Data Structures

For BCA 2nd Semester

Lecture 1

[Linked List, Insertion and Deletion Algorithms for Singly]

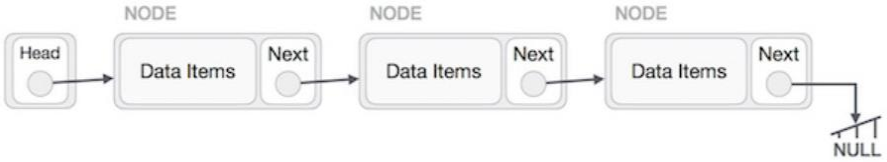
Compiled
By
Sakhi Bandyopadhyay
Dept. of Computer Science & BCA,
Kharagpur College,
Kharagpur 721305

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.

Link – Each link of a linked list can store a data called an element.
Next – Each link of a linked list contains a link to the next link called Next.
LinkedList – A Linked List contains the connection link to the first link called First.



Advantages of Linked Lists:

1) They are a dynamic in nature which allocates the memory when required.
2) Insertion and deletion operations can be easily implemented.
3) Stacks and queues can be easily executed.
4) Linked List reduces the access time.

Disadvantages:

1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists efficiently with its default implementation.

2) Extra memory space for a pointer is required with each element of the list.

3) Reverse Traversing is difficult in linked list.

Why use linked list over array?

Array contains following limitations:

- The size of array must be known in advance before using it in the program.
- Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
- All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all its predecessors.

- Linked list is the data structure which can overcome all the limitations of an array. Using linked list is useful because,
- It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
- Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.

Types of Linked List:

Following are the various types of linked list.

Simple Linked List – Item navigation is forward only.

Doubly Linked List – Items can be navigated forward and backward.

Circular Linked List – Last item contains link of the first element as next and the first element has a link to the last element as previous.

Basic Operations:

Following are the basic operations supported by a list.

Insertion – Adds an element at the beginning of the list.

Deletion – Deletes an element at the beginning of the list.

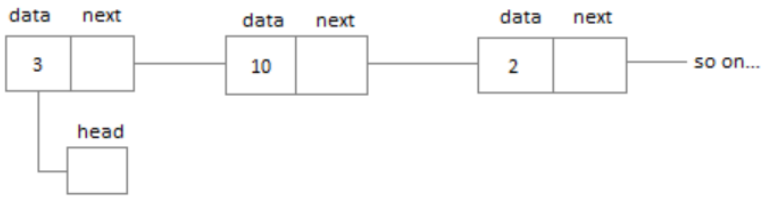
Display – Displays the complete list.

Search – Searches an element using the given key.

Delete – Deletes an element using the given key.

Singly Linked List

Singly linked lists contain nodes which have a **data** part as well as an **address part** i.e. next, which points to the next node in the sequence of nodes.



Node Creation

```
struct node
{
    int data;
    struct node *next;
};
struct node *head, *ptr;
ptr = (struct node *)malloc(sizeof(struct node *));
```

Insertion:

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

Insertion in singly linked list at beginning:

Algorithm:

Step 1: IF PTR = NULL
Write OVERFLOW
Go to Step 7
[END OF IF]

Step 2: SET NEW_NODE = PTR

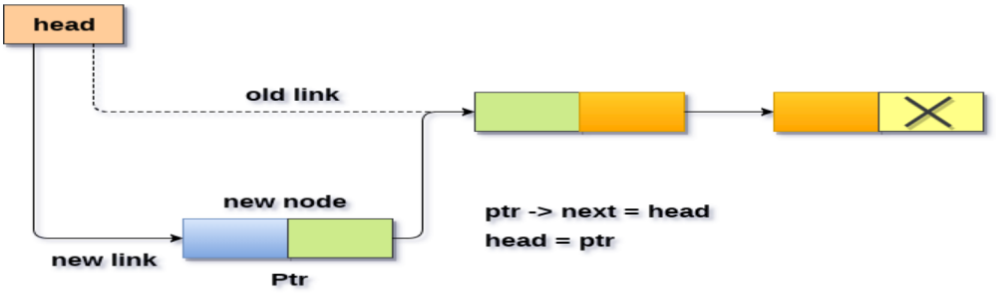
Step 3: SET PTR = PTR → NEXT

Step 4: SET NEW_NODE → DATA = VAL

Step 5: SET NEW_NODE → NEXT = HEAD

Step 6: SET HEAD = NEW_NODE

Step 7: EXIT



C Code:

```
void beginsert(int item)
  {
     struct node *ptr = (struct node *)malloc(sizeof(struct node *));
     if(ptr == NULL)
     {
        printf("\nOVERFLOW\n");
     }
     else
     {
        ptr->data = item;
        ptr->next = head;
        head = ptr;
        printf("\nNode inserted\n");
     }

  }
```

Insertion in singly linked list at the end

In order to insert a node at the last, there are two following scenarios which need to be mentioned.

