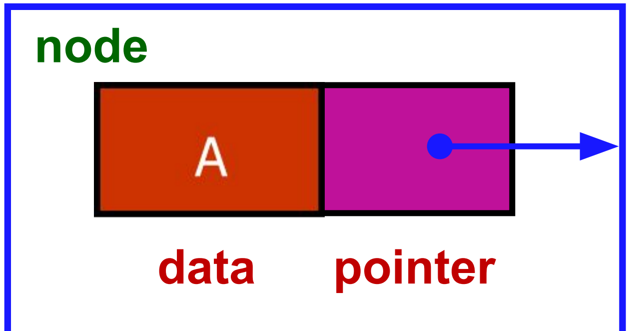
بسم الله الرحمن الرحيم

Linked Lists

Dr. Tahir Maqsood



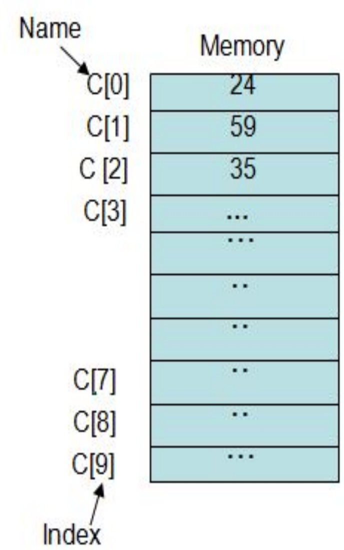
Department of Computer Science
CUI, Lahore Campus

Outline

- Static vs Dynamic Data Structure
  - Limitations of List ADT with Array Implementation.
- Dynamic List (Linked lists)
- Variations of linked lists
  - Single linked lists
  - Circular linked lists
  - Doubly linked lists
- Basic operations of linked lists
  - Insert
    - Iinsert at start
    - Insert at end
    - Insert at specific position
  - delete,
  - print, find etc.

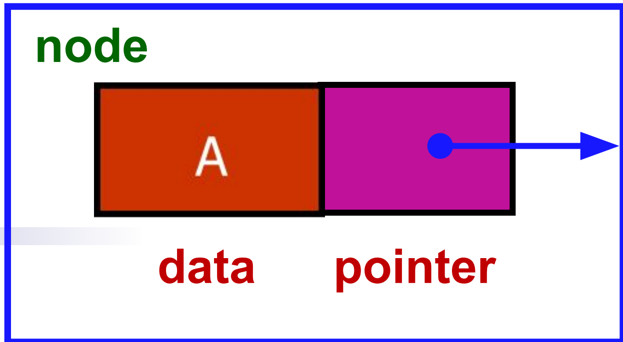
Array Limitations

- What are the limitations of an array, as a data structure?
  - Fixed size(e.g. int L[10])
    - What is the drawback of fixed size?
    - Can not grow or shrink as needed
    - **Solution:** Use dynamic Data Structure
  - Physically stored in consecutive memory locations
    - What is drawback of consecutive memory
    - Suppose we need 10 memory location
    - The 10 locations are available but not consecutive
    - Solution: store elements where space available

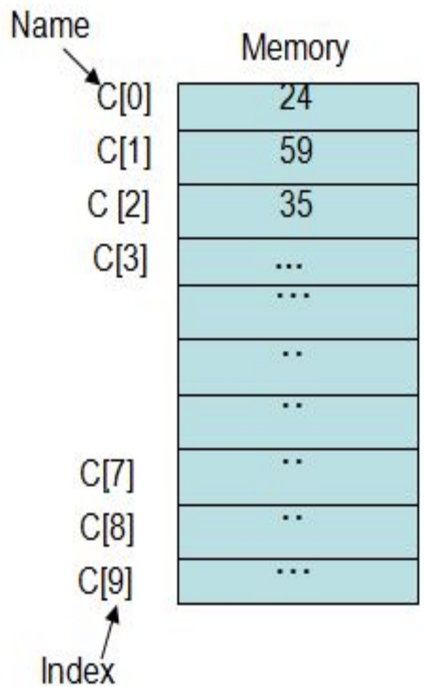


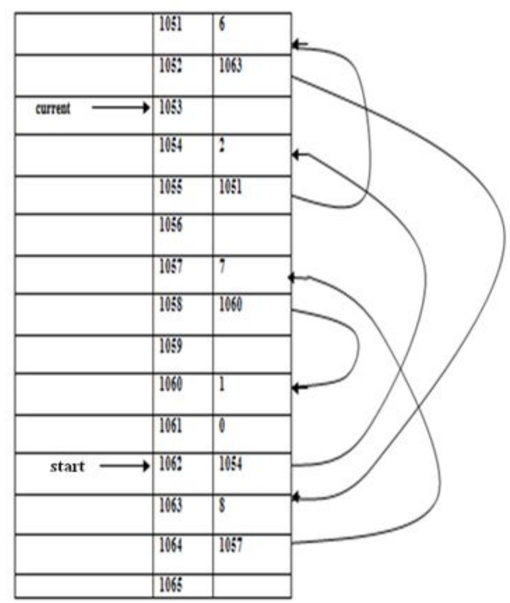
Linked Data Structures

- How we ensure these two solutions?
  - Use Linked Data Structure
  - A **linked** data structure consists of items that are linked to other items(how?)
  - each item **points to** another item
- Now what is **linked list**?
  - A linked list is an **ordered sequence** of items called ***nodes***
  - Each node contains at least
    - A piece of data (any type)
    - Pointer to the next node in the list

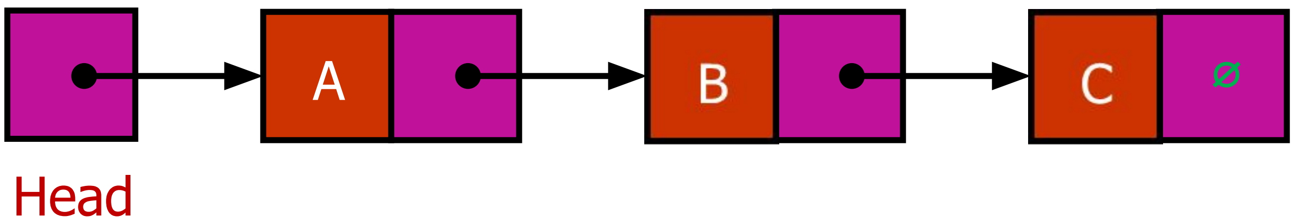


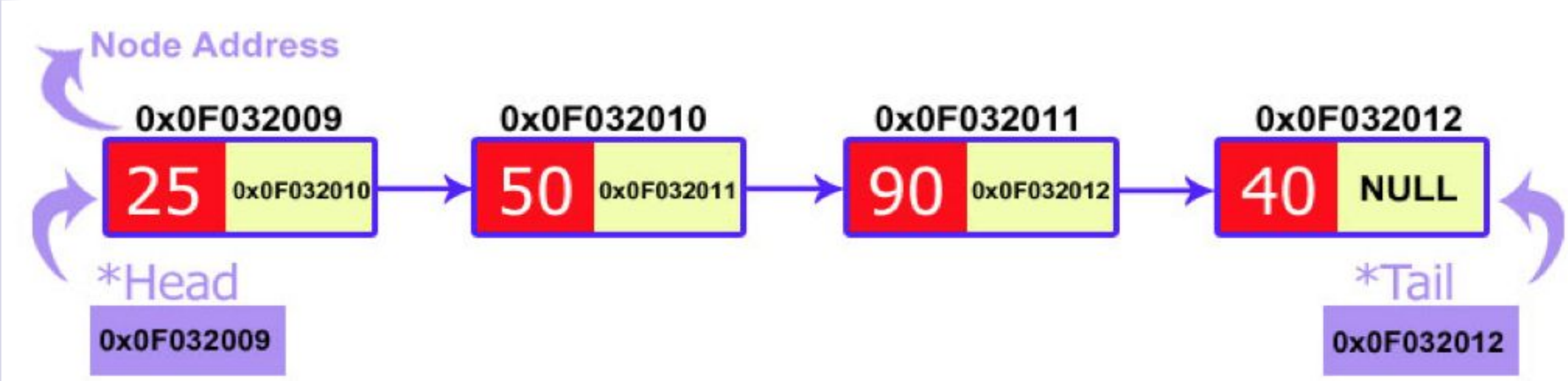
- In linked list, adjacency between the elements are maintained by means of link or pointers





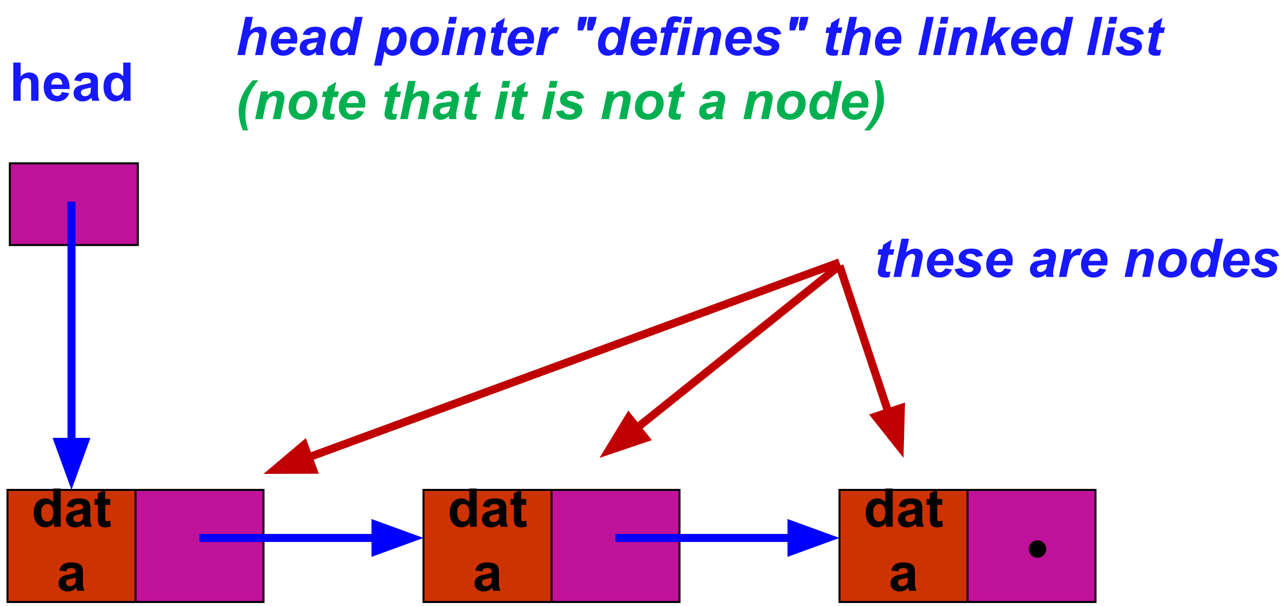
Conceptual Diagram of a Singly-Linked List



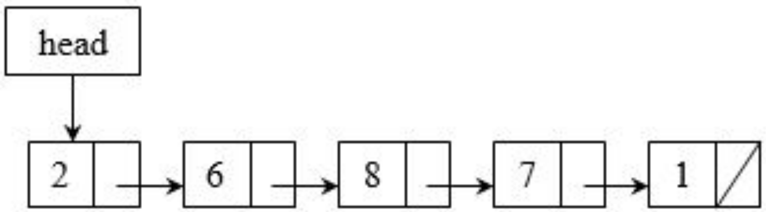


- The first item (node) in the linked list is accessed via a **front** or **head** pointer
  - The linked list is defined by its head (this is its starting point)
- Head: pointer to the first node
- The last node points to `NULL`

