Batch:B-2 Roll No.:16010122151

Experiment No. 6

Grade: AA / AB / BB / BC / CC / CD /DD

Title: Implementation of Linked List

Objective: To understand the use of linked list as data structures for various application.

Expected Outcome of Experiment:

CO	Outcome
CO 2	Apply linear and non-linear data structure in application development.

Books/ Journals/ Websites referred:

Introduction:

Define Linked List

A linked list is a sequence of data structures, which are connected together via links. Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array.

Types of linked list:

Algorithm for creation, insertion, deletion, traversal and searching an element in assigned linked list type:

Singly Linked List

Insertion

In a single linked list, the insertion operation can be performed in three ways. They are as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the single linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **newNode→next** = **NULL** and **head** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, set **newNode→next** = **head** and **head** = **newNode**.

Inserting At End of the list

We can use the following steps to insert a new node at end of the single linked list...

- **Step 1 -** Create a **newNode** with given value and **newNode → next** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**).
- **Step 3 -** If it is **Empty** then, set **head** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).
- **Step 6 -** Set **temp → next** = **newNode**.

Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the single linked list...

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **newNode → next** = **NULL** and **head** = **newNode**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- **Step 6 -** Every time check whether **temp** is reached to last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
- **Step 7 -** Finally, Set '**newNode → next** = **temp → next**' and '**temp → next** = **newNode**'

Deletion

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the single linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Check whether list is having only one node (**temp** → **next** == **NULL**)
- **Step 5 -** If it is **TRUE** then set **head** = **NULL** and delete **temp** (Setting **Empty** list conditions)
- **Step 6 -** If it is **FALSE** then set **head** = **temp** → **next**, and delete **temp**.

Deleting from End of the list

We can use the following steps to delete a node from end of the single linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2'** and initialize '**temp1**' with **head**.
- **Step 4 -** Check whether list has only one Node (**temp1** → **next** == **NULL**)
- **Step 5 -** If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate the function. (Setting **Empty** list condition)
- **Step 6 -** If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until it reaches to the last node in the list. (until **temp1** → **next** == **NULL**)
- **Step 7 -** Finally, Set **temp2** → **next** = **NULL** and delete **temp1**.

Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the single linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.
- **Step 4 -** Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
- **Step 5 -** If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
- **Step 6 -** If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
- **Step 7 -** If list has only one node and that is the node to be deleted, then set **head** = **NULL** and delete **temp1** (**free(temp1)**).
- **Step 8 -** If list contains multiple nodes, then check whether **temp1** is the first node in the list (**temp1 == head**).
- **Step 9 -** If **temp1** is the first node then move the **head** to the next node (**head = head → next**) and delete **temp1**.
- **Step 10 -** If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == NULL**).
- **Step 11 -** If **temp1** is last node then set **temp2 → next** = **NULL** and delete **temp1** (**free(temp1)**).
- **Step 12 -** If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).

Displaying a Single Linked List

We can use the following steps to display the elements of a single linked list...

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Keep displaying **temp → data** with an arrow (**--->**) until **temp** reaches to the last node

- **Step 5 -** Finally display **temp → data** with arrow pointing to **NULL** (**temp → data ---> NULL**).

Doubly Linked List

Insert At Beginning

1. Start
2. Input the DATA to be inserted
3. Create a new node.
4. NewNode → Data = DATA NewNode →Lpoint =NULL
5. IF START IS NULL NewNode→ Rpoint = NULL
6. Else NewNode → Rpoint = START START→Lpoint = NewNode
7. START =NewNode
8. Stop

ii. Insertion at location:

1. Start
2. Input the DATA and POS
3. Initialize TEMP = START; i = 0
4. Repeat the step 4 if (i less than POS) and (TEMP is not equal to NULL)
5. TEMP = TEMP → RPoint; i = i +1
6. If (TEMP not equal to NULL) and (i equal to POS)