1. The node is being added to an empty list
2. The node is being added to the end of the linked list

Algorithm:

Step 1: IF PTR = NULL
Write OVERFLOW
Go to Step 1
[END OF IF]

Step 2: SET NEW_NODE = PTR

Step 3: SET PTR = PTR - > NEXT

Step 4: SET NEW_NODE - > DATA = VAL

Step 5: SET NEW_NODE - > NEXT = NULL

Step 6: SET PTR = HEAD

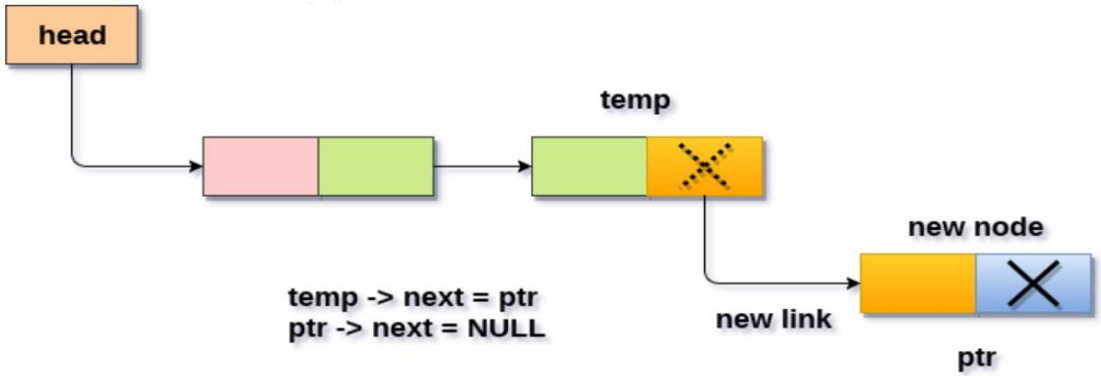
Step 7: Repeat Step 8 while PTR - > NEXT != NULL

Step 8: SET PTR = PTR - > NEXT

[END OF LOOP]

Step 9: SET PTR - > NEXT = NEW_NODE

Step 10: EXIT



C Code:

```
void lastinsert(int item)
  {
    struct node *ptr = (struct node*)malloc(sizeof(struct node));
    struct node *temp;
    if(ptr == NULL)
    {
      printf("\nOVERFLOW");
    }
    else
    {
      ptr->data = item;
      if(head == NULL)
      {
        ptr -> next = NULL;
        head = ptr;
        printf("\nNode inserted");
      }
      else
      {
        temp = head;
        while (temp -> next != NULL)
        {
          temp = temp -> next;
        }
        temp->next = ptr;
        ptr->next = NULL;
        printf("\nNode inserted");

      }
```

```
    }
  }
```

Insertion in singly linked list after specified Node:

Algorithm:

STEP 1: IF PTR = NULL
WRITE OVERFLOW
GOTO STEP 12
END OF IF

STEP 2: SET NEW_NODE = PTR

STEP 3: NEW_NODE → DATA = VAL

STEP 4: SET TEMP = HEAD

STEP 5: SET I = 0

STEP 6: REPEAT STEP 5 AND 6 UNTIL I

STEP 7: TEMP = TEMP → NEXT

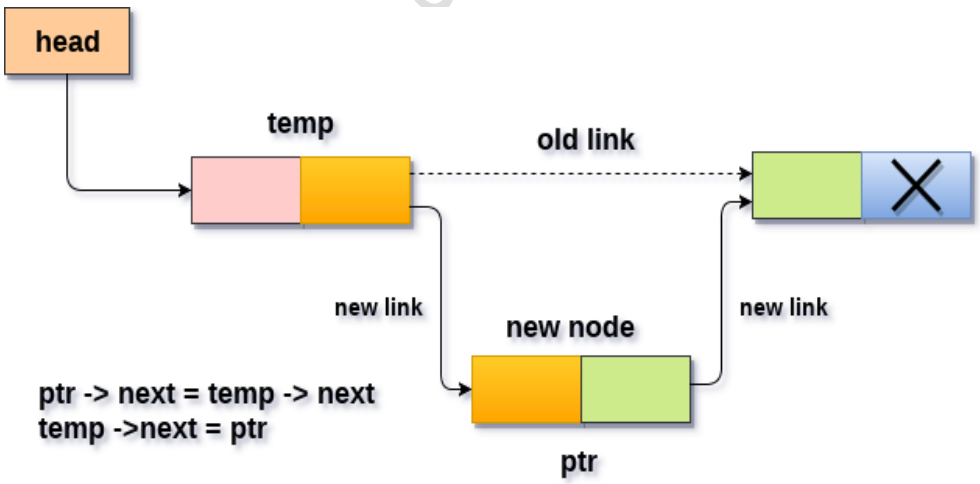
STEP 8: IF TEMP = NULL
WRITE "DESIRED NODE NOT PRESENT"
GOTO STEP 12
END OF IF
END OF LOOP

