Chapter Table of Contents

Chapter 4.2

Operations on Linked List

Aim....231
Instructional Objectives....231
Learning Outcomes....231
Introduction....232
4.2.1 Operations on Singly List....232
(i) Creating a Linked List....233
(ii) Insertion of a Node in Linked List....234
(iii) Deletion of a Node from the Linked List....238
(iv) Searching and Displaying Elements from the Linked List....242
Self-assessment Questions....245
Summary....246
Terminal Questions....247
Answer Keys....248
Activity....248
Bibliography....249
e-References....249
External Resources....249
Video Links....249

Aim

To educate the students on the importance of using Linked List data structure in computer science

Instructional Objectives

After completing this chapter, you should be able to:

- Demonstrate the traversing in the linked list*
- Illustrate insertion and deletion operations at the end and beginning of linked list
- Demonstrate the approach to search and display a value in the linked list

Learning Outcomes

At the end of this chapter, you are expected to:

- Identify the traversal path of the linked list
- Discuss insertion and deletion operations in singly linked
- Determine the position of the value to be searched in the linked list

Introduction

In the previous chapter, the basic concepts and fundamentals of the linked lists were covered. Linked lists are very robust and dynamic data structures and therefore, different nodes can be added, deleted, or can be updated at miniscule cost. Moreover, while using linked lists, there is no need of large contiguous memory at the compile time; rather it accesses memory at runtime based on requirements. These properties make linked lists a primary choice of programmers for many practical applications.

This chapter focuses on understanding the most important and fundamental operations on linked lists such as creation of linked list, adding or insertion a node, deleting a node from the list, searching a node element and displaying elements of linked lists.

4.2.1 Operations on Singly List

Deleting and adding an element in an array requires shifting of array elements to create or fill the holes (empty spaces). Also, updating an array element only requires accessing that particular value at index and simply overwriting or replacing the value. Moreover, one of the distinguishing features of the arrays is its random access which enables it to access any index value directly i.e. given an access to particular index; we can easily find the value at that index.

In case of linked lists, the entry point constitutes head of the linked lists. Head or start of the list is actually not the node, but a reference to the first node in the linked list. That means, a head constitutes a value. For an empty linked list, the value of head is null. It is also known that linked list always ends with a null pointer (except for circular linked list where last node is connected to the start or head node). Great care must be taken while manipulating linked lists, as any wrong link in the middle makes the entire list inaccessible. That is because the only way to traverse a list is by using a reference to the next node from the current node. This concept of linking wrong nodes is called as “memory leaks”. In case of lost memory reference, the entire list from that point becomes inaccessible.

Some of the basic operations on linked lists discussed in this chapter are as follows:

- Creation of linked lists
- Insertion of node from the linked lists
- Deletion of node from the linked lists

- Searching an elements from the lists
- Displaying all linked lists elements

Did you know?

Many programming languages such as Lisp and Scheme have singly linked lists built in. In many functional languages, these lists are constructed from nodes, each called a cons or cons cell. The cons has two fields: the car, a reference to the data for that node, and the cdr, a reference to the next node. Although cons cells can be used to build other data structures, this is their primary purpose.

In languages that support abstract data types or templates, linked list ADTs or templates are available for building linked lists. In other languages, linked lists are typically built using references together with records.

(i) Creating a Linked List

The most simple of the operations of linked lists is creating a linked list. It includes a simple step of getting a free node and copying a data item into the "data" field of the node. The next step is to update the links of the node *i.e.,* in case of entirely a new list, start and null pointers are supposed to be updated and in case of merging lists, the references must be updated accordingly.

