DATA STRUCTURES

UNIT-2

Linked List

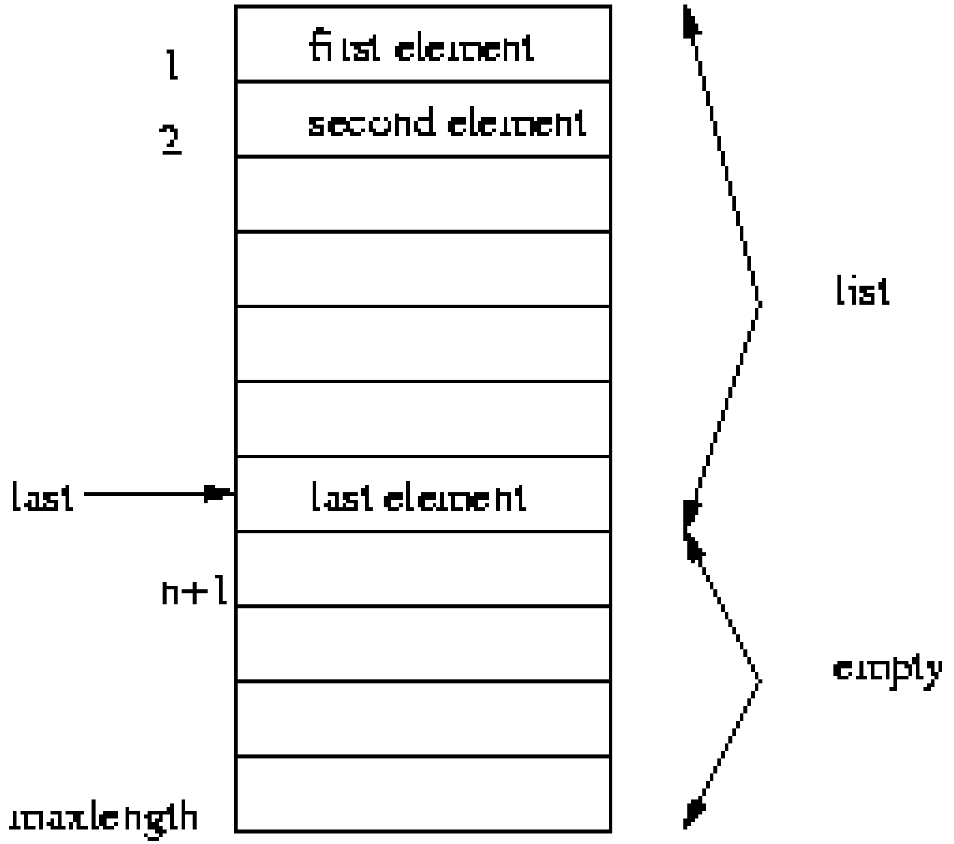
Dr G.KALYANI

Topics

- **Limitations of the Array Implementation**
- Motivation for Linked List
- Types of Linked List
- Single Linked List Characteristics
- Linked List ADT
- Operations on Linked List and their implementation
- Other Types of Linked List
- Summary

Array Implementation

- Elements are stored in contiguous array positions



Array Implementation...

- Requires an estimate of the maximum size of the list
- waste of space
- Print List and find: linear
- Find Kth element: constant
- insert and delete: slow
  - e.g. insert at position k (making a new element)
    - requires first pushing the array down one spot from k to n to make space at k position
  - e.g. delete at position k
    - requires shifting the elements from k+1 to n in the list up one
  - On average, half of the lists needs to be moved for either operation

Array Limitations

- Arrays are suitable for:
  - Inserting/deleting an element at the end.
  - Randomly accessing any element.
  - Searching the list for a particular value.

Topics

- Limitations of the Array Implementation
- **Motivation for Linked List**
- Types of Linked List
- Single Linked List Characteristics
- Linked List ADT
- Operations on Linked List and their implementation
- Other Types of Linked List
- Summary

Array versus Linked Lists

- Arrays are suitable for:
  - Inserting/deleting an element at the end.
  - Randomly accessing any element.
  - Searching the list for a particular value.
- Linked lists are suitable for:
  - Inserting an element at required location.
  - Deleting an element from required location.
  - Applications where sequential access is required.
  - In situations where the number of elements cannot be predicted beforehand.

Topics

- Limitations of the Array Implementation
- Motivation for Linked List
- **Types of Linked List**
- Single Linked List Characteristics
- Linked List ADT
- Operations on Linked List and their implementation
- Other Types of Linked List
- Summary

Types of Linked list

- Single Linked list

  There is a head pointer, and one next pointer per element. The last element's pointer is null. Traversed in only one direction

- Double Linked list

  There is a head pointer, and each element contains two pointers, one to the previous element and one to the next element. Traversed in two directions, making insertion and deletion a bit easier, at the cost of extra memory.

- Circular Linked list

  Same as Singly/Doubly linked, except that the last element's pointer points back to the first element's pointer. These used as queues.

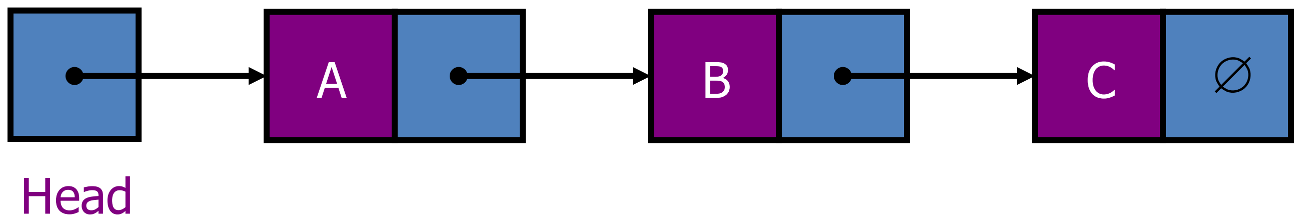
Topics

- Limitations of the Array Implementation
- Motivation for Linked List
- Types of Linked List
- **Single Linked List Characteristics**
- Linked List ADT
- Operations on Linked List and their implementations
- Other Types of Linked List
- Summary

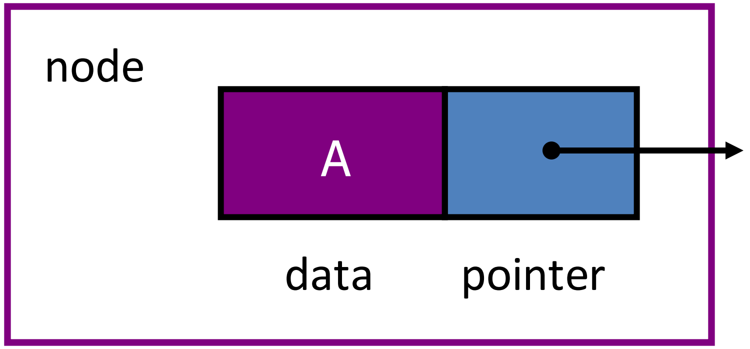
Linked Lists

- A **Linked List** is an ordered collection of data in which each element contains the location of the next element.
- In a linked list, each element contains two parts: data and one or more links
- A linked list is simply a chain of structures which contain a pointer to the next element.
- It is dynamic in nature. Items may be added to it or deleted from it at any location.
- The last node has a reference to null. The entry point into a linked list is called the **head** of the list. It should be noted that head is not a separate node, but the reference to the first node

Linked Lists

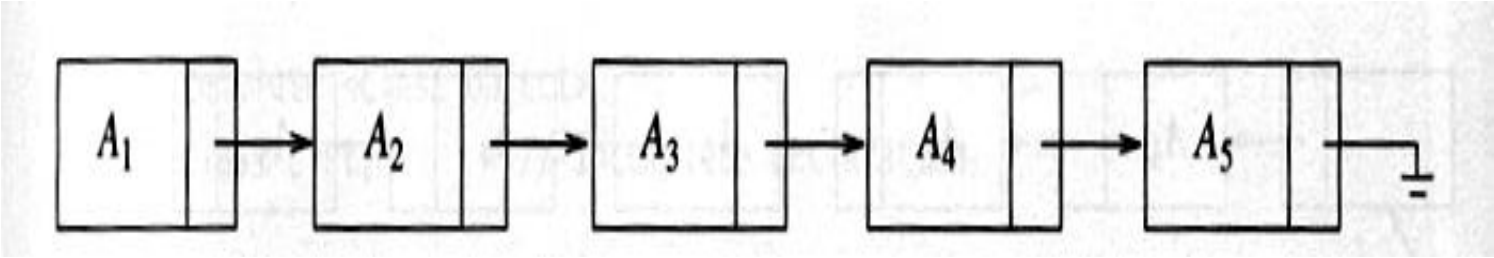


- A linked list is a series of connected nodes
- Each node contains at least
  - A piece of data (any type)
  - Pointer to the next node in the list
- Head: pointer to the first node
- The last node points to NULL



Linked List Implementation

- Ensure that the list is not stored contiguously
  - a series of structures that are not necessarily adjacent in memory



Topics

- Limitations of the Array Implementation
- Motivation for Linked List
- Types of Linked List
- Single Linked List Characteristics
- **Linked List ADT**
- Operations on Linked List and their Implementations
- Other Types of Linked List
- Summary

Linked List ADT

- **ADT Linked List**

  Objects: a finite list with zero or more elements

  Functions:

  - **Create node():** creates a new node which is empty.
  - **Insert(item):** adds a new item to the linked list at the specified position.
  - **Delete(item):** removes the specified item from linked list.
  - **Is Empty(queue):** tests to see whether the queue is empty. It returns a Boolean value.
  - **Search(item):** search for an item in the linked list.