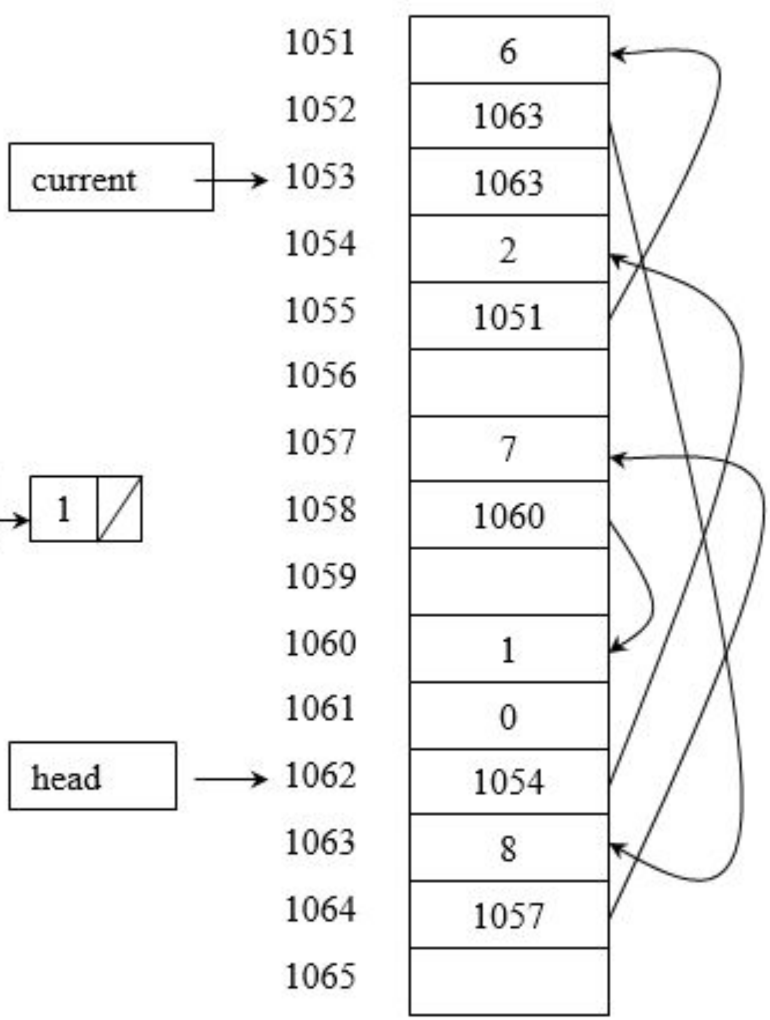
Conceptual Diagram of a Singly-Linked List



Linked List inside Computer Memory



Conceptual diagram



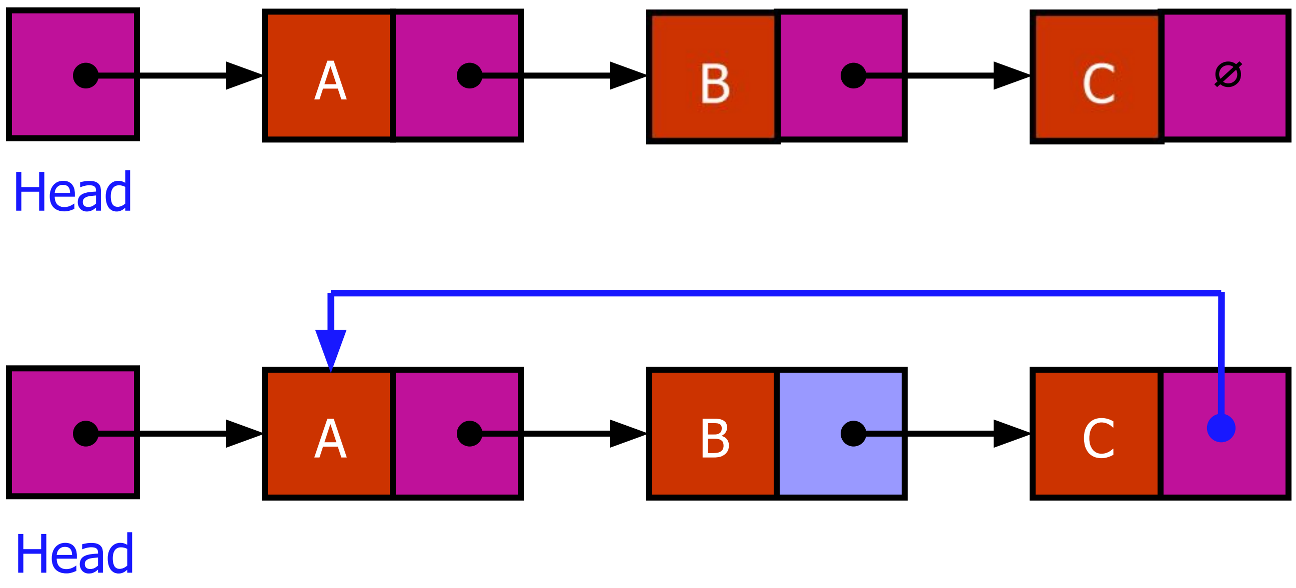
Linked list in memory

Advantages of Linked Lists

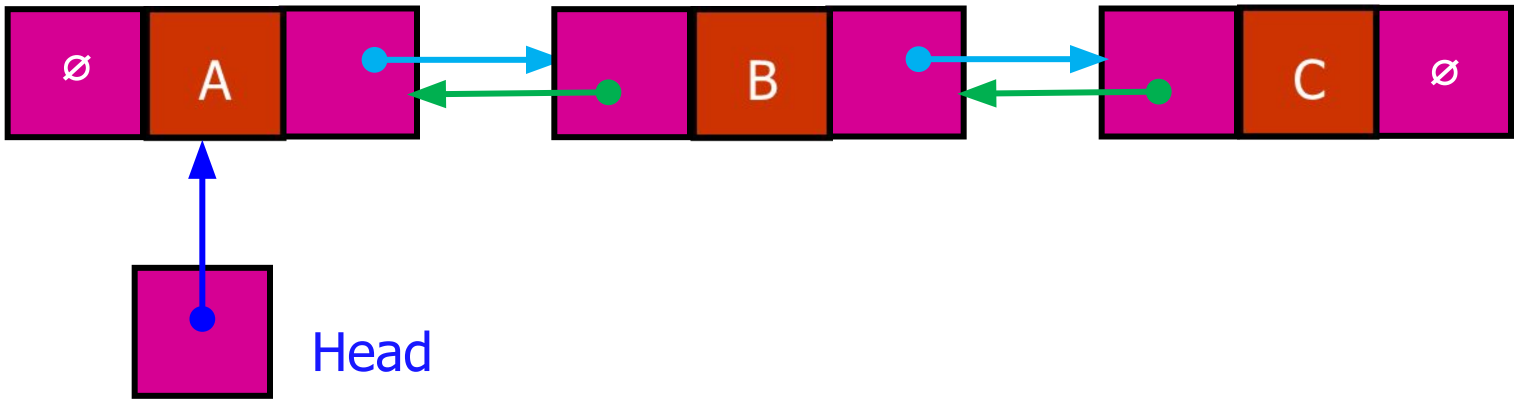
- The items do not have to be stored in consecutive memory locations: the successor can be anywhere physically
  - So, can insert and delete items without shifting data
  - Can increase the size of the data structure easily
- Linked lists can grow dynamically (i.e. at run time) – the amount of memory space allocated can grow and shrink as needed

Variations of Linked Lists

- **Singly linked list:** each item points to the next item
- Circular or Non circular



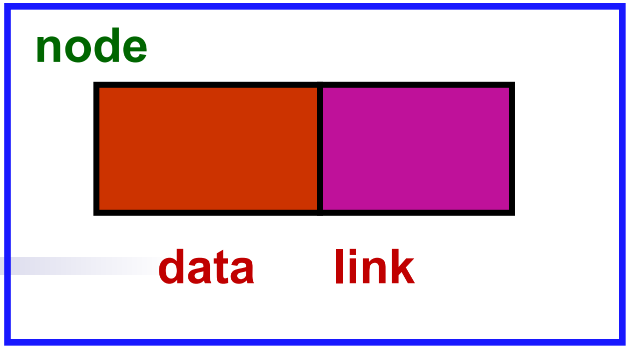
- **Doubly linked list:** each item points to the next item and to the previous item



Implementation of Linked List Using C++

- Declare `Node` structure for the nodes
  - `data`: `int`-type data e.g. example
  - `link`: a pointer to the next node in the list

```
struct Node
{
    int       data;     // data
    Node*     link;     // pointer to next
};
```



Creating single Linked List

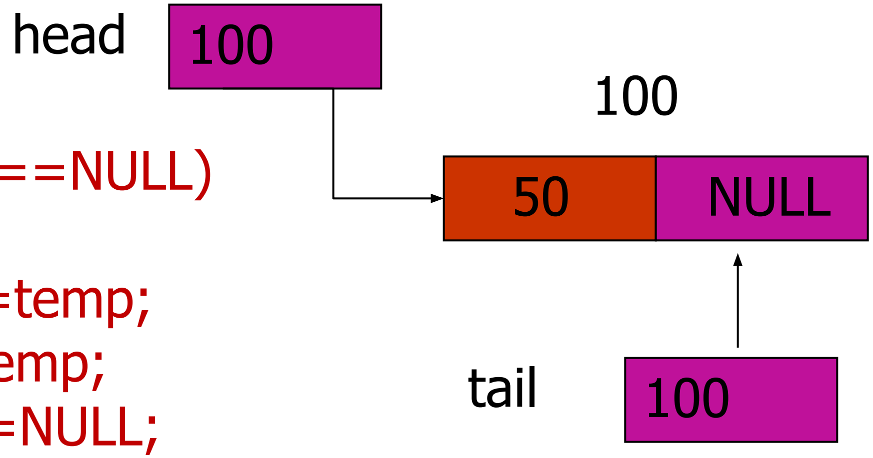
- We use two important pointers, i.e. head and tail
  - **head** points to first node
  - **tail** points to last node.
- Demo

head NULL tail NULL

- Initially Linked list has no node, i.e. Linked list is empty
  - It means if the head is equal to NULL then we can conclude that the linked list is empty.

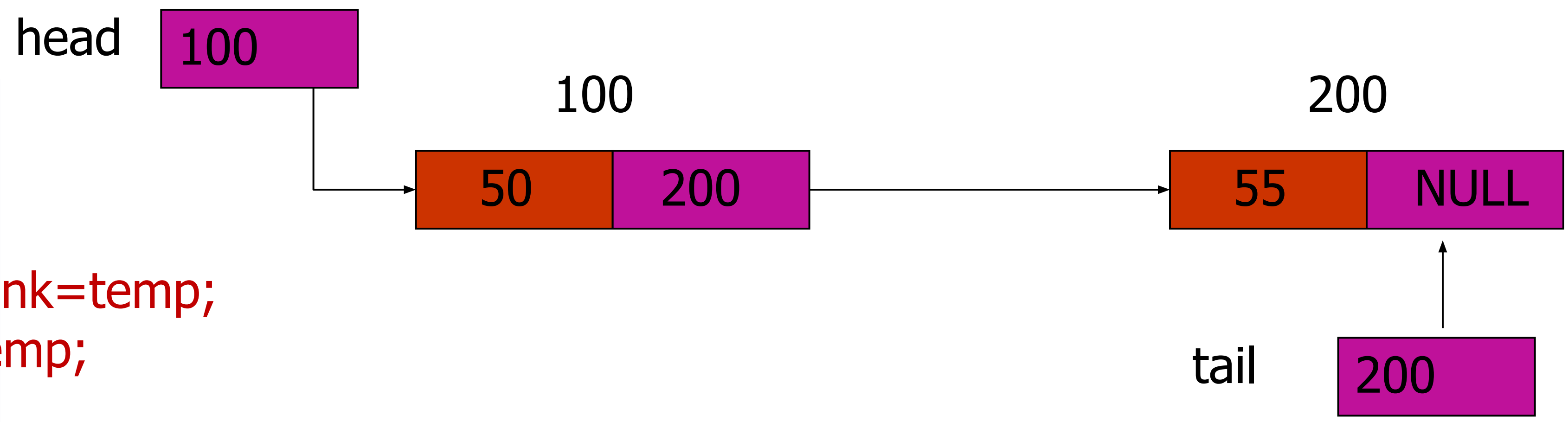
Creating single Linked List

```
node *temp=new node;
temp->data=value;
temp->link=NULL;
```




```
if(head==NULL)
{
  head=temp;
  tail=temp;
  temp=NULL;
}
```

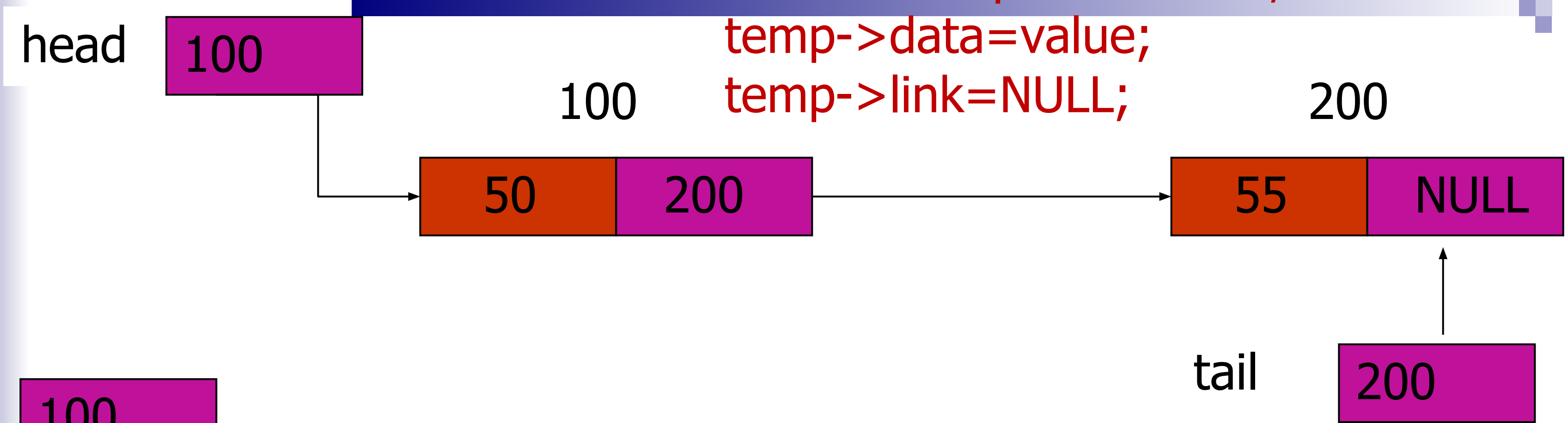
if there is just one node in linked lists, then it is called both head and tail.

