Chapter 3

Linked List Data Structure and its applications

Lists - Introduction

- Some applications require processing data in a strict chronological order.

Example:

- Processing objects in the order that they arrived.

  OR

- Processing objects in the reverse of the order that they arrived.

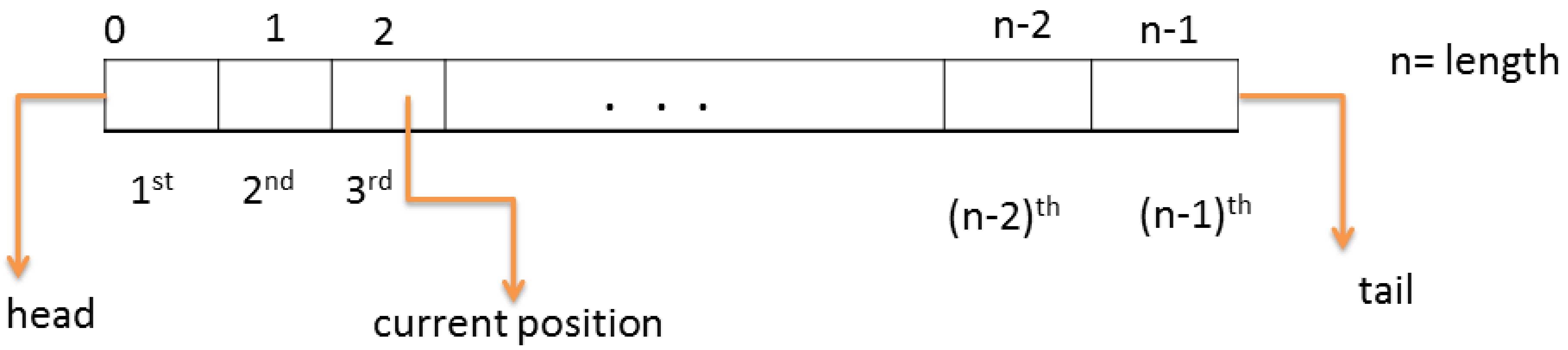
▪ **For all these situations, a simple list structure is appropriate.**

Lists – Definition

- A list is a finite, ordered sequence of data items called elements.
- **Important concept:** List elements have a position.
  - [1st element in the list, 2nd element, and so on.]
- **Notation: $<a_0, a_1, \ldots, a_{n-1}>$**
  - ["Ordered": the *positions* are ordered, NOT the *values*.]
- Each list element has a data type (all same data type, but not necessarily).

Lists – Definition

- The empty list contains no elements.
- The length of the list is the number of elements currently stored.
- The beginning of the list is called the head, the end of the list is called the tail.



Lists – Definition

- Sorted lists have their elements positioned in ascending or descending order of value, while unsorted lists have no necessary relationship between element values and positions.

- What operations should we implement?

  [Design the basic operations first]

List Implementation Concepts

- Our list implementation will support the concept of a current position.

- **Basic operations:**
  - Add, find and delete element anywhere, next, previous, clear (reinitialize), test for empty.

Types of Data Structures to implement list

- There are two broad types of data structure based on their memory allocation:
  - Static Data Structures
  - Dynamic Data Structures

I. Static Data Structures

- Are data structures that are defined & allocated before execution, thus the size cannot be changed during time of execution.

  Example: **Array** implementation of ADTs.

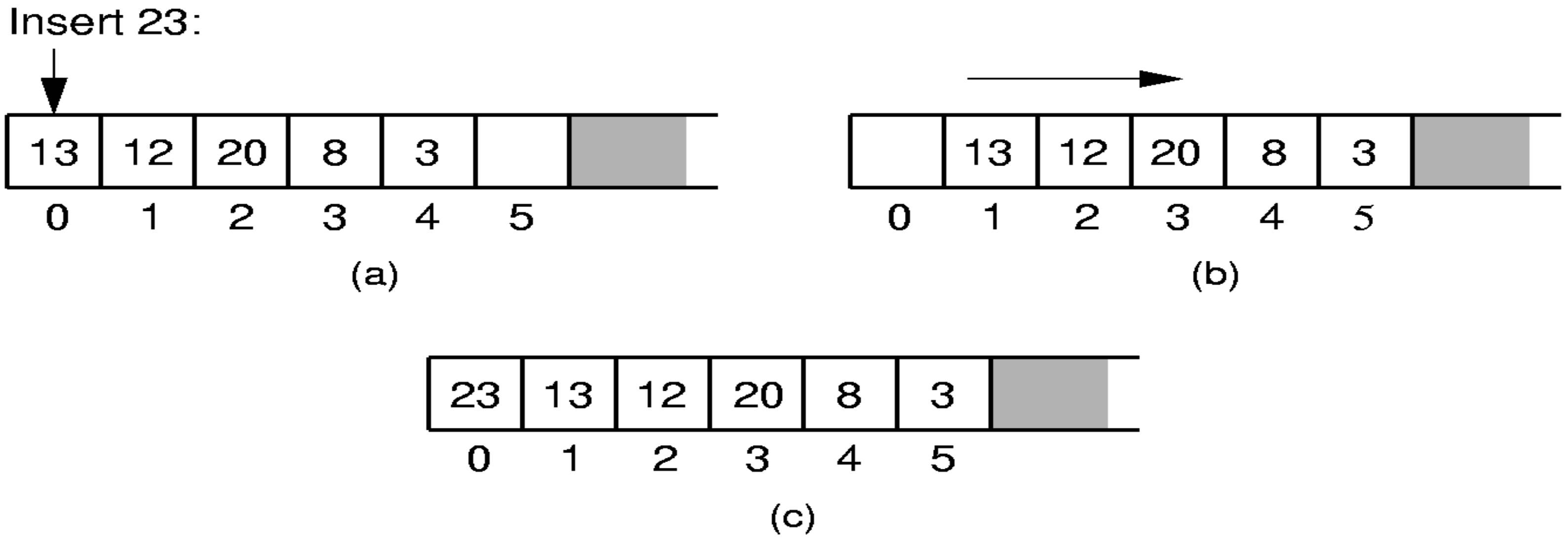
II. Dynamic Data Structure

- Are data structure that can grow and shrink in size or permits discarding of unwanted memory during execution time.

  Example: **Linked list** implementation of ADTs.

Array-Based List implementation

- There are two standard approaches to implementing lists, the array-based list and the linked list.
- Array-Based List Insert:
  - Push items up/down. Cost: $\Theta(n)$.



- Inserting an element at the **<u>head</u>** of an array-based list requires **shifting all existing elements** in the array by one position toward the tail.

Revision to Structure

- Structure is a collection of data items, and the data items can be of different data type.
- The data item of structure is called **member of the structure.**

<u>Declaration of structure</u>

- Structure is defined using the **struct** keyword.

```
struct name {
                    data type1 member 1;
                    data type2 member 2
                       .
                       .
                    data type n member n;
                };
```

Example :

```
struct student {
                char name[20];
                int age;
                char Dept[20];
              };
```

- The **struct** keyword creates a new **user defined data type** that is used to declare variable of an aggregated data type.

Accessing Members of Structure Variables

- *The Dot operator (.):* to access data members of structure variables.
- *The Arrow operator (->):* to access data members of pointer variables pointing to the structure.

```
struct student {
                char name[20];
                int age;
                char Dept[20];
            };
```

<u>Example</u>:

```
        struct student stud;
        struct student *studptr;

        cout<<stud.name;
              OR
        cout<<studptr->name;
```

Self-Referential Structures

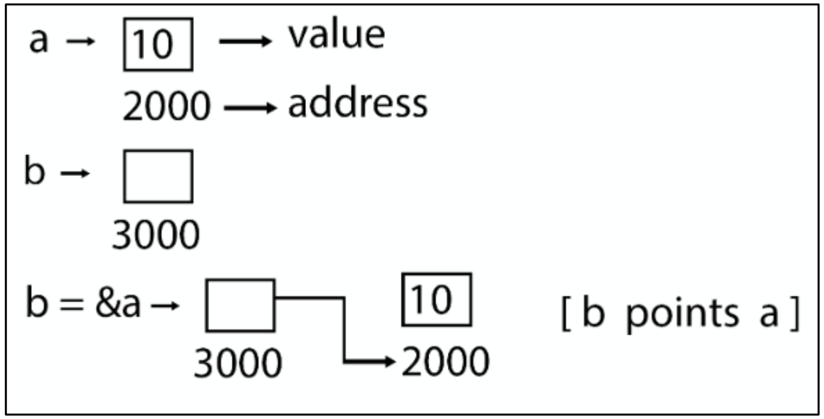
- Structures can hold pointers to instances of themselves.