Below is a C programming code segment for declaring a linked list.

```
struct node
{
      int data;
      struct node *next;
}*start=NULL;
```

Below is a C programming function for creating a new node

```
void create()
{
char c;
 do
 {
  struct node *new_node,*current;
  new_node=(struct node *)malloc(sizeof(struct node));
```

```
  printf("nEnter the data : ");
  scanf("%d",&new_node->data);
  new_node->next=NULL;
  if(start==NULL)
  {
     start=new_node;
     current=new_node;
  }
  else
  {
     current->next=new_node;
     current=new_node;
  }
 printf("nDo you want to create another node : ");
 c=getch();
 }while(c!='n');
}
```

(ii) Insertion of a Node in Linked List

Insertion is an important operation while working with the data structures. This operation involves adding data to the data structure. For carrying out the operation there can be notably three different cases mentioned as below:

- Insertion in front of the list
- Insertion at any given location within the list
- Insertion at the end of the list.

The general procedure for inserting a node in the linked list is detailed below:

Step 1: Input the value of the new node and the position where it is supposed to be inserted.

Step 2: Check if the linked list is full. If YES then display an error message "Overflow". Else continue.

Step 3: Create a new node and Insert the data value in the data field of new node.

Step 4: Add the new node to the desired location in the linked list.

Below is C programming function for inserting a node in the linked list.

```
void insert(node *ptr, int data)
{
        /* Iterate through the list till we encounter the last node.*/
```

```
        while(ptr->next!=NULL)
        {
                ptr = ptr -> next;
        }
        /* Allocate memory for the new node and put data in it.*/
        ptr->next = (node *)malloc(sizeof(node));
        ptr = ptr->next;
        ptr->data = data;
        ptr->next = NULL;
}
```

Insertion of node at the starting on the linked list

Figure 4.2.1 below shows the case for insertion of node in starting of the linked list.

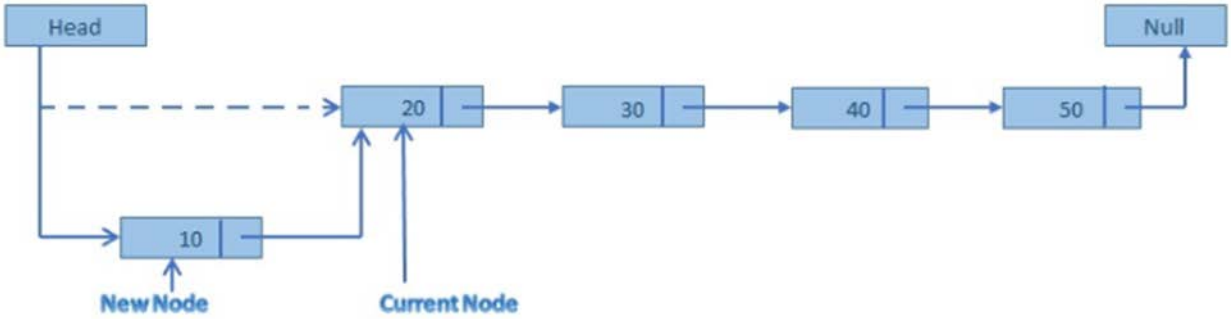

Figure 4.2.1: Insertion of Node in Starting of the Linked List

Algorithm below shows insertion of node at the starting of the linked list.

Step 1. Create a new node and assign the address to any node say ptr.

Step 2. OVERFLOW,IF(PTR = NULL)

write : OVERFLOW and EXIT.

Step 3. ASSIGN INFO[PTR] = ITEM

Step 4.

```
IF(START = NULL)
      ASSIGN NEXT[PTR] = NULL
       ELSE
      ASSIGN NEXT[PTR] = START
```

Step 5. ASSIGN START = PTR

Step 6. EXIT

Following is the C programming function for implementing above algorithm.

```
void insertion(struct node *nw)
{
      struct node start, *previous, *new1;
      nw = start.next;
      previous = &start;
      new1 = (struct node* ) malloc(sizeof(struct node));
      new1->next = nw ;
      previous->next = new1;
      printf("\n Input the fisrt node value: ");
      scanf("%d", &new1->data);
}
```

Insertion of node at any given position in the linked list

Figure 4.2.2 below shows the case for insertion of new node at any given location.