Topics

- Limitations of the Array Implementation
- Motivation for Linked List
- Types of Linked List
- Single Linked List Characteristics
- Linked List ADT
- **Operations on Linked List and their Implementations**
- Other Types of Linked List
- Summary

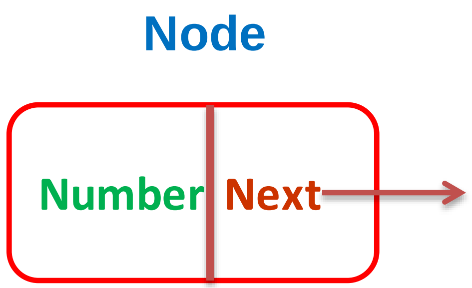
Operations on Linked List

- Create
- Insert
  - At Starting
  - In the specified location
  - At the End
- Delete
  - At Starting
  - At the End
  - Delete Specified
- Search
- Traversing

Creating a Node

- Node can be created using structure Data type.

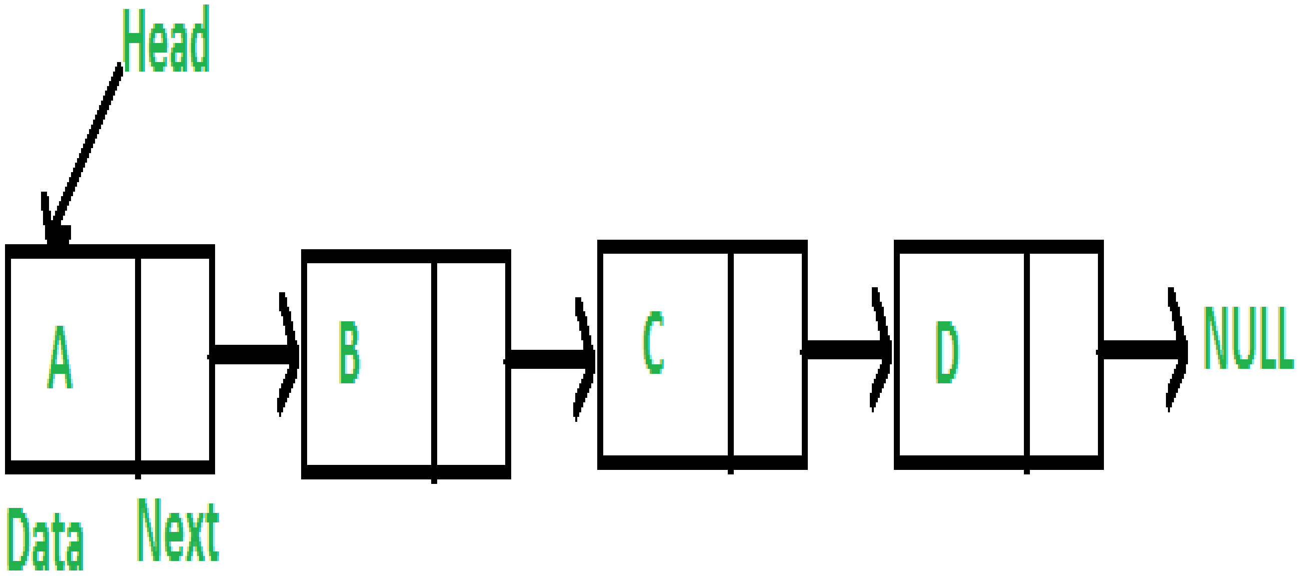
```
Struct node
{
Int number;
Struct node *next;
}
struct node *head=null;
```



Traversing a linked list

- Repeat the process of printing the data values of the nodes until the next of the node is NULL.

Traversing a linked list



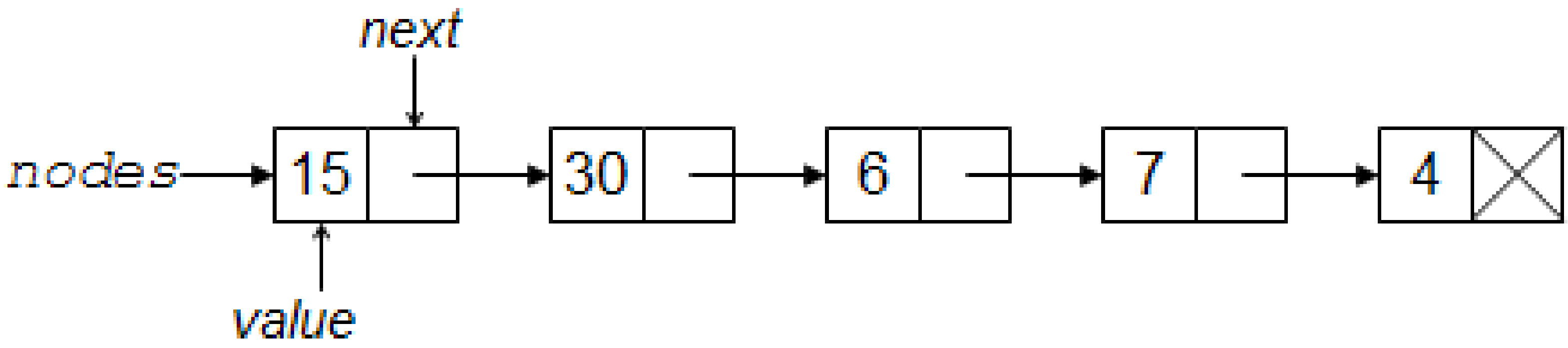
Traversing a linked list

```
Algorithm display()
{
    struct node *temp;
    temp = head;
    if(temp == NULL) then
    {
        write Nothing to print;
    }
    else
    {

        while (temp!=NULL) do
        {
            write temp->data;
            temp = temp -> next;
        }
    }
}
```

Searching in a linked list

- Repeat the process of comparing the search element with data values of the nodes until match occurs or the next of the node is NULL.

Searching in a linked list

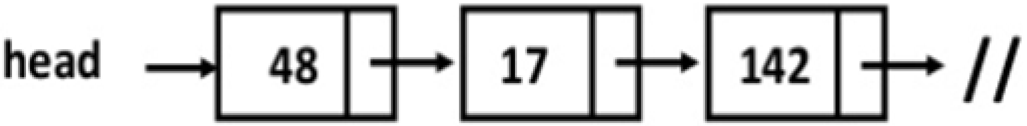


Searching a linked list

```
Algorithm search()
{
    struct node *temp;
    temp = head;
    if(temp == NULL) then
    {
        Write Empty List;
    }
    else
    {
        Read Item
        while (temp!=NULL) do
        {
            if(temp->data == item) then
            {
                write item found;
                flag=0;
                return;
            }
            else
            {
                flag=1;
                temp = temp -> next;
            }
        }
        if(flag==1) then
        {
            write Item not found;
        }
    }

}
```

