Bharatiya Vidya Bhavan's
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Autonomous Institute Affiliated to University of Mumbai)
Munshi Nagar, Andheri (W), Mumbai – 400 058.
COMPS Department

Experiment	3
Aim	**Remove Duplicates from Sorted List: Given a linked list, first sort the link list (ascending/descending), then delete all duplicates such that each element appears only once**
Name	David Daniels
UID	202330038
Class	Div -A
Batch	C
Date of Submission	9-9-24

Theory	A **linked list** is a linear data structure that consists of a series of nodes connected by pointers C Each node contains data and a pointer/reference to the next node in the list. Linked lists can be considered to be upgrade to the array Data structure. **Linear Data Structure:** Data structure where data elements are arranged sequentially or linearly where each and every element is attached to its previous and next adjacent is called a **linear data structure**. In linear data structure, single level is involved. Therefore, we can traverse all the elements in single run only. Linear data structures are easy to implement because computer memory is arranged in a linear way. Its examples are array, stack, queue, linked list, etc. **Arrays** store elements in contiguous memory locations, resulting in easily calculable addresses for the elements stored and this allows faster access to an element at a specific index. **Linked lists** are less rigid in their storage structure and elements are usually not stored in contiguous locations, hence they need to be stored with additional tags giving a reference to the next element. **Linked List** is a linear data structure, in which elements are not stored at a contiguous location, rather they are linked using pointers. Linked List forms a series of connected nodes, where each node stores the data and the address of the next node. There are many types of linked lists such as • Singly Linked lists • Doubly Linked lists • Circular linked lists • Doubly Circular linked lists

but we shall only focus on singly linked lists, which only have a connection to the next element in the list.

Node Structure: A node in a linked list typically consists of two components:

1. **Data:** It holds the actual value or data associated with the node.
2. **Next Pointer:** It stores the memory address of the next node in the sequence.

   We wrap both the data item and the next node reference in a struct as:

```
struct node
{
  int data;
  struct node *next;
};
```

Head: The linked list is accessed through the head node, which points to the first node in the list. The Header node does not have any data. The last node in the list points to NULL indicating the end of the list. This node is known as the tail node. All nodes are in some way connected with head node, thus we don't have to pass every single node in any function on linked list

A linked list is an ADT as we can implement many operations in it to insert elements at any location or delete at any location or simply traverse the list or sort an array.

1. Insertion: This to add an element in the linked list. The implementation of this function will be different depending on the location. Insertion can be done on first , last and middle node
2. Deletion: This to remove an element in the linked list. The implementation of this function will be different depending on the location. Deletion can be done on first , last and middle node
3. Sorting: This is to sort the linked list in ascending or descending order. We can use bubble sort to do so.
4. Traversal: This is to pass through all elements of list , doing some operation on all of them

There are Many advantages of Using a linked list over an Array:

1. **Dynamic Data structure:** The size of memory can be allocated or de-allocated at run time based on the operation insertion or deletion. Otherwise in Arrays it is always fixed
2. **Ease of Insertion/Deletion:** The insertion and deletion of elements are simpler than arrays since no elements need to be shifted after insertion and deletion, Just the address needed to be updated.

	3. **Efficient Memory Utilization:** As we know Linked List is a dynamic data structure the size increases or decreases as per the requirement so this avoids the wastage of memory. There is One major Disadvantage of using a linked list as compared to arrays. 1. That is we cannot randomly access index in the list. We must traverse every single previous node from head to get there. In an array, all elements are continuous in memory, so we can just jump to memory address (initial + size_of_datatype *index). 2. Also because we have to make a new pointer in every node, it is not very space efficient. **Applications of Linked Lists:** • Linked Lists can be used to implement stacks, queue, deque, sparse matrices and adjacency list representation of graphs. • Dynamic memory allocation in operating systems and compilers (linked list of free blocks). • Manipulation of polynomials (addition , multiplication,etc) • Arithmetic operations on long integers. • In operating systems, they can be used in Memory management, process scheduling (for example circular linked list for round robin scheduling) and file system. • Algorithms that need to frequently insert or delete items from large collections of data. • LRU cache, which uses a doubly linked list to keep track of the most recently used items in a cache.
Algorithm	1. Insertion: 1) For Insertion at the front of linked list • Make the first node of Linked List linked to the new node • Remove the head from the original first node of Linked List • Make the new node as the Head of the Linked List. 2) For insertion at the end of linked list • Go to the last node of the Linked List • Change the next pointer of last node from NULL to the new node • Make the next pointer of new node as NULL to show the end of Linked List 3) For insertion at middle of linked list at specific position • Traverse the Linked list upto position-1 node. • Once all the position-1 nodes are traversed, allocate memory and the given data to the new node. • Point the next pointer of the new node to the next of current node. • Point the next pointer of current node to the new node.