<u>Example</u>:

```
struct student{
    char name[20];
    int age;
    char Dept[20];
    struct student *next;
  };
```

What is Pointer?

- A pointer is a variable that **stores the memory address of an object**.
- Pointer is used **to points the address of the value** stored anywhere in the computer memory.
- Every memory location has its address defined which can be accessed using **ampersand (&)** operator which denotes an address in memory.
- To obtain the value stored at the location is known as **dereferencing** the pointer



Linked List-Based List implementation

- The second approach for implementing lists makes use of pointers and is called a linked list.

What is Linked List?

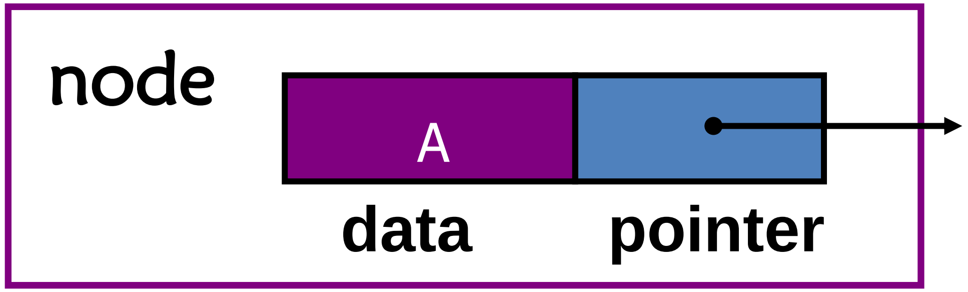
- It is a linear data structure that includes a **series of connected nodes.**
- It can be defined as the **nodes that are randomly stored in the memory.**
- The linked list uses dynamic memory allocation,
  - i.e. it allocates memory for new list elements as needed.
- It is **self-referential** structure.

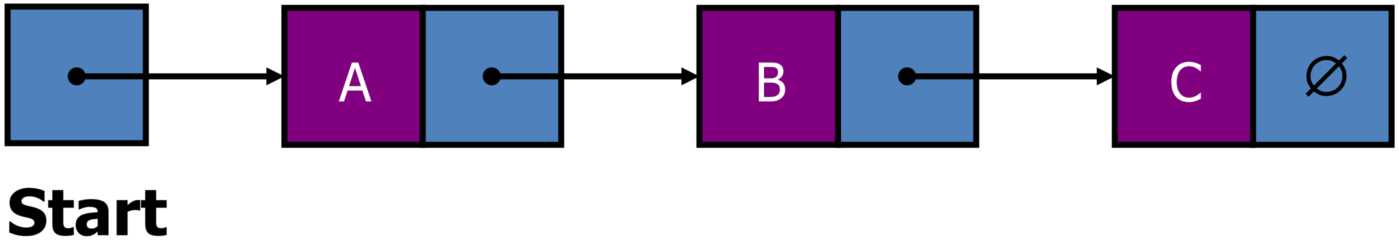
Linked List-Based List implementation

- A node in the linked list contains two parts,
  - i.e., first is the **data** part and second is the **address** part.
  - The data field holds the actual elements on the list.
  - The address (pointer) field contains the address of the next node in the list.
- The last node of the list contains a pointer to the **null**.
- In a linked list, every link contains a connection to another link.
- After array, linked list is the **second most used data structure.**

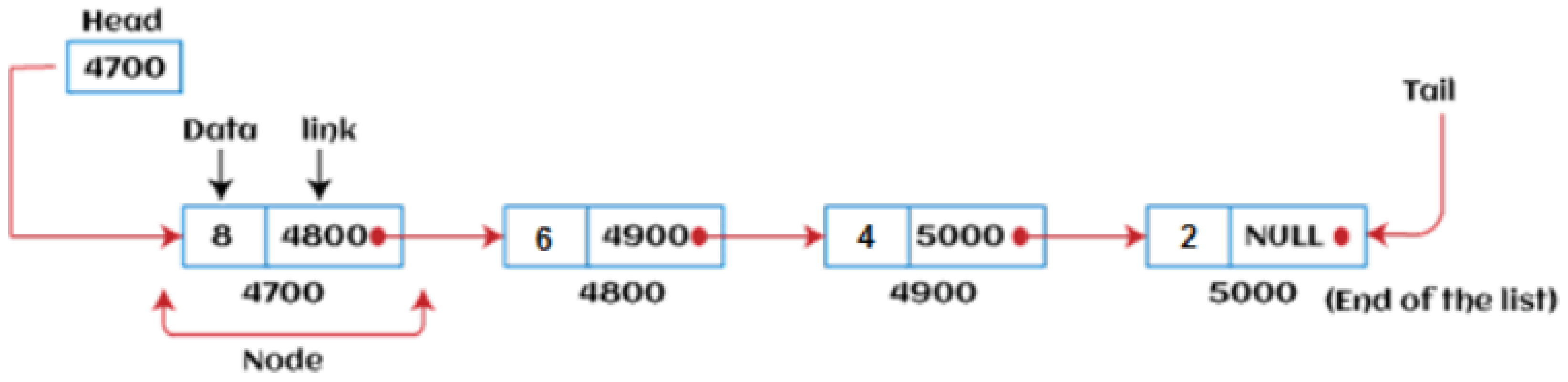
Linked List-Based List implementation

- Linked list can be represented as the connection of nodes in which **each node points to the next node of the list.**
- The representation of the linked list is shown below -





Linked List-Based List implementation



- Start (Head): Special pointer that points to the first node of a linked list, so that we can keep track of the linked list.
- The last node should point to NULL to show that it is the last link in the chain (in the linked list).

Why use linked list over array?

- Now, the question arises why we should use linked list over array?
- Linked list is a data structure that overcomes the limitations of arrays.
- Limitations of arrays –
  - The size of the array must be known in advance before using it in the program.
  - Increasing the 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 an element in the array needs shifting of all its predecessors.

Why use linked list over array?

- Linked list is useful because –
  - Linked list allocates the memory dynamically.
  - All the nodes of the linked list are non-contiguously stored in the memory and linked together with the help of pointers.
  - In linked list, size 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.

Advantages of Linked list

- The advantages of using the Linked list are given as follows -
  - Dynamic data structure - The size of the linked list may vary according to the requirements. Linked list <u>does not have a fixed size</u>.
  - Memory efficient - The size of a linked list can grow or shrink according to the requirements, so <u>memory consumption in linked list is efficient</u>.
  - Implementation - We can implement both <u>stacks</u> and <u>queues</u> using linked list.

Advantages of Linked list

- The advantages of using the Linked list are given as follows -
  - Insertion and deletion - Unlike arrays, insertion, and deletion in linked list is easier.
    - Array elements are stored in the consecutive location, whereas the elements in the linked list are stored at a random location.
    - To insert or delete an element in an array, we have to shift the elements for creating the space. Whereas, in linked list, instead of shifting, we just have to update the address of the pointer of the node.

Disadvantages of Linked list

- The limitations of using the Linked list are given as follows -
  - Memory usage - In linked list, node occupies more memory than array.
    - Each node of the linked list occupies two types of variables, i.e., one is a simple variable, and another one is the pointer variable.
  - Traversal - Traversal is not easy in the linked list.
    - If we have to access an element in the linked list, we cannot access it randomly, while in case of array we can randomly access it by index.
    - For example, if we want to access the 3rd node, then we need to traverse all the nodes before it. So, the *time required to access a particular node is large.*

Applications of Linked list

- The applications of the Linked list are given as follows -
  - A linked list can be used to represent the polynomials and to perform the operations on the polynomial.
  - A linked list can be used to represent the sparse matrix.
  - The various operations like student's details, employee's details, or product details can be implemented using the linked list as the linked list uses the structure data type that can hold different data types.
  - Using linked list, we can implement stack, queue, tree, and other various data structures.
  - A linked list can be used to implement dynamic memory allocation.
    - The dynamic memory allocation is the memory allocation done at the run-time.

Operations performed on Linked list

- The basic operations that are supported by a list are mentioned as follows -
  - Creation – Creating or defining a list (with elements).
  - Insertion - This operation is performed to add an element into the list.
  - Deletion - It is performed to delete an operation from the list.
  - Display - It is performed to display the elements of the list.
  - Search - It is performed to search an element from the list using the given key.

Complexity of Linked list

The time complexity of the linked list for the operations search, insert, and delete.

Operation	Average case time complexity	Worst case time complexity
Insertion	O(1)	O(1)
Deletion	O(1)	O(1)
Search	O(n)	O(n)

How to declare a linked list?