(a) Create a New Node

(b) NewNode → DATA = DATA

(c) NewNode → RPoint = TEMP → RPoint

(d) NewNode → LPoint = TEMP

(e) (TEMP → RPoint) → LPoint = NewNode

1. (f) TEMP → RPoint = New Node
2. Else

(a) Display “Position NOT found”

1. Stop

iii. Insert at End

1. Start

2. Input DATA to be inserted
3. Create a NewNode
4. NewNode → DATA = DATA
5. NewNode → RPoint = NULL
6. If (SATRT equal to NULL)

a. START = NewNode

b. NewNode → LPoint=NULL

1. Else

a. TEMP = START

b. While (TEMP → Next not equal to NULL)

i. TEMP = TEMP → Next

c. TEMP → RPoint = NewNode

d. NewNode → LPoint = TEMP

1. Stop

iv. Forward Traversal

1. Start
2. If (START is equal to NULL)

a) Display "The list is Empty"

b) Stop

1. Initialize TEMP = START
2. Repeat the step 5 and 6 until (TEMP == NULL)
3. Display "TEMP → DATA"
4. TEMP = TEMP → Next
5. Stop

v. Backward Traversal

1. Start
2. If (START is equal to NULL)
3. Display "The list is Empty"
4. Stop
5. Initialize TEMP = TAIL
6. Repeat the step 5 and 6 until (TEMP == NULL)

7. Display “TEMP → DATA”
8. TEMP = TEMP → Prev
9. Stop

Implementation of an application using linked list:

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


struct node

{

  int data;

  struct node *next;

  struct node *prev;

};
```

```
struct node *front = NULL;
```

```
int isEmpty()

{

  if (front == NULL)

  {

    return 1;

  }

  return 0;

}
```

```
void insertend()

{

  int new_data;

  printf("Enter the data to be inserted: ");
```

```
  scanf("%d", &new_data);
  struct node *newnode = malloc(sizeof(struct node));
  struct node *ptr;
  newnode->data = new_data;
  newnode->prev = NULL;
  newnode->next = NULL;
  if (isEmpty() == 1)
  {
    front = newnode;
  }
  else
  {
    ptr = front;
    while (ptr->next != NULL)
    {
      ptr = ptr->next;
    }
    ptr->next = newnode;
    newnode->prev = ptr;
    newnode->next = NULL;
  }
}
```

```
void insertbegin()
{
  int new_data;
  printf("Enter the data to be inserted: ");
```

```
  scanf("%d", &new_data);
  struct node *newnode = malloc(sizeof(struct node));
  newnode->data = new_data;
  newnode->prev = NULL;
  newnode->next = NULL;
  if (isEmpty() == 1)
  {
    front = newnode;
  }
  else
  {
    front->prev = newnode;
    newnode->next = front;
    front = newnode;
  }
}
```

```
void deletebegindoubly()
{
  struct node *temp;
  temp = front;
  temp = temp->next;
  if (isEmpty() == 1)
  {
    printf("The list is empty");
  }
  else
```

```
  {
    temp->prev = NULL;
  }
  free(temp);
}
```

```
void deleteenddoubly()
{
  struct node *temp;
  temp = front;
  while (temp->next != NULL)
  {
    temp = temp->next;
  }
  temp->next = NULL;
  temp->prev = NULL;
  free(temp);
}
```

```
void displaydoubly()
{
  struct node *ptr;
  if (front == NULL)
  {
    printf("The list is empty");
  }
```

```
  else
  {
    ptr = front;
    printf("The list is: ");
    while (ptr != NULL)
    {
      printf("%d\t", ptr->data);
      ptr = ptr->next;
    }
  }
}
```