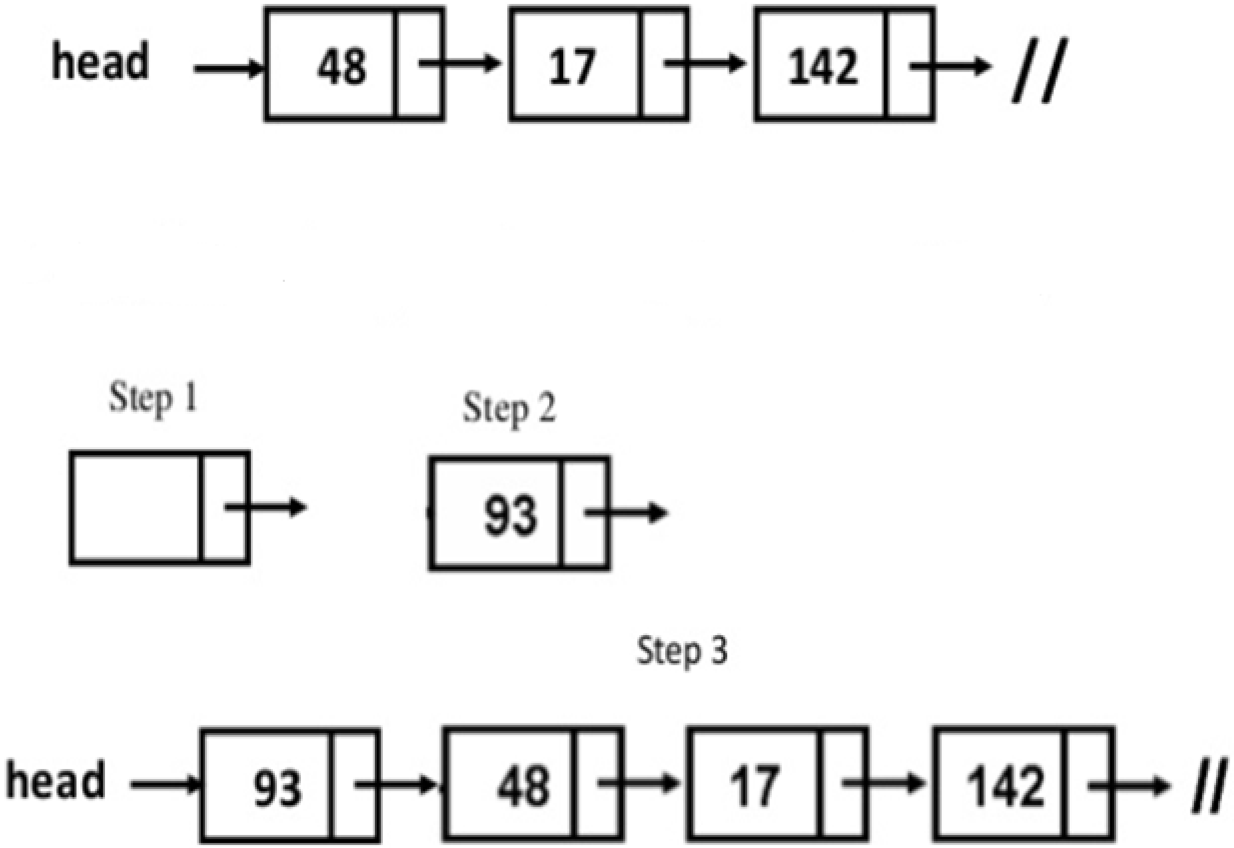
Insertion a node

Linked List:



Node to be added with value 93

Insertion at the beginning



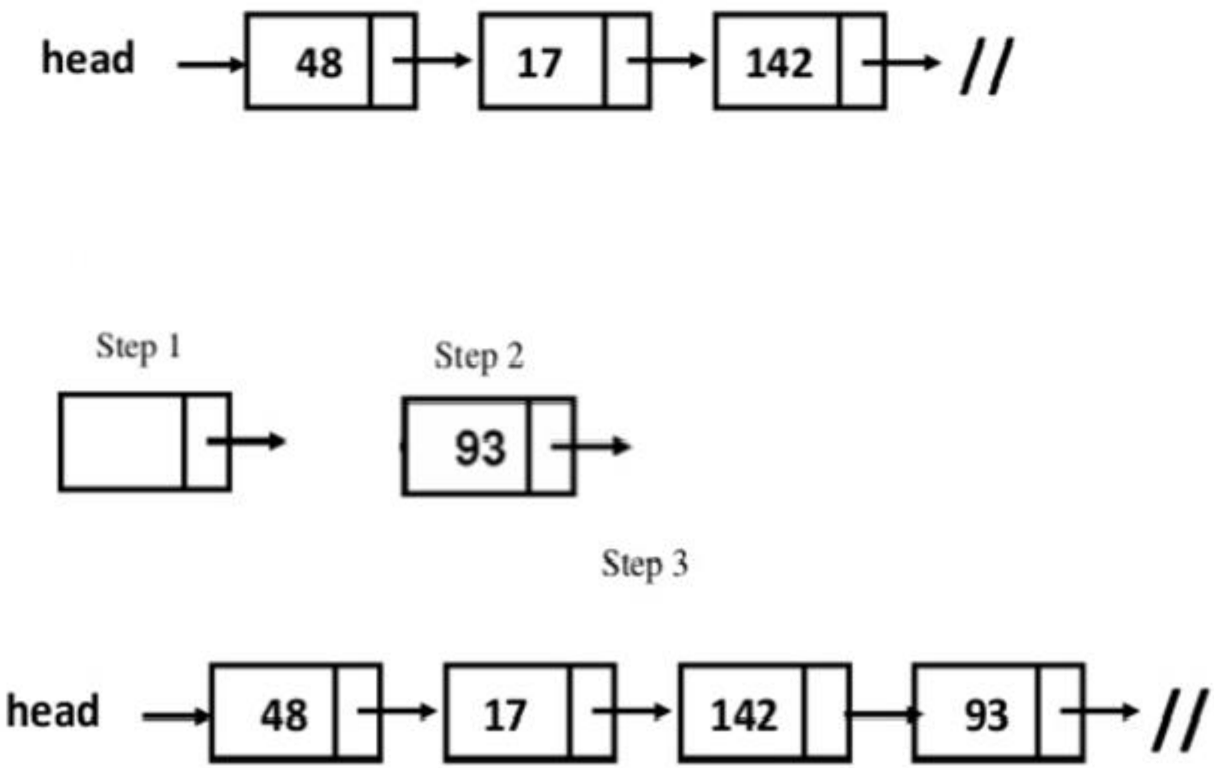
Insertion at the beginning

- Step 1: create a node
- Step 2: place the item in the data field
- Step 3: Adjust the pointers
  - Place the new node next as current header/starting
  - Make new node as header

Insertion at the beginning

```
Algorithm Begin insert()
{
    struct node *newnode;
    Allocate the memory to newnode;
    if(newnode == NULL) then
    {
        write memory not allocated;
    }
    else
    {
        read item;
        newnode ->data = item;
        newnode ->next = head;
        head =  newnode;
        //use traverse to check whether the item inserted or not ;
     }
}
```

Insertion at the End



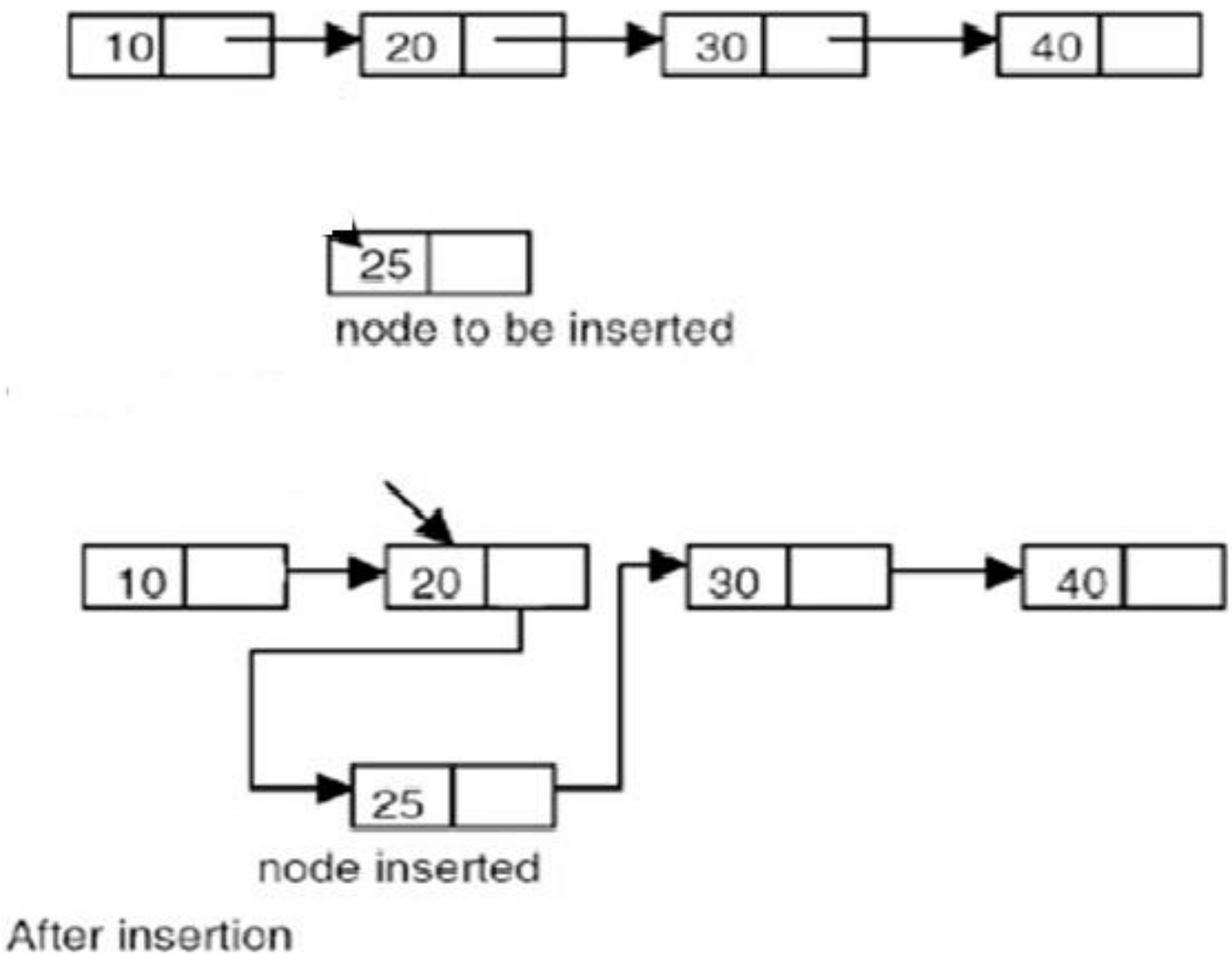
Insertion at the End

- Step 1: create a node
- Step 2: place the item in the data field
- Step 3: Adjust the pointers
  - Traverse up to the last node
  - Place the new node as the next of last node
  - Make new node next as NULL