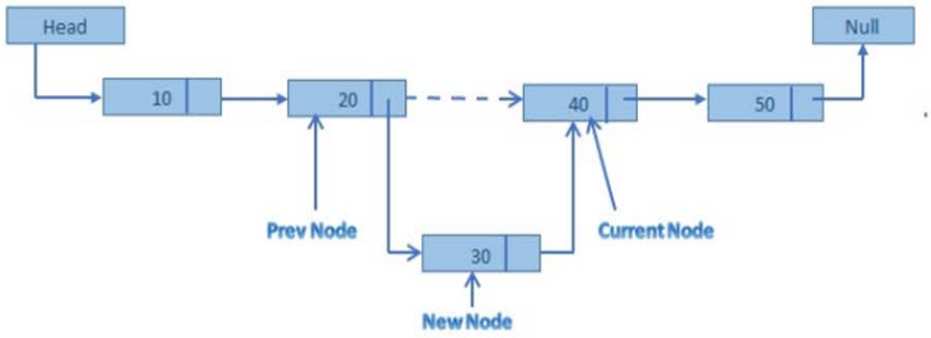

Figure 4.2.2: Insertion of Node at any Desired Location

The following algorithm shows insertion of node at any given location within the list.

```
InsertAtlocDll(info,next,start,end,loc,size)
1.set nloc = loc-1 , n=1
2.create a new node and address in assigned to ptr.
3.check[overflow] if(ptr=NULL)
      write:overflow and exit
4.set Info[ptr]=item;
5.if(start=NULL)
      set next[ptr] = NULL
      set start = ptr
  else if(nloc<=size)
      repeat steps a and b while(n != nloc)
            a.    loc = next[loc]
            b.    n = n+1
```

```
            [end while]
            next[ptr] = next[loc]
            next[loc] = ptr
   else
            set last = start;
            repeat step (a) while(next[last]!= NULL)
            a. last=next[last]
            [end while]
            last->next = ptr ;
      [end if]
6.Exit.
```

The below is the C programming function for implementing the above algorithm.

```
void insertAtloc(node **start,int item , int i,int k )
{
      node *ptr,*loc,*last;
      int n=1 ;
      i=i-1;
      ptr=(node*)malloc(sizeof(node));
      ptr->info=item;
      loc = *start ;
      if(*start==NULL)
      {
            ptr->next = NULL ;
      *start = ptr ;
      }
      else if(i<=k)
      {   while(n != i)
            {
                  loc=loc->next;
                  n++;
            }
            ptr->next = loc->next ;
            loc->next = ptr ;
      }
      else
      {
            last = *start;
            while(last->next != NULL)
            {last=last->next;
            }
            last->next = ptr ;
      }
}
```

Insertion of node at end of the linked list

Figure 4.2.3 below shows the case for insertion of node at end of the list.

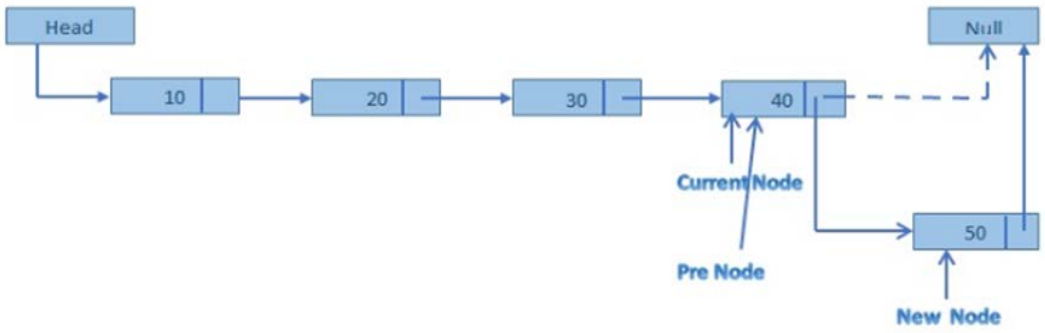

Figure 4.2.3: Insertion of Node at the End of the List

(iii) Deletion of a Node from the Linked List

The deletion of a node in linked is similar to insertion. Again in deletion operation there are three cases as listed below:

- Deletion of the front node
- Deletion of any intermediate node
- Deletion of the last node.

