EXPERIMENT 7

Aim:

1. Write a program to insert a new node into the linked list. A node can be added into the linked list using three ways: [Write code for all the three ways.]
    a. At the front of the list
    b. After a given node
    c. At the end of the list.
2. Write a program to delete a node from the linked list. A node can be deleted from the linked list using three ways: [Write code for all the three ways.]
    a. Delete from the beginning
    b. Delete from the end
    c. Delete from the middle.
3. Write a program that takes two sorted lists as inputs and merge them into one sorted list. For example, if the first linked list A is 5 =>10 =>15, and the other linked list B is 2 => 3 => 20, then output should be 2 => 3 => 5 => 10 => 15 => 20.
4. Implement the circular linked list and perform the operation of traversal on it. In a conventional linked list, we traverse the list from the head node and stop the traversal when we reach NULL. In a circular linked list, we stop traversal when we reach the first node again.
5. Implement the doubly linked list and perform the deletion and/ or insertion operation on it. Again, you can perform insertion deletion according to the three ways as given above. Implement all of them according to availability of time.

Theory (Linked List Insert):

In a **singly linked list**, each node contains two parts: **data (info)** and a **link** to the next node. The insertion can be performed at three different positions:

1. **Insert at Beginning:**
    - Create a new node.
    - Set the new node's link to the current head (start).
    - Update the head to point to the new node.
2. **Insert After a Node:**
    - Traverse the list to find the node with previnfo.
    - Create a new node.
    - Set the new node's link to the next node.
    - Update the found node's link to the new node.
3. **Insert at End:**
    - Traverse to the last node.
    - Create a new node with its link set to NULL.
    - Update the last node's link to point to the new node.

Time Complexity:

- **Insert at Beginning:** $O(1)$ — The operation is performed directly without traversal.
- **Insert After a Node:** $O(n)$ — The list needs to be traversed to find the node with the specified value.
- **Insert at End:** $O(n)$ — The list is traversed to the last node before inserting.

Program:

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

typedef struct List
{
   int info;
   struct List *link;
} List;

List *insertbeg(List *start, int info);
void printlist(List *start);
List *insertafter(List *start, int previnfo, int info);
List *insertend(List *start, int info);
void freelist(List *start);

int main(void)
{

   List *start = NULL;

   start = insertbeg(start, 10);
   printlist(start);
   start = insertbeg(start, 20);
   printlist(start);
   start = insertbeg(start, 30);
   printlist(start);
   start = insertafter(start, 20, 12);
   printlist(start);
   start = insertend(start, 45);
   printlist(start);
   start = insertend(start, 72);

   printlist(start);

   freelist(start);
   return 0;
}

List *insertbeg(List *start, int info)
{
   List *node = (List *)malloc(sizeof(List));

   if (node == NULL)
   {
```

```
        printf("Unable to allocate memory for new node\n");
        return start;
    }

    node->info = info;
    node->link = start;
    return node;
}

List *insertafter(List *start, int previnfo, int info)
{
    List *node = (List *)malloc(sizeof(List));
    if (node == NULL)
    {
        printf("Unable to allocate memory for new node\n");
        return start;
    }
    List *current = start;

    while (current->info != previnfo && current != NULL)
    {
        current = current->link;
    }
    if (current->info == previnfo)
    {
        node->info = info;
        node->link = current->link;
        current->link = node;
    }
    else
    {
        printf("Node with value %d not found!\n", previnfo);
        return start;
    }

    return start;
}

List *insertend(List *start, int info)
{
    List *current = start;
    List *node = (List *)malloc(sizeof(List));
    if (node == NULL)
    {
        printf("Unable to allocate memory for new node\n");
        return start;
    }

    node->info = info;
    node->link = NULL;
```

```
    if (start == NULL)
    {
        return node;
    }

    while (current->link != NULL)
    {
        current = current->link;
    }

    current->link = node;

    return start;
}

void printlist(List *start)
{
    while (start != NULL)
    {
        printf("%d", start->info);
        start = start->link;

        if (start != NULL)
        {
            printf("->");
        }

    }
    printf("->NULL\n");
}

void freelist(List *start)
{
    List *current = start;

    if (start == NULL)
    {
        return;
    }

    while (current != NULL)
    {
        List *node = current;
        current = current->link;
        // printf("removed %d from memory\t", current->info);
        free(node);
    }

    return;
}
```

Output:

```
PS B:\sem3\ds\23bcp153_dsa\lab7> gcc insertall.c -o insertall
PS B:\sem3\ds\23bcp153_dsa\lab7> ./insertall
10->NULL
20->10->NULL
30->20->10->NULL
30->20->12->10->NULL
30->20->12->10->45->NULL
30->20->12->10->45->72->NULL
```

Theory (Linked List Delete):

This code handles three types of deletions in a singly linked list:

1. **Delete from the Beginning (deletebegthis)**:
   The first node is removed by updating the start pointer to the second node, and the first node's memory is freed.

2. **Delete a Specific Node (deletenodethis)**:
   The list is traversed to find the node with a given value (info). Once found, the previous node is linked to the next one, and the target node is freed.

3. **Delete from the End (deleteendthis)**:
   The last node is removed by traversing to the second-to-last node, setting its link to NULL, and freeing the last node.

Time Complexity:

- **Delete from Beginning**: $O(1)$
- **Delete a Specific Node**: $O(n)$
- **Delete from End**: $O(n)$

Program:

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

#include "mylistlib.h"
// compile command:
// gcc deletelist.c mylistlib.c -o deletelist

List *deletebegthis(List *start);
List *deletenodethis(List *start, int info);
List *deleteendthis(List *start);

int main(void)
{
   List *start = NULL;

   start = insertbeg(start, 10);
   start = insertbeg(start, 20);
   start = insertbeg(start, 30);
```

```
    start = insertafter(start, 20, 12);
    start = insertend(start, 45);
    start = insertend(start, 72);
    printlist(start);

    start = deletebegthis(start);
    printlist(start);
    start = deletenodethis(start, 45);
    printlist(start);

    start = deleteendthis(start);

    printlist(start);

    start = deleteend(start);

    printlist(start);

    freelist(start);

    return 0;
}

List *deletebegthis(List *start)
{
    if (start == NULL)
    {
        printf("List is Empty\n");
        return start;
    }
    // To Avoid Memory Leaks my friend
    List *temp = start;
    start = start->link;
    free(temp);
    return start;
}

List *deletenodethis(List *start, int info)
{
    if (start == NULL)
    {
        printf("List is Empty\n");
        return start;
    }
    List *current = start;

    if (current->info == info)
    {
        start = current->link;
        free(current);
        return start;
```

```
    }

    while (current->link->info != info && current->link != NULL)
    {
        current = current->link;
    }

    if (current->link == NULL)
    {
        printf("Node with value %d was not found!\n", info);
        return start;
    }

    List *temp = current->link;
    // Bypassing the node
    current->link = current->link->link;
    free(temp);

    return start;
}

List *deleteendthis(List *start)
{
    // If only one node
    if (start->link == NULL)
    {
        free(start);
        return NULL;
    }

    List *current = start;

    while (current->link->link != NULL)
    {
        current = current->link;
    }
    // Freeing last node my friend - prevent memory leaks
    // always check for memory leaks
    free(current->link);
    current->link = NULL;

    return start;
}
```

Output:

```
PS B:\sem3\ds\23bcp153_dsa\lab7> gcc deletelist.c mylistlib.c -o deletelist
PS B:\sem3\ds\23bcp153_dsa\lab7> ./deletelist
30->20->12->10->45->72->NULL
20->12->10->45->72->NULL
20->12->10->72->NULL
20->12->10->NULL
20->12->NULL
```

Theory (Merging Two Sorted Lists):

The function mergeSortedlist takes two sorted linked lists (A and B) and merges them into a single sorted linked list. It compares the nodes of both lists, attaches the smaller node to the merged list, and continues until all nodes from both lists are merged. A dummy node is used to simplify the merging process.

Steps:

1. If either list is empty, the other list is returned.
2. A dummy node is used to build the merged list.
3. The nodes from A and B are compared and linked accordingly.
4. Any remaining nodes from A or B are appended after one list is exhausted.