Inserting at the End

```
Algorithm Insert-End ()
{
    struct node * newnode,*temp;
    Allocate the memory to newnode;
    if(newnode == NULL) then
    {
        write memory not allocated;
    }
    else
    {
        read item;
        newnode ->data = item;
        if(head == NULL) then
        {
            newnode  -> next = NULL;
            head = newnode;

        }
        else
        {
            temp = head;
            while (temp -> next != NULL) do
            {
                temp = temp -> next;
            }
            temp->next =  newnode;
             newnode ->next = NULL;

        }
    }
}
```

Inserting after the specified location



Inserting after the specified location

- Step 1: create a new node
- Step 2: place the item in the data field of new node
- Step 3: Adjust the pointers
  - Traverse up to the specified location
  - Store the next of the specified node in new node next
  - Place the new node as the next of specified node

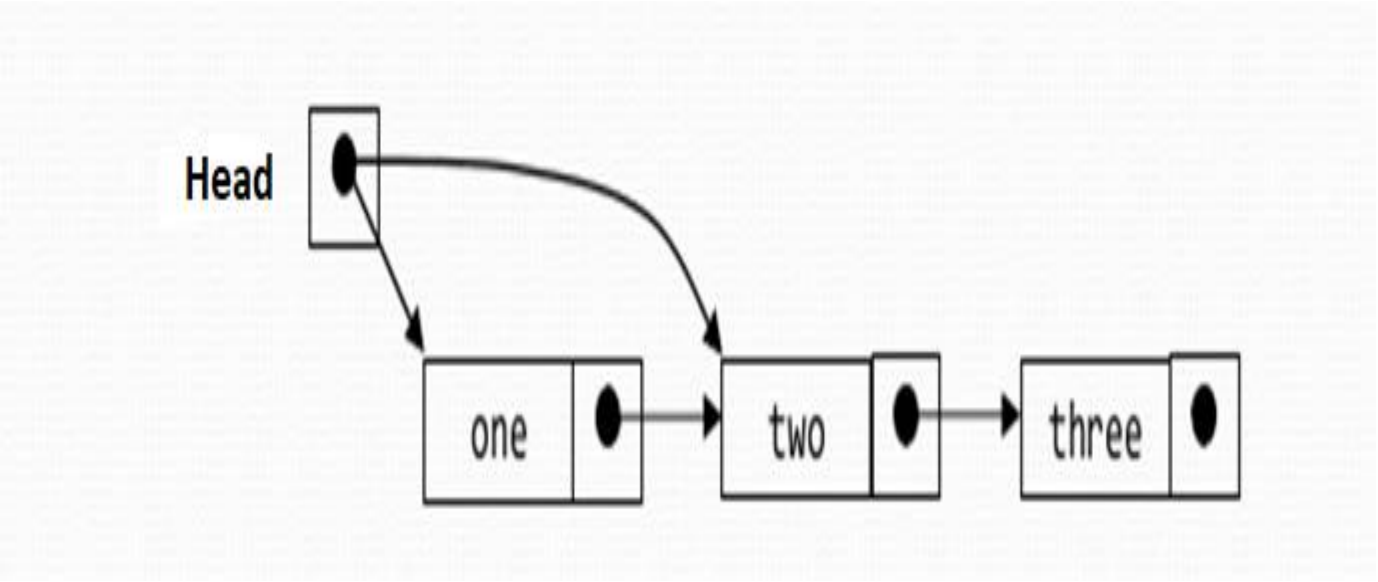
Insertion after the specified location

```
Algorithm random-insert()
{
    struct node * newnode, *temp;

  Allocate the memory to newnode;

    if(newnode == NULL)  then
    {
        write memory not allocated;
    }
    else
    {
        Read Item;
        Read the location
        newnode ->data = item;
        temp=head;
        for i=1 to loc-1 do
        {
            temp = temp->next;
            if(temp == NULL)
            {
                write cannot insert;
            }

        }
        newnode  ->next = temp ->next  ;
        temp ->next =  newnode;
    }
}
```

Deleting a Node

- Delete at beginning
- Delete at end
- Delete the specified location

Deletion at the beginning



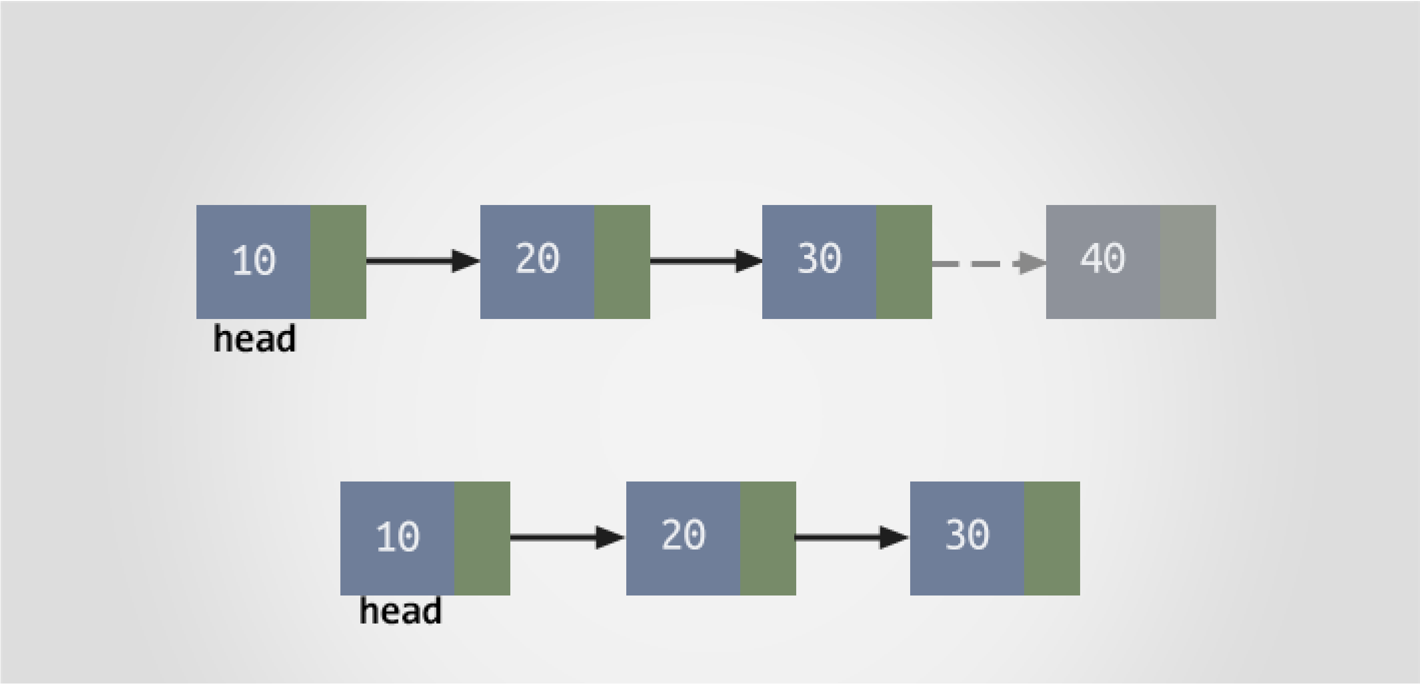
Deletion at the beginning

- Step 1: check whether the list is empty or not.
- Step 2: If empty display that list is empty
- Step 3: Adjust the pointers
  - Save head as temp
  - Place the next of temp as current header
  - Delete the temp

Deletion at the beginning

```
Algorithm  begin_delete()
{
   struct node *temp;
   if(head == NULL) then
   {
       Write List is empty;
   }
   else
   {
      temp = head;
      head = temp->next;
      free(temp);

   }
}
```