```
void searchdoub()
{
  int c;
  printf("Enter the element you want to search in the linked 
list: ");
  scanf("%d", &c);
  struct node *p;
  p = front;
  int i = 1;
  while (p->data != c)
  {
    p = p->next;
    i++;
  }
  if (p->data == c)
```

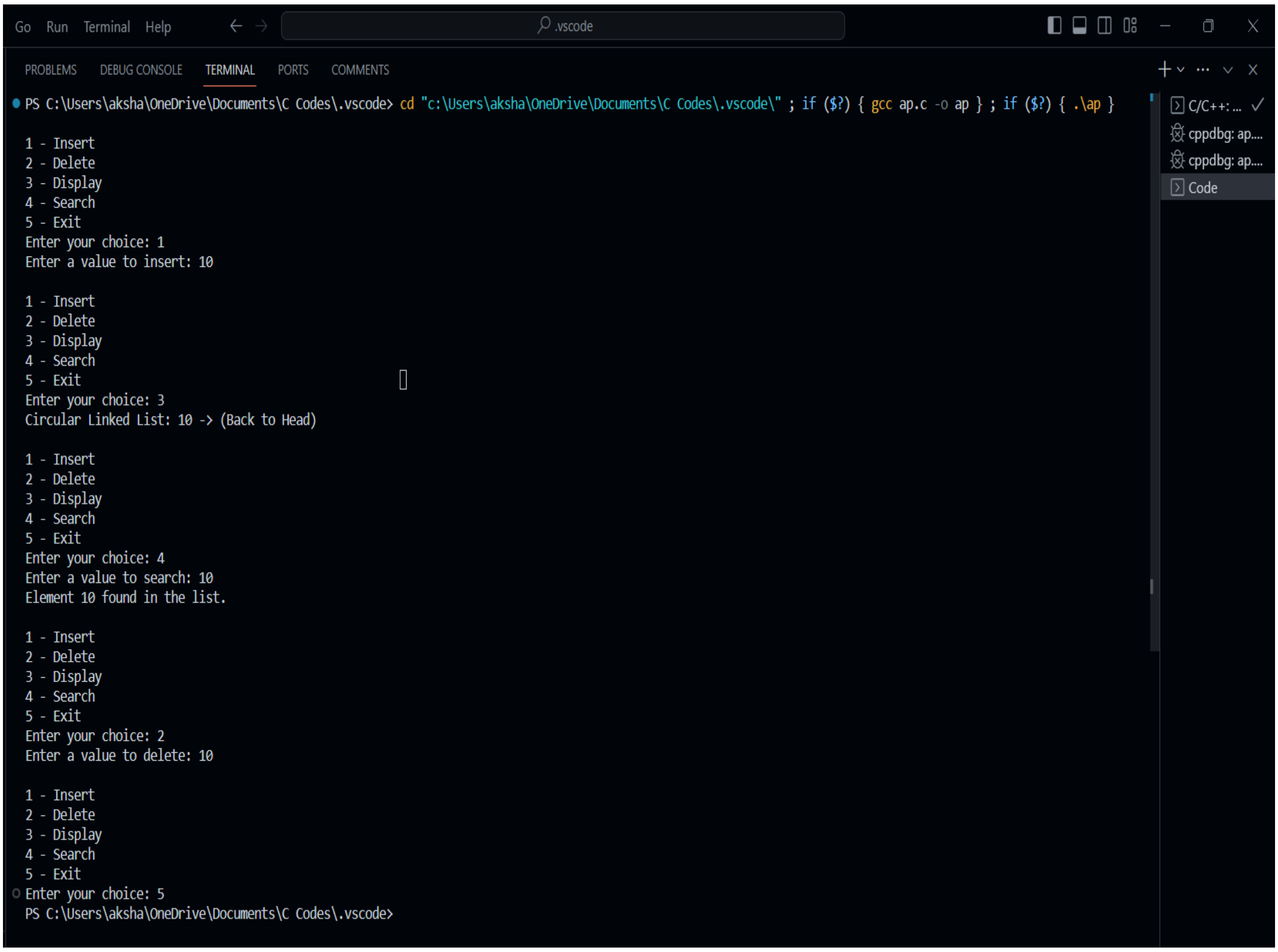
```
  {
    printf("The position of the element %d in the list is: %d",
p->data, i);
  }
  else
  {
    printf("The element is not in the list");
  }
}
```

