: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
```

Output

```
1) Push
2) Pop
3) Traversal
4) Exit
Enter your choice: 1
Enter the value to push: 9
```

```
Pushed: 9

1) Push
2) Pop
3) Traversal
4) Exit
Enter your choice: 1
Enter the value to push: 4
Pushed: 4

1) Push
2) Pop
3) Traversal
4) Exit
Enter your choice: 1
Enter the value to push: 6
Pushed: 6

1) Push
2) Pop
3) Traversal
4) Exit
Enter your choice: 1
Enter the value to push: 5
Pushed: 5

1) Push
2) Pop
```

3) Traversal

4) Exit

Enter your choice: 1

Enter the value to push: 7

Pushed: 7

1) Push

2) Pop

3) Traversal

4) Exit

Enter your choice: 1

Enter the value to push: 8

Pushed: 8

1) Push

2) Pop

3) Traversal

4) Exit

Enter your choice: 3

Stack elements: 8 -> 7 -> 5 -> 6 -> 4 -> 9 -> NULL

1) Push

2) Pop

3) Traversal

4) Exit

Enter your choice: 2

Popped: 8

1) Push

2) Pop

3) Traversal

4) Exit

Enter your choice: 2

Popped: 7

1) Push

2) Pop

3) Traversal

4) Exit

Enter your choice: 3

Stack elements: 5 -> 6 -> 4 -> 9 -> NULL

1) Push

2) Pop

3) Traversal

4) Exit

Enter your choice: 4

Exiting program.

Experiment 16 **Date:04/12/2024**

Doubly Linked List

Aim: To implement the perform following operations on a Doubly linked list.

a) Creation
b) Count the number of nodes
c) Searching
d) Traversal

Algorithm

Main()

1. Start.
2. Initialize front = NULL and rear = NULL.
3. Display the menu:
   a) Creation
   b) Count the number of nodes
   c) Searching
   d) Traversal
   e) Exit
4. Read choice
5. Switch case:
   a) Case 1: Call create_list()
   b) Case 2: Call count_nodes()
   c) Case 3: Call search().
   d) Case 4: Call traverse().
   e) Case 5: Exit.
   f) Default: Print "Invalid choice."
6. Stop.

void create_list()

1. Start.
2. Create a new node using malloc.

3. If memory allocation fails, print "Memory allocation failed." and exit.
4. Read the value
5. Set newNode->data = value, newNode->prev = NULL, and newNode->next = NULL.
6. If head == NULL:
7. Set head = newNode.
8. Else:
9. Set last->next = newNode and newNode->prev = last.
10. Print "Node created."
11. Exit.

void count_nodes()

1. Start.
2. If head == NULL, print "List is empty." and exit.
3. Initialize count = 0 and temp = head.
4. While temp != NULL:
5. temp = temp->next.
6. Print "Total number of nodes: count."
7. Exit.

void search()

1. Start.
2. If head == NULL, print "List is empty." and exit.
3. Read key.
4. Initialize pos = 1 and a pointer temp = head.
5. While temp != NULL:
6. If temp->data == key, print "Element found at position".
7. temp = temp->next.

8. Else, print "Element not found."
9. Exit.

void traverse()

1. Start.
2. If head == NULL, print "List is empty." and exit.
3. Set temp = head.
4. Print "List elements:".
5. While temp != NULL:
6. Print temp->data.
7. temp = temp->next.
8. Print "NULL".
9. Exit.

Program

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
};
struct Node* head = NULL;
void create_list();
void count_nodes();
void search();
void traverse();
void main() {
    int ch;
```

```
    do {
        printf("\n1) Creation\n");
        printf("2) Count the number of nodes\n");
        printf("3) Searching\n");
        printf("4) Traversal\n");
        printf("5) Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &ch);
        switch (ch) {
            case 1:
                create_list();
                break;
            case 2:
                count_nodes();
                break;
            case 3:
                search();
                break;
            case 4:
                traverse();
                break;
            case 5:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice.\n");
        }
    } while (ch != 5);
```

```
}
void create_list() {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   if (newNode == NULL) {
      printf("Memory allocation failed.\n");
      return;
   }
   printf("Enter data: ");
   scanf("%d", &newNode->data);
   newNode->prev = NULL;
   newNode->next = NULL;
   if (head == NULL) {
      head = newNode;
   } else {
      struct Node* temp = head;
      while (temp->next != NULL) {
         temp = temp->next;
      }
      temp->next = newNode;
      newNode->prev = temp;
   }
   printf("Node created.\n");
}
void count_nodes() {
   int count = 0;
   struct Node* temp = head;
   if (head == NULL) {
      printf("List is empty.\n");
```

```
            return;
        }
        while (temp != NULL) {
            count++;
            temp = temp->next;
        }
        printf("Total number of nodes: %d\n", count);
    }
    void search() {
        int key, position = 1;
        printf("Enter the element to search: ");
        scanf("%d", &key);
        struct Node* temp = head;
        while (temp != NULL) {
            if (temp->data == key) {
                printf("Element %d found at position %d.\n", key, position);
                return;
            }
            temp = temp->next;
            position++;
        }
        printf("Element %d not found in the list.\n", key);
    }
    void traverse() {
        if (head == NULL) {
            printf("List is empty.\n");
            return;
        }
```

```
    struct Node* temp = head;
    printf("List elements: ");
    while (temp != NULL) {
        printf("%d <-> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
```