Deletion at the End



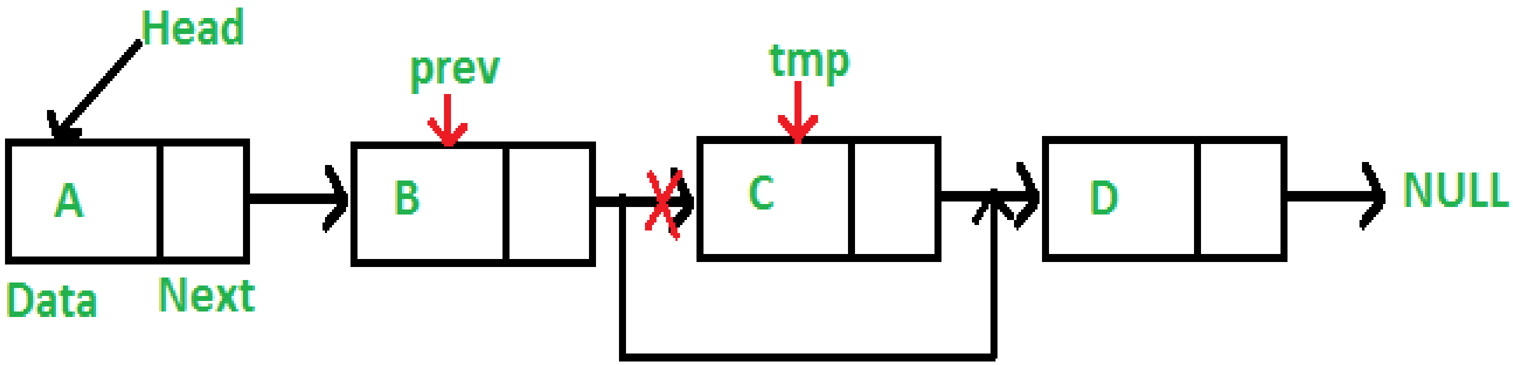
Deletion at the end

- Step 1: check whether the list is empty or not.
- Step 2: If empty display that list is empty
- Step 3: if the list contains single node, delete it and make the head as null.
- Step 4: Otherwise Adjust the pointers
  - Save head as temp
  - Move up to the last
  - Place the next of last but one as null
  - Delete the temp

Deletion at the End

```
Algorithm last_delete()
{
    struct node *temp,*prev;

    if(head == NULL) then
    {
        Write List is empty;
    }
    else if(head -> next == NULL) then
    {
        head = NULL;
        free(head);

    }
else
    {
      temp = head;
      while(temp->next != NULL)do
        {
            prev = temp;
            temp = temp ->next;
        }
        prev->next = NULL;
        free(temp);

    }
}
```

Deletion at the specified location



Deletion at the specified location

- Step 1: check whether the list is empty or not.
- Step 2: If empty display that list is empty
- Step 3: Otherwise Adjust the pointers
  - Save head as temp
  - Move up to the specified location
  - Place the next of specified location in previous of specified location
  - Delete the temp

Deletion at the specified location

```
Algorithm random_delete()
{
    struct node *temp,*prev;

    read location ;
    temp=head;
    for i= 1 to loc-1 do
    {
        prev = temp;
        temp = temp->next;
        if(temp == NULL)
        {
            Write cannot delete;
            return;
        }
    }
    prev ->next = temp ->next;
    free(temp);
  }
```

Topics

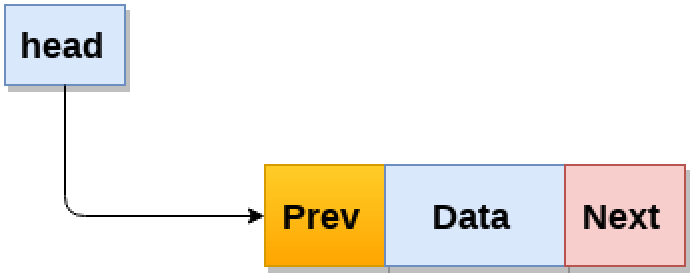
- Limitations of the Array Implementation
- Motivation for Linked List
- Types of Linked List
- Single Linked List Characteristics
- Linked List ADT
- Operations on Linked List and their Implementations
- **Other Types of Linked List**
- Summary

Double Linked List

- Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence.

- Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer).

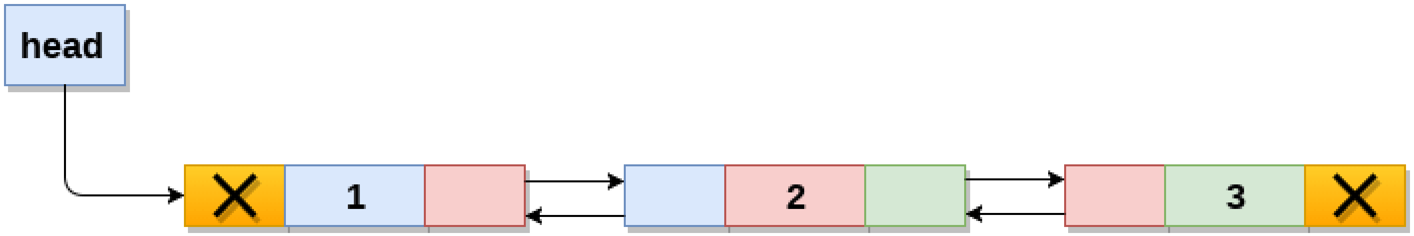
- A sample node in a doubly linked list is shown in the figure.



Node

Double Linked List

- A doubly linked list containing three nodes having numbers from 1 to 3 in their data part.



Doubly Linked List

Double Linked List

Head

1

	Data	Prev	Next
1	13	-1	4
2			
3			
4	15	1	6
5			
6	19	4	8
7			
8	57	6	-1

Memory Representation of a Doubly linked list

Operations on Double Linked List

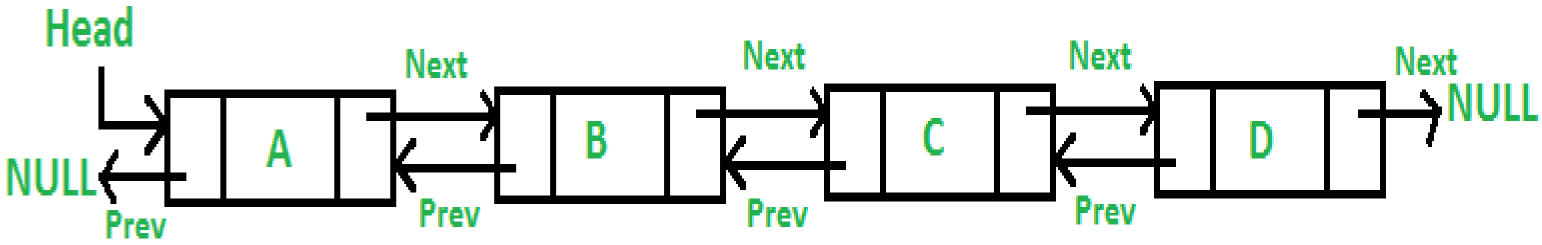
- Create
- Insert
  - At Starting
  - In the specified location
  - At the End
- Delete
  - At Starting
  - At the End
  - Delete Specified
- Search
- Traversing

Creating a node in double linked list

- structure of a node in doubly linked list can be given as :

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

Traversing a double linked list



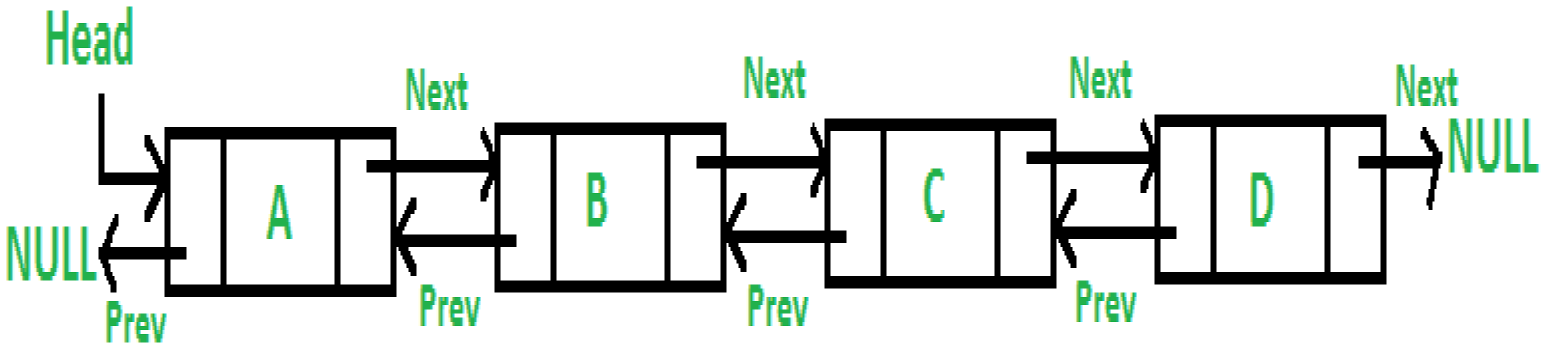
Traversing a double linked list

- Repeat the process of printing the data values of the nodes until the next of the node is NULL.