```
int main()
{
  int x, y;
  while (1)
  {
    printf("\n**************Doubly Linked
List**************\n");
    printf("1.Insert at the begin\n2.Insert at the end\n3.Delete
at the begin\n4.Delete at the
end\n5.Traverse\n6.Search\n7.Exit\n");
    printf("Enter the number in front of the operation in Doubly
Linked List: ");
    scanf("%d", &y);
    if (y == 1)
    {
      insertbegin();
    }
    else if (y == 2)
    {
```

```
    insertend();
  }
  else if (y == 3)
  {
    deletebegindoubly();
  }
  else if (y == 4)
  {
    deleteenddoubly();
  }
  else if (y == 5)
  {
    displaydoubly();
  }
  else if (y == 6)
  {
    searchdoub();
  }
  else if (y == 7)
  {
    printf("Exiting the Doubly Linked List operations...\n");
    break;
  }
  else
  {
    printf("Invalid option\n");
  }
```

```
    }
```

```
    return 0;
}
```



Go Run Terminal Help
.vscode
PROBLEMS DEBUG CONSOLE TERMINAL PORTS COMMENTS
PS C:\Users\aksha\OneDrive\Documents\C Codes\.vscode> cd "c:\Users\aksha\OneDrive\Documents\C Codes\.vscode\" ; if ($?) { gcc ap.c -o ap } ; if ($?) { .\ap }
1 - Insert
2 - Delete
3 - Display
4 - Search
5 - Exit
Enter your choice: 1
Enter a value to insert: 10
1 - Insert
2 - Delete
3 - Display
4 - Search
5 - Exit
Enter your choice: 3
Circular Linked List: 10 -> (Back to Head)
1 - Insert
2 - Delete
3 - Display
4 - Search
5 - Exit
Enter your choice: 4
Enter a value to search: 10
Element 10 found in the list.
1 - Insert
2 - Delete
3 - Display
4 - Search
5 - Exit
Enter your choice: 2
Enter a value to delete: 10
1 - Insert
2 - Delete
3 - Display
4 - Search
5 - Exit
Enter your choice: 5
PS C:\Users\aksha\OneDrive\Documents\C Codes\.vscode>
C/C++: ...
cppdbg: ap....
cppdbg: ap....
Code

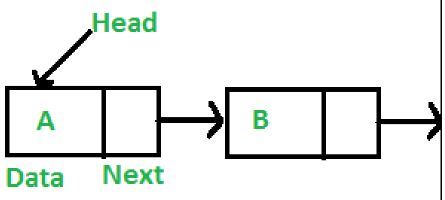
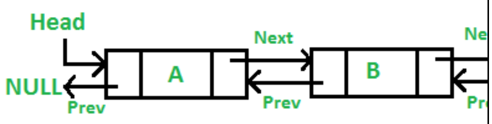
Conclusion:-

Hence we successfully implemented various type of Linked Lists.

Post lab questions:

1. Compare and contrast SLL and DLL

1.

Singly linked list (SLL)	Doubly linked list (DLL)	
SLL nodes contains 2 field -data field and next link field.	DLL nodes contains 3 fields -data field, a previous link field and a next link field.	
		
In SLL, the traversal can be done using the next node link only. Thus traversal is possible in one direction only.	In DLL, the traversal can be done using the previous node link or the next node link. Thus traversal is possible in both directions (forward and backward).	
The SLL occupies less memory than DLL as it has only 2 fields.	The DLL occupies more memory than SLL as it has 3 fields.	
Complexity of insertion and deletion at a given position is O(n).	Complexity of insertion and deletion at a given position is O(n / 2) = O(n) because traversal can be made from start or from the end.	
Complexity of deletion with a given node is O(n), because the previous node needs to be known, and traversal takes O(n)	Complexity of deletion with a given node is O(1) because the previous node can be accessed easily	
We mostly prefer to use singly linked list for the execution of stacks.	We can use a doubly linked list to execute heaps and stacks, binary trees.	
A singly linked list consumes less memory as compared to the doubly linked list.	The doubly linked list consumes more memory as compared to the singly linked list.	
Cannot point to previous element	.Can point to previous element.	