- It is simple to declare an array, as it is of single type, while the declaration of linked list is a bit more typical than array.
- Linked list contains two parts, and both are of different types, i.e., one is the simple variable, while another is the pointer variable.
- We can declare the linked list by using the user-defined data type structure.

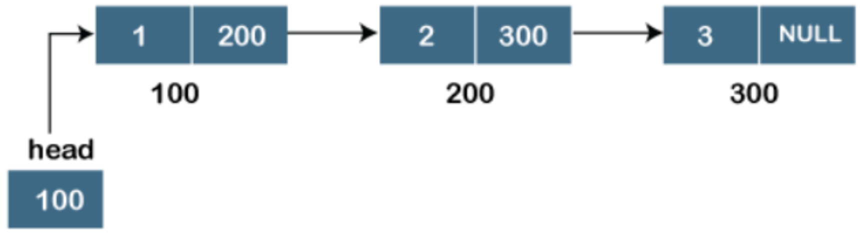
```
struct node
{
int data;
struct node *next;
}
```

we have defined a structure named as node that contains two variables, one is data that is of integer type, and another one is next that is a pointer which contains the address of next node.

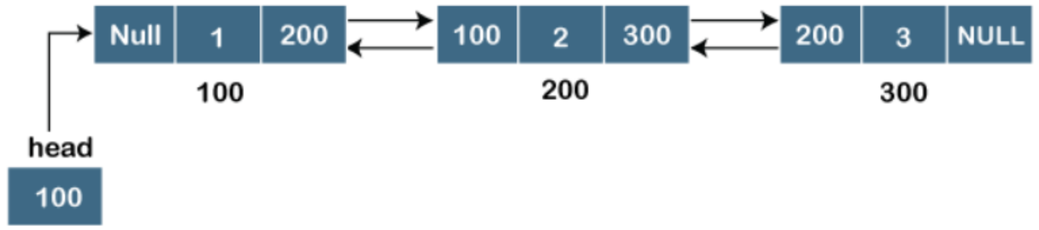
Types of linked Lists

There are three basic types of linked list

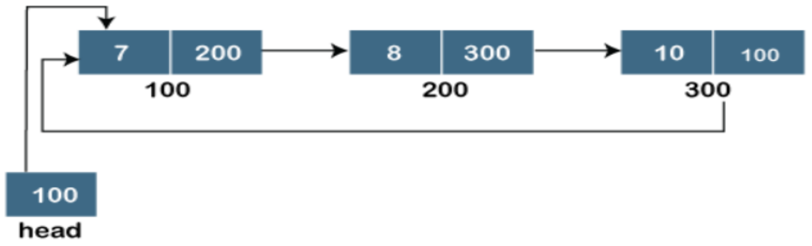
- Singly Linked list:- Navigation is forward only



- Doubly linked list:- Forward and backward navigation is possible

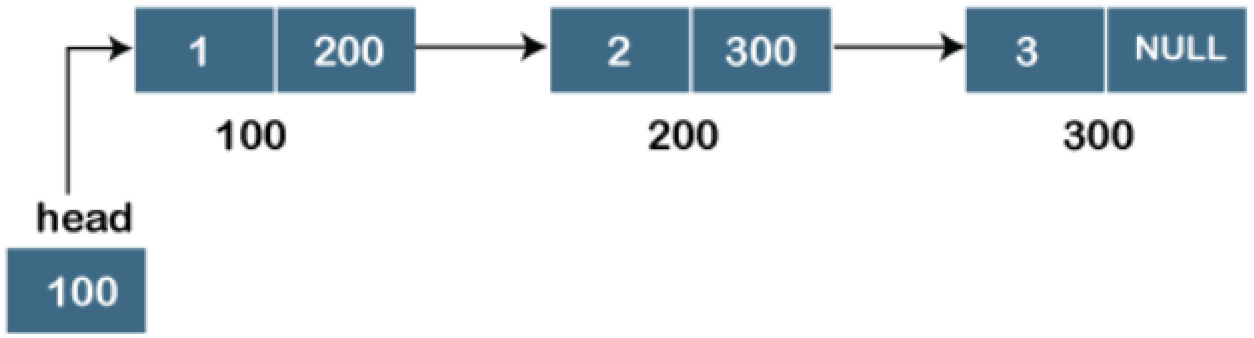


- Circular linked list:- last element is linked to the first element



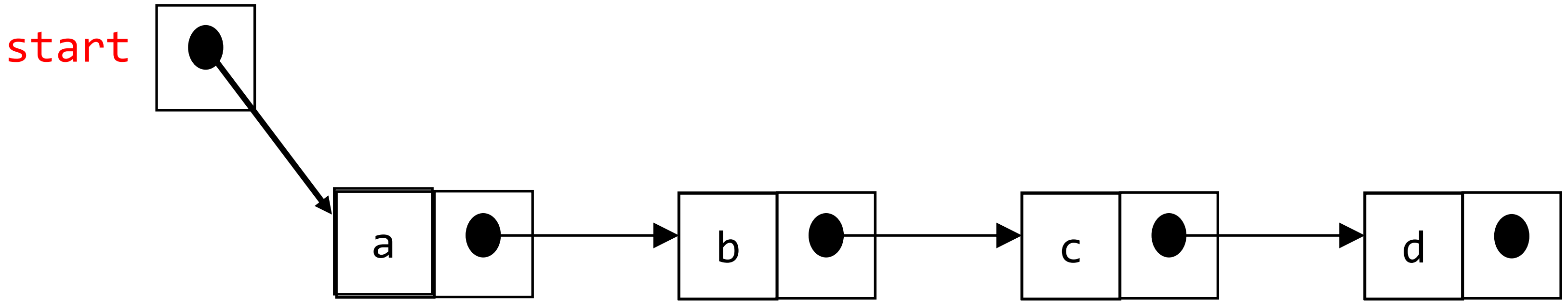
Singly Linked List

- Each node has only one link part.
- Each link part contains the address of the next node in the list.
- Link part of the last node contains NULL value (special value) which signifies the end of the node.



Schematic Representation

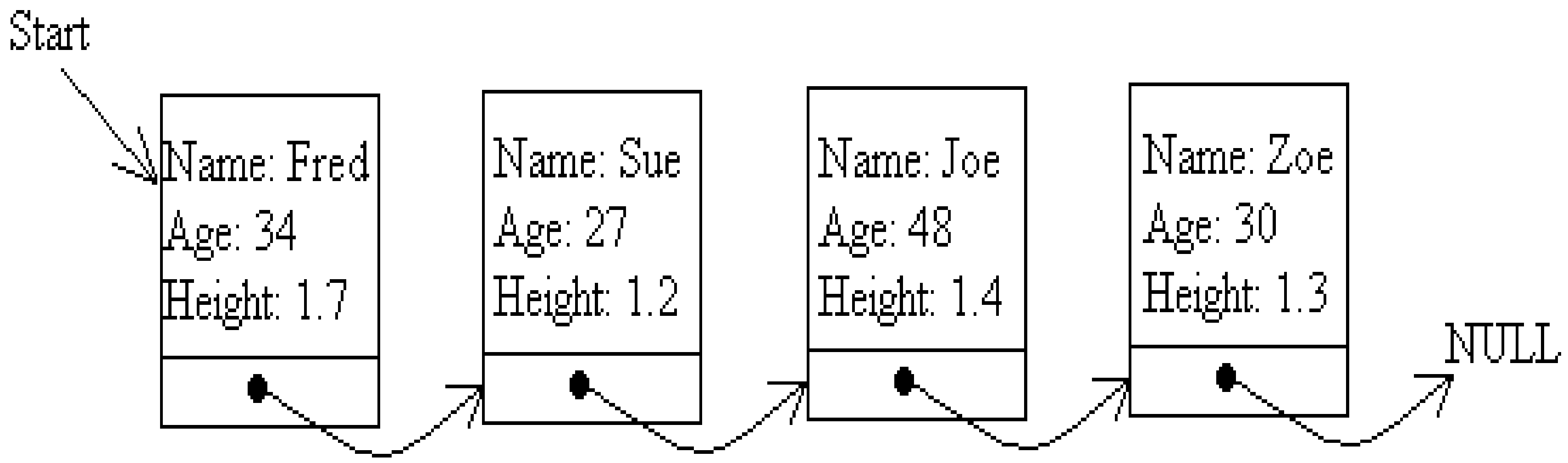
- Here is a singly-linked list (SLL):



- Each node contains a value(data) and a pointer to the next node in the list.

- Start is the header pointer which points at the first node in the list.

Example



- This linked list has four nodes in it, each with a link to the next node in the series (in the linked list).