Traversing a double linked list

```
Algorithm display()
{
    struct node *temp;
    temp = head;
    if(temp == NULL) then
    {
        write Nothing to print;
    }
    else
    {

        while (temp!=NULL) do
        {
            write temp->data;
            temp = temp -> next;
        }
    }
}
```

Searching in a double linked list



Searching in a double linked list

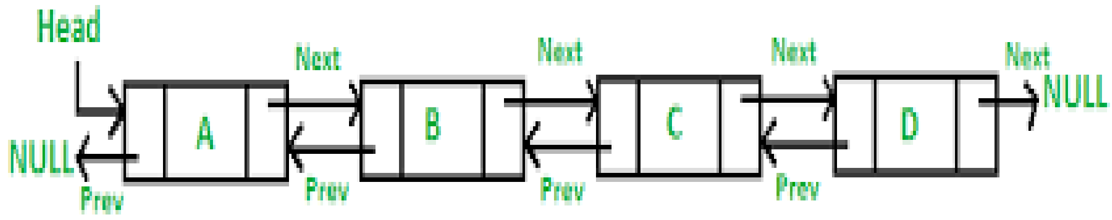
- Repeat the process of comparing the search element with data values of the nodes until match occurs or the next of the node is NULL.

Searching a double linked list

```
Algorithm search()
{
    struct node *temp;
    temp = head;
    if(temp == NULL) then
    {
        Write Empty List;
    }
    else
    {
        Read Item
        while (temp!=NULL) do
        {
            if(temp->data == item) then
            {
                write item found;
                return;
            }
            else
            {

                temp = temp -> next;
            }
        }
        write Item not found;

    }

}
```

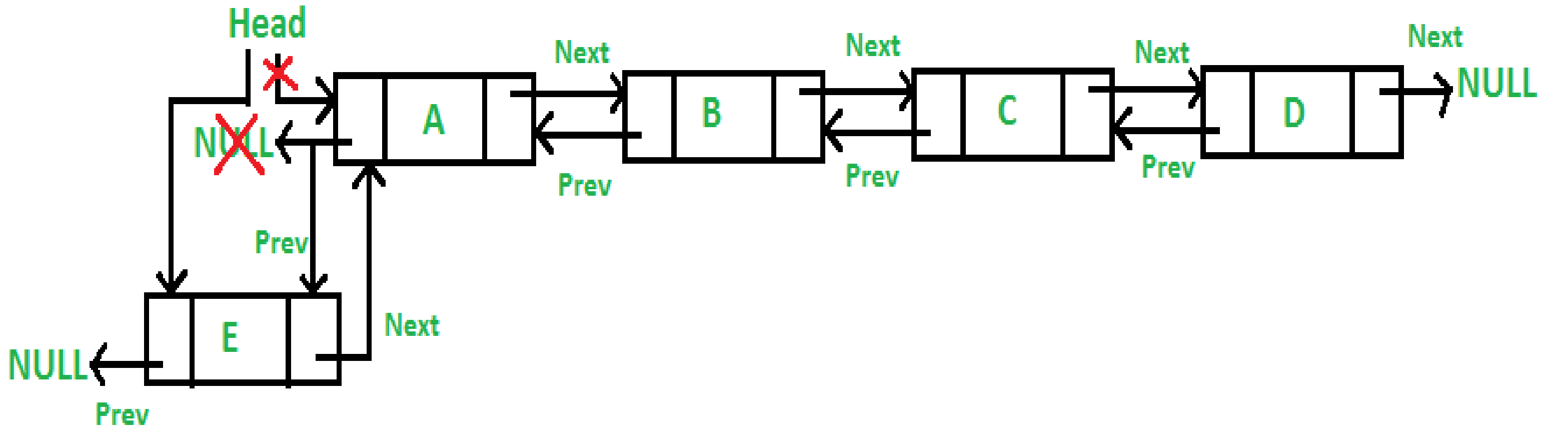
Insertion a node into double linked list

Double Linked List



Node to be added with value E

Insertion at the beginning



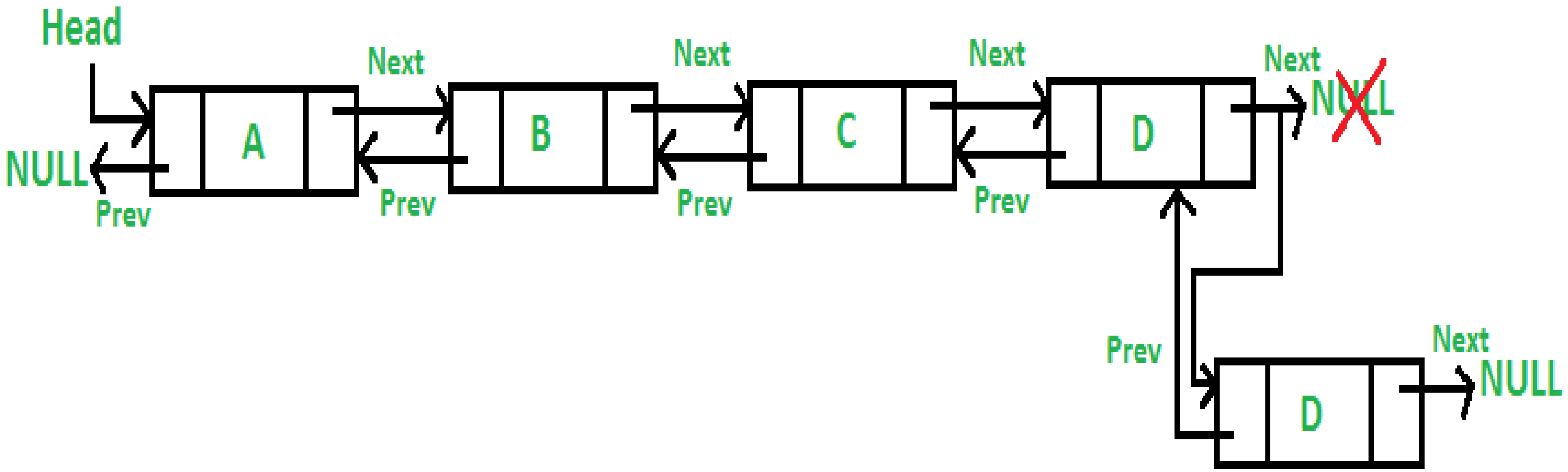
Insertion at the beginning

- Step 1: create a node
- Step 2: place the item in the data field
- Step 3: Adjust the pointers
  - Make the prev of current header as new node
  - Place the new node next as current header/starting
  - Make the prev of new node as null
  - Make new node as header

Insertion at the beginning

```
Algorithm insertion_beginning()
{
  struct node *newnode;
  Allocate the memory to newnode;
  if(newnode == NULL)
  {
     Write memory not allocated;
  }
  else
  {
       read item;
       newnode->data=item;
       if(head==NULL)
       {
              newnode->next  = NULL;
              newnode->prev=NULL;
              head=newnode;
       }
       else
       {
              newnode->prev=NULL;
              newnode->next = head;
              head->prev=newnode;
              head=newnode;
       }
  }
}
```