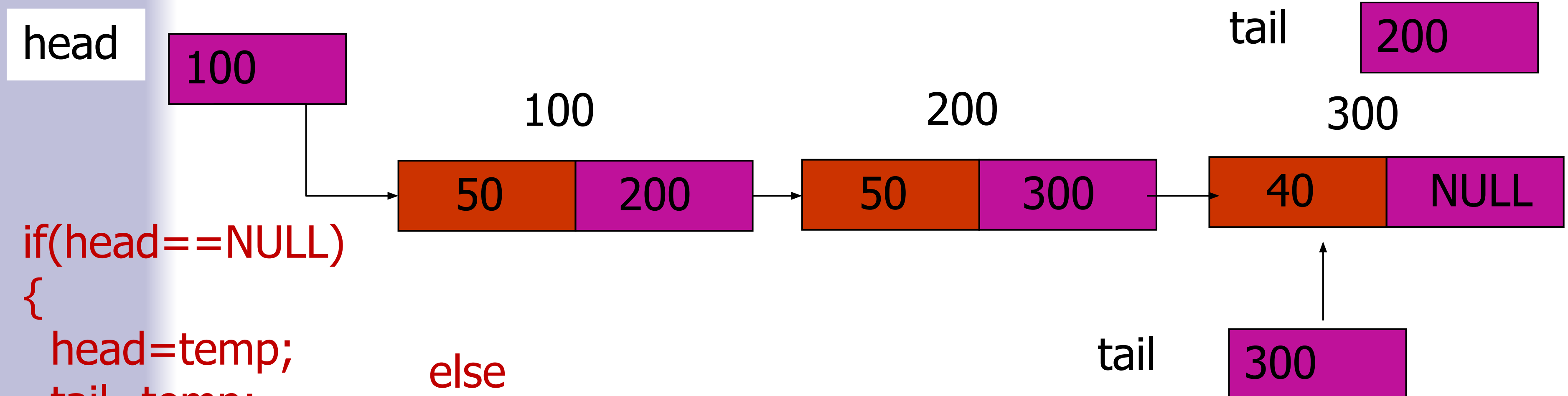



```
else
{
 tail->link=temp;
 tail=temp;
}
```

Creating single Linked List

```
node *temp=new node;
temp->data=value;
temp->link=NULL;
```






```
if(head==NULL)
{
  head=temp;
  tail=temp;
  temp=NULL;
}
```

```
else
{
  tail->link=temp;
  tail=temp;
}
```

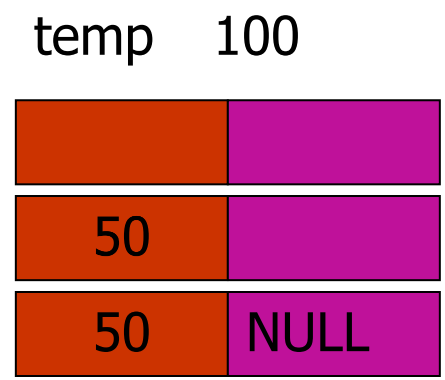
Creating single Linked List

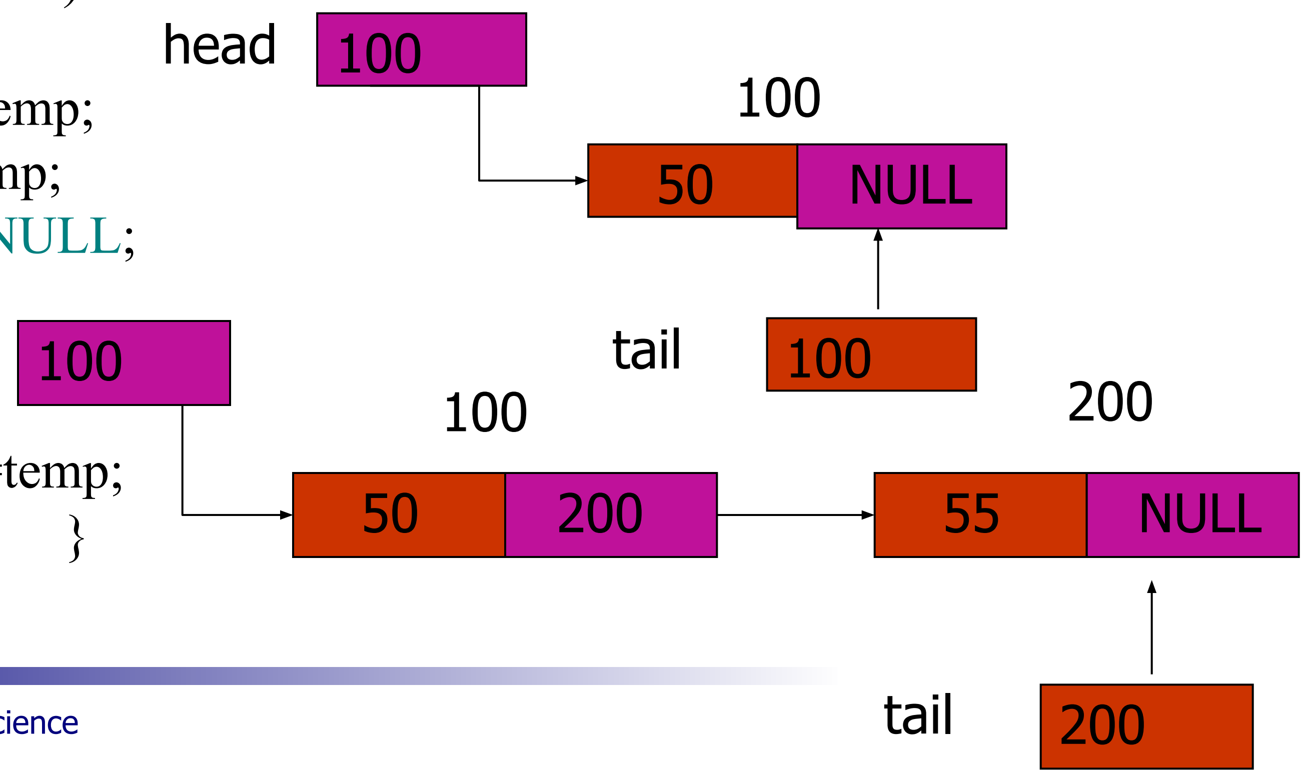
- Now, we will write a function for creating LinkedList(node will be inserted at end)

```
void createnode(int value)
{
    node *temp=new node;
    temp->data=value;
    temp->link=NULL;
    if(head==NULL)
    {
     head=temp;
     tail=temp;
     temp=NULL;
    }
  else
  {
     tail->link=temp;
     tail=temp;
 }
}
```

Creating single Linked List

```
void createnode(int value)
  {
     node *temp=new node;
     temp->data=value;
     temp->next=NULL;
     if(head==NULL)
        {
            head=temp;
            tail=temp;
            temp=NULL;
        }
    else
      {
         tail->next=temp;
         tail=temp;      }
     }
```





Displaying single Linked List

- Now we have a working linked list which allows creating nodes.
- If we want to see that what is placed in our linked list then we will have to make a display function.

Displaying single Linked List

- The logic behind this function is that:
  - Make a temporary node   node *temp=new node;
  - pass the address of the head node to it.

    temp=head;
  - Now we want to print all the nodes on the screen. So we need a loop which runs as many times as nodes exist.
    - Every node contains the address of the next node so the temporary node walks through the whole linked list.
    - If the temporary node becomes equal to NULL then the loop would be terminated.

```
while(temp!=NULL)
{
    Print "temp->data";
    temp=temp->link;
}
```

Displaying single Linked List

```
void display()
{
   node *temp=new node;
   temp=head;
   while(temp!=NULL)
     {
        cout<<temp->data<<"\t";
        temp=temp->link;
     }
}
```

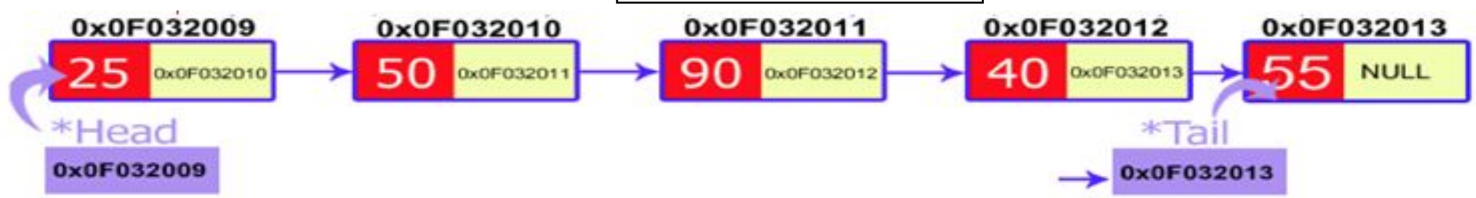
25

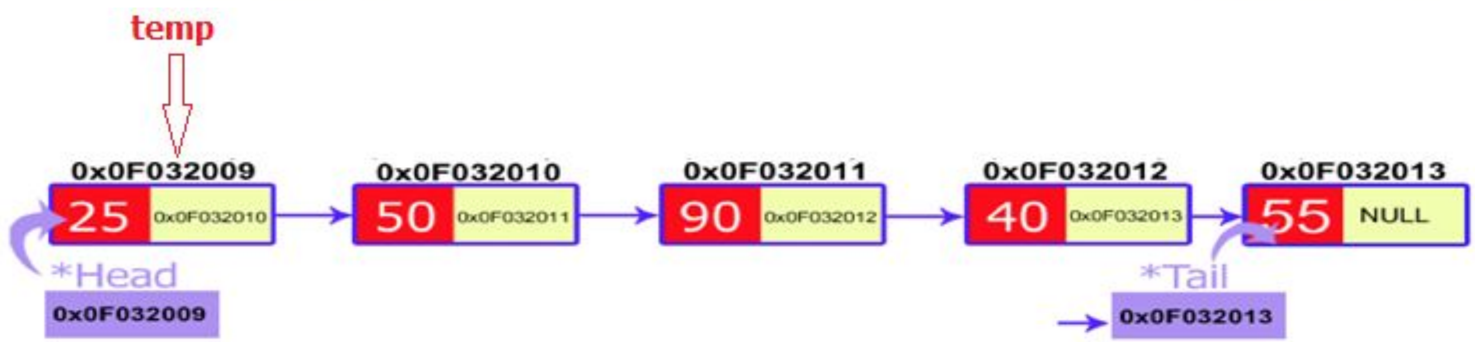
temp

temp 0x0F032009

25

temp 0x0F032010





Displaying single Linked List

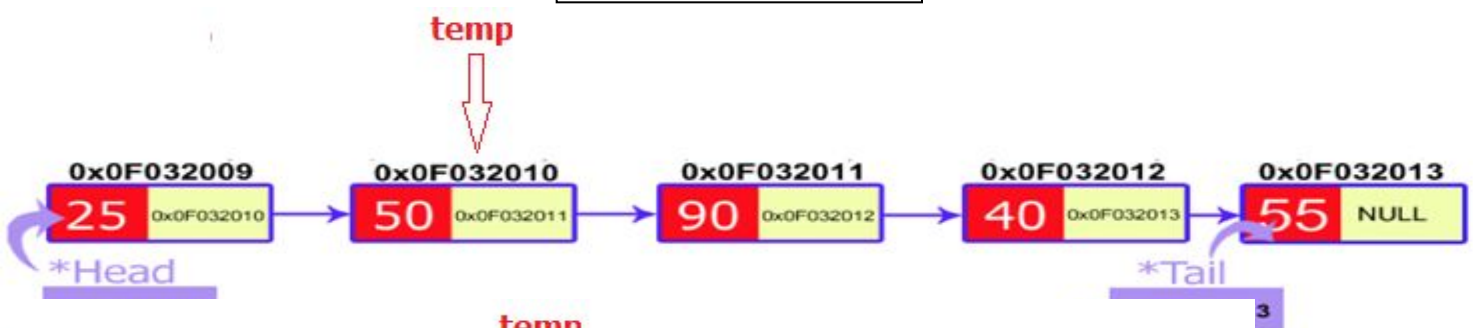
```
void display()
{
    node *temp=new node;
    temp=head;
    while(temp!=NULL)
      {
          cout<<temp->data<<"\t";
          temp=temp->link;
      }
}
```