Defining(Creating) the data structure for SLL

```
struct student
 {
     char name[20];
     int age;
     char Dept[20];
     struct student *next;
 };
     struct student *start = NULL;
```

Inserting a node in a SLL

- First we declare the space for a pointer item and assign a pointer (*p) to it.
- This is done using the new statement as follow

```
 struct student
    {
        char name[20];
        int age;
        char Dept[20];
         struct student *next;
    };
    struct student *start = NULL;
    struct student *p;
    p = new student;
```

Inserting a node in a SLL

- Having declared the node, we ask the user to fill in the details of the student i.e. the name, age and department or others…

```
cout<<"Enter the name of the student:n";
cin>>p->name;
cout<<"Enter the age of the student:\n";
cin>>p->age
cout<<"Enter the dept of the student:\n";
cin>>p->Dept;
p->next= NULL;//if the node is to be inserted at the end
```

Inserting a node in a SLL

- There are 3 cases here:-
  - Insertion at the beginning
    - It involves inserting any element at the front of the list.
    - We just need to a few link adjustments to make the new node as the head of the list.
  - Insertion at the end of the list
    - It involves insertion at the last of the linked list.
    - The new node can be inserted as the only node in the list or it can be inserted as the last one.
    - Different logics are implemented in each scenario.
  - Insertion after a particular node
    - It involves insertion after the specified node of the linked list.
    - We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted.

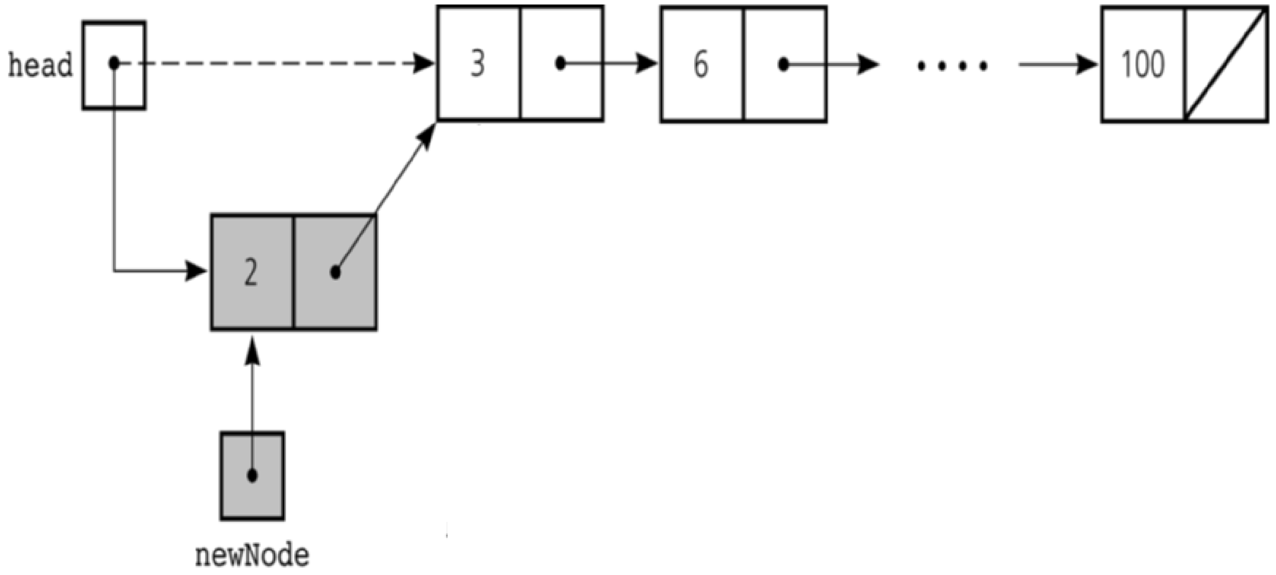
Insertion at the beginning

There are two steps to be followed:-

a) Make the next pointer of the new node point towards the first node of the list.

b) Make the start pointer point towards this new node.

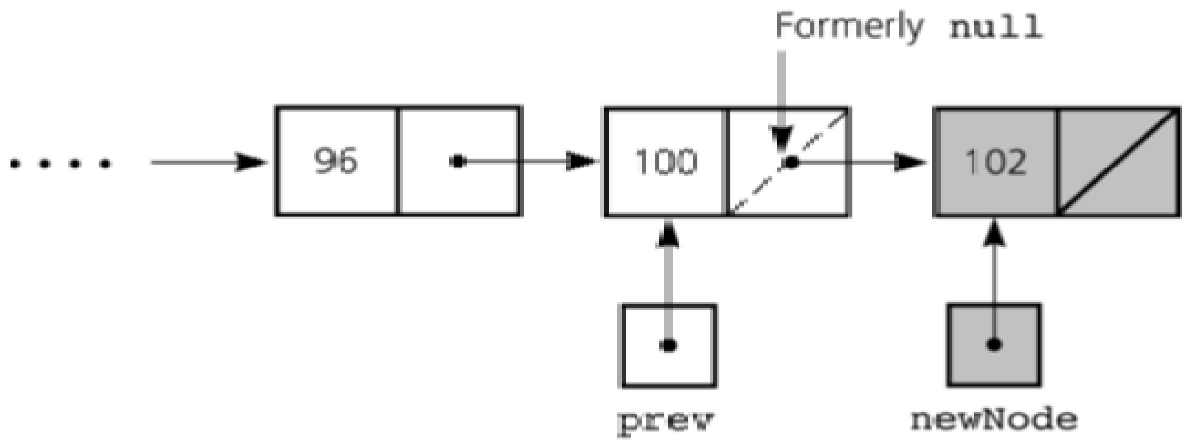
- If the list is empty, simply make the start pointer point towards the new node;



```
void insert_beg(student *p)
{
    student *temp;
    if(start==NULL) //if the list is empty
    {
       start=p;
       cout<<"Node inserted at the beginning";
    }
    else
    {
     temp=start;
     start=p;
     p->next=temp;//making new node point at the first node of the list
    }
}
```

Inserting at the end of the list

- Here we simply need to make the next pointer of the last node point to the new node.
- To do this we need to declare a second pointer, q, to step through the list until it finds the last node



```
void insert_end(student *p)
{
   student *q=start;
   if(start==NULL)
   {
    start=p;
    cout<<"Node inserted at the end…!\n";
   }
   else
   {// the loop terminate when q points to the last node
      while(q->next!=NULL)
               q=q->next;

      q->next=p;//the new node is made the last node
    }
  }
```

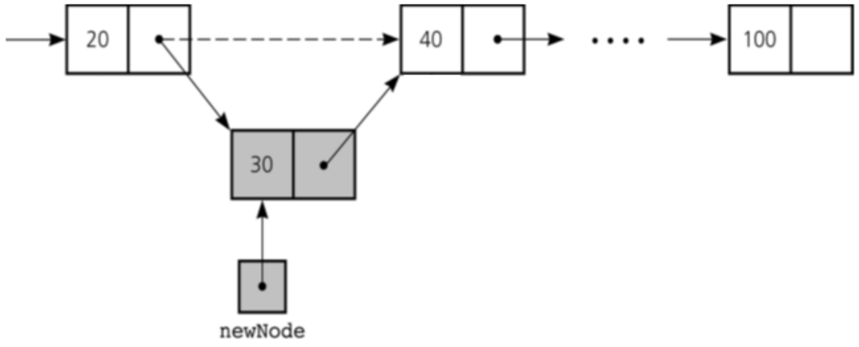
Inserting after a specified node

Here we again need to do 2 steps :-

- Make the next pointer of the node to be inserted point to the next node of the node after which you want to insert the node.
- Make the next pointer of the node after which the node is to be inserted, point to the node to be inserted.

or

1. Create a Node
2. Set the node data Values
3. Break pointer connection
4. Re-connect the pointers



```
void insert_after(int c, student *p){
   student *q;
   q=start;
   for(int i=1;i<c; i++)
            q=q->next;
   if(q==NULL)
      cout<<”Less than “<<c<<” nodes…!”;
   p->next=q->next;
   q->next=p;
   cout<<”Node inserted successfully”;
}
```

Displaying the list of nodes

- Having added one or more nodes, we need to display the list of nodes on the screen using the following steps

Steps

1. Set a temporary pointer to point to the same thing as the start pointer
2. If the pointer points to NULL, display the message “End of list” and stop
3. Otherwise, display the details of the node pointed by the start node
4. Make the temporary pointer point to the same thing as the next pointer of the node it is currently indicating
5. Jump back to step 2

```
void display()
{
 student *q=start;
 do
   { if(q==NULL)
        cout<<"End of List\n";
     else
     { // Display details for what q points to
        cout<<"Name:"<<q->name<<endl;
        cout<<"Age:"<<q->age<<endl;
        cout<<"Department:"<<q->Dept<<endl;
        // Move to next node (if present)
        q=q->next;
     }
   }while(q!=NULL);
}
```

Navigating/Traversing through the list

To Move Forward:

1. Set a pointer to point to the same thing as the start pointer.
2. If the pointer points to NULL, display the message “list is empty" and stop.
3. Otherwise, move to the next node by making the pointer point to the same thing as the next pointer of the node it is currently indicating (using current pointer).