Insertion at the End



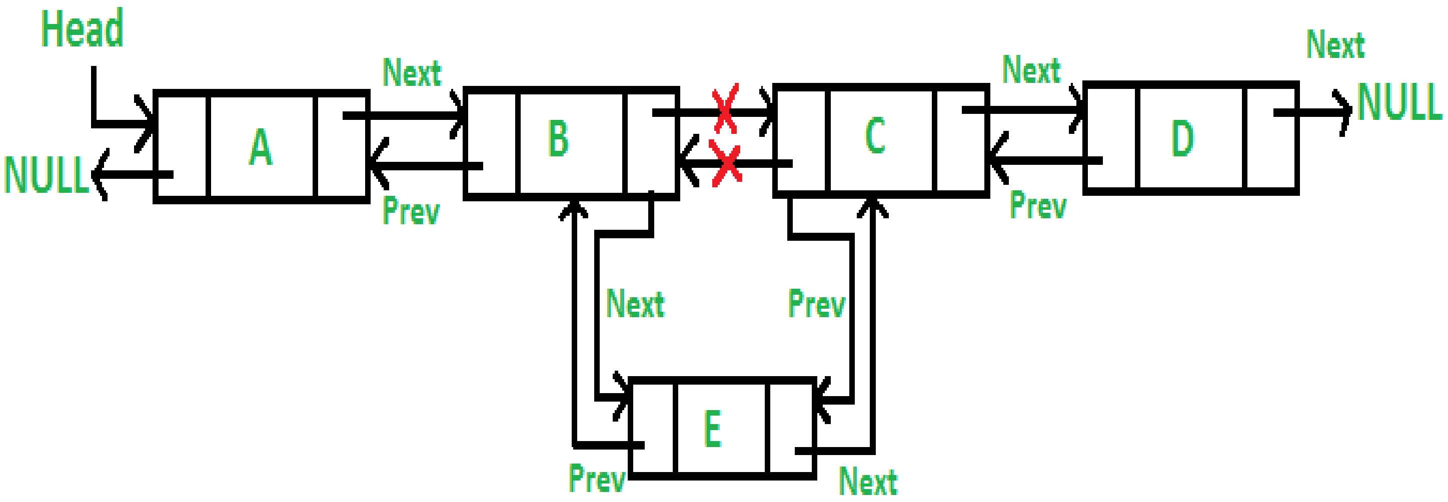
Insertion at the end

- Step 1: create a node
- Step 2: place the item in the data field
- Step 3: Adjust the pointers
  - Traverse to the last node
  - Make the last node next as new node
  - Place the new node prev as current last
  - Make the next of new node as null

Inserting at the End

```
void insertion_last()
{
  struct node *newnode,*temp;
  Allocate memory to newnode;
  if(newnode  == NULL)
  {
     write memory not created;
  }
  else
  {
      read item;
       newnode ->data=item;
     if(head == NULL)
     {
         newnode ->next = NULL;
         newnode ->prev = NULL;
         head =  newnode;
     }
      else
      {
        temp = head;
        while(temp->next!=NULL)
        {
           temp = temp->next;
        }
        temp->next =  newnode;
         newnode  ->prev=temp;
         newnode ->next = NULL;
        }

      }
}
```

Inserting after the specified location



Inserting after the specified location

- Step 1: create a new node
- Step 2: place the item in the data field of new node
- Step 3: Adjust the pointers
    - Traverse up to the specified location
    - Store the next of the specified node in new node next
    - Make the specified node as prev of new node
    - Place the new node as the next of specified node
    - Make the prev of specified next as newnode

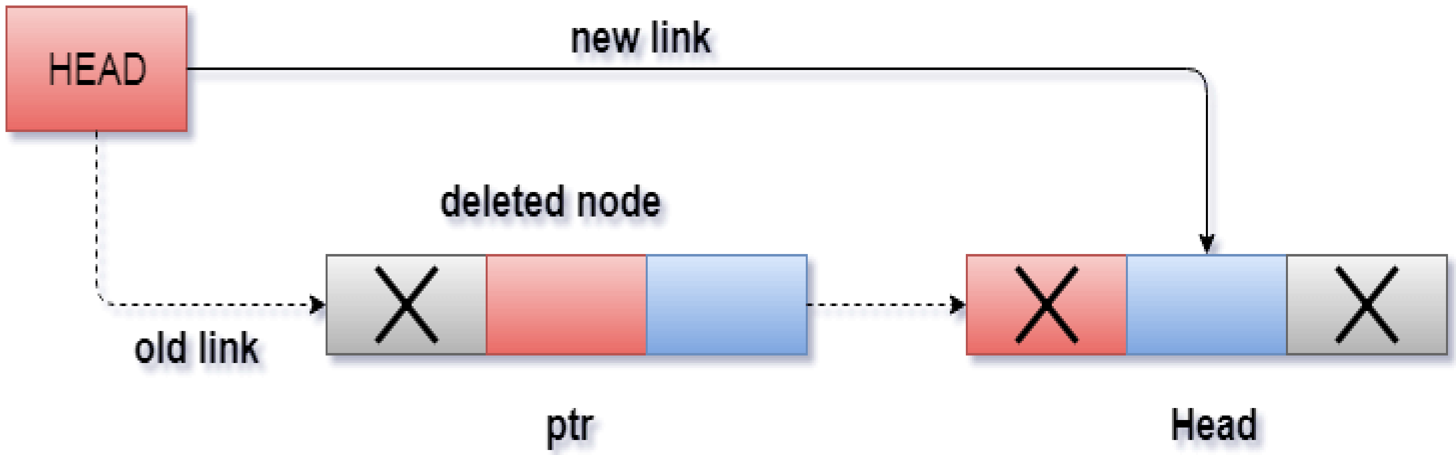
Inserting after the specified location

```
Algorithm  insertion_specified()
{
   struct node *newnode,*temp;
  Allocate the memory to newnode;
   if(newnode  == NULL)
   {
      write memory not allocated;
   }
   else
   {
      Read item, location;
      newnode ->data = item;
      temp=head;
      for i=1 to loc-1 do
        {
          temp = temp->next;
          if(temp == NULL)
          {
            write   less than the required elements;
             return;
          }
        }

         newnode ->next = temp->next;
         newnode  -> prev = temp;
        temp->next =  newnode;
         newnode ->next->prev= newnode;

   }
}
```

Deleting a Node

- Delete at beginning
- Delete at end
- Delete the specified location

Deletion at the beginning



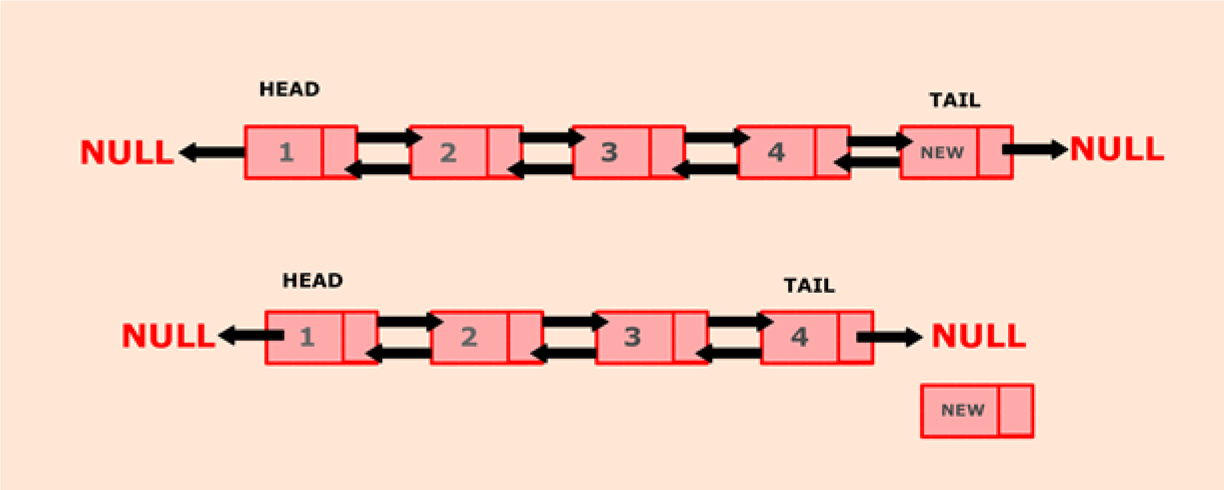
Deletion at the beginning

- Step 1: check whether the list is empty or not.
- Step 2: If empty display that list is empty
- Step 3: Adjust the pointers
  - Save head as temp
  - Place the next of temp as current header
  - Delete the temp

Deletion at the beginning

```
void deletion_beginning()
{
    struct node *temp;
    if(head == NULL)
    {
        write UNDERFLOW;
    }
    else if(head->next == NULL)
    {
        head = NULL;
        free(head);
        write node deleted;
    }
    else
    {
        temp  = head;
        head =  temp  -> next;
        head -> prev = NULL;
        free(temp);
        write node deleted;
    }

}
```