Time Complexity:

Merging two sorted lists: $O(n + m)$, where n is the number of nodes in list A and m is the number of nodes in list B.

Program:

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

#include "mylistlib.h"

List *mergeSortedlist(List *A, List *B);

int main(void)
{
   List *A = NULL;

   List *B = NULL;

   A = insertend(A, 5);
   A = insertend(A, 10);
   A = insertend(A, 15);

   B = insertend(B, 2);
   B = insertend(B, 3);
   B = insertend(B, 20);
```

```
    printlist(A);
    printlist(B);

    List *merged = NULL;

    merged = mergeSortedlist(A, B);

    printlist(merged);

    freelist(merged);
    freelist(A);
    freelist(B);

    return 0;
}

List *mergeSortedlist(List *A, List *B)
{
    if (A == NULL)
    {
        return B;
    }

    if (B == NULL)
    {
        return A;
    }

    List *dummy = (List *)malloc(sizeof(List));
    if (dummy == NULL)
    {
        printf("Memory allocation failed\n");
        return NULL;
    }

    dummy->link = NULL;
    List *current = dummy;

    while (A != NULL && B != NULL)
    {
        if (A->info <= B->info)
        {
            current->link = A;
            A = A->link;
        }
        else
        {
            current->link = B;
            B = B->link;
        }
```

```
        current = current->link;
    }

    if (A != NULL)
    {
        current->link = A;
    }

    if (B != NULL)
    {
        current->link = B;
    }

    List *merged = dummy->link;
    free(dummy);

    return merged;
}
```

Output:

```
PS B:\sem3\ds\23bcp153_dsa\lab7> gcc mergelist.c mylistlib.c -o mergelist
PS B:\sem3\ds\23bcp153_dsa\lab7> ./mergelist
5->10->15->NULL
2->3->20->NULL
2->3->5->10->15->20->NULL
```

Theory (Circular Linked List):

In a **circular linked list**, the last node points back to the first node, forming a loop. Unlike a standard linked list, it doesn't end with NULL.

1. **Insertion at Beginning** (insertbegcircthis): A new node is inserted at the beginning, and the last node's link is updated to point to the new start.
2. **Insertion at End** (insertendcircthis): A new node is inserted at the end, and its link is set to point to the head of the list, maintaining the circular nature.
3. **Printing the List** (printlistcircthis): It traverses the list and prints until it circles back to the first node.
4. **Freeing the List** (freelistcircthis): It frees all nodes while maintaining the circular structure until all nodes are removed.

Time Complexity:

- **Insertion at Beginning**: $O(n)$ due to the traversal required to update the last node's link.
- **Insertion at End**: $O(n)$ as it requires traversal to the last node.
- **Printing**: $O(n)$, where n is the number of nodes in the circular list.
- **Freeing**: $O(n)$, as all nodes must be visited and freed.

Program:

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

#include "mylistlib.h"

List *insertbegcircthis(List *start, int info);
List *insertendcircthis(List *start, int info);
void printlistcircthis(List *start);
void freelistcircthis(List *start);

int main(void)
{
   List *start = NULL;

   start = insertbegcircthis(start, 10);
   printlistcircthis(start);
   start = insertendcircthis(start, 30);
   printlistcircthis(start);
   start = insertendcircthis(start, 20);
   printlistcircthis(start);
   start = insertbegcircthis(start, 45);
   printlistcircthis(start);

   freelistcircthis(start);

   return 0;
}

List *insertbegcircthis(List *start, int info)
{
   List *node = (List *)malloc(sizeof(List));
   if (node == NULL)
   {
      printf("Unable to allocate memory for new node\n");
      return start;
   }

   node->info = info;

   if (start == NULL)
   {
      node->link = node;
      return node;
   }

   List *current = start;
   while (current->link != start)
   {
      current = current->link;
   }
```

```
   current->link = node;
   node->link = start;

   return node;
}

List *insertendcircthis(List *start, int info)
{
   List *node = (List *)malloc(sizeof(List));
   if (node == NULL)
   {
      printf("Unable to allocate memory for new node\n");
      return start;
   }

   node->info = info;

   if (start == NULL)
   {
      node->link = node;
      return node;
   }

   List *current = start;
   while (current->link != start)
   {
      current = current->link;
   }

   current->link = node;
   node->link = start;

   return start;

}

void printlistcircthis(List *start)
{
   if (start == NULL)
   {
      printf("The list is empty.\n");
      return;
   }

   List *current = start;

   do
   {
      printf("%d->", current->info);
      current = current->link;
```

```
    }
    while (current != start);

    printf("(back to %d)\n", start->info);
}

void freelistcircthis(List *start)
{
    if (start == NULL)
    {
        return;
    }

    List *current = start;
    List *node;

    do
    {
        node = current->link;
        printf("Removing %d from memory\t", current->info);
        free(current);
        current = node;
    }
    while (current != start);

    return;
}
```