Navigating/Traversing through the list

To Move to Backward: (single linked list)

1. Set a pointer to point to the same thing as the start pointer.

2. If the pointer points to NULL, display the message “list is empty" and stop.

3. Set a new pointer (previous) and assign the same value as start pointer and move forward until you find the node before the one we are considering at the moment(using current pointer).

4. Set current pointer equal to the new pointer (previous pointer).

Deleting a node in SLL

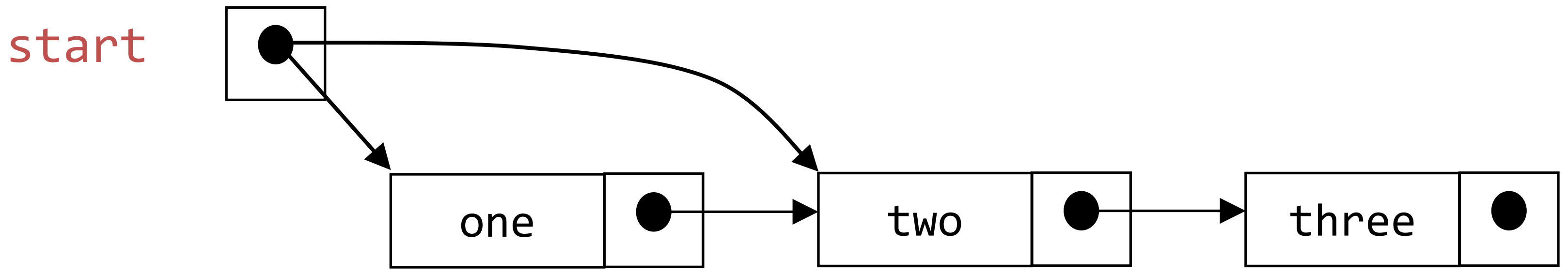
Here also we have three cases:-

- Deleting the first node
    - It involves deletion of a node from the beginning of the list.
    - This is the simplest operation among all.
    - It just need a few adjustments in the node pointers.
- Deleting the last node
    - It involves deleting the last node of the list.
    - The list can either be empty or full.
    - Different logic is implemented for the different scenarios.
- Deleting after a specified node
    - It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted.
    - This requires traversing through the list.

Deleting the first node

Here we apply 2 steps:-

- Making the start pointer point towards the 2nd node.
- Deleting the first node using delete keyword.

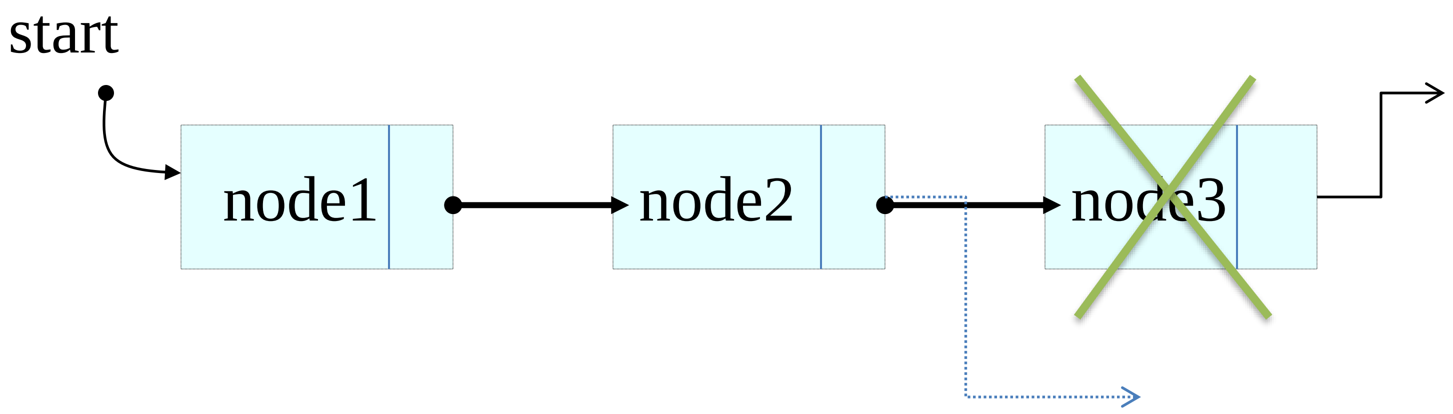


```
void del_first(){

   if(start==NULL)
      cout<<"\n Error……List is empty\n";

   else
     {
        student *temp=start;
        start=temp->next;
        delete temp;
        cout<<"\n First node deleted!";
      }
}
```

Deleting the last node

Here we apply 2 steps:-

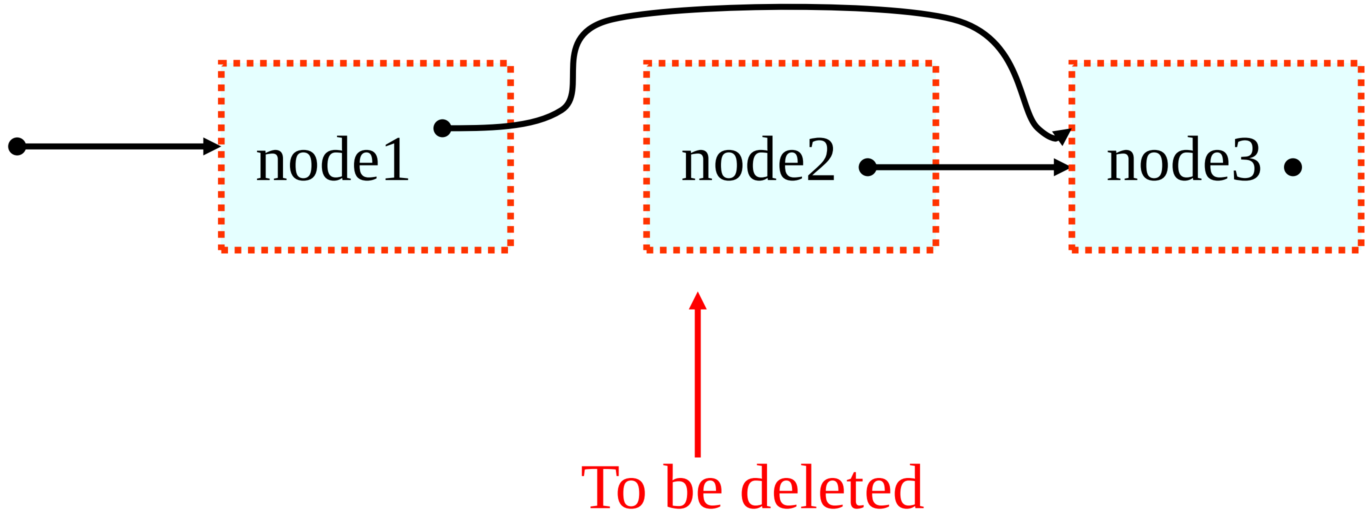
- Making the second last node's next pointer point to NULL
- Deleting the last node via delete keyword



```
void del_last()
{
      if(start==NULL)
          cout<<"\n Error….List is empty";
     else
           {
              student *q=start;
               while(q->next->next!=NULL)
                             q=q->next;
               student *temp=q->next;
                     q->next=NULL;
              delete temp;
             cout<<"\n Deleted successfully…";
         }
 }
```

Deleting a particular node

- Here we make the next pointer of the node previous to the node being deleted, point to the successor node of the node to be deleted and then delete the node using delete keyword.



```
void del(int c)
{
   node *q=start;
   for(int i=1;i<c; i++)
    {
         q=q->next;
         if(q==NULL)
         cout<<"\n Node not found\n";
    }
    if(i==c)
    {
        node *p=q->next; //node to be deleted
        q->next=p->next;//disconnecting the node p
        delete p;
        cout<<"Deleted Successfully";
    }
}
```

Video Animation

Arrays Vs Linked Lists

Arrays	Linked list
Fixed size: Resizing is expensive	Dynamic size
Insertions and Deletions are inefficient: Elements are usually shifted	Insertions and Deletions are efficient: No shifting
Random access i.e., efficient indexing	No random access → Not suitable for operations requiring accessing elements by index such as sorting
No memory waste if the array is full or almost full; otherwise may result in much memory waste.	Since memory is allocated dynamically(acc. to our need) there is no waste of memory.
Sequential access is faster [Reason: Elements in contiguous memory locations]	Sequential access is slow [Reason: Elements not in contiguous memory locations]