The general algorithm for deletion of any given node is given below:

Step 1: Search for the appropriate node to be deleted

Step 2: Remove the node

Step 3: Reconnect the linked list

Step 4: Update all the links.

Deleting the front node

Figure 4.2.4 shows deletion of node from the front of the linked list.

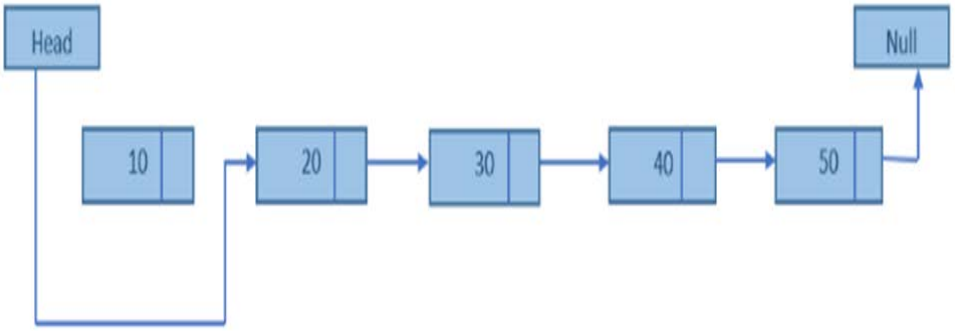

Figure: 4.2.4: Deleting front Node

Following algorithm shows deletion of the front node from the linked list.

DELETE AT BEG(INFO,NEXT,START)

```
1.IF(START=NULL)
2.ASSIGN PTR = STRAT
3.ASSIGN TEMP = INFO[PTR]
4.ASSIGN START = NEXT[PTR]
5.FREE(PTR)
6.RETURN(TEMP)
```

Below is the C programming function for implementing above algorithm.

```
void deleteatbeg(node **start)
{
      node *ptr;
      int temp;
      ptr = *start ;
      temp = ptr->info;
      *start = ptr->next ;
      free(ptr);
      printf("\nDeleted item is %d : \n",temp);
}
```

Deletion of Intermediate node

Figure 4.2.5 shows deletion of intermediate node from the linked list.

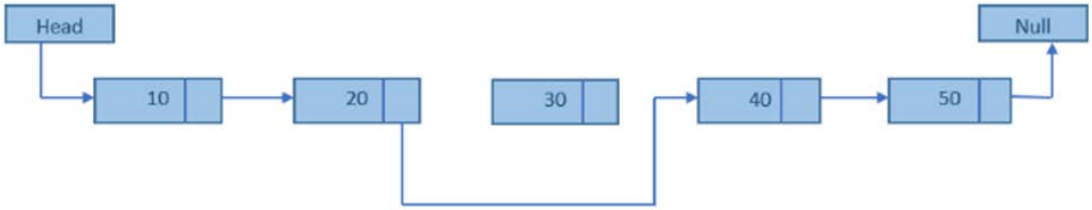

Figure 4.2.5: Deletion of Intermediate Node from the Linked List

Following is the algorithm for deleting an intermediate node from the linked list.

Step 1: Traverse to the node to be deleted in the linked list.

Step 2: Examine the next node by using the node pointers to move from node to node until the correct node is identified.

Step 3: Copy the pointer of the removed node into temporary memory.

Step 4: Remove the node from the list and mark the memory it was using as free once more.

Step 5: Update the previous node's pointer with the address held in temporary memory.