2. Deleting Duplicates

- Make a temp pointer point to next element of head
- If the data of temp and data of temp's next node is the same It means that we have reached a duplicate
- Point the next pointer of temp to next of temp's next keeping in mind to free the duplicated node
- Keep on doing this till we reach end of the list

3. Deletion of a node

1) Deletion at the front

- Keep a temp pointer to point to head->next
- Make head = head->next->next to skip the element to be deleted
- Free temp

2) Deletion at the end

- Create a pointer temp1 = head and temp2 = temp1->next
- While temp 2 is not null temp 1 and temp 2 =ptr->next
- This ensures temp 2 is end and temp 1 is second last
- Free temp 2 and make temp1->next=Null

3) Deletion at the middle at position

- Create temp pointer = head
- Make a for loop till index – 1 to ensure that we reach to node just before the node we must delete , making temp=temp->next inside the loop
- Create temp 2 pointer = temp-> next to store element to be deleted
- Temp = temp->next->next to create linking
- Free temp2

4. Traversal

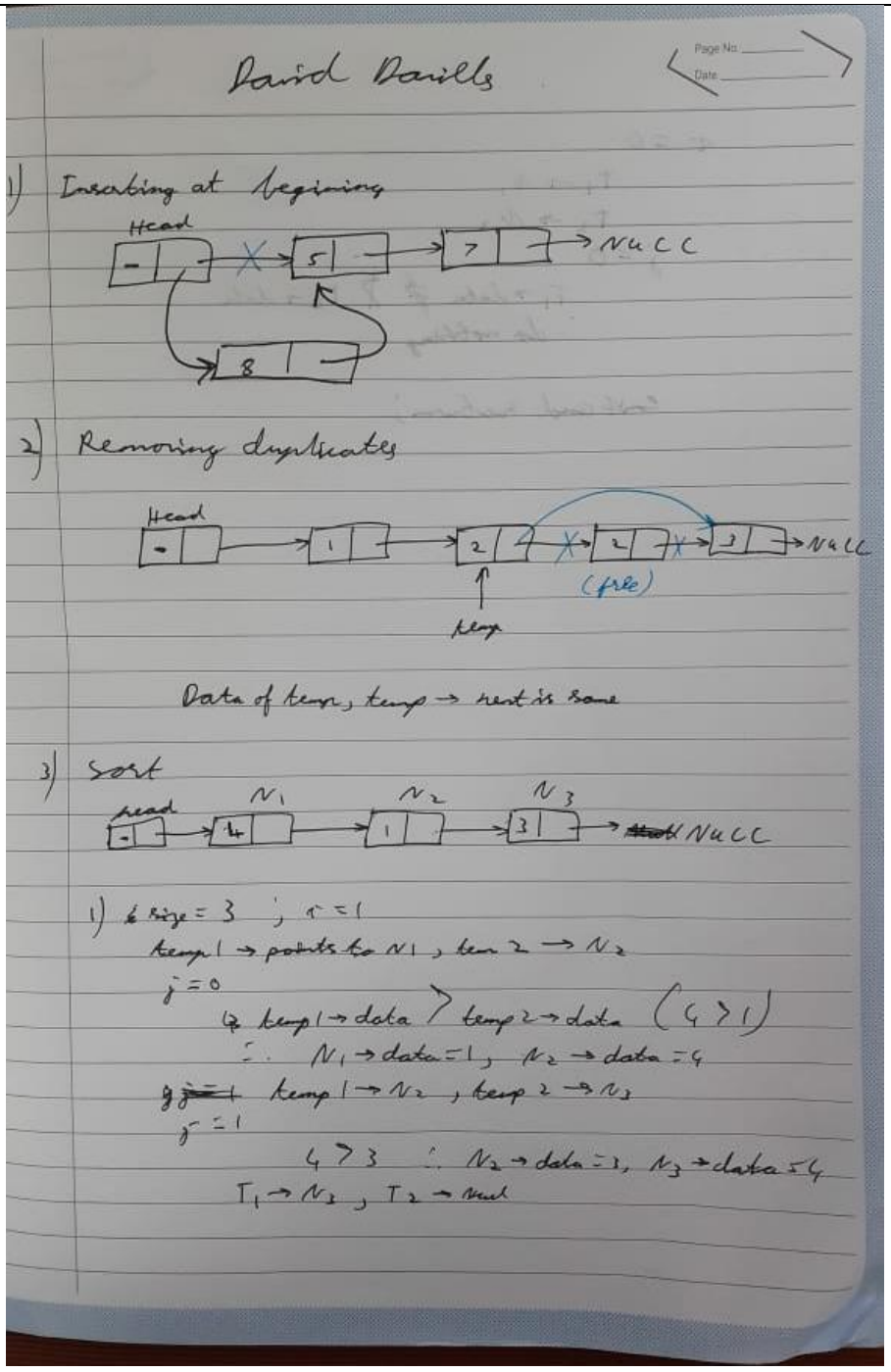
- Make a temp pointer point to next element of head
- While The next element of temp is not Null
- Print the data of the node and advance to the next node