COMPLEXITY OF VARIOUS OPERATIONS IN ARRAYS AND SLL

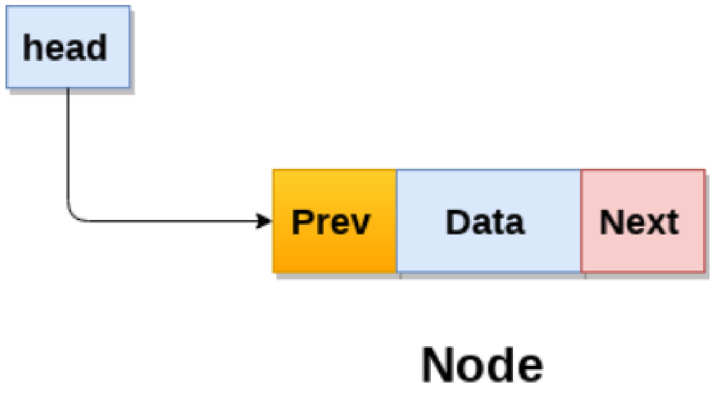
Operation	ID-Array Complexity	Singly-linked list Complexity
Insert at beginning	O(n)	O(1)
Insert at end	O(1)	O(1) if the list has **tail** reference O(n) if the list has no **tail** reference
Insert at middle	O(n)	O(n)
Delete at beginning	O(n)	O(1)
Delete at end	O(1)	O(n)
Delete at middle	O(n): O(1) access followed by O(n) shift	O(n): O(n) search, followed by O(1) delete
Search	O(n) linear search O(log n) Binary search	O(n)
Indexing: What is the element at a given position k?	O(1)	O(n)

Doubly linked lists

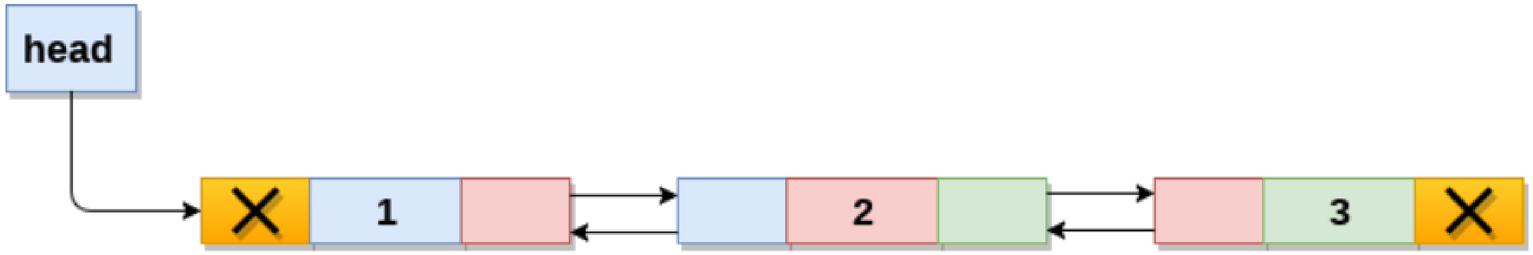
- Each node points not only to Successor node (Next node), but also to Predecessor node (Previous node).
- There are two NULL: at the first and last nodes in the linked list.
- Therefore, in a doubly linked list, a node consists of three parts:
  - node data
  - pointer to the next node in sequence (next pointer)
  - pointer to the previous node (previous pointer)

Doubly linked lists

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

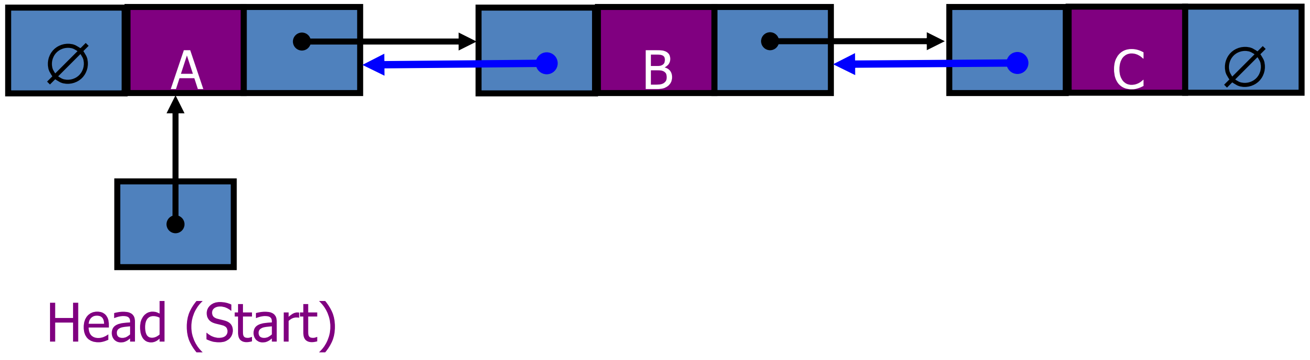


- A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.



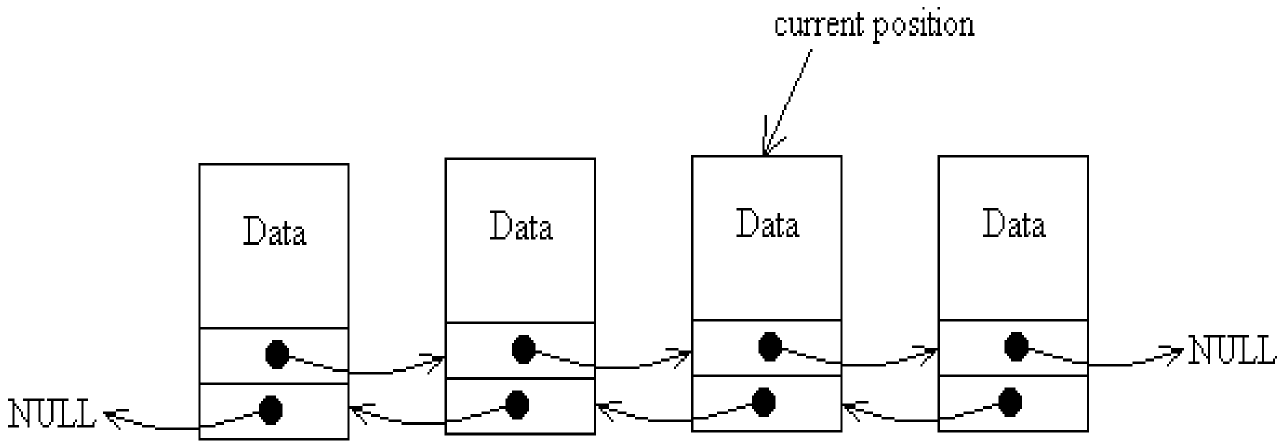
Doubly linked lists

- Advantage: given a node,
  - It is easy to visit its predecessor (previous) node.
  - It is convenient to traverse linked lists Forwards and Backwards.



- It is not necessary to have start pointer, we can have any pointer(current) pointing to one of the node in the list

Doubly linked lists



Here, there is no pointer to the start of the list, there is simply a pointer to some position in the list that can be moved left or right.

The reason we needed a start pointer in the ordinary linked list is because, having moved on from one node to another, we can't easily move back, so without the start pointer, we would lose track of all the nodes in the list that we have already passed.

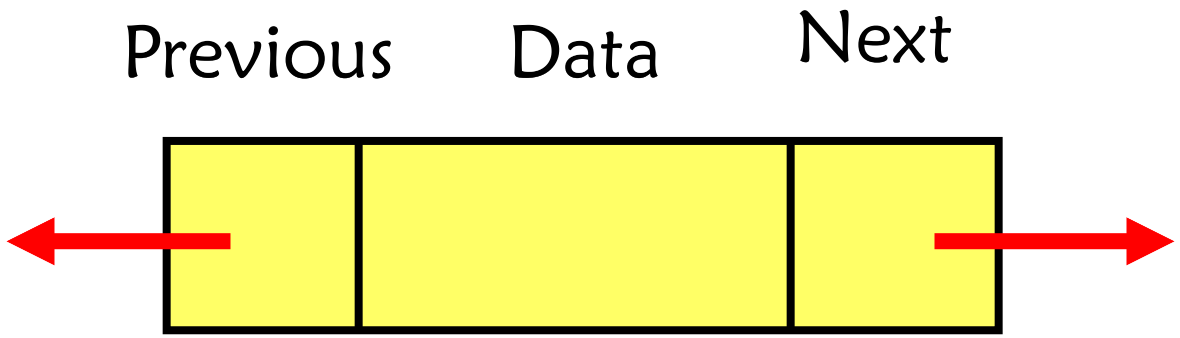
With the doubly linked list, we can move the current pointer backwards and forwards at will.