Below is the C programming function for implementation of above algorithm.

```
//Function to delete any node from linked list.
void delete_any()
{
      int key;

      if(header->link == NULL)
      {
            printf("\nEmpty Linked List. Deletion not possible.\n");
      }
      else
      {
            printf("\nEnter the data of the node to be deleted: ");
            scanf("%d", &key);

            ptr = header;
            while((ptr->link != NULL) && (ptr->data != key))
            {
```

```
                ptr1 = ptr;
                ptr = ptr->link;
            }
            if(ptr->data == key)
            {
                ptr1->link = ptr->link;
                free(ptr);
                printf("\nNode with data %d deleted.\n", key);
            }
            else
            {
                printf("\nValue %d not found. Deletion not possible.\n",
key);
            }
      }
}
```

Deletion of end node from the linked list

Figure 4.2.6 below shows deletion of the end node from the linked list.

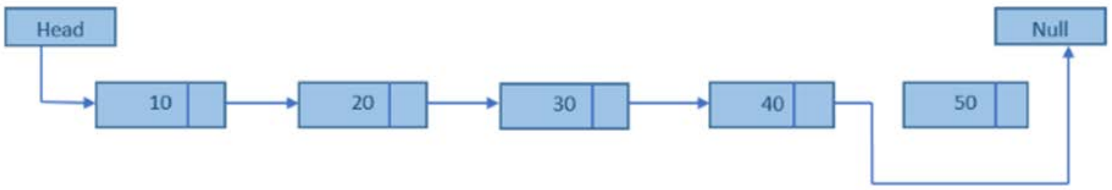

Figure 4.2.6: Deletion of Last Node from the Linked List

Below is the algorithm for deletion of last node from the linked list.

```
Delete End(info,next,start)


1.if(start=NULL)
   Print Underflow and Exit.
2.if(next[start]==NULL)
   Set ptr =start and start=NULL.
set temp = info[ptr].
else cptr = start and ptr = next[start].
Repeat steps(a) and (b) while(next[ptr]!=NULL)
      (a) set cptr = ptr
      (b) set ptr = next[ptr]
```

```
      [end while]
set next[cptr] = NULL
      temp = info[ptr]
   [end if]
3. free(ptr)
4. return temp
5. exit
```

Following is the C programming function foe implementation of above algorithm.

```
void deleteatlast(node **start)
{
      node *ptr,*cptr;
      int temp;

      if((*start)->next == NULL)
      {
            ptr = *start ;
            *start = NULL;
            temp = ptr->info;
      }
      else
      {
            cptr = *start ;
            ptr =(*start)->next;
            while(ptr->next != NULL)
            {
                  cptr = ptr;
                  ptr = ptr->next;
            }
            cptr->next = NULL;
            temp = ptr->info;
      }
      free(ptr);
      printf("\nDeleted item is %d : \n",temp);
}
```

(iv) Searching and Displaying Elements from the Linked List

Searching and displaying elements of the linked list are very similar and simple as both operations involve simple traversal across the linked list.

Searching: For linked list, it is simple linear search which begins from starting of the linked list from start pointer till we find the NULL pointer.

Algorithm for searching an element in the linked is as below.

Step 1: Input the element to be searched KEY.

Step 2: Initiate the current pointer with the beginning of the list.

Step 3: Compare KEY with the value in the data field of current node.

Step 4: If they match then quit.

Step 5: else go to Step 3.

Step 6: Move the current pointer to point to the next node in the list and go to step 3; till the list is not over or else quit.

```
int search(int item)
{
      int count=1;
      nw=&first;
      while(nw->next!=NULL)
      {
            if(nw->data==item)
                  break;
            else
                  count++;
                  nw=nw->next;
      }
      return count;
}
```

The above code fragment explains how searching of an element takes place in a single linked list.

Searching in a single linked list follows linear search algorithm. Here, based on the search key which is given by the user, the pointer will point to the first node of the list. It will then compare the search key with the data found at every node. If the match is found then the search is complete. If not found, then the pointer moves forward in a linked list and the process continues till the last node of the list.

Did you know?

Finding a specific element in a linked list, even if it is sorted, normally requires O (n) time. This is one of the primary disadvantages of linked lists over other data structures. One of the better way to improve search time is move-to-front heuristic, which simply moves an element to the beginning of the list once it is found. This scheme, handy for creating simple caches, ensures that the most recently used items are also the quickest to find again.