STEP 9: PTR → NEXT = TEMP → NEXT

STEP 10: TEMP → NEXT = PTR

STEP 11: SET PTR = NEW_NODE

STEP 12: EXIT



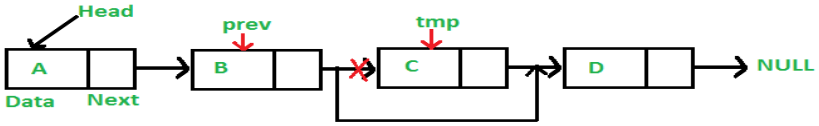
C Code:

```
void randominsert(int item)
  {
    struct node *ptr = (struct node *) malloc (sizeof(struct node));
    struct node *temp;
    int i,loc;
```

```
if(ptr == NULL)
{
   printf("\nOVERFLOW");
}
else
{

   printf("Enter the location");
   scanf("%d",&loc);
   ptr->data = item;
   temp=head;
   for(i=0;i<loc;i++)
   {
      temp = temp->next;
      if(temp == NULL)
      {
         printf("\ncan't insert\n");
         return;
      }

   }
   ptr ->next = temp ->next;
   temp ->next = ptr;
   printf("\nNode inserted");
}

}
```

Deleting a node:

To delete a node from linked list, we need to do following steps.
1) Find previous node of the node to be deleted.
2) Change the next of previous node.
3) Free memory for the node to be deleted.



Algorithm:

STEP 1: IF HEAD = NULL
WRITE UNDERFLOW
GOTO STEP 10
END OF IF

STEP 2: SET TEMP = HEAD

STEP 3: SET I = 0

STEP 4: REPEAT STEP 5 TO 8 UNTIL I<loc< li=""></loc<>

STEP 5: TEMP1 = TEMP

STEP 6: TEMP = TEMP → NEXT

STEP 7: IF TEMP = NULL
WRITE "DESIRED NODE NOT PRESENT"
GOTO STEP 12
END OF IF

STEP 8: I = I+1
END OF LOOP

STEP 9: TEMP1 → NEXT = TEMP → NEXT

STEP 10: FREE TEMP

STEP 11: EXIT

Deletion in singly linked list at beginning

Algorithm:

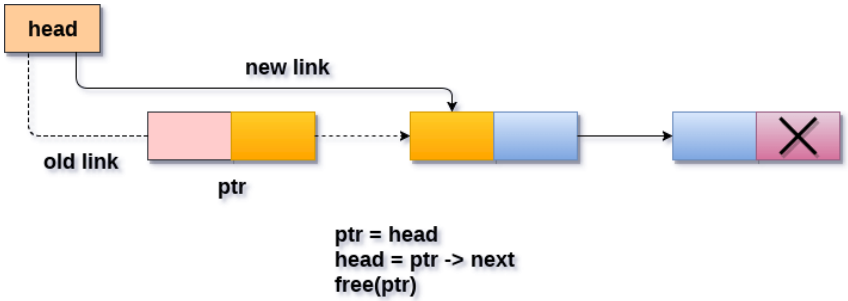
Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 5
[END OF IF]

Step 2: SET PTR = HEAD

Step 3: SET HEAD = HEAD -> NEXT

Step 4: FREE PTR

Step 5: EXIT



Deleting a node from the beginning

Deletion in singly linked list at the end

There are two scenarios in which, a node is deleted from the end of the linked list.

1. There is only one node in the list and that needs to be deleted.
2. There are more than one node in the list and the last node of the list will be deleted.

Algorithm:

Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 8
[END OF IF]

Step 2: SET PTR = HEAD

Step 3: Repeat Steps 4 and 5 while PTR -> NEXT!= NULL

Step 4: SET PREPTR = PTR

Step 5: SET PTR = PTR -> NEXT
[END OF LOOP]

Step 6: SET PREPTR -> NEXT = NULL

Step 7: FREE PTR

Step 8: EXIT