DLL's compared to SLL's

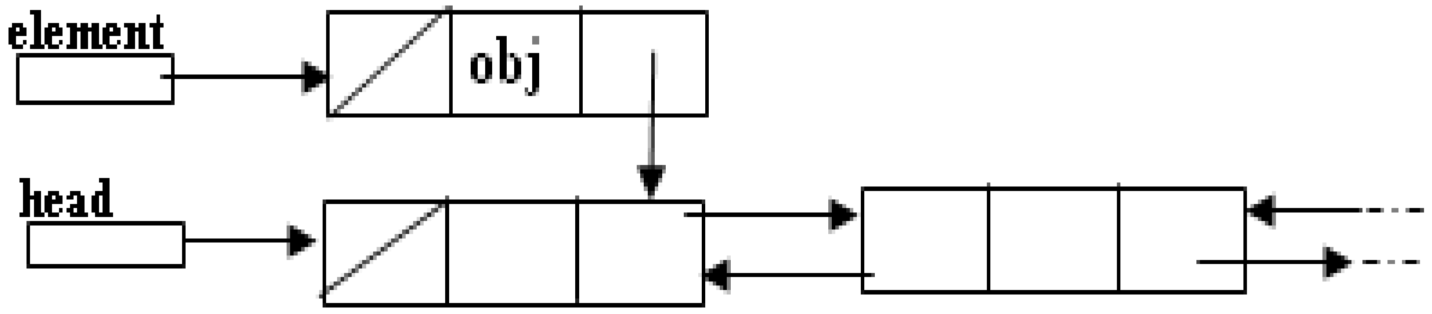
- **Advantages:**
  - Can be traversed in either direction (may be essential for some programs).
  - Some operations, such as deletion and inserting before a node, become easier.
- **Disadvantages:**
  - Requires more space.
  - List manipulations are slower (because more links must be changed).
  - Greater chance of having bugs (because more links must be manipulated).

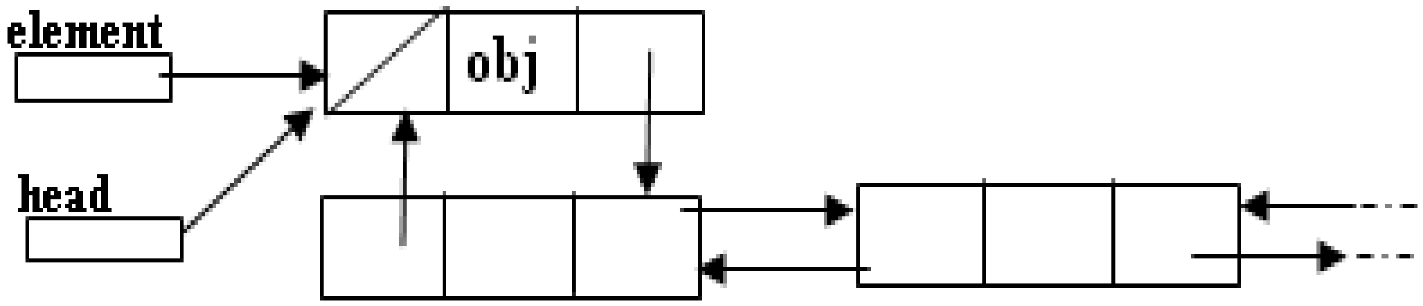
Structure of DLL

```
struct student
 {
  char name[20];
  int age;
  struct node *next;
  struct node *previous;//holds the address of
 previous node
 };
```



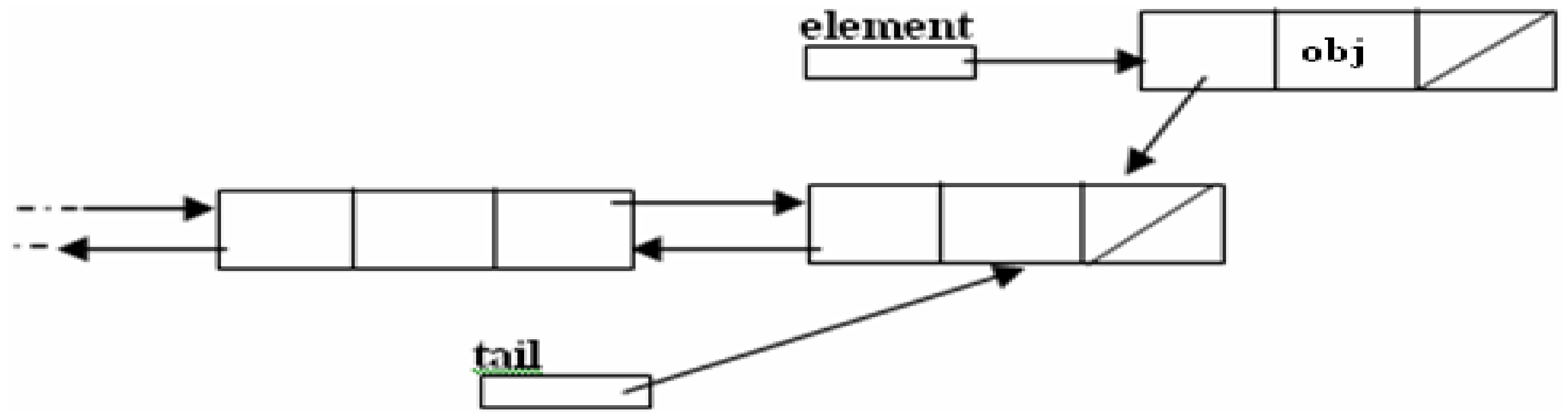
Inserting at the beginning

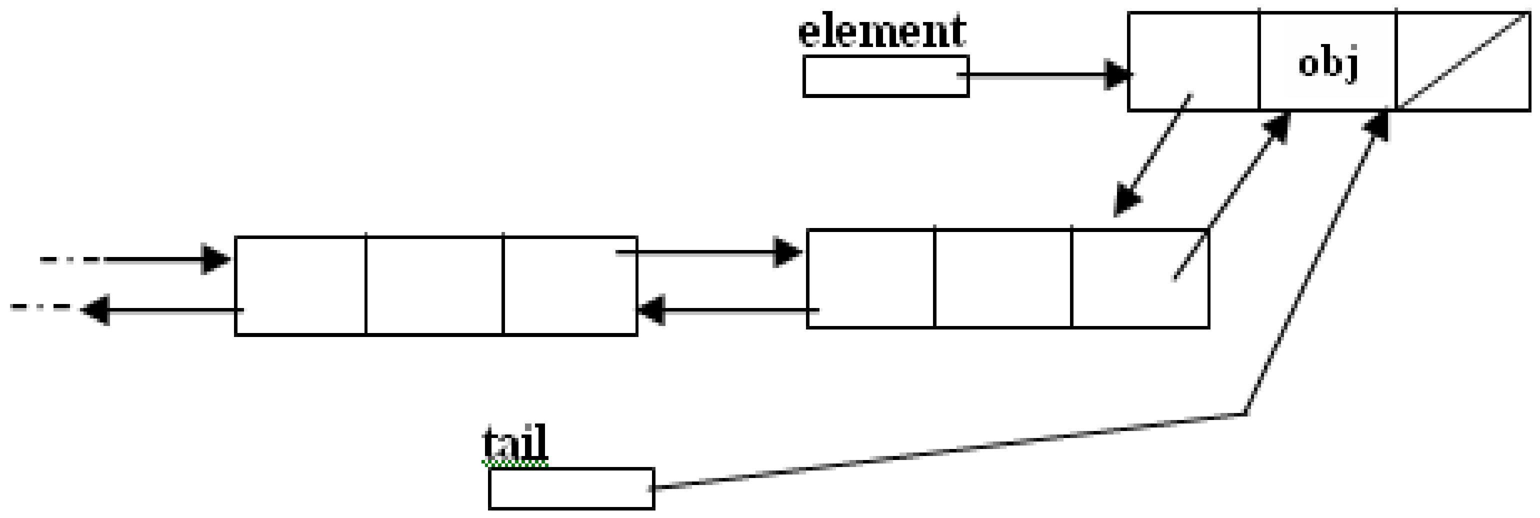




```
void insert_beg(student  *p)
{
     if(start==NULL)
     {
       start=p;
       cout<<"Node inserted successfully at the beginning";
     }
     else
     {
      student * temp=start;
      start=p;
      temp->previous=p;  //making 1st node's previous point to the new node
      p->next=temp; //making next of the new node point to the 1st node
      cout<<"\nNode inserted successfully at the beginning\n";
   }
}
```