Another common approach is to "index" a linked list using a more efficient external data structure. ***For example,*** one can build a red-black tree or hash table whose elements are references to the linked list nodes. Multiple such indexes can be built on a single list. The disadvantage is that these indexes may need to be updated each time a node is added or removed (or at least, before that index is used again).

Displaying the linked list content

Similar to search function, display function begins by traversing from the starting of the list to display the value in the data field of each and every node till it reaches the last node.

Below is the algorithm for displaying the content of the linked list.

Step 1: Initiate the current pointer from the beginning of the list.

Step 2: Display the value in the data field of current pointer.

Step 3: Increment the current pointer to pint to the next node.

Step 4: Repeat step 3 until the end of the list.

Following is the algorithm for implementation of the above algorithm.

```
void display()
{
    struct node *temp;
    temp=start;
    while(temp!=NULL)
    {
       printf("%d",temp->data);
       temp=temp->next;
    }
}
```

Self-assessment Questions

1) While inserting a node at the first position in the linked list the link part of the node to be inserted should point to _____.

a) Any random node in the list
b) NULL pointer
c) Start pointer
d) Midpoint of all the nodes

2) Trying to insert a node where the linked list has reached its maximum capacity is called _____

a) Underflow
b) Overflow
c) Mid flow
d) Overcover

3) The concept of linking wrong nodes is called as _____

a) Memory leaks
b) Memory holes
c) Memory pits
d) Memory wastage

4) Trying to delete a node from an empty linked list is called _____

a) Underflow
b) Overflow
c) Mid flow
d) Overcover

Summary

- A linked list is a collection of nodes which consists of two fields namely data and link. The data field contains the information or the data to be stored by the node. The link field contains the addresses of the next node.
- The basic operations of linked list are creation of linked lists, insertion of node into the linked lists, deletion of node from the linked lists, searching an element from the lists and displaying all linked lists elements.
- Insertion operation involves addition of a node to a linked which happens either at the start of the linked list or at any given position in the linked list or at the end of the linked list.
- Similarly, deletion operation involves deletion of a node from a linked which happens either at the start of the linked list or at any given position in the linked list or at the end of the linked list.
- Searching in a single linked list is based on a simple linear search technique which traverses from the start till the end of the list until the desired element is found.
- Display operation is responsible for traversing the whole list and then displaying the data component of the list in a linear fashion.

Terminal Questions

1. Write a C program for inserting node in the linked list
2. Write and explain the algorithm for deleting a node in the linked list
3. Write a C program for searching a node in the linked list.
4. Explain the algorithm for displaying all the nodes in the linked list.

Answer Keys

Self-assessment Questions	
Question No.	Answer
1	c
2	b
3	a
4	a

Activity

Activity Type: Offline **Duration:** 20 Minutes

Description:

Write a function to sort an existing linked list of integers using insertion sort.

Bibliography

e-Reference

- cs.cmu.edu, (2016). *Lecture 10 Linked List Operations Concept of a Linked List Revisited*. Retrieved on 19 April 2016, from http://www.cs.cmu.edu/~ab/15-123S09/lectures/Lecture%2010%20-%20%20Linked%20List%20Operations.pdf

External Resources

- Kruse, R. (2006). *Data Structures and program designing using 'C'* (2nd ed.). Pearson Education.
- Srivastava, S. K., & Srivastava, D. (2004). *Data Structures Through C in Depth* (2nd ed.). BPB Publications.
- Weiss, M. A. (2001). *Data Structures and Algorithm Analysis in C* (2nd ed.). Pearson Education.

Video Links

Topic	Link
Operations on singly linked list	https://www.youtube.com/watch?v=McgL6JuWUpM
Singly Linked List-Deletion of Last Node	https://www.youtube.com/watch?v=Hn8Hs9sVSCM
Linked Lists in 10 minutes	https://www.youtube.com/watch?v=LOHBGyK3Hbs

Notes: