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

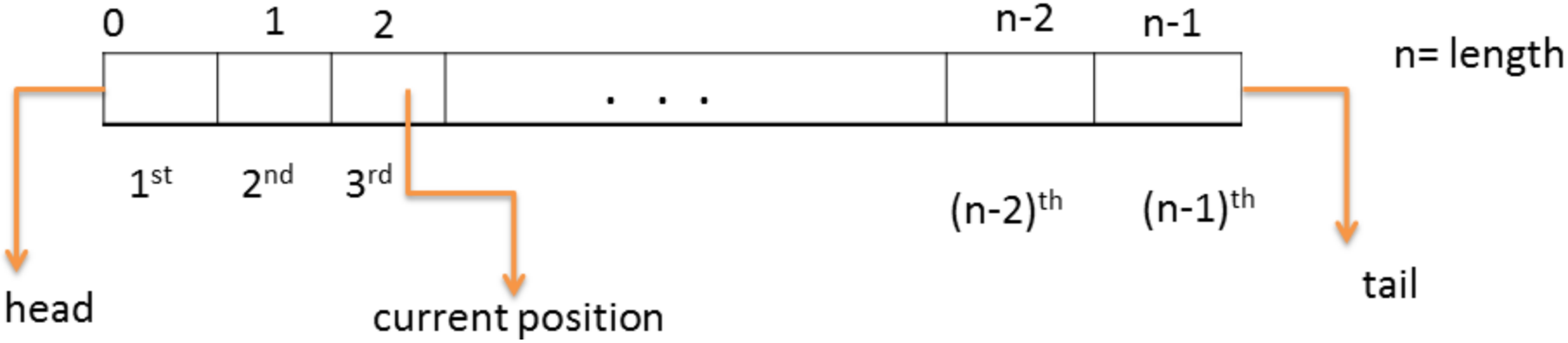
- Processing objects in the reverse of the order that they arrived.
- For all these situations, a simple list structure is appropriate.

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).

Cont'd

- 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.



Cont'd

- **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.

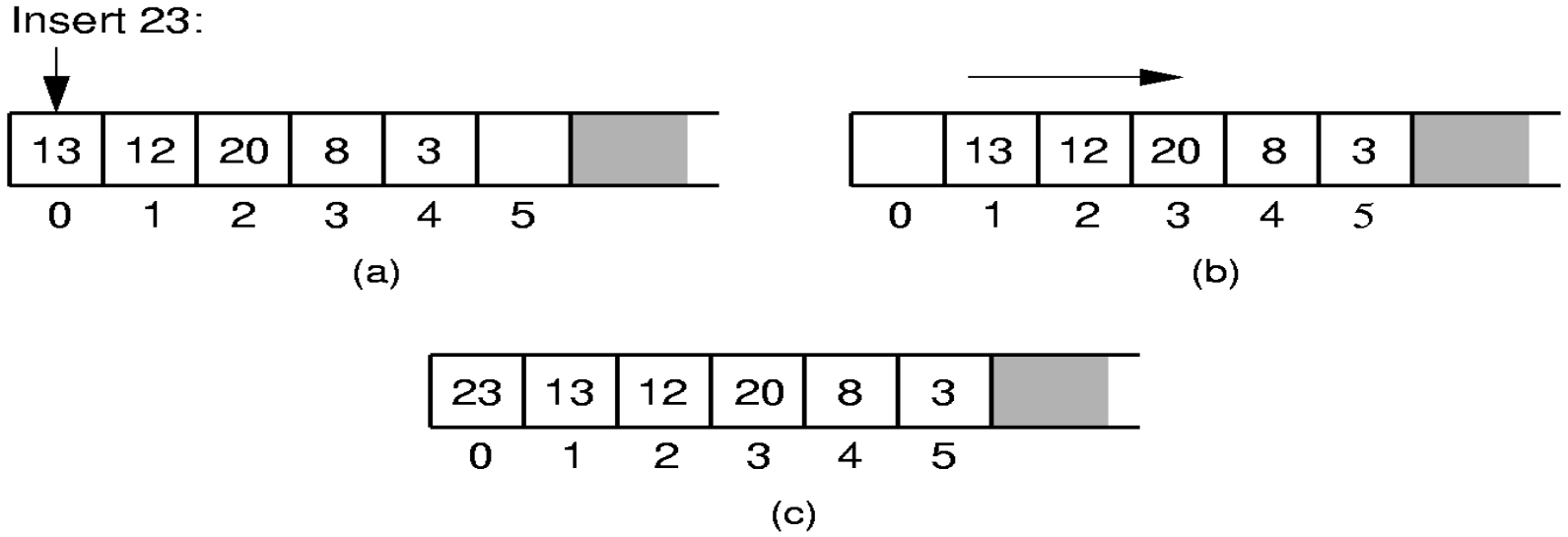
  Example: Linked list implementation of ADTs.

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 head of an array-based list requires shifting all existing elements in the array by one position toward the tail.

Array-Based List Class (1)

```
//Array-based list
class AList : public List{
private:
  int maxSize;      // Maximum size of list
  int listSize;     // Actual elem count
  int curr;        // Position of curr
  datatype listArray; // Array holding list
public:
  AList(int size = DefaultListSize){// Constructor
    maxSize = size;
    listSize = curr = 0;
    listArray = new datatype[maxSize];
  }
  ~AList() { delete [] listArray; }// Destructor
```

Array-Based List Class (2)

```
void clear() { //Reinitialize the list
  delete [] listArray;
  listSize = curr = 0;
  listArray = new datatype[maxSize];
}
void setStart() { curr = 0; }
void setEnd() { curr = listSize; }
void prev()   { if (curr != 0) curr--; }
void next()   { if (curr <= listSize)
                  curr++; }
int length() { return listSize; }

bool setPos(int pos) {
  if ((pos >= 0) && (pos <= listSize))
    curr = pos;
  return (pos >= 0) && (pos <= listSize);
}
```

Array-Based List Class (3)

```
//get current value
bool getValue(){
  if (listSize == 0)return false;
  else{
    it = listArray[curr];
    return true;}
  }
// Insert at current position
bool insert(datatype item) {
  if (listSize == maxSize) return false;
  for(int i=listSize; i>curr; i--)
    // Shift Elems up to make room
    listArray[i] = listArray[i-1];

  listArray[curr] = item;
  listSize++; // Increment list size
  return true;
}
```

Array-Based List Class (4)

```
// Append element to end of the list
bool append(datatype item) {
  if (listSize == maxSize) return false;
  listArray[listSize++] = item;
  return true;
}
// Remove and return curr element
bool remove() {
  if (length() == 0) return false;
  it = listArray[curr]; // Copy element
  for(int i=curr; i<listSize-1; i++)
    // Shift them down
    listArray[i] = listArray[i+1];
  listSize--;    // Decrement size
  return true;
 }
……
}
```

Revision to 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.**

Declaration of structure

- 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.

Example:

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

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

Self-Referential Structures

- Structures can hold pointers to instances of themselves.

Example:

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

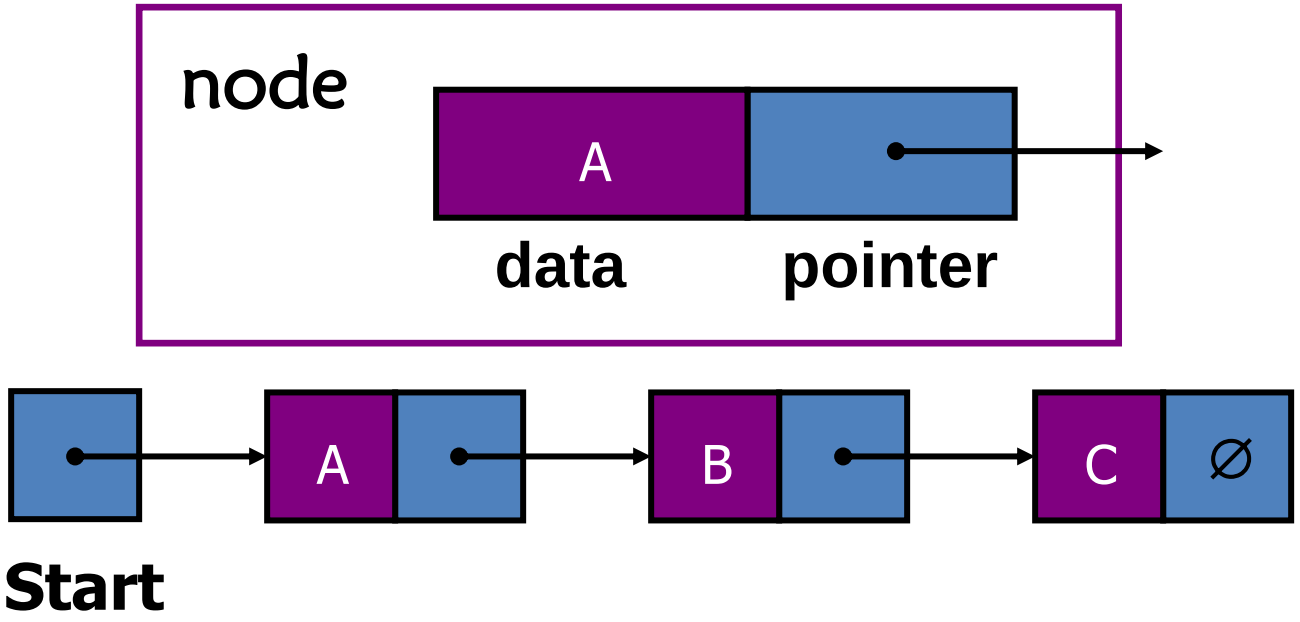
Linked List- Based List Implementation

- The second approach for implementing lists makes use of pointers and is called a linked list.
- The linked list uses dynamic memory allocation, i.e. it allocates memory for new list elements as needed
- Is self-referential structure.
- Is a collection of elements called **nodes,** each of which stores two types of fields. **Data items** and a **pointer to next node.**

The data field: holds the actual elements on the list.

The pointer field: contains the address of the next node in the list.

- A linked list can be represented by a diagram like this one:

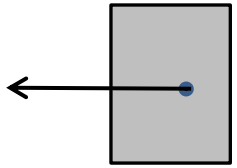


- **Start (Head)**: Special pointer that points to the first node of a linked list, so that we can keep track of the linked list.
- The last node should points to **NULL** to show that it is the last link in the chain (in the linked list).

Types of Lists

There are two basic types of linked list

- Singly Linked list
- Doubly linked list

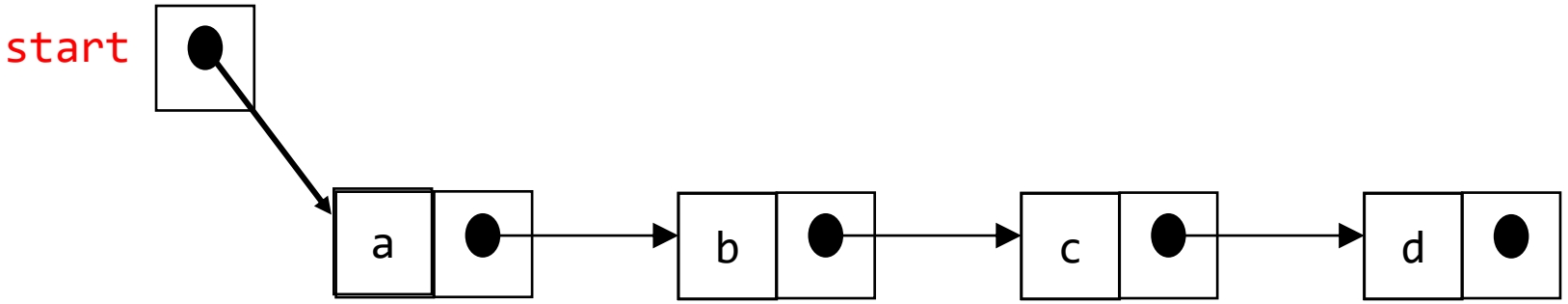


Singly Linked List

- 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

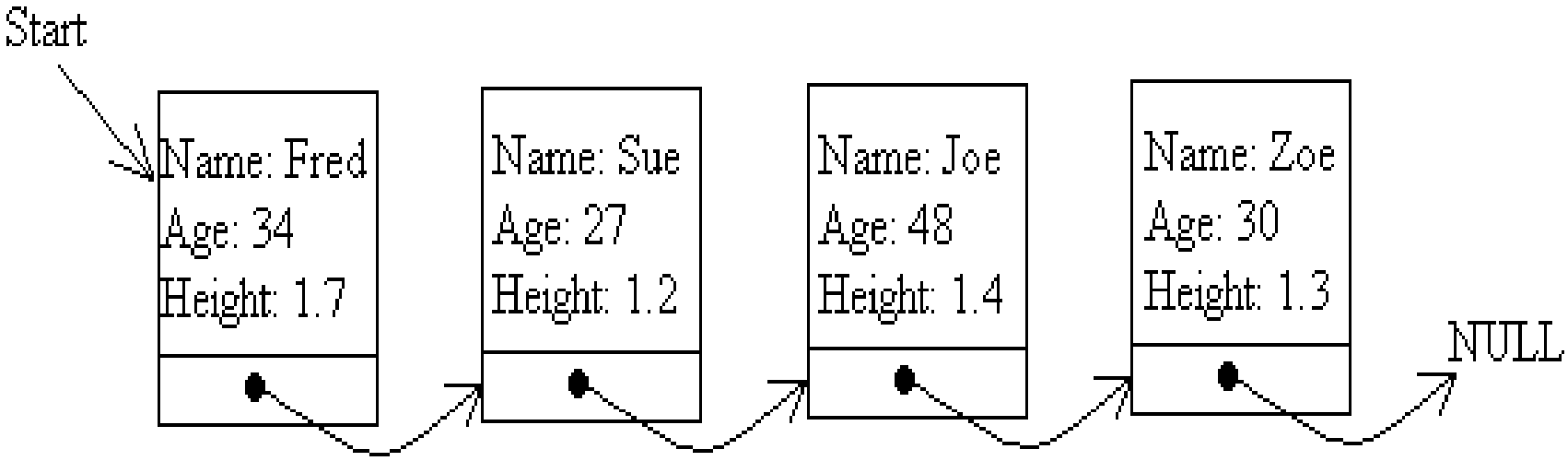
- Here is a singly-linked list (SLL):



- 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).

Basic Operations on a List

- Creating a List
- Inserting an element in a list
- Deleting an element from a list
- Searching a list

Defining(Creating) the data structure for linked lists

```
struct student
 {
    char name[20];
    int age;
    char Dept[20];
    student *next;
    student *prev;//Double linked list
 };
    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];
      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
- Insertion at the end
- Insertion after a particular node

Insertion at the beginning

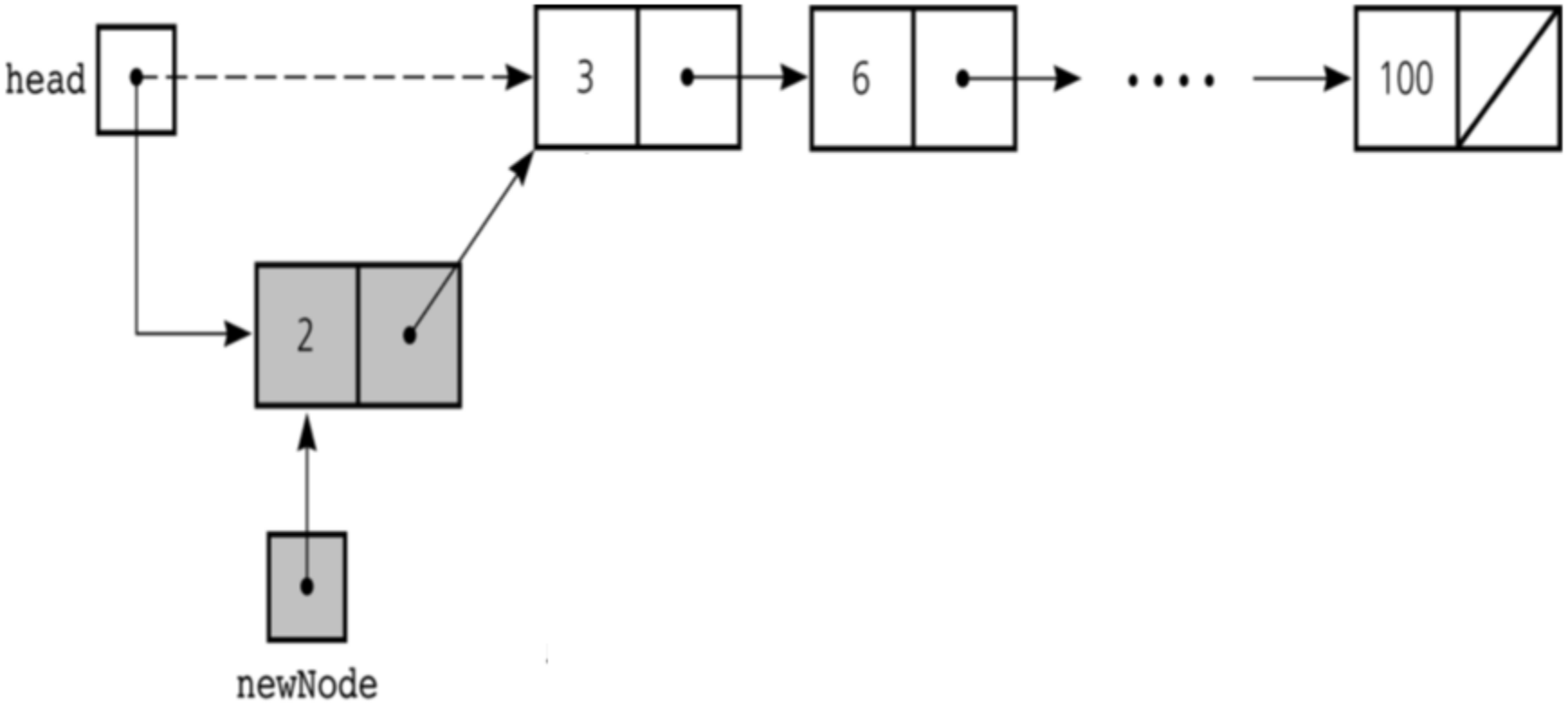
There are two steps to be followed:-

a) Make the next pointer of the 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;

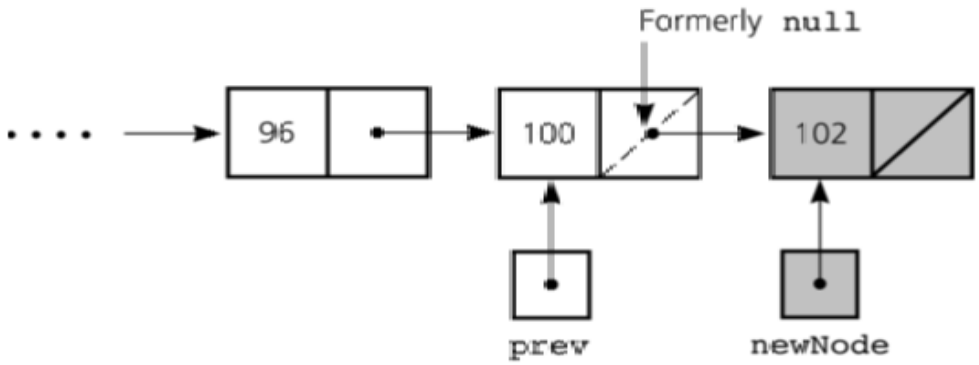
Cont…



```
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

- 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
   }
 }
```

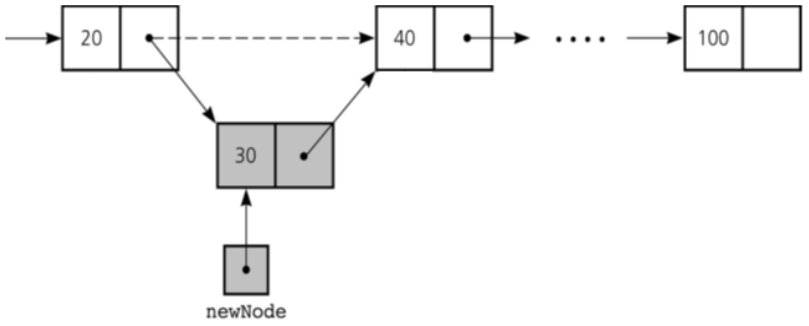
Inserting after an element

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).

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

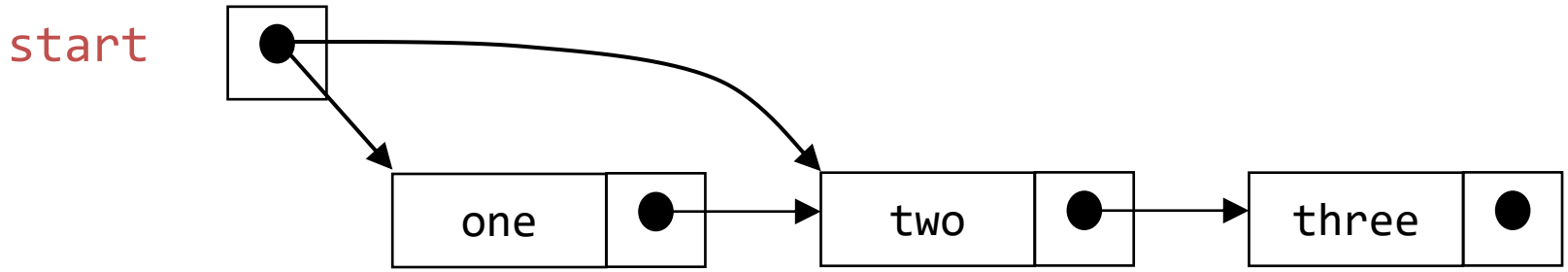
Here also we have three cases:-

- Deleting the first node
- Deleting the last node
- Deleting the intermediate node

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.

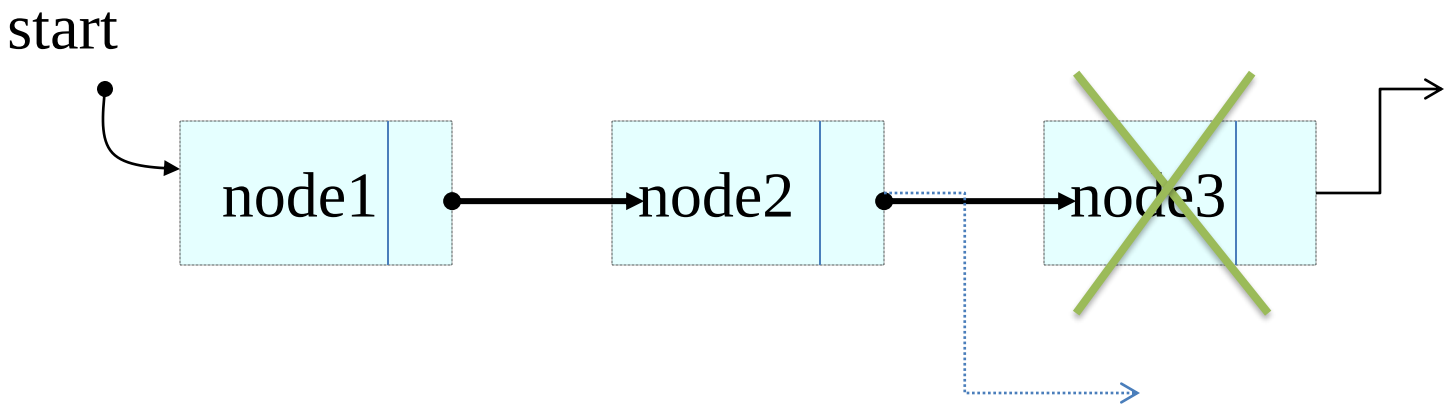


```
void del_first(){

   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:-

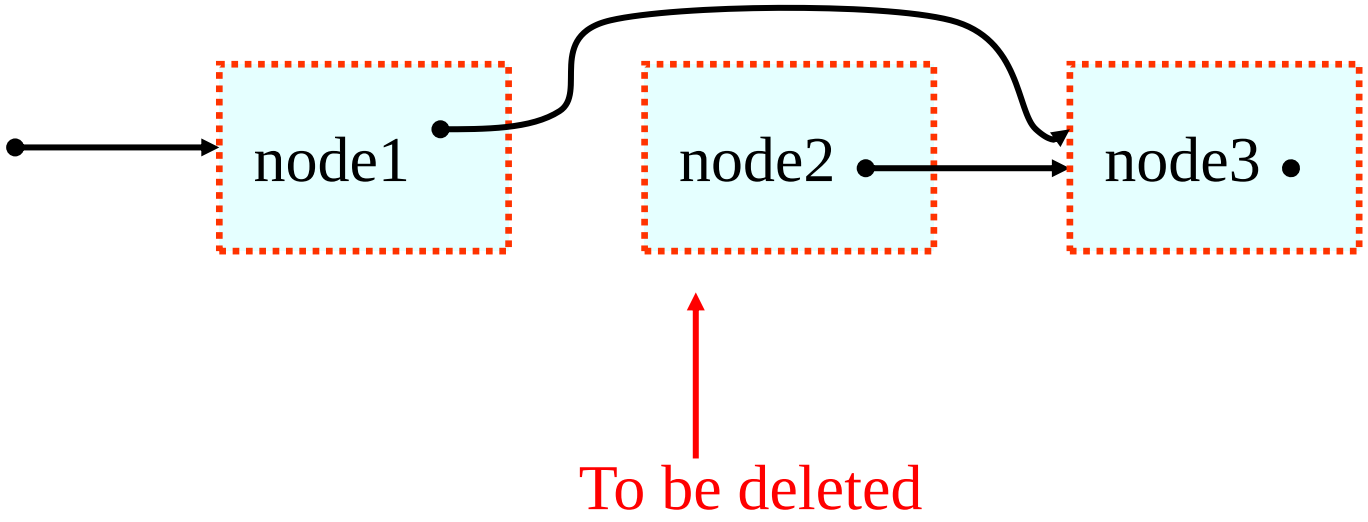
- Making the second last node's next pointer point to NULL
- Deleting the last node via delete keyword



```
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";
    }
}
```

Arrays Vs Linked Lists

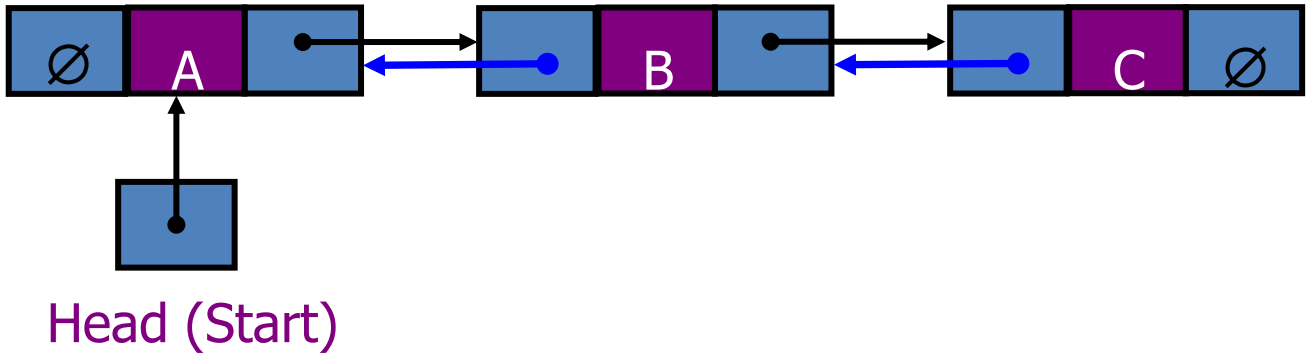
Arrays	Linked list
Fixed size: Resizing is expensive	Dynamic size
Insertions and Deletions are inefficient: Elements are usually shifted	Insertions and Deletions are efficient: No shifting
Random access i.e., efficient indexing	No random access → Not suitable for operations requiring accessing elements 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)

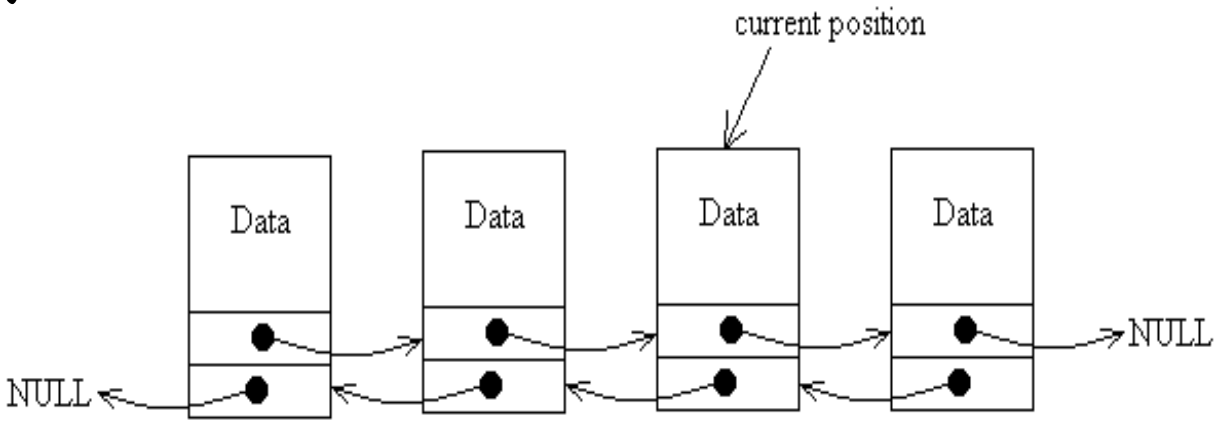
Doubly linked lists

- 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.
- 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.

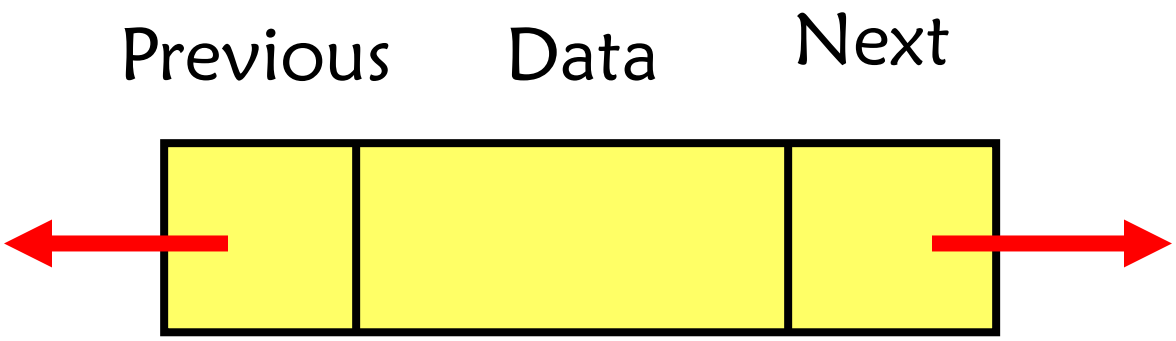
The reason we needed a start pointer in the ordinary linked list is because, having moved on from one node to another, we can't easily move back, so without the start pointer, we would lose track of all the nodes in the list that we have already passed. With the doubly linked list, we can move the current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

DLL's compared to SLL's

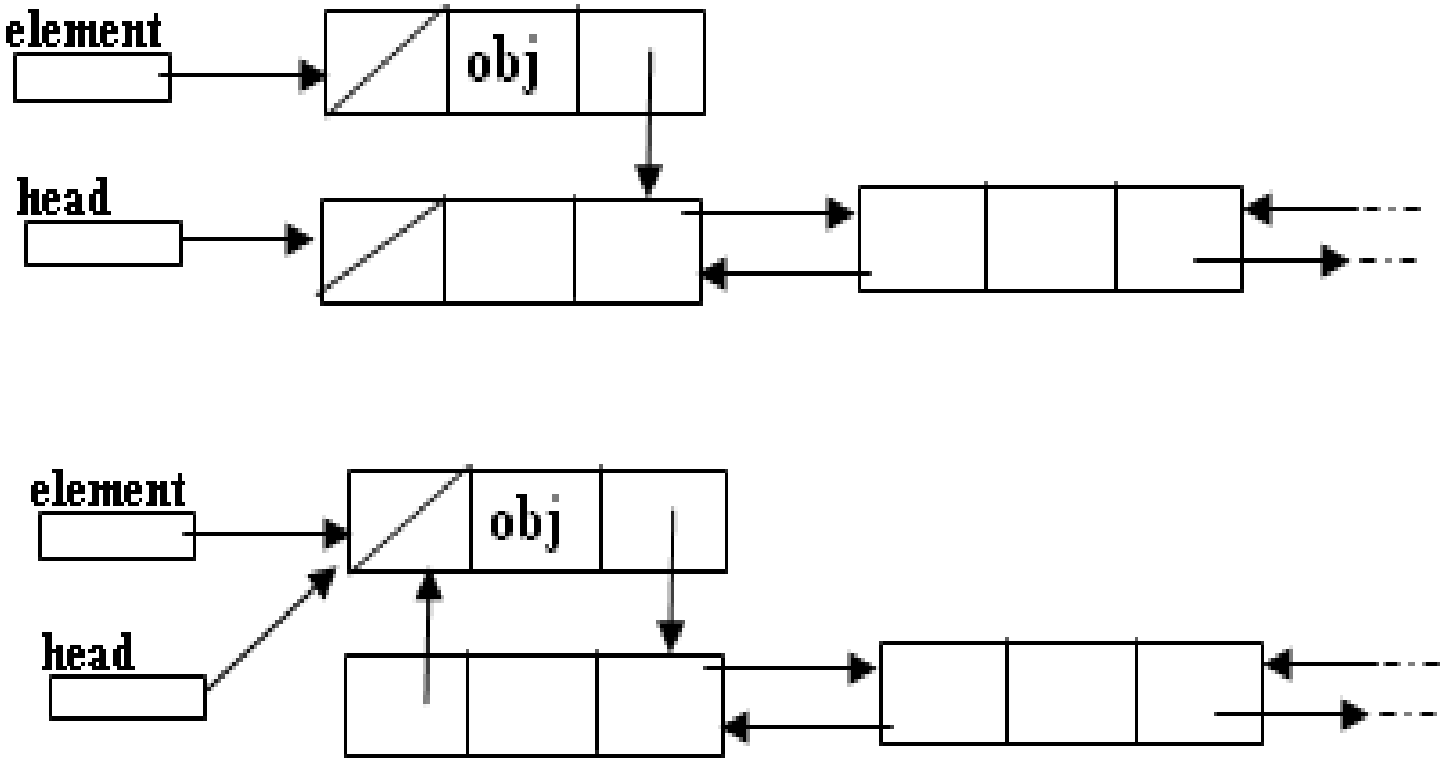
- **Advantages:**
  - Can be traversed in either direction (may be essential for some programs).
  - Some operations, such as deletion and inserting before a node, become easier.
- **Disadvantages:**
  - Requires more space.
  - List manipulations are slower (because more links must be changed).
  -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

Structure of DLL

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

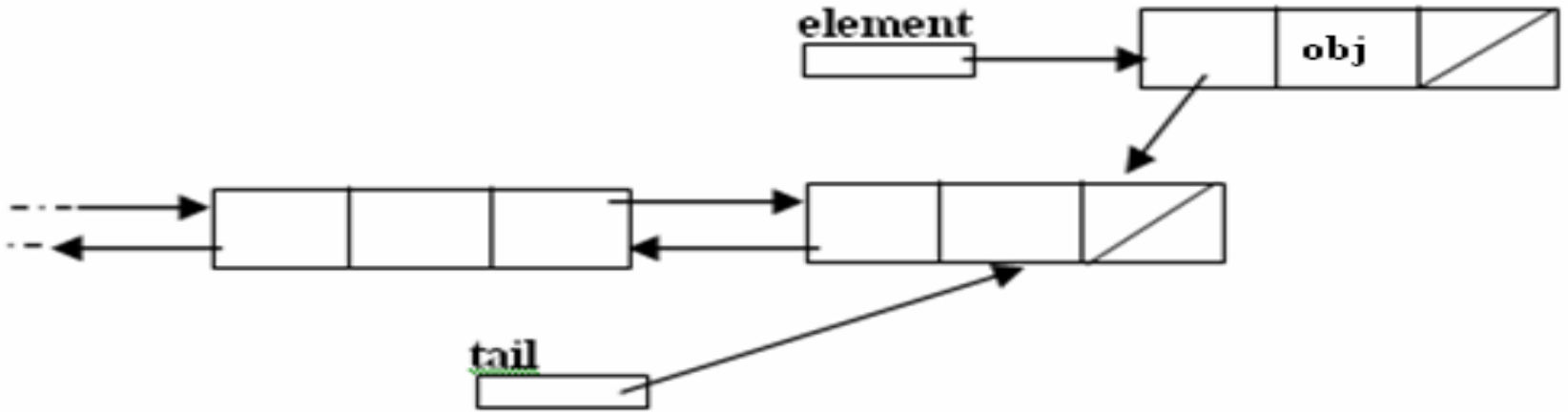


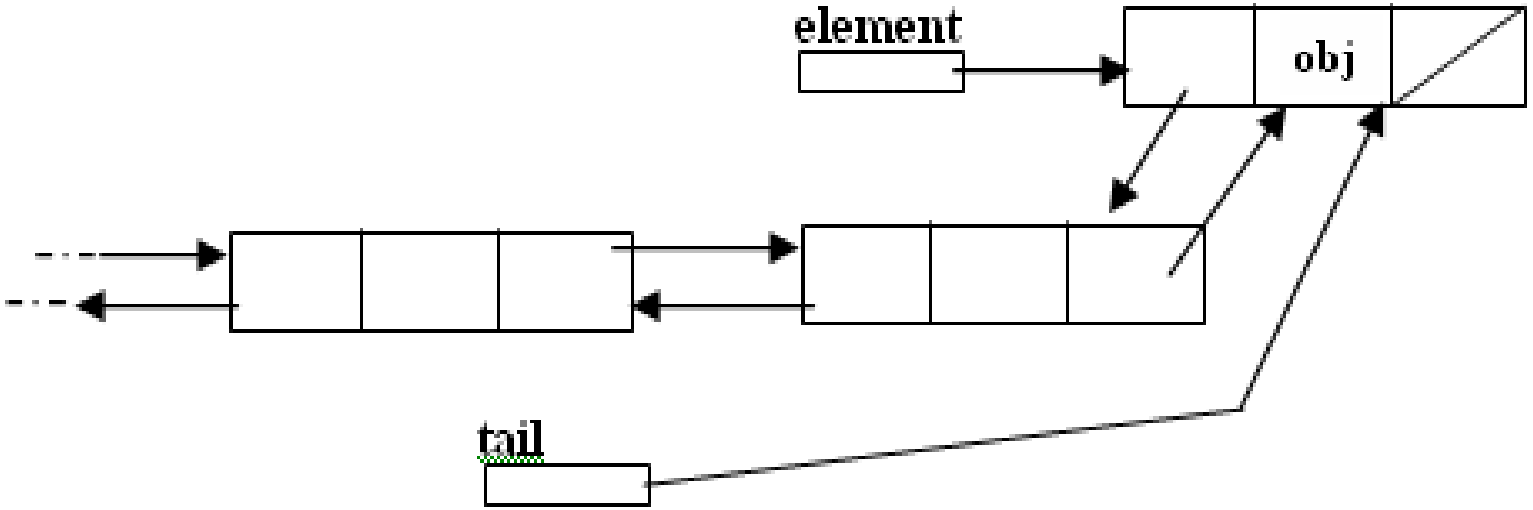
Inserting at beginning



```
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";
   }
}
```

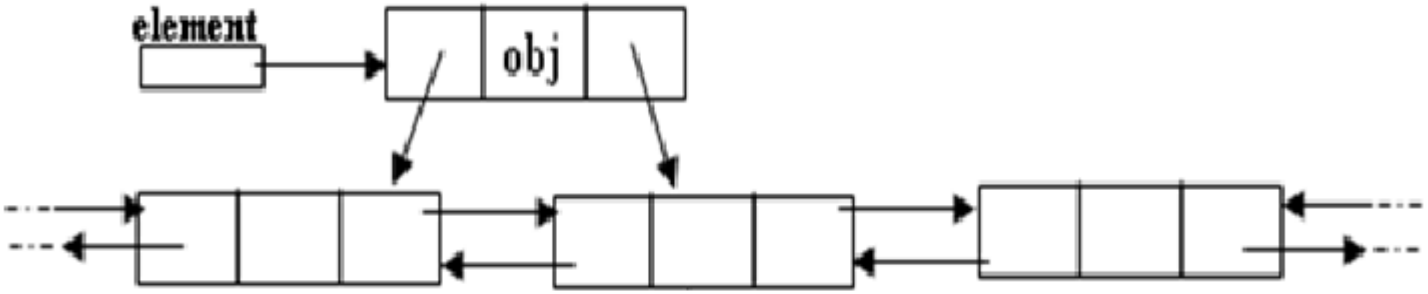
Inserting at the end



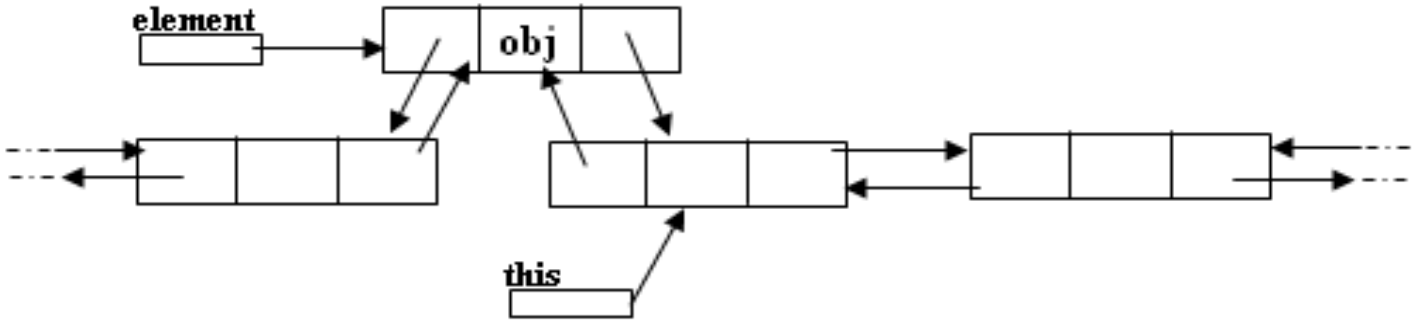


```
void insert_end(student *p)
{
    if(start==NULL)
    {
       start=p;
       cout<<"\nNode inserted successfully at the end";
    }
    else
    {
        student *temp=start;
        while(temp->next!=NULL)
         {
           temp=temp->next;
         }
          temp->next=p;
          p->previous=temp;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

Inserting after a 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

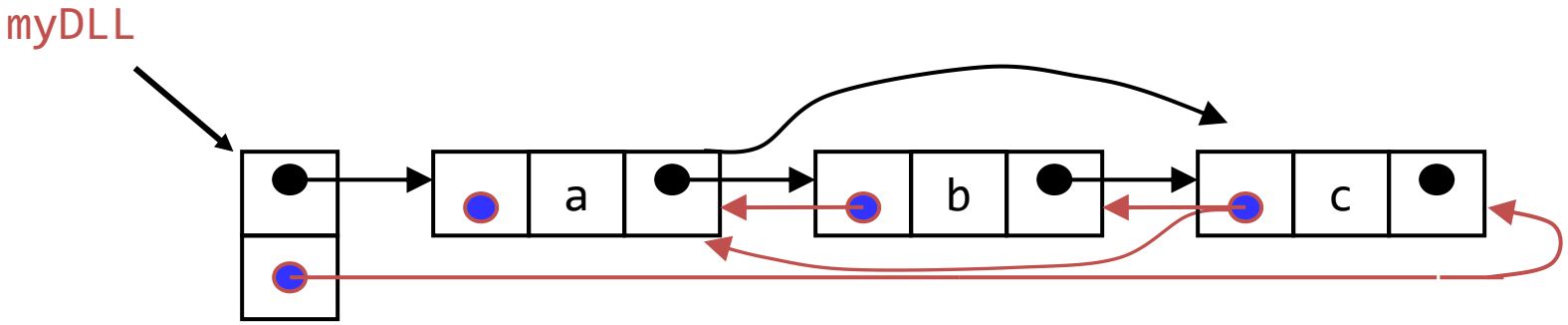
```
void insert_after(int c, node *p)
{
    temp=start;
    for(int i=1;i<c-1;i++)
    {
      temp=temp->next;
    }
       p->next=temp->next;
       temp->next->previous=p;
       temp->next=p;
       p->previous=temp;
       cout<<"\nInserted successfully";
}
```

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

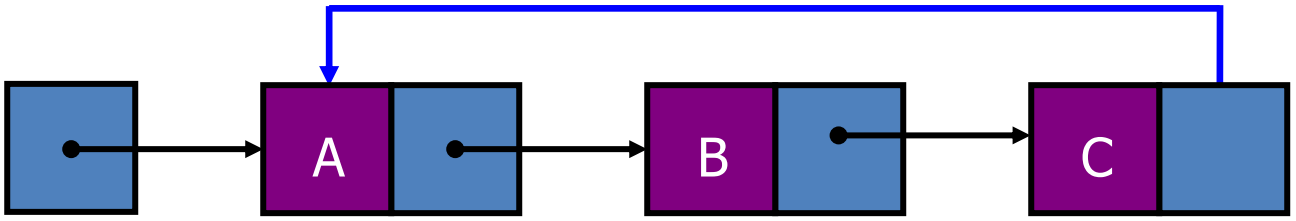
```
void del_at(int c)
{
    node *s=start;
      {
          for(int i=1;i<c-1;i++)
          {
           s=s->next;
          }
    node *p=s->next;
    s->next=p->next;
    p->next->previous=s;
    delete p;
    cout<<"\nNode number "<<c<<" deleted successfully";
      }
   }
}
```

APPLICATIONS OF LINKED LIST

1. Applications that have an MRU (Most Recently Used) list (a linked list of file names).
2. The cache in your browser that allows you to hit the BACK button (a linked list of URLs).
3. Undo functionality in Photoshop or Word (a linked list of state).
4. A stack, hash table, and binary tree can be implemented using a doubly linked list.

Variations of Linked Lists

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



Start

  - How do we know when we have finished traversing the list?
  - (Hint: 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**, and **Age, Department**. The program should include the following operations.
   a) Add a new student data from the keyboard(front, middle, last)
   b) Dele a node (the first, last, middle)
   c) Display the whole list of students
   d) Search a specific node