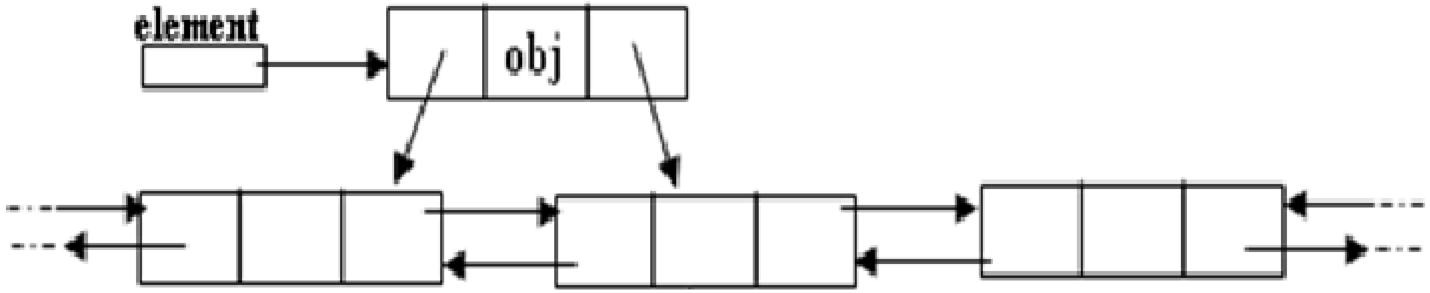
Inserting at the end of the list



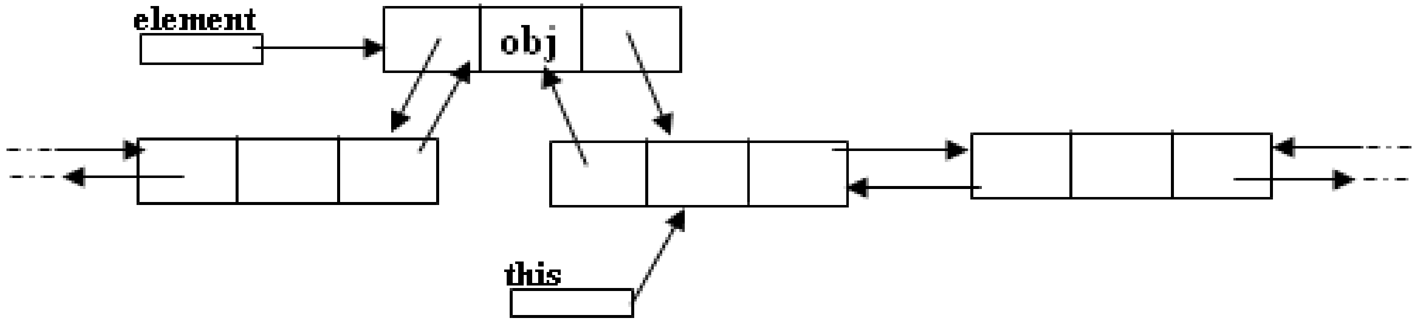


```
void insert_end(student *p)
{
     if(start==NULL)
     {
        start=p;
        cout<<"\nNode inserted successfully at the end";
     }
     else
     {
         student *temp=start;
         while(temp->next!=NULL)
          {
            temp=temp->next;
          }
           temp->next=p;
           p->previous=temp;
           cout<<"\nNode inserted successfully at the end\n";
       }
  }
```

Inserting after a specified node



Making next and previous pointer of the node to be inserted point accordingly



Adjusting the next and previous pointers of the nodes b/w which the new node accordingly

```
void insert_after(int c, node *p)
{
    temp=start;
    for(int i=1;i<c-1;i++)
    {
      temp=temp->next;
    }
      p->next=temp->next;
      temp->next->previous=p;
      temp->next=p;
      p->previous=temp;
      cout<<"\nInserted successfully";
}
```

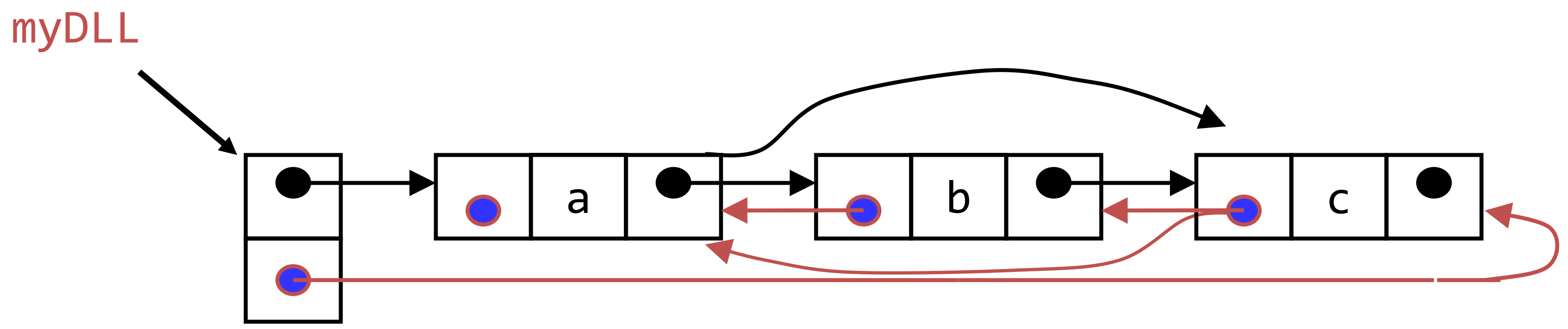
Navigating through DDL

To Move to Backward:

1. Set a pointer to point to the same thing as the start pointer.
2. If the pointer points to NULL, display the message “list is empty" and stop.
3. Otherwise, move back to the previous node by making the pointer point to the same thing as the previous pointer of the node it is currently indicating.

Deleting a node

- Node deletion from a DLL involves changing *two* links
- In this example, we will delete node b

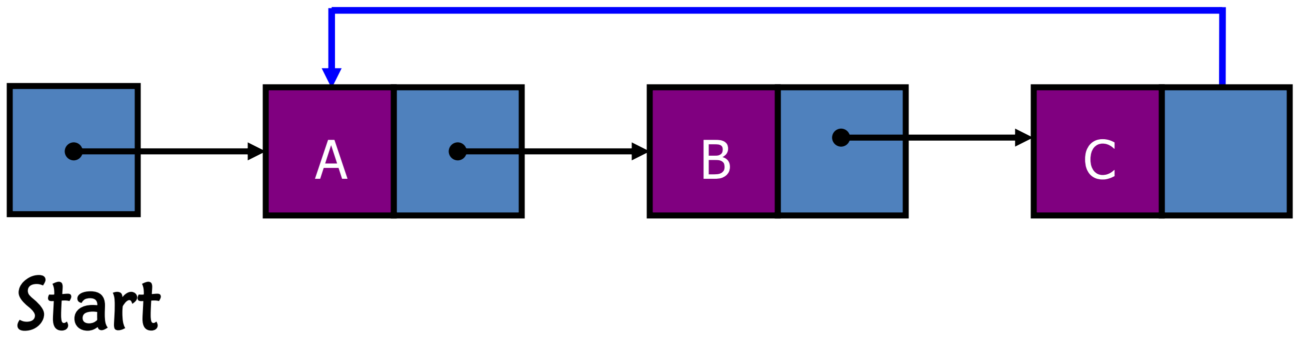


- Deletion of the first node or the last node is a special case

```
void del_at(int c)
{
    node *s=start;
      {
           for(int i=1;i<c-1;i++)
           {
            s=s->next;
           }
    node *p=s->next;
    s->next=p->next;
    p->next->previous=s;
    delete p;
    cout<<"\nNode number "<<c<<" deleted successfully";
      }
   }
}
```

Variations of Linked Lists

- Circular linked lists
  - The last node points to the first node of the list.



  - How do we know when we have finished traversing the list?
  - (<u>Hint:</u> check if the pointer of the current node is equal to the Start (head) pointer).

Exercises

1. Write a C++ program using single and double linked list data structure that keeps track of student record. Each student has Name, ID, Age, and Department. The program should include the following operations.
   a) Add a new student data from the keyboard(front, middle, last)
   b) Delete a node (the first, last, middle)
   c) Display the whole list of students
   d) Search a specific node

Thank You

Question?