5. Sorting

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high. In this algorithim , every element is compared with every other element and the largest elements bubble to the top of list.

- Take the size of linked list by traversing the list and incrementing count by 1
- Run the outer loop from size -2 to 0 to select node one by one

	• Inside the outer loop initialise temp1 to head->next and temp 2 to the next of temp1 • Now run an inner loop from 0 to current pass index of outer loop to check every node with every other node • Inside the loop compare temp 1 with temp 2 (value ahead of temp 1) • If they are greater (or lesser) for ascending/descending sorting Then swap the values of the nodes • Inside the Inner loop only advance temp 1 and temp 2 to compare with the rest of the list • Finally return the Head

Problem Solving	

David Davills

Page No.
Date

1) Inserting at beginning

2) Removing duplicates

Data of temp, temp → next is same

3) Sort

i) & size = 3 ; i = 1

temp1 → points to N1 , temp 2 → N2

j = 0

& temp1 → data > temp2 → data (4 > 1)

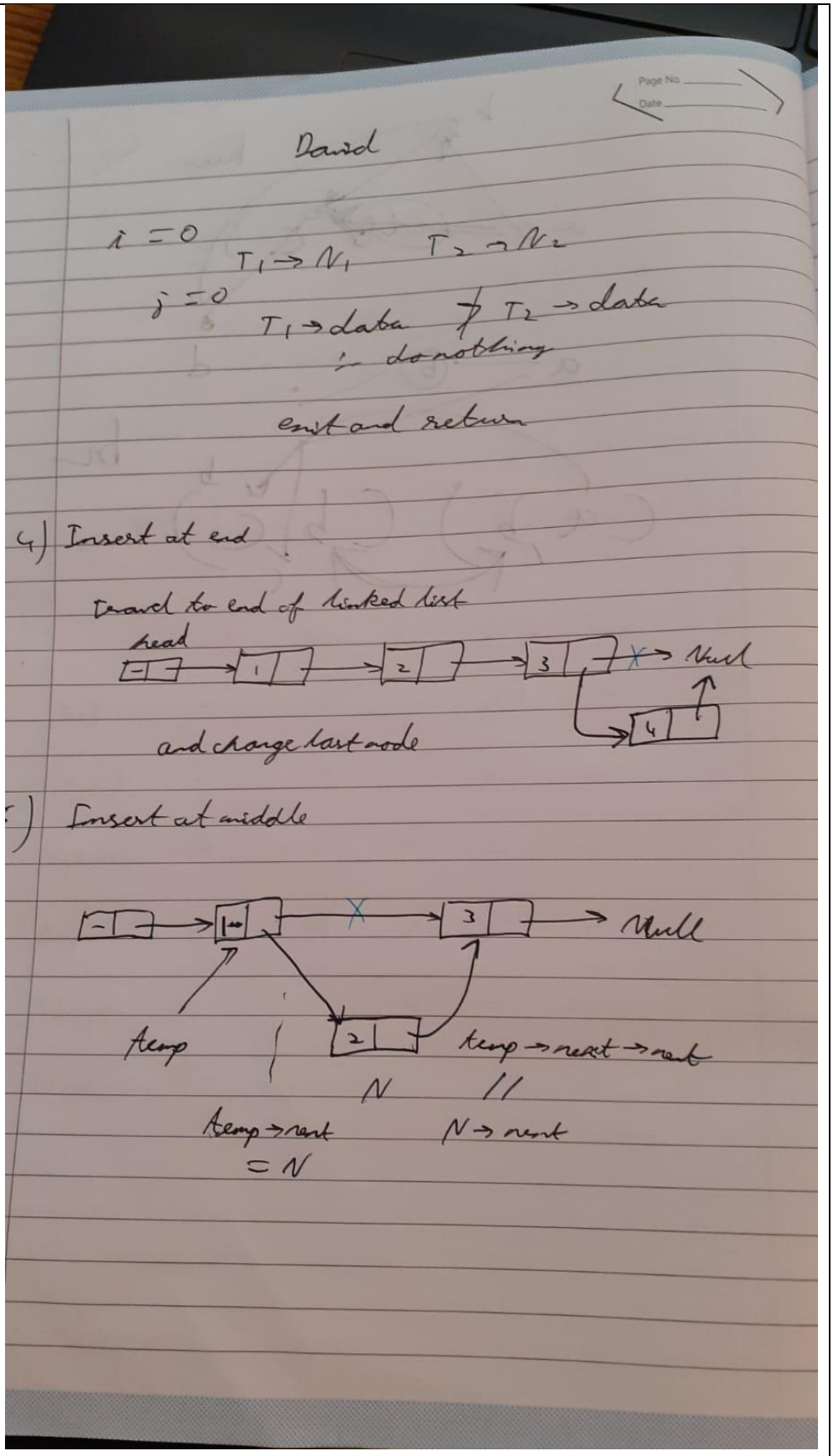
∴ N1 → data = 1, N2 → data = 4

~~j = 1~~ temp1 → N2 , temp 2 → N3

j = 1

4 > 3 ∴ N2 → data = 3, N3 → data = 4

T1 → N2 , T2 → Null



Program(Code)	#include <stdio.h> #include <stdlib.h>

```
#include <stdbool.h>

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

void display(struct node * head)
{
   printf("\n\n");
   struct node * temp= head->next;

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

int tell_size(struct node * head)
{
   int size=0;
   struct node * temp= head->next;

   while (temp!=NULL)
   {
      size++;
      temp=temp->next;
   }
   return size;
}

struct node * insert_at_start(struct node * head,int d)
{
   struct node * new = (struct node * ) malloc(sizeof(struct node));
   new->next=head->next;
   head->next=new;
   new->data=d;
   return head;
}

struct node* insert_at_index(struct node * head, int d, int index)
{
   struct node * ptr_insert = malloc(sizeof(struct node));
   ptr_insert->data=d;

   struct node * ptr =head;
   //go to just before the index where we want to place
   for (int i = 0; i < index-1; i++)
```

```
    {
        ptr= ptr->next;
    }
    ptr_insert->next=ptr->next;
    ptr->next=ptr_insert;
    return head;

}

struct node* insert_at_end(struct node * head, int d)
{
    struct node * new_end = malloc(sizeof(struct node));
    struct node * ptr =head;
    new_end->data=d;

    while(ptr->next!= NULL)
    {
        ptr = ptr->next;
    }
    ptr->next=new_end;
    new_end->next=NULL;
    return head;
}

struct node * sort(struct node * head)
{
    int size=tell_size(head);
    struct node* temp1=NULL;
    struct node* temp2=NULL;
    int k=0;
    for (int i = size - 2; i >= 0; i--) {
        temp1 = head->next;
        temp2 = temp1->next;
        for (int j = 0; j <= i; j++) {
            if (temp1->data > temp2->data) {
                k = temp1->data;
                temp1->data = temp2->data;
                temp2->data = k;
            }
            temp1 = temp2;
            temp2 = temp2->next;
        }
    }
    return head;
}



struct node * remov_dupli(struct node * head)
{
```

```
    struct node * temp1=head->next;
    while (temp1->next!=NULL)
    {
     if (temp1->data==temp1->next->data)
     {
        struct node * hold_temp=temp1->next;
        temp1->next=temp1->next->next;
        free(hold_temp);
     }
     temp1=temp1->next;
    }
    return head;
}

void choice(struct node * head)
{
    int choice=0;
    printf("Welcome to choice \n Enter 1 to print \n enter 2 to insert
at start \n enter 3 to sort \n enter 4 to remove duplicates \n enter 5 to insert
at position \n enter 6 to insert at end\n enter 0 to end\n");
    int num=0,idx=0;
    while (true)
    {
       printf("Enter choice \n");
       scanf("%d",&choice);
       if (choice==0)
       {
          printf("Goodbye");
          break;
       }

       switch (choice)
       {
       case 1:
          display(head);
          break;
       case 2:
          printf("Enter number to insert at start ");
          scanf("%d",&num);
          insert_at_start(head,num);
          break;
       case 3:
          sort(head);
          break;
       case 4:
          remov_dupli(head);
          break;
       case 5:
          printf("Enter number to insert at middle");
          scanf("%d",&num);
```

```
                printf("Enter Index: \n");
                scanf("%d",&idx);
                insert_at_index(head ,num,idx);
                break;
            case 6:
                printf("Enter number to insert at end ");
                scanf("%d",&num);
                insert_at_end(head,num);
                break;
            default:
                printf("Error");
                break;
            }
        }


    }

    int main(int argc, char const *argv[])
    {
        struct node * head = (struct node * ) malloc(sizeof(struct node));
        struct node * first = (struct node * ) malloc(sizeof(struct node));
        head->next=first;
        int no1=0;
        printf("Enter first element (manditory): \n");
        scanf("%d",&no1);
        first->data=no1;
        first->next=NULL;
        choice(head);
        return 0;
    }
```

Output	

```
Enter first element (manditory):
5
Welcome to choice
 Enter 1 to print
 enter 2 to insert at start
 enter 3 to sort
 enter 4 to remove duplicates
 enter 5 to insert at position
 enter 6 to insert at end
 enter 0 to end
Enter choice
2
Enter number to insert at start 6
Enter choice
6
Enter number to insert at end 2
Enter choice
6
Enter number to insert at end 95
Enter choice
5
Enter number to insert at middle 12
Enter Index:
3
```

```
Enter choice
1


6
5
12
2
95
Enter choice
2
Enter number to insert at start 12
Enter choice
2
Enter number to insert at start 2
Enter choice
1


2
12
6
5
12
2
95
Enter choice
3
```

```
Enter choice
3
Enter choice
1


2
2
5
6
12
12
95
Enter choice
4
Enter choice
1


2
5
6
12
95
Enter choice
0
Goodbye
```

Conclusion	Thus I have learned how to implement linked lists in C along with inserting elements at the front and middle and end. I have also learned deletion algorithm in a singly linked list I have also learned how to sort a linked list using bubble sort and remove duplicates from a linked list