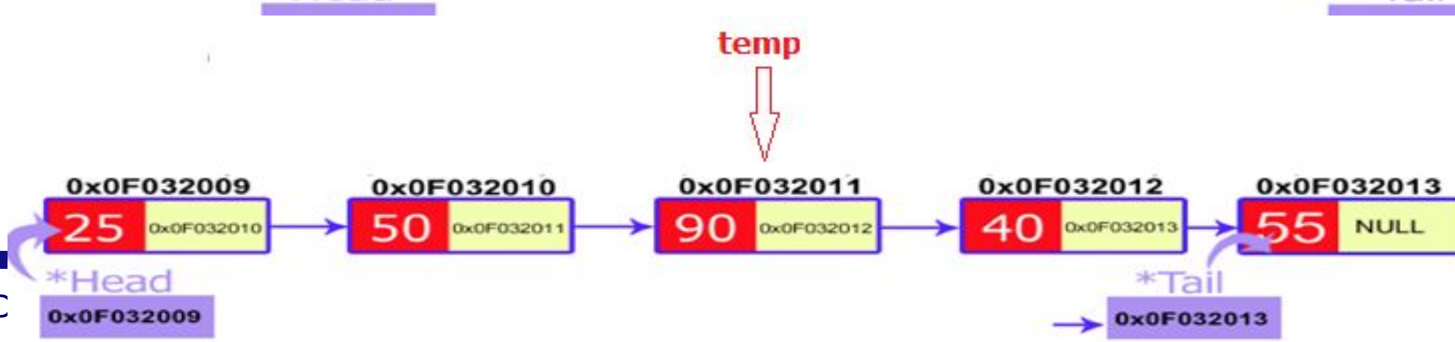
25, 50

temp $\boxed{\text{0x0F032010}}$

50

temp $\boxed{\text{0x0F032011}}$





Displaying single Linked List

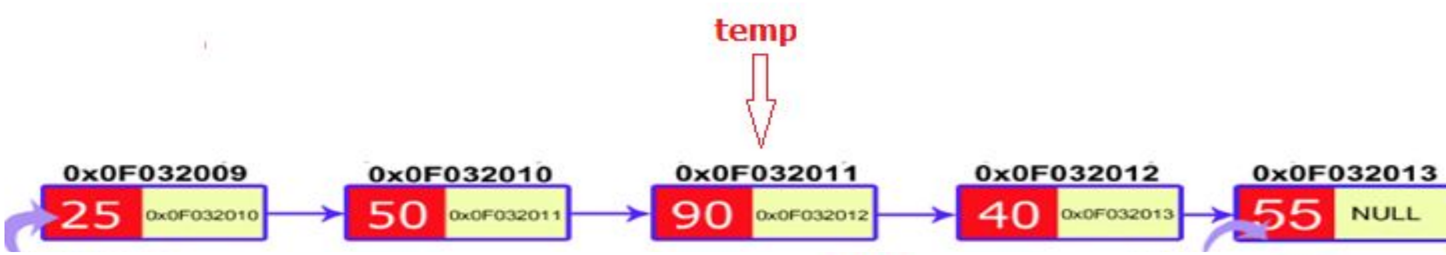
```
void display()
{
    node *temp=new node;
    temp=head;
    while(temp!=NULL)
     {
        cout<<temp->data<<"\t";
        temp=temp->link;
     }
}
```

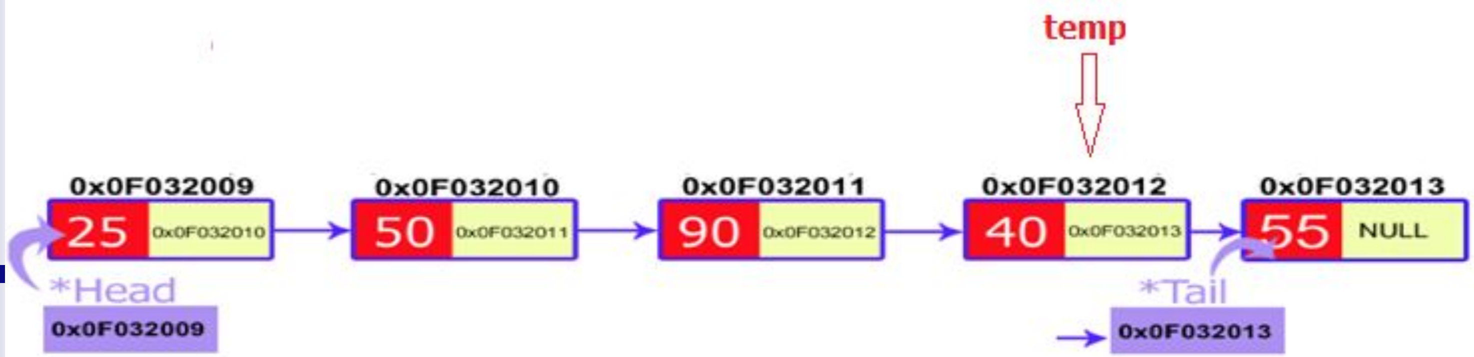
25, 50,90

temp 0x0F032011

90

temp 0x0F032012





Displaying single Linked List

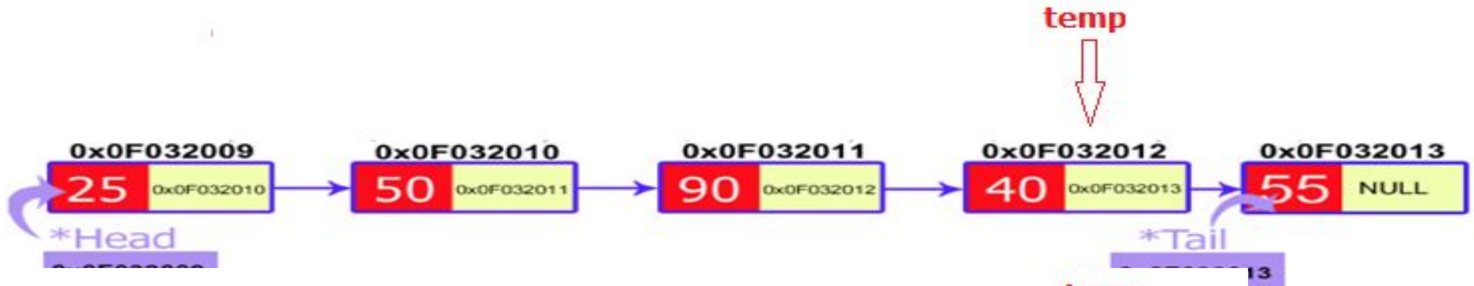
```
void display()
{
   node *temp=new node;
   temp=head;
   while(temp!=NULL)
     {
        cout<<temp->data<<"\t";
        temp=temp->link;
     }
}
```

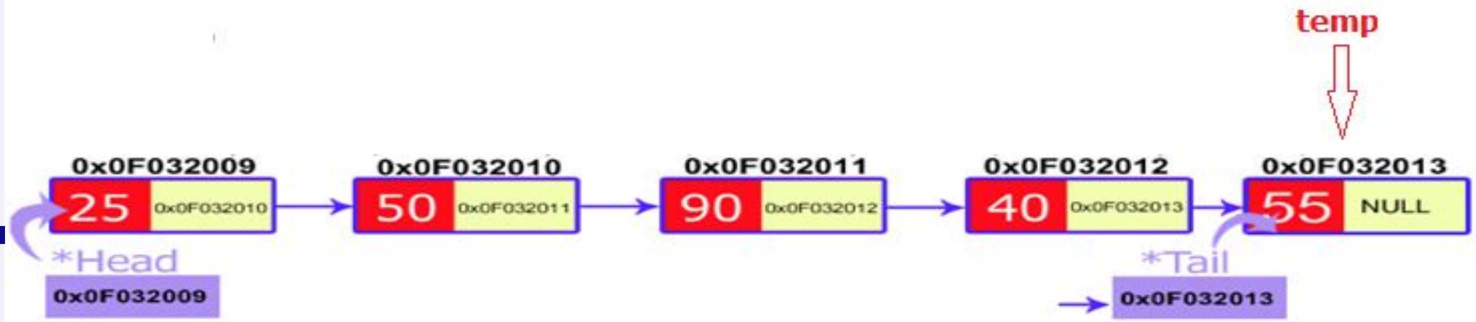
25, 50,90,40

temp 0x0F032012

40

temp 0x0F032013





Displaying single Linked List

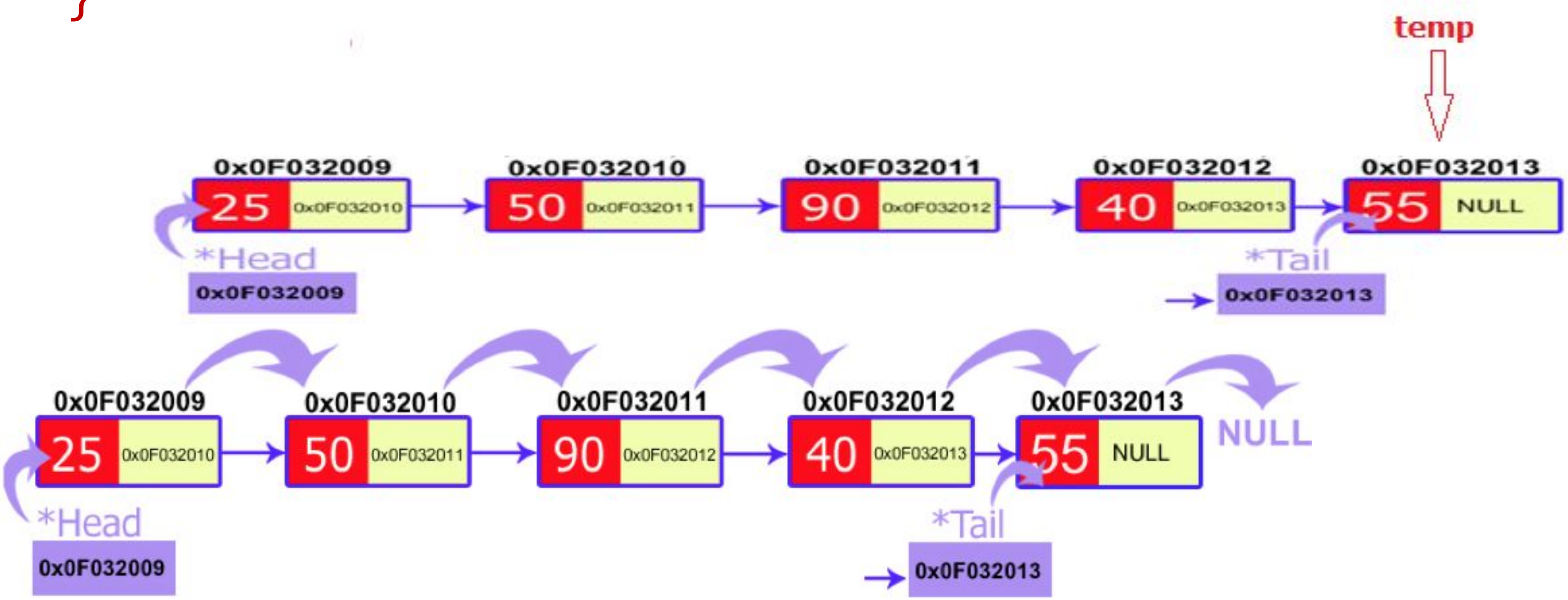
```
void display()
{
    node *temp=new node;
    temp=head;
    while(temp!=NULL)
     {
        cout<<temp->data<<"\t";
        temp=temp->link;
     }
}
```

25, 50,90,40,55

temp 0x0F032013

55

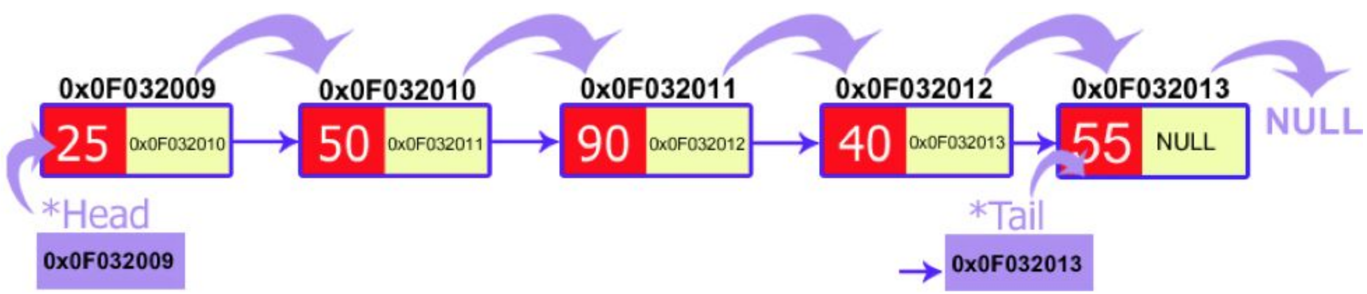
temp NULL



Displaying single Linked List

```
void display()
{
   node *temp=new node;
   temp=head;
   while(temp!=NULL)
     {
         cout<<temp->data<<"\t";
         temp=temp->link;
     }
}
```

temp | NULL



25, 50,90,40,55

Operations on single Linked List

We will now examine linked list operations:

- **Insert** an item to the linked list
  - We have 3 situations to consider:
    - insert a node at the front
    - insert a node at the end(Already done)
    - insert a node at particular position
- **Delete** an item from the linked list
  - We have 3 situations to consider:
    - delete the node at the front
    - delete an interior node
    - delete the last node

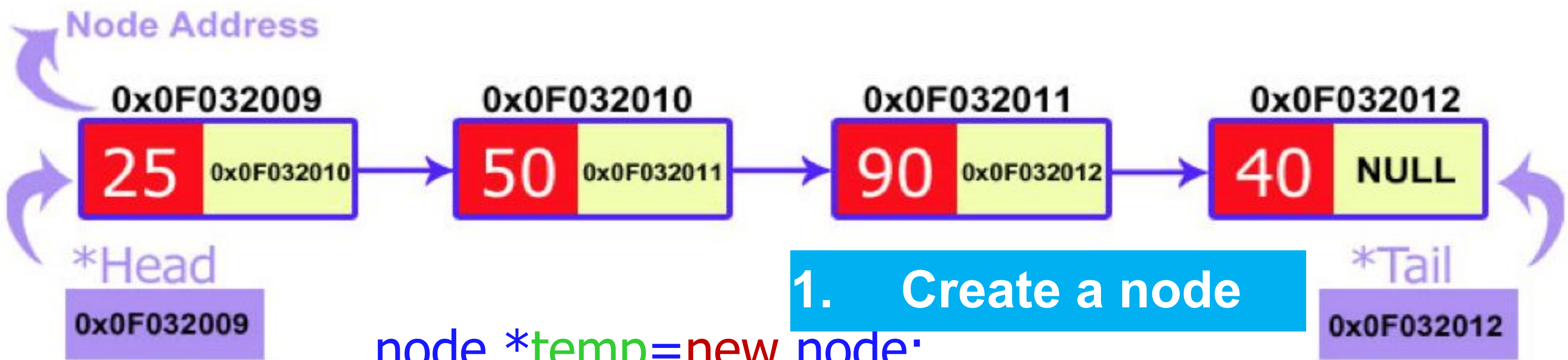
Inserting a Node

- The following steps are involved in inserting an element into linked list
- **Creation of the node**
  - Before insertion, the node is created. Using **new** operator memory space for the node is allocated
- **Assignment of data**
  - Once the node is created, data values are assigned to members
- **Adjusting pointers**
  - The insertion operation changes the sequence.
  - Hence, according to the sequence the address of the next elements is assigned to the inserted node. The address of the current node (inserted node) is assigned to the previous node.

Inserting a Node at the front

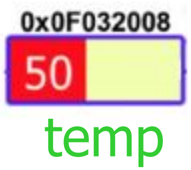
- It is just a 2-step algorithm which is performed to insert a node at the start of a singly linked list.
  - Created a node called temp
  - Connect the newly created node to the first node,
    - This can be achieved by putting the address of the head in the link field of the new node.
  - New node should be considered as a head.
    - It can be achieved by declaring head equals to a new node.

Inserting a Node at the front

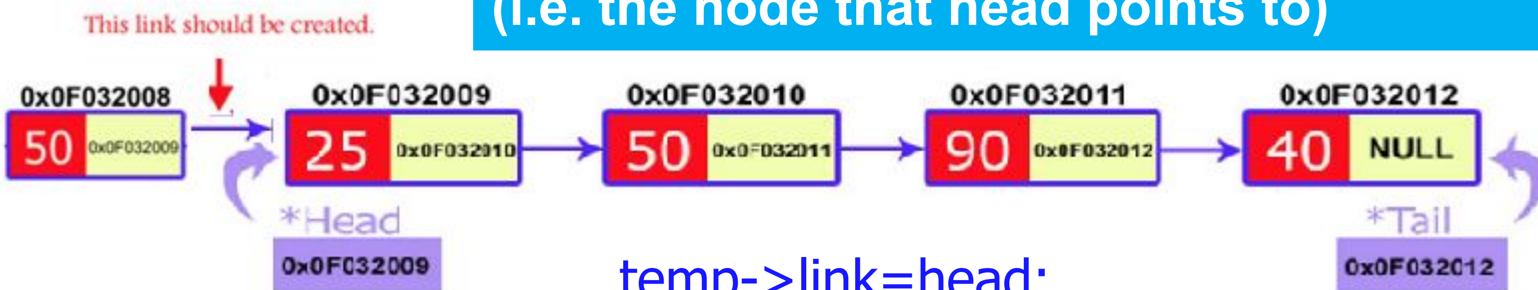


1. Create a node

```
node *temp=new node;
temp->data=value;
```

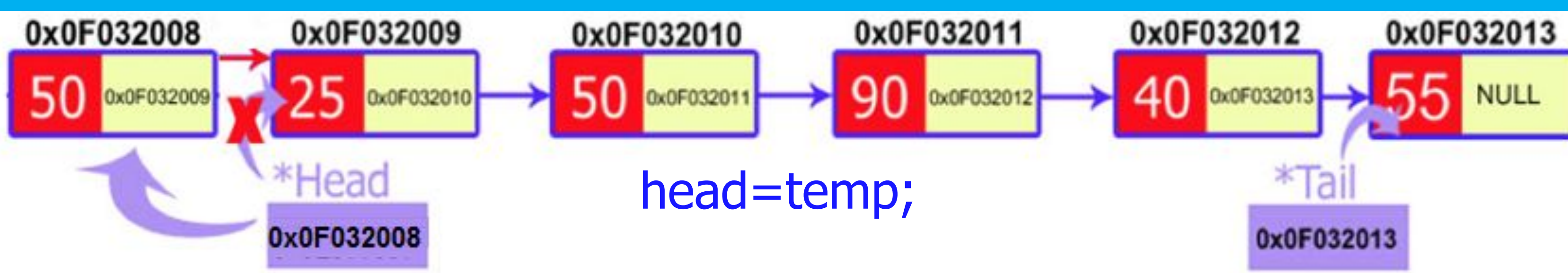


2. Make the new node point to the first node (i.e. the node that head points to)




```
temp->link=head;
```

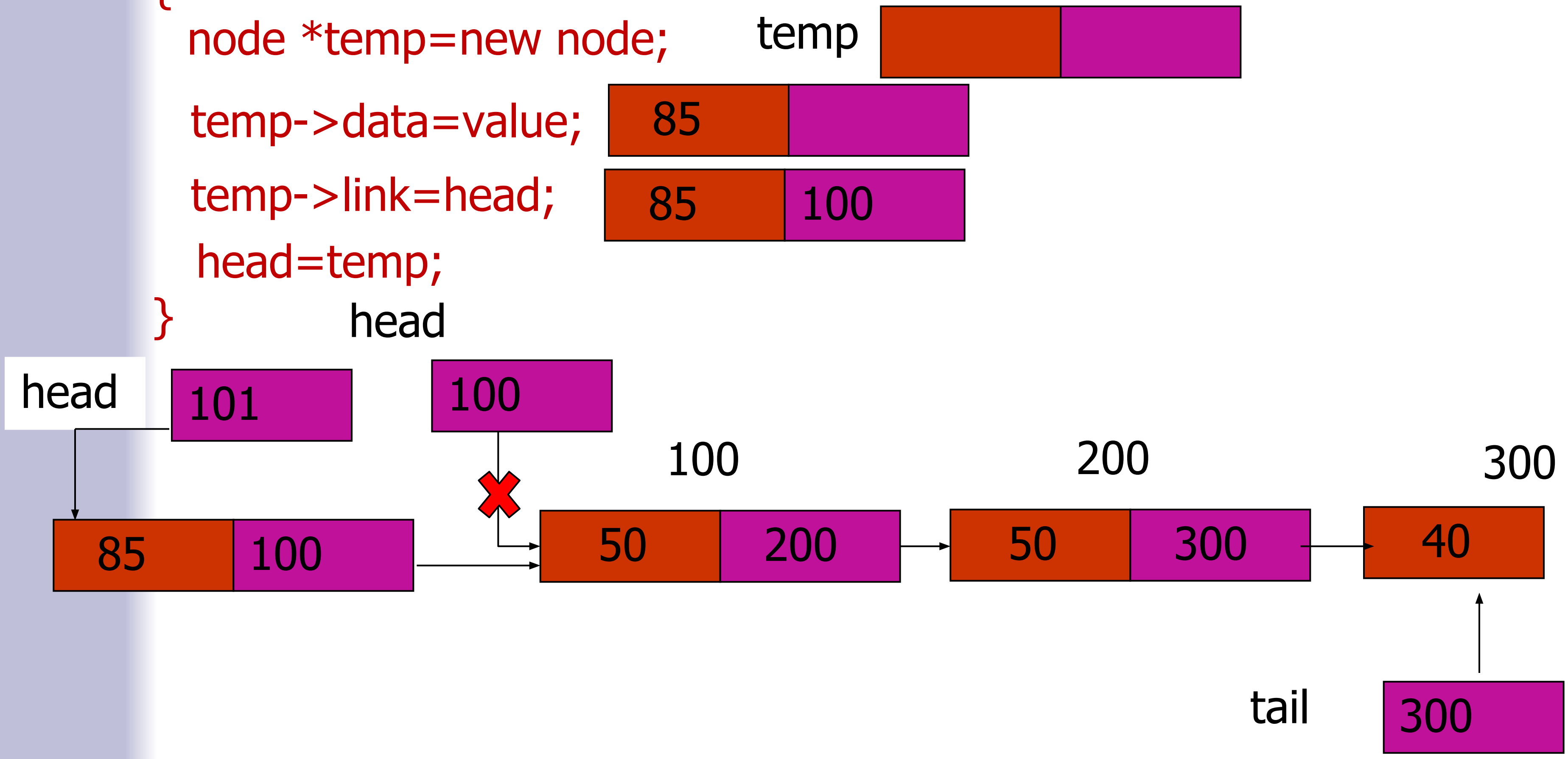
3. Make headt point to the new node (i.e the node that node points to)




```
head=temp;
```

Inserting a Node at the front

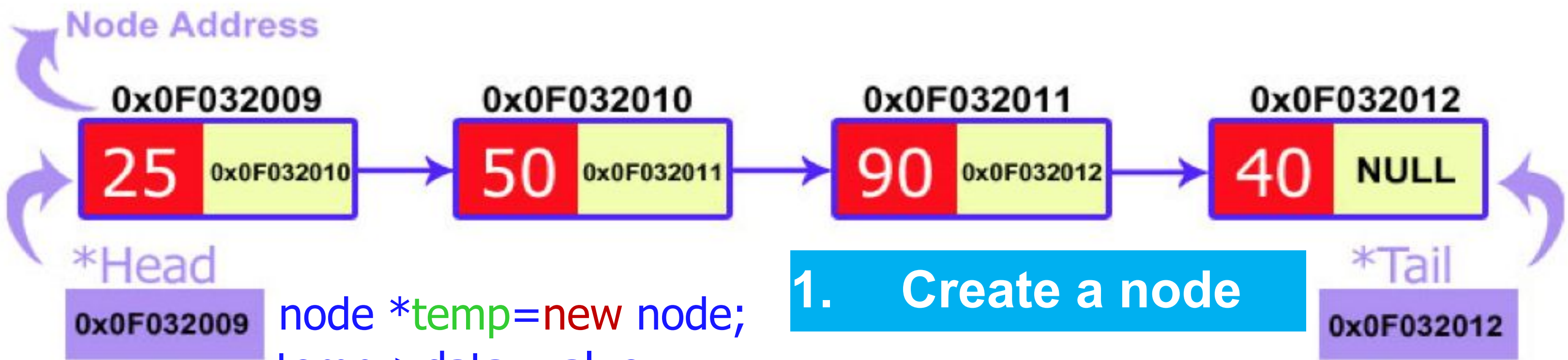
```
void insert_start(int value)
{
  node *temp=new node;
   temp->data=value;
   temp->link=head;
    head=temp;
}
```



Inserting a Node at the end

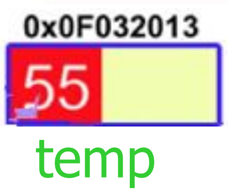
- The insertion of a new node at the end of linked list has 2 steps:
- Create a node called temp
- Linking the newly created node with tail node.
  - This can be achieved by assigning the address of a new node to the link filed of a tail node.
- The tail pointer should always point to the last node.
  - This can be achieved by assigning the address of a new node to tail pointer.

Inserting a Node at the end



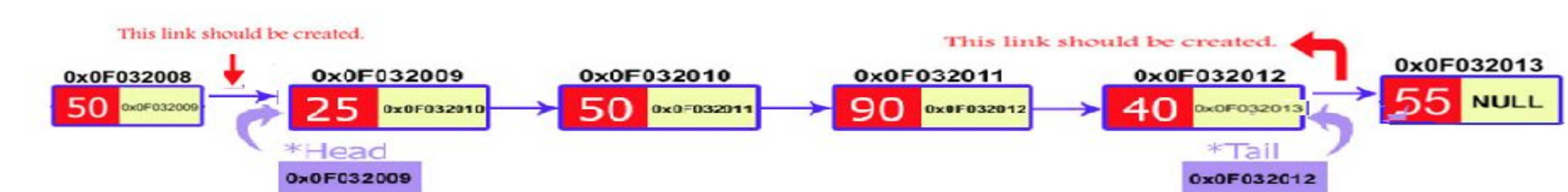
1. Create a node

node *temp=new node;
temp->data=value;

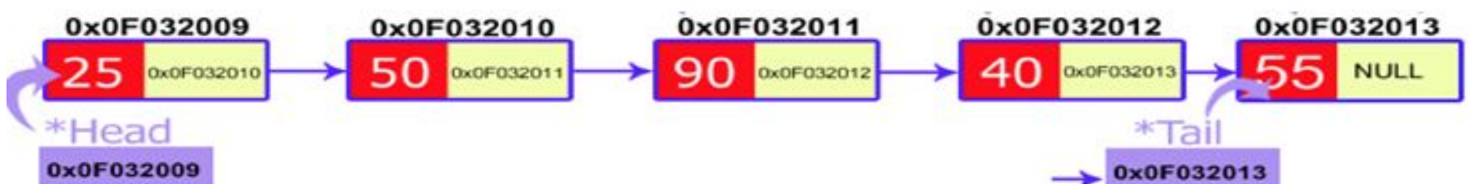


2. Make the new node last node

temp->link=NULL;
tail-> link = temp;



3. Make tail point to the new node (i.e the node that node points to)

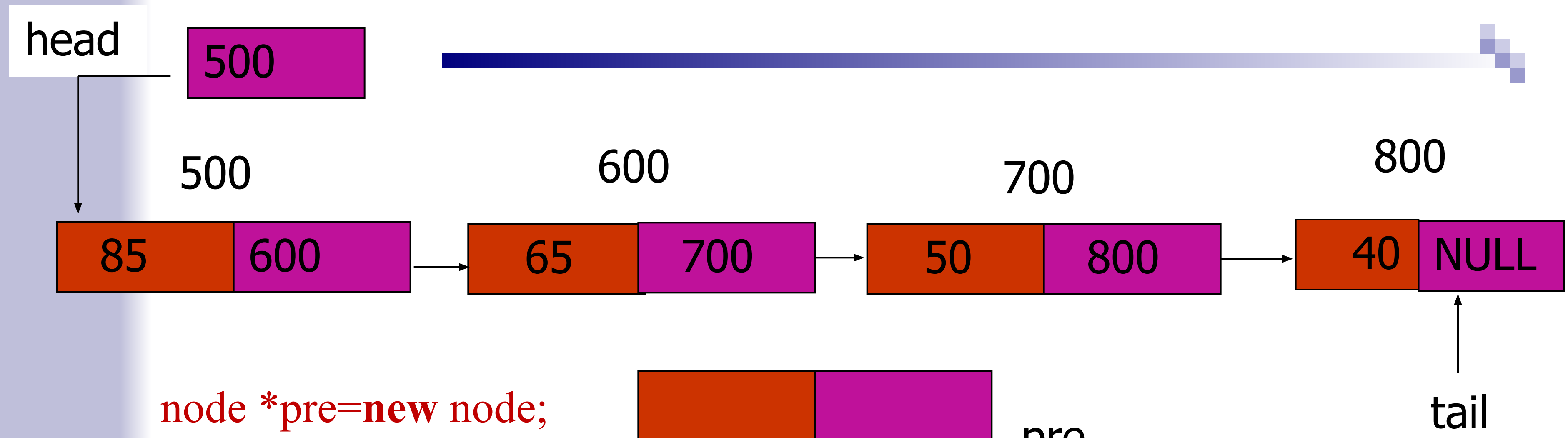


tail= temp;

Inserting a Node at Specific

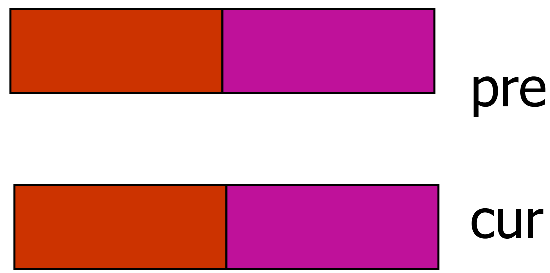
- In this case, the new node is inserted between two consecutive nodes.
  - We will access these nodes by asking the user at what position (s)he wants to insert the new node.
- We call one node as current and the other as previous, and the new node is placed between them.

Inserting a Node at Specific



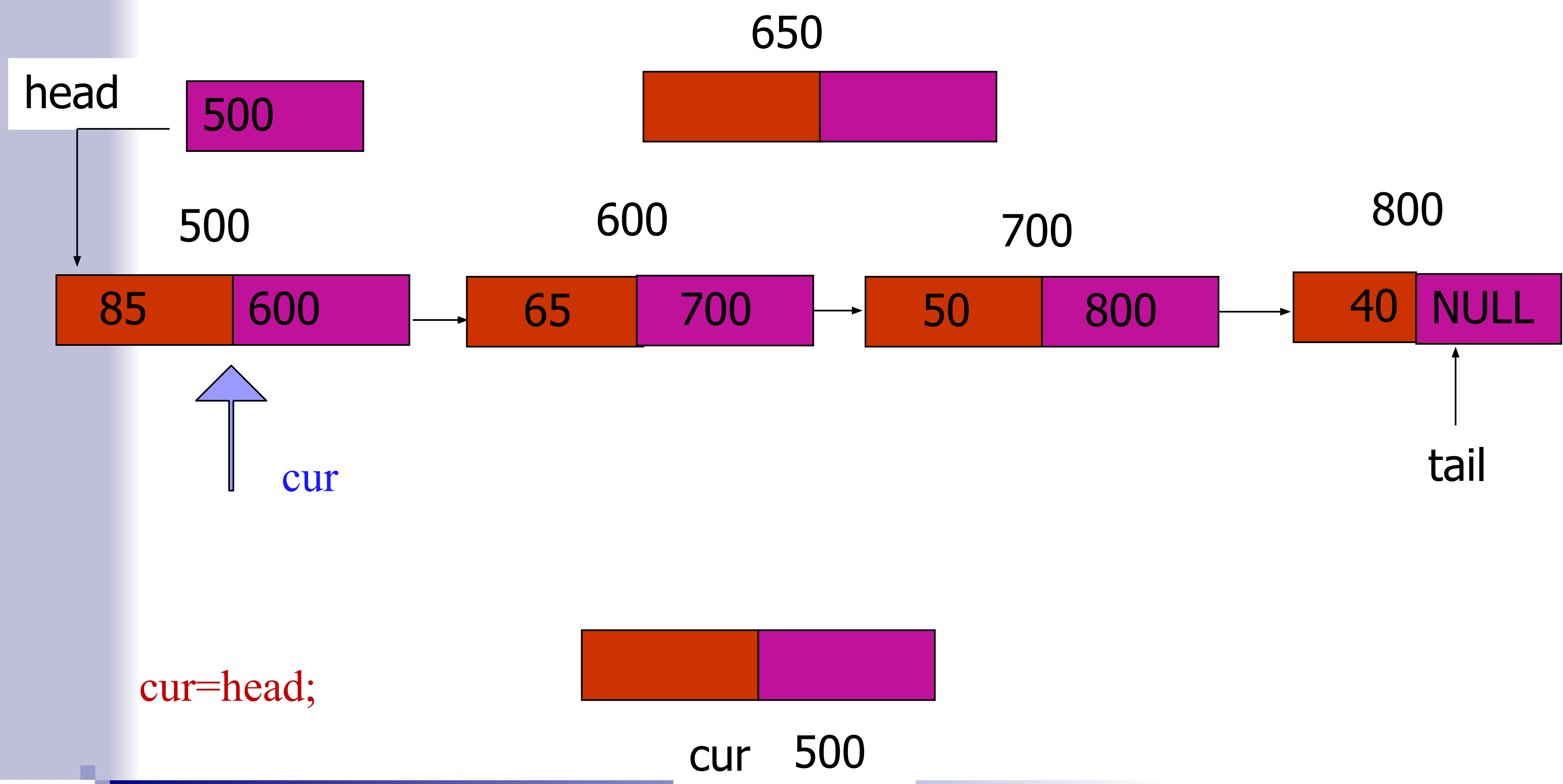
node *pre=**new** node;

node *cur=**new** node;



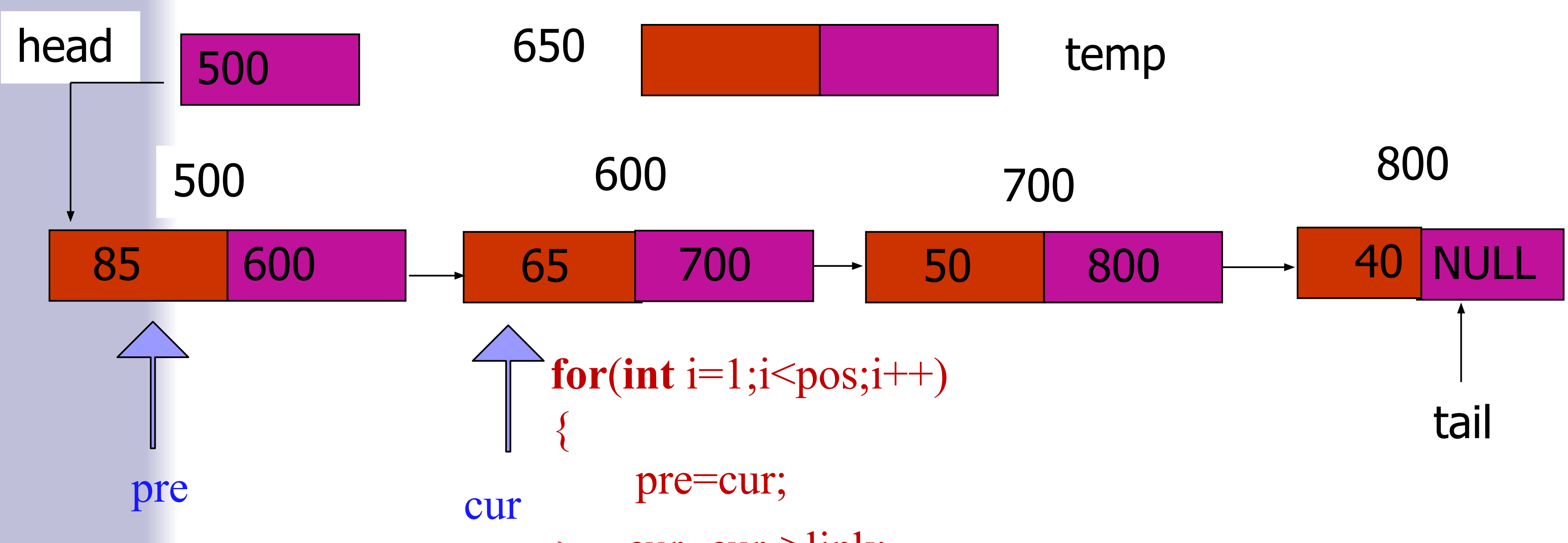
Inserting a Node at Specific

- We initialized our current node by the head



Inserting a Node at Specific

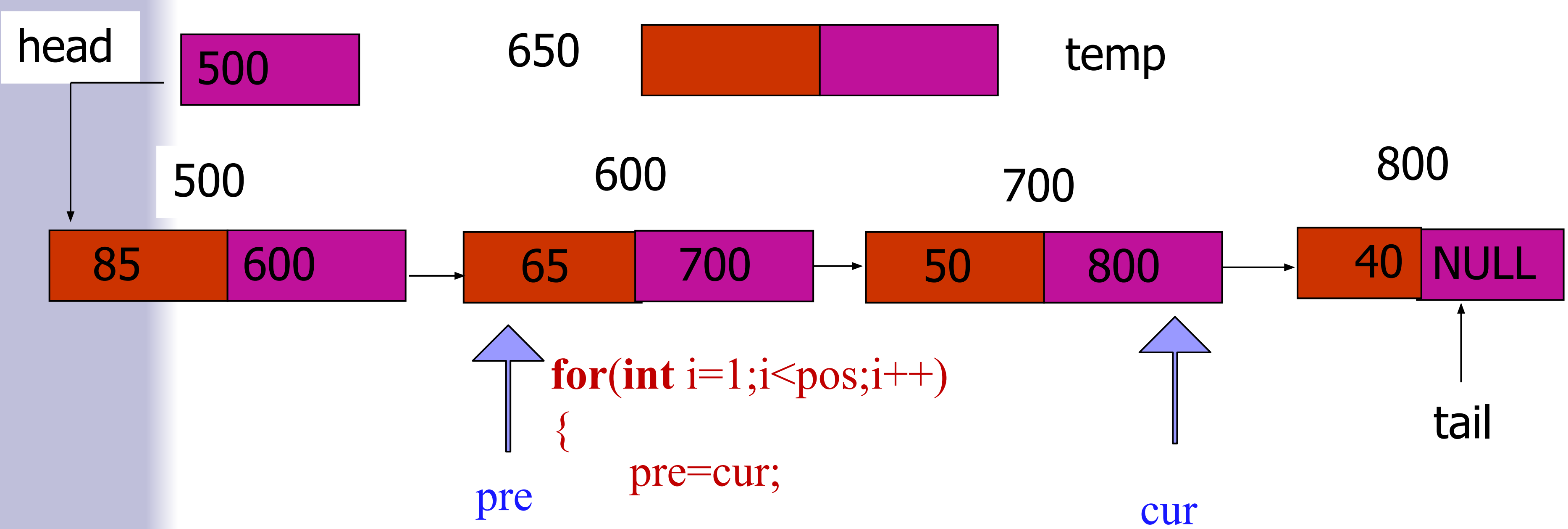
- Now, we will start a loop to reach those specific nodes.
- We move current node through the linked list




```
for(int i=1;i<pos;i++)
{
    pre=cur;
    cur=cur->link;
}
```

Inserting a Node at Specific

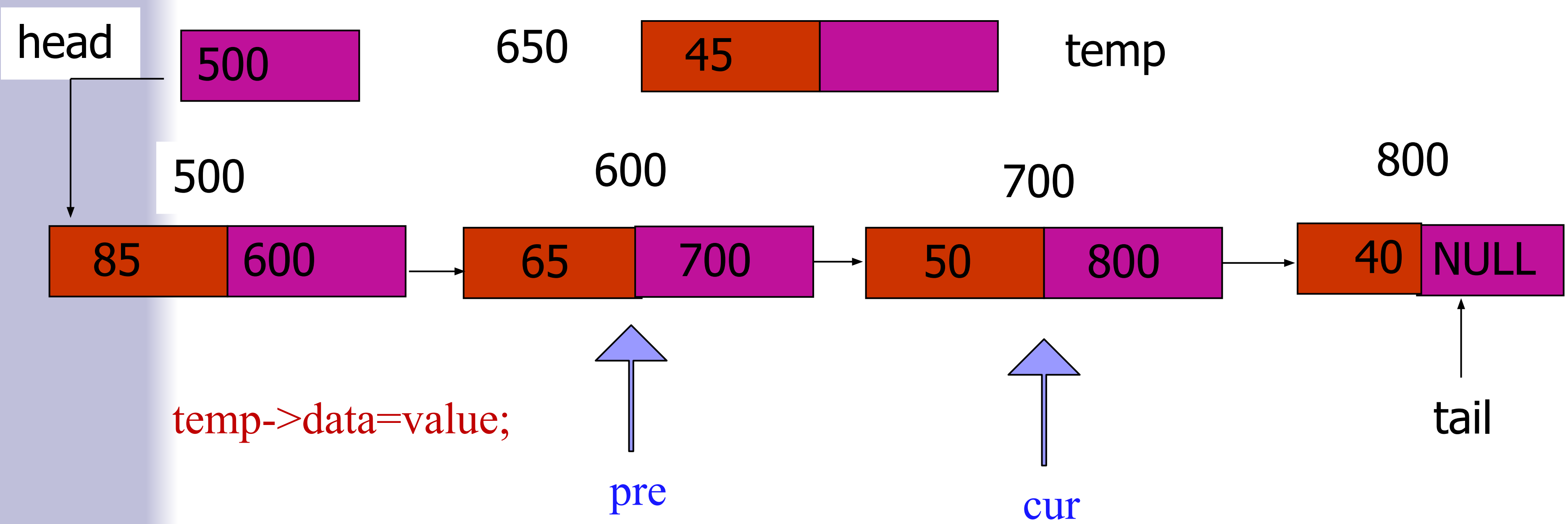
- Now, we will start a loop to reach those specific nodes.
- We move current node through the linked list




```
for(int i=1;i<pos;i++)
{
    pre=cur;
    cur=cur->link;
}
```

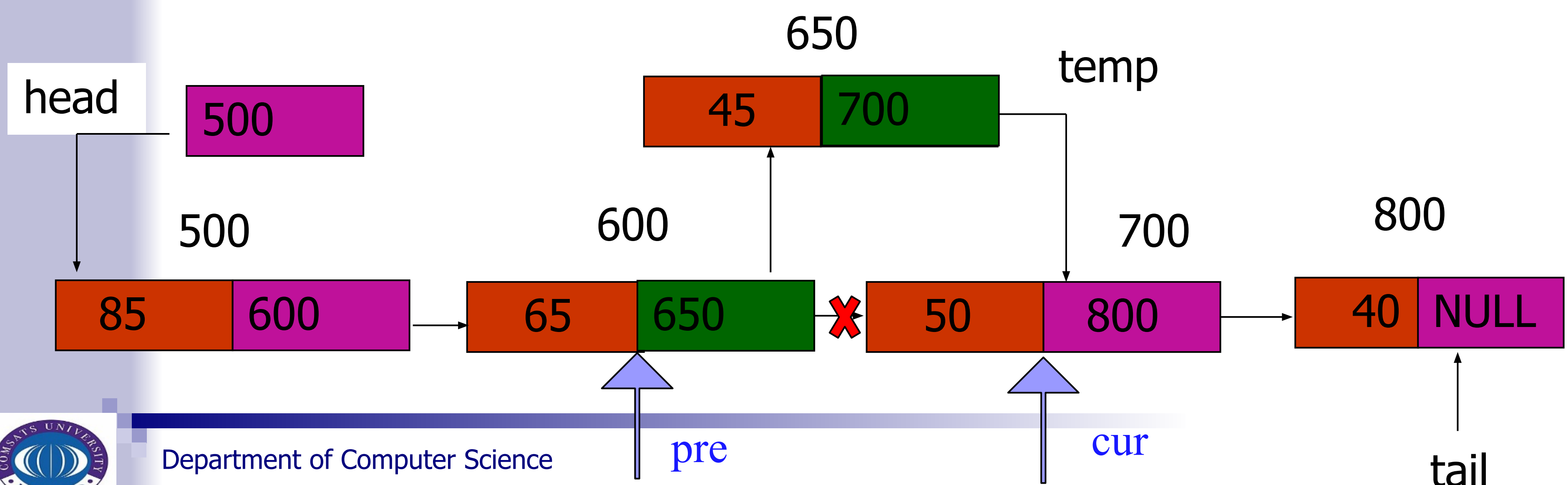
Inserting a Node at Specific

- Now, we will start a loop to reach those specific nodes.
- We move current node through the linked list

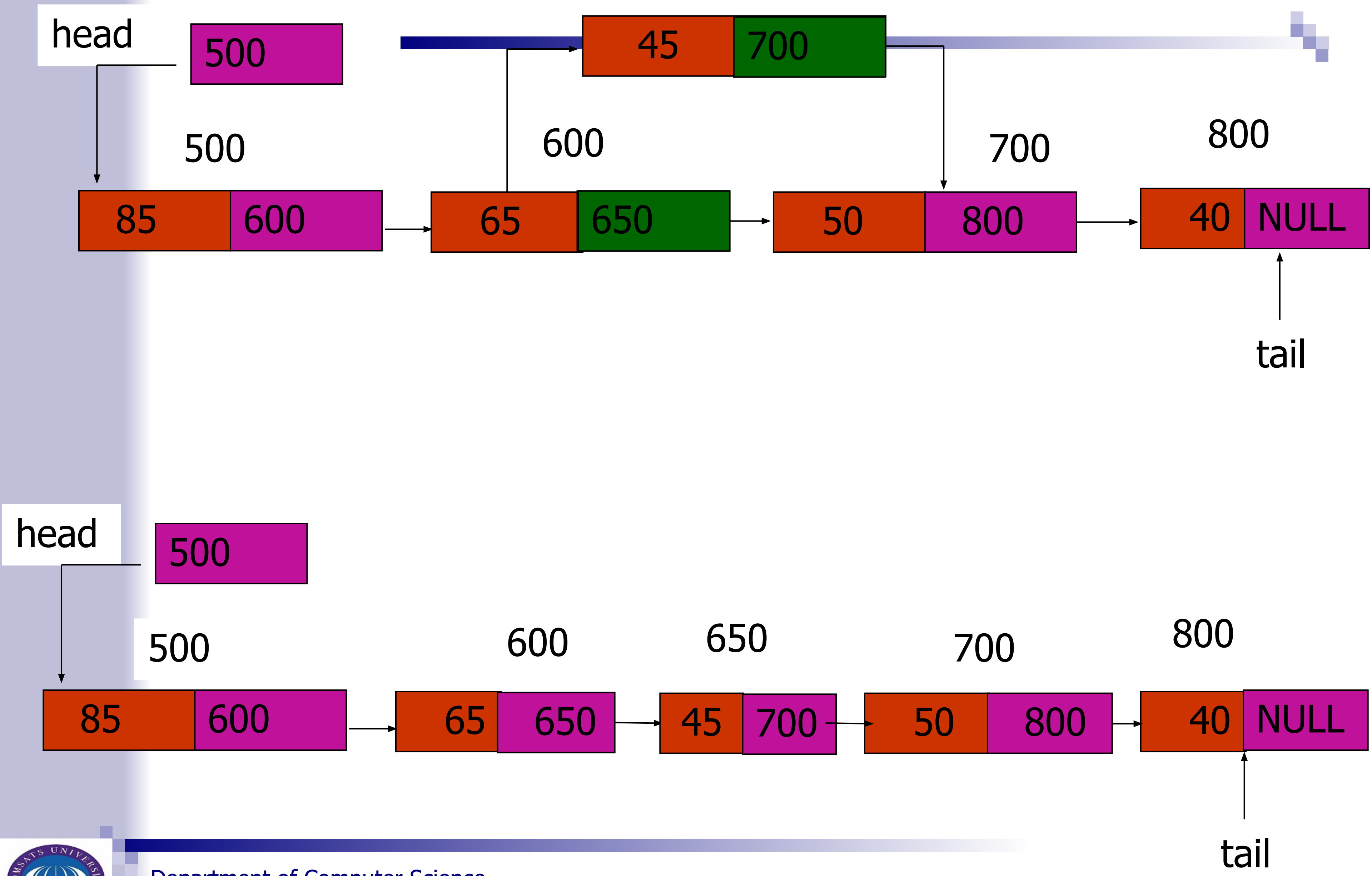


Inserting a Node at Specific

- Now the new node can be inserted between the previous and current node by just performing two steps:
  - Pass the address of the new node in the link field of the previous node. pre->link=temp;
  - Pass the address of the current node in the link field of the new node. temp->link=cur;



Inserting a Node at Specific



;

Inserting a Node at Specific

```
void insert_position(int pos, int value)
{
  node *pre=new node;
  node *cur=new node;
  node *temp=new node;
  cur=head;
  for(int i=1;i<pos;i++)
  {
    pre=cur;
    cur=cur->next;
  }
  temp->data=value;
  pre->next=temp;
  temp->next=cur;
}
```

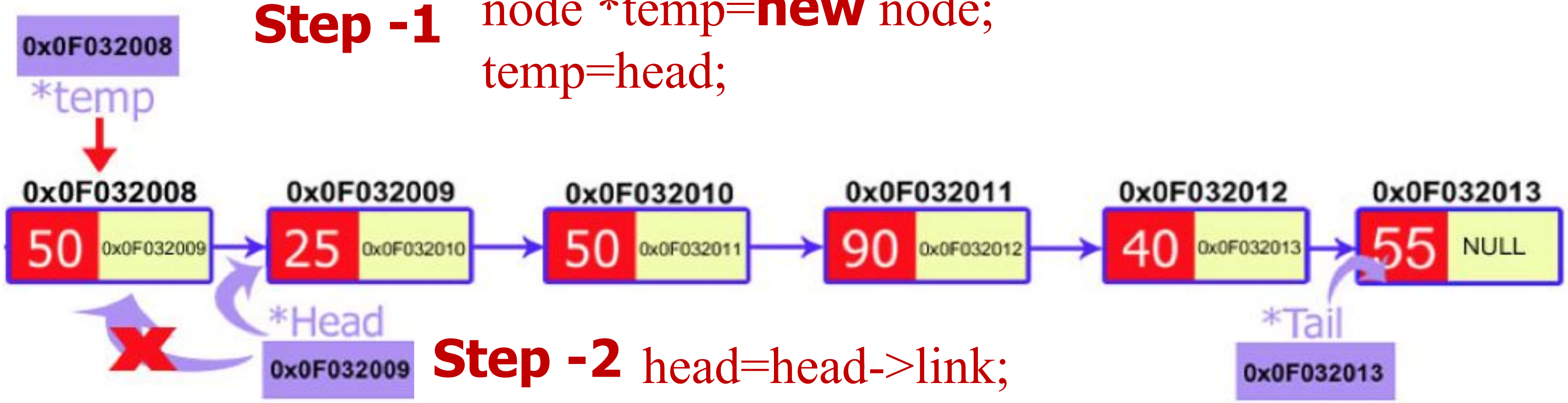
;

Deleting a node from Linked List

- There are also three cases in which a node can be deleted:
  - Deletion at the start
  - Deletion at the end
  - Deletion at a particular position

- The process of deletion includes:
  - Declare a **temp** pointer and pass the address of the first node, i.e. head to this pointer.
  - Declare the second node of the list as head as it will be the first node of linked list after deletion.
  - Delete the temp node.

Step -1 node *temp=**new** node;
temp=head;



Step -2 head=head->link;

Step -3 **delete** temp

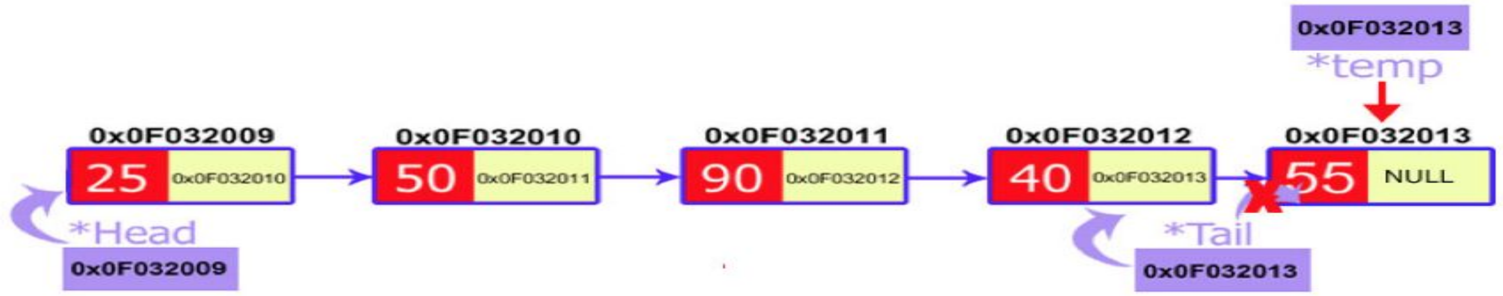
;

Deleting a node from end

- In the case find a node that comes before the last node.
  - This can be achieved by traversing the linked list.
  - We would make two temporary pointers(previous and current) and let them move through the whole linked list.
  - At the end, the previous node will point to the second to the last node and the current node will point to the last node, i.e. node to be deleted.
- We would delete current node and make the previous node as the tail.

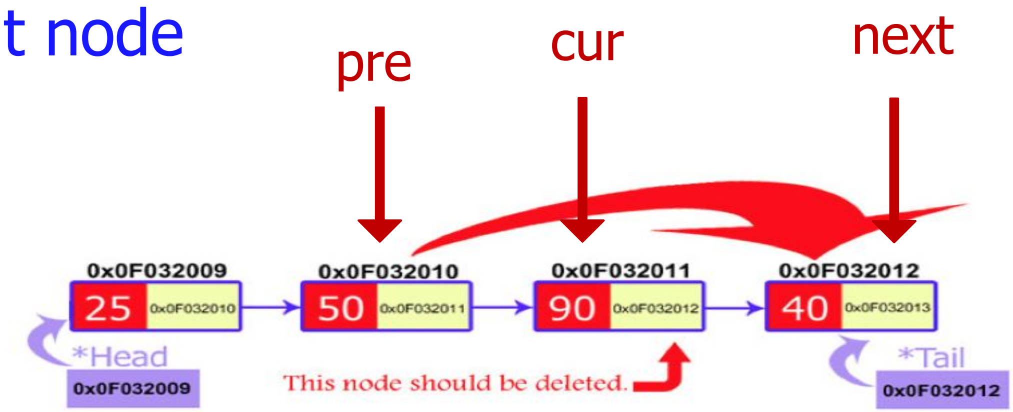
Deleting a node from end

```
void delete_last()
{
  node *current=new node;
  node *previous=new node;
  current=head;
  while(current->next!=NULL)
  {
    previous=current;
    current=current->next;
  }
  tail=previous;
  previous->next=NULL;
  delete current;
}
```



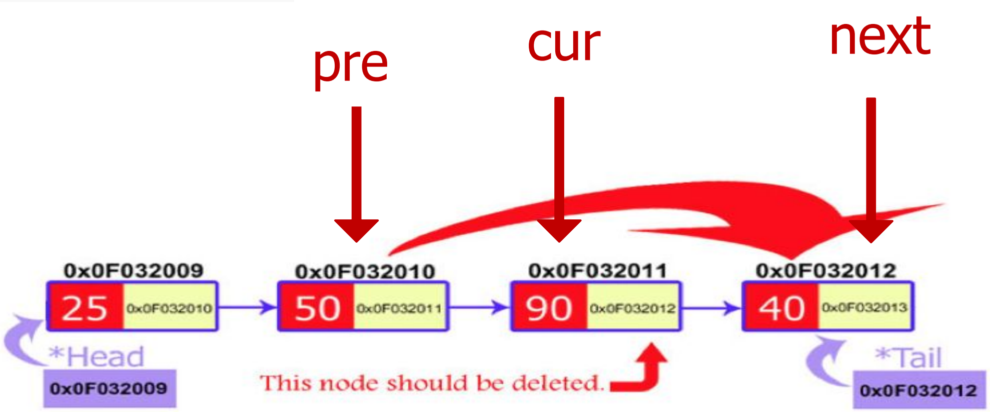
Deleting a node from specific position

- We ask the user to input the position of the node to be deleted.
- After that:
  - Just move two temporary pointers(previous and current) through the linked list until we reach our specific node.
  - Established the link between previous and next node.
    - pass the address of the node that is after current node to the previous pointer.
  - delete current node



Deleting a node from specific position

```
void delete_position(int pos)
{
  node *current=new node;
  node *previous=new node;
  current=head;
  for(int i=1;i<pos;i++)
  {
    previous=current;
    current=current->link;
  }
  previous->link=current->link;
}
```



Analysis of Link List

- As stated earlier, we will be going to analyze each data structure.
- We will see whether it is useful or not.
  - We will see its cost and benefit with respect to time and memory.
- Let us analyze the link list which we have created with the dynamic memory allocation in a chain form.

Insertion operation

Dynamic List(Linked List	Static List(Array Based)
Insert at start: we simply insert the new node after the current node. So 'add' is a one-step operation.	**Insert at start:** Suppose if we have to insert the element in the start of the array, all the elements to the right one spot are shifted.
Insert at end: We insert a new node after the current node in the chain.	**Insert at end:** Suppose if we have to insert the element in the start of the array, all the elements to the left one spot are shifted.
Insert at specific position: For this, we have to change two or three pointers while changing the values of some pointer variables.	**Insert at specific position:** if we have to add an element in the centre of the array, the space for it is created at first. For this, all the elements that are after the current pointer in the array, should be shifted one place to the right.

Insertion operation

```
/*-------------------------------------
 Member function that adds a node at the start
        of  Linked List
---------------------------------------*/
    void insert_start(int value)
    {
        node *temp=new node;
        temp->data=value;
        temp->link=head;
        head=temp;
    }
/*-------------------------------------
```

```
void insert_at_start()
{
cout<<"\n Insertion method" ;
if(si==0 && li==size-1)
{
    cout<<"\n Array is full ";
}
else if(si==-1)
{
    cout<<"\n Array is empty till now and we changed si and li to 0";
    si=li=0;
    cout<<"\n Enter value at "<<li <<" index: ";
    cin>>arr[li];
}
else if(si>0){
    si--;
    cout<<"\n Enter value at "<<--si <<" index: ";
    cin>>arr[si];
}
else{
        shift_right(si, li);
        cout<<"\n Enter value at "<<si<<" index: ";
        cin>>arr[si];
}

}
```

```
void shift_right(int x, int y)
{
    for(int i=y; i>=x; i--)
    {
        arr[i+1]=arr[i];
    }
    li++;
}
```

Insertion operation

```
/*-----------------------------------------
 Member function that adds a node at specific
        position of  Linked List
------------------------------------------*/
    void insert_position(int pos, int value)
    {
        node *pre=new node;
        node *cur=new node;
        node *temp=new node;
        cur=head;
        for(int i=1;i<pos;i++)
        {
            pre=cur;
            cur=cur->link;
        }
        temp->data=value;
        pre->link=temp;
        temp->link=cur;
    }
/*-----------------------------------------
```

Insertion operation

```
/*---------------------------------------
Member function to create a Linked List
---------------------------------------*/
    void createnode(int value)
    