Output

1) Creation

2) Count the number of nodes

3) Searching

4) Traversal

5) Exit

Enter your choice: 1

Enter data: 3

Node created.

1) Creation

2) Count the number of nodes

3) Searching

4) Traversal

5) Exit

Enter your choice: 1

Enter data: 8

Node created.

1) Creation

2) Count the number of nodes

3) Searching

4) Traversal

5) Exit

Enter your choice: 1

Enter data: 4

Node created.

1) Creation

2) Count the number of nodes

3) Searching

4) Traversal

5) Exit

Enter your choice: 1

Enter data: 5

Node created.

1) Creation

2) Count the number of nodes

3) Searching

4) Traversal

5) Exit

Enter your choice: 1

Enter data: 6

Node created.

1) Creation

2) Count the number of nodes

3) Searching

4) Traversal

5) Exit

Enter your choice: 1

Enter data: 9

Node created.

1) Creation

2) Count the number of nodes

3) Searching

4) Traversal

5) Exit

Enter your choice: 4

List elements: 3 <-> 8 <-> 4 <-> 5 <-> 6 <-> 9 <-> NULL

1) Creation

2) Count the number of nodes

3) Searching

4) Traversal

5) Exit

Enter your choice: 2

Total number of nodes: 6

1) Creation

2) Count the number of nodes

3) Searching

4) Traversal

5) Exit

```
Enter your choice: 3
Enter the element to search: 5
Element 5 found at position 4.

1) Creation
2) Count the number of nodes
3) Searching
4) Traversal
5) Exit
Enter your choice: 5
Exiting program.
```

Experiment 17 **Date:04/12/2024**

Doubly Linked List

Aim: To implement the perform following operations on a Doubly linked list.

a) Creation
b) Insert a node at first position
c) Insert a node at last
d) Delete a node from the first position
e) Delete a node from last
f) Traversal

Algorithm

Main()

1. Start.
2. Initialize head = NULL.
3. Display the menu:
   a) Creation
   b) Insert at first
   c) Insert at last
   d) Delete from first
   e) Delete from last
   f) Traversal
   g) Exit
4. Read choice.
5. Switch case:
   a) Case 1: Call create_list().
   b) Case 2: Call insert_first().
   c) Case 3: Call insert_last().
   d) Case 4: Call delete_first().
   e) Case 5: Call delete_last().
   f) Case 6: Call traverse().

g) Case 7: Exit switch.

6. Stop.

void create_list()

1. Start.
2. Create a new node.
3. If head == NULL, set head = newNode.
4. else, traverse till the end
5. Stop.

void insert_first()

1. Start.
2. Create a new node.
3. Set newNode->next = head.
4. If head != NULL, update head->prev = newNode.
5. head = newNode.
6. Stop.

void insert_last()

1. Start.
2. Create a new node.
3. If head == NULL, set head = newNode.
4. Otherwise, traverse till end.
5. Stop.

void delete_first()

1. Start.
2. If head == NULL, print "List is empty." and exit.

3. head = head->next.
4. If head != NULL, set head->prev = NULL.
5. Stop.

void delete_last()

1. Start.
2. If head == NULL, print "List is empty." and exit.
3. Traverse to the last node.
4. Remove the last node and update the second-last node's next pointer to NULL.
5. Stop.

void traverse()

1. Start.
2. Traverse the list and print all elements.
3. Stop.

Program

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
   int data;
   struct Node* prev;
   struct Node* next;
};
struct Node* head = NULL;
void create_list();
void insert_first();
void insert_last();
void delete_first();
void delete_last();
```

```
void traverse();
void main() {
   int ch;
   do {
      printf("\n1) Creation\n");
      printf("2) Insert a node at the first position\n");
      printf("3) Insert a node at the last position\n");
      printf("4) Delete a node from the first position\n");
      printf("5) Delete a node from the last position\n");
      printf("6) Traversal\n");
      printf("7) Exit\n");
      printf("Enter your choice: ");
      scanf("%d", &ch);
      switch (ch) {
         case 1:
            create_list();
            break;
         case 2:
            insert_first();
            break;
         case 3:
            insert_last();
            break;
         case 4:
            delete_first();
            break;
         case 5:
            delete_last();
```

```
                break;
            case 6:
                traverse();
                break;
            case 7:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice.\n");
        }
    } while (ch != 7);
}
void create_list() {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        return;
    }
    printf("Enter data: ");
    scanf("%d", &newNode->data);
    newNode->prev = NULL;
    newNode->next = NULL;
    if (head == NULL) {
        head = newNode;
    } else {
        struct Node* temp = head;
        while (temp->next != NULL) {
            temp = temp->next;
```

```
            }
            temp->next = newNode;
            newNode->prev = temp;
        }
        printf("Node created.\n");
    }
    void insert_first() {
        struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
        if (newNode == NULL) {
            printf("Memory allocation failed.\n");
            return;
        }
        printf("Enter data: ");
        scanf("%d", &newNode->data);
        newNode->prev = NULL;
        newNode->next = head;
        if (head != NULL) {
            head->prev = newNode;
        }
        head = newNode;
        printf("Node inserted at the first position.\n");
    }
    void insert_last() {
        struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
        if (newNode == NULL) {
            printf("Memory allocation failed.\n");
            return;
        }
```

```
    printf("Enter data: ");
    scanf("%d", &newNode->data);
    newNode->next = NULL;
    if (head == NULL) {
        newNode->prev = NULL;
        head = newNode;
    } else {
        struct Node* temp = head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->prev = temp;
    }
    printf("Node inserted at the last position.\n");
}
void delete_first() {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    struct Node* temp = head;
    head = head->next;
    if (head != NULL) {
        head->prev = NULL;
    }
    printf("Deleted: %d\n", temp->data);
    free(temp);
```

```
}
void delete_last() {
   if (head == NULL) {
      printf("List is empty.\n");
      return;
   }
   if (head->next == NULL) {
      printf("Deleted: %d\n", head->data);
      free(head);
      head = NULL;
      return;
   }
   struct Node* temp = head;
   while (temp->next != NULL) {
      temp = temp->next;
   }
   printf("Deleted: %d\n", temp->data);
   temp->prev->next = NULL;
   free(temp);
}
void traverse() {
   if (head == NULL) {
      printf("List is empty.\n");
      return;
   }
   struct Node* temp = head;
   printf("List elements: ");
   while (temp != NULL) {
```

```
        printf("%d <-> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
```

Output

1) Creation

2) Insert a node at the first position

3) Insert a node at the last position

4) Delete a node from the first position

5) Delete a node from the last position

6) Traversal

7) Exit

Enter your choice: 1

Enter data: 5

Node created.

1) Creation

2) Insert a node at the first position

3) Insert a node at the last position

4) Delete a node from the first position

5) Delete a node from the last position

6) Traversal

7) Exit

Enter your choice: 1

Enter data: 9

Node created.

1) Creation

2) Insert a node at the first position

3) Insert a node at the last position

4) Delete a node from the first position

5) Delete a node from the last position

6) Traversal

7) Exit

Enter your choice: 1

Enter data: 8

Node created.

1) Creation

2) Insert a node at the first position

3) Insert a node at the last position

4) Delete a node from the first position

5) Delete a node from the last position

6) Traversal

7) Exit

Enter your choice: 1

Enter data: 3

Node created.

1) Creation

2) Insert a node at the first position

3) Insert a node at the last position

4) Delete a node from the first position

5) Delete a node from the last position

6) Traversal

7) Exit

Enter your choice: 2

Enter data: 6

Node inserted at the first position.

1) Creation

2) Insert a node at the first position

3) Insert a node at the last position

4) Delete a node from the first position

5) Delete a node from the last position

6) Traversal

7) Exit

Enter your choice: 3

Enter data: 7

Node inserted at the last position.

1) Creation

2) Insert a node at the first position

3) Insert a node at the last position

4) Delete a node from the first position

5) Delete a node from the last position

6) Traversal

7) Exit

Enter your choice: 6

List elements: 6 <-> 5 <-> 9 <-> 8 <-> 3 <-> 7 <-> NULL

1) Creation

2) Insert a node at the first position

3) Insert a node at the last position

4) Delete a node from the first position

5) Delete a node from the last position

6) Traversal

7) Exit

Enter your choice: 4

Deleted: 6

1) Creation

2) Insert a node at the first position

3) Insert a node at the last position

4) Delete a node from the first position

5) Delete a node from the last position

6) Traversal

7) Exit

Enter your choice: 5

Deleted: 7

1) Creation

2) Insert a node at the first position

3) Insert a node at the last position

4) Delete a node from the first position

5) Delete a node from the last position

6) Traversal

7) Exit

Enter your choice: 6

List elements: 5 <-> 9 <-> 8 <-> 3 <-> NULL

1) Creation

2) Insert a node at the first position

3) Insert a node at the last position

4) Delete a node from the first position

5) Delete a node from the last position

6) Traversal

7) Exit

Enter your choice: 7

Exiting program.