Output:

```
PS B:\sem3\ds\23bcp153_dsa\lab7> gcc circll.c mylistlib.c -o circll
PS B:\sem3\ds\23bcp153_dsa\lab7> ./circll
10->(back to 10)
10->30->(back to 10)
10->30->20->(back to 10)
45->10->30->20->(back to 45)
```

Theory (Doubly Linked List):

A doubly linked list allows traversal in both directions using two pointers (prev and next). Insertion at the end involves traversing to the last node and linking the new node, while deletion from the beginning removes the first node and adjusts the pointers of the new start node.

Time Complexity:

- Insertion at the end: $\boldsymbol{O(n)}$ (for traversing to the last node).
- Deletion from the beginning: $\boldsymbol{O(1)}$ (adjusting pointers).
- Traversal/printing: $\boldsymbol{O(n)}$.

Program:

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

typedef struct DoubListthis {
   int info;
   struct DoubListthis *prev;
   struct DoubListthis *next;
} DoubListthis;

DoubListthis *insertendDoubthis(DoubListthis *start, int info);
DoubListthis *deletebegDoubthis(DoubListthis *start);
void printlistDoubthis(DoubListthis *start);

int main(void)
{
   DoubListthis *start = NULL;

   start = insertendDoubthis(start, 45);
   printlistDoubthis(start);
   start = insertendDoubthis(start, 63);
   printlistDoubthis(start);
   start = insertendDoubthis(start, 81);
   printlistDoubthis(start);
   start = insertendDoubthis(start, 90);
   printlistDoubthis(start);
   start = deletebegDoubthis(start);
   printlistDoubthis(start);
   start = deletebegDoubthis(start);
   printlistDoubthis(start);
   start = deletebegDoubthis(start);
   printlistDoubthis(start);

   return 0;
}

DoubListthis *insertendDoubthis(DoubListthis *start, int info)
{
   DoubListthis *node = (DoubListthis *)malloc(sizeof(DoubListthis));
   if (node == NULL)
   {
      printf("Unable to allocate memory for new node\n");
      return start;
   }

   node->info = info;
   node->prev = NULL;
   node->next = NULL;

   if (start == NULL)
   {
```

```
    return node;
  }

  DoubListthis *current = start;
  while (current->next != NULL)
  {
    current = current->next;
  }

  current->next = node;
  node->prev = current;

  return start;
}

DoubListthis *deletebegDoubthis(DoubListthis *start)
{
  if (start == NULL)
  {
    printf("List is Empty\n");
    return start;
  }

  DoubListthis *temp = start;

  if (start->next == NULL)
  {
    free(start);
    return NULL;
  }

  start = start->next;
  start->prev = NULL;

  free(temp);

  return start;
}

void printlistDoubthis(DoubListthis *start)
{
  DoubListthis *current = start;

  while (current != NULL)
  {
    printf("%d", current->info);
    current = current->next;
    if (current != NULL)
    {
      printf("<->");
    }
```

```
    }

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

Output:

```
PS B:\sem3\ds\23bcp153_dsa\lab7> gcc doubll.c -o doubll
PS B:\sem3\ds\23bcp153_dsa\lab7> ./doubll
45
45<->63
45<->63<->81
45<->63<->81<->90
63<->81<->90
81<->90
90
```