Deletion at the End



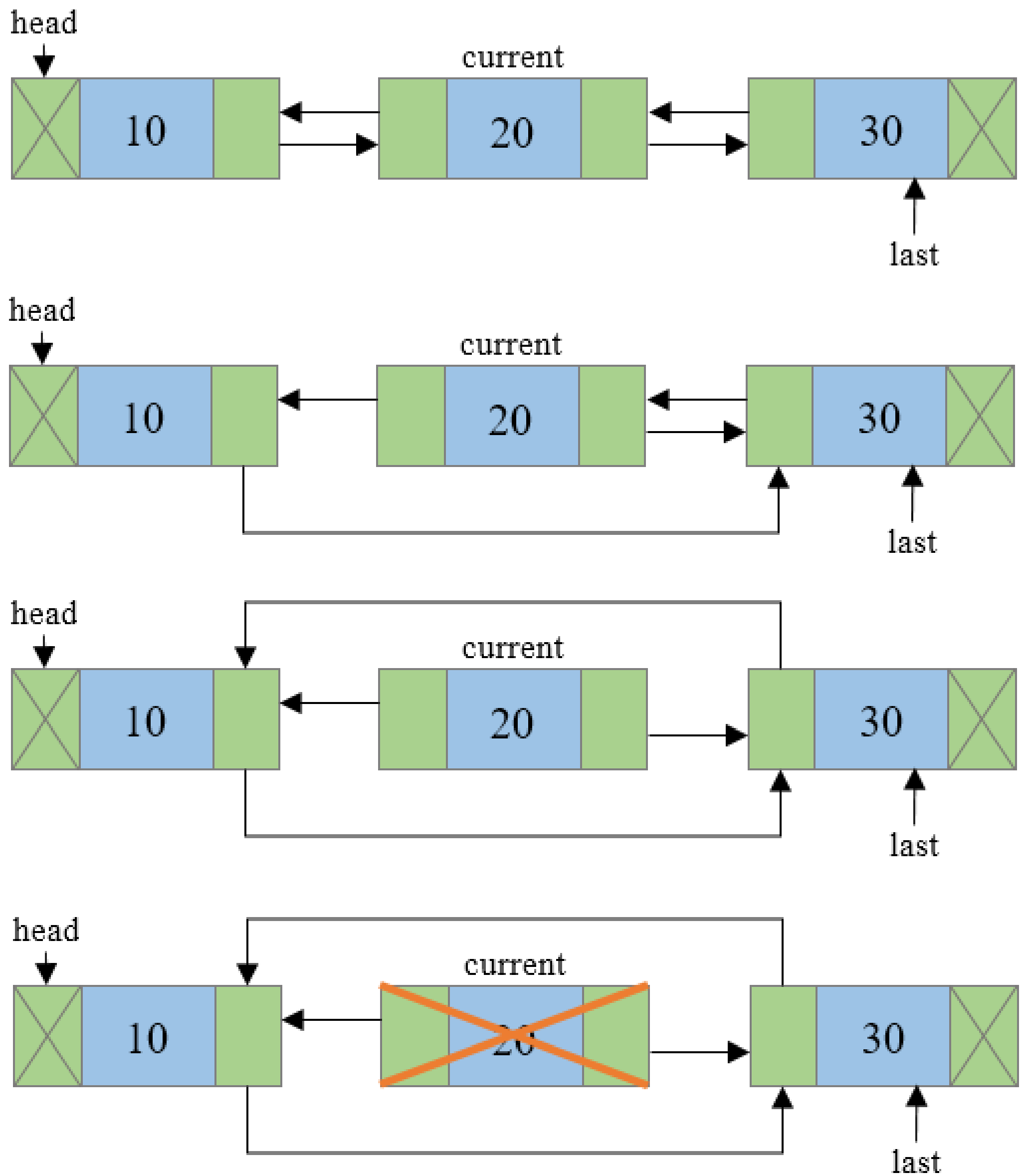
Deletion at the end

- Step 1: check whether the list is empty or not.
- Step 2: If empty display that list is empty
- Step 3: if the node contains single node, delete it and make the head as null.
- Step 4: Otherwise Adjust the pointers
  - Save head as temp
  - Move up to the last
  - Make last but one node next as null
  - Delete the temp

Deletion at the End

```
void deletion_last()
{
    struct node *temp;
    if(head == NULL)
    {
        write UNDERFLOW;
    }
    else if(head->next == NULL)
    {
        head = NULL;
        free(head);
        write node deleted;
    }
    else
    {
        temp  = head;
        if(temp ->next != NULL)
        {
            pnode=temp;
            temp  =  temp  -> next;
        }
        pnode -> next = NULL;
        free(temp);
        write node deleted;
    }
}
```

Deletion at the specified location



Deletion at the specified location

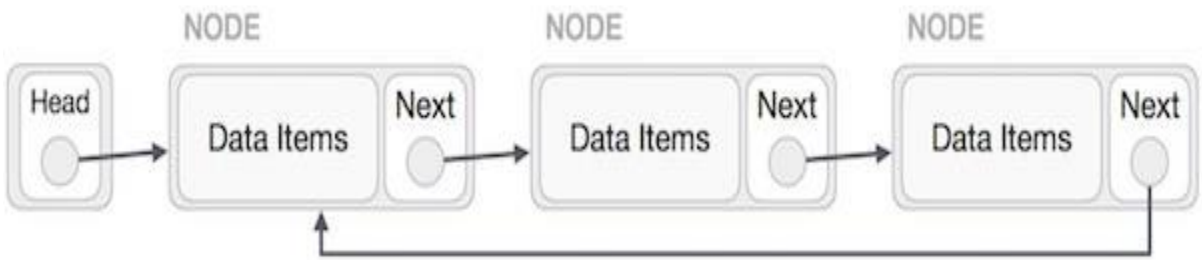
- Step 1: check whether the list is empty or not.
- Step 2: If empty display that list is empty
- Step 3: Otherwise Adjust the pointers
  - Save head as temp
  - Move up to the specified location
  - Place the next of specified location as next of the previous node
  -

Deletion at the specified location

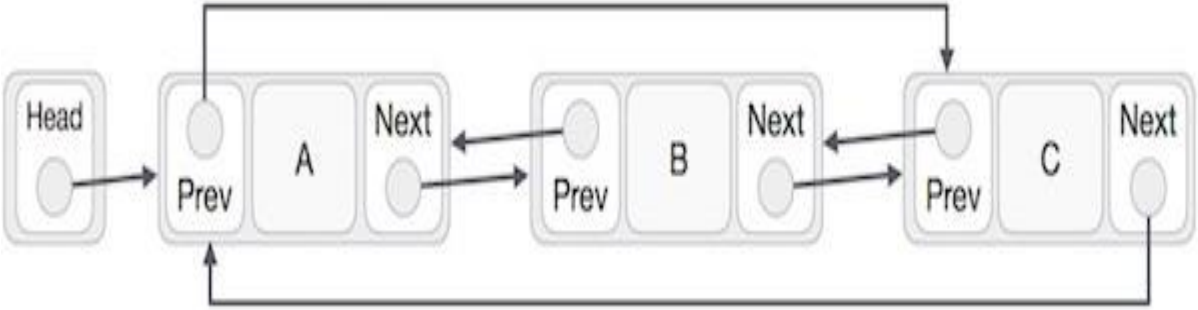
```
Algorithm random_delete()
{
    struct node *temp,*pnode;
    if(head == NULL)
    {
        write UNDERFLOW;
    }
    else
    {
        read location ;
        temp=head;
        for i= 1 to loc-1 do
        {
            pnode = temp;
            temp = temp->next;
            if(temp == NULL)
            {
                Write cannot delete;
                return;
            }
        }
        pnode ->next = temp ->next;
        temp->next->prev=pnode;
        free(temp);
    }
```

Circular linked list

- Singly Linked List as Circular list:
  the next pointer of the last node points to the first node.



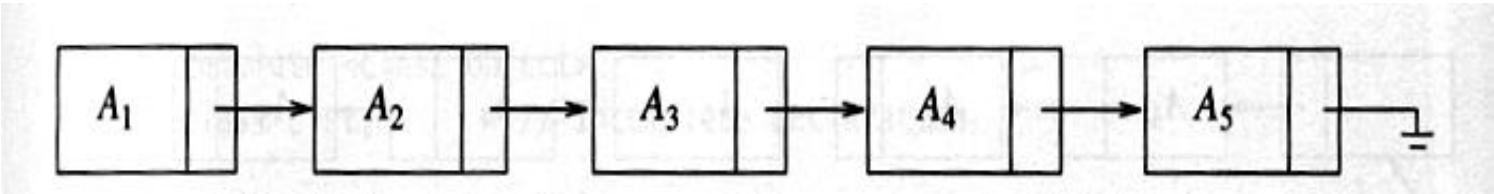
- Doubly Linked List as Circular list:
  the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node



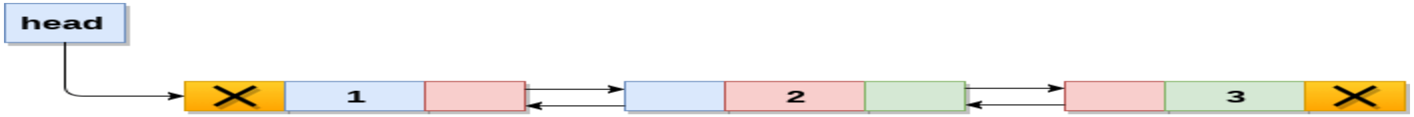
Summary

- Limitation of arrays
  - Arrays are suitable for:
    - Inserting/deleting an element at the end.
    - Randomly accessing any element.
    - Searching the list for a particular value.

- Single linked list



- Double linked list



Doubly Linked List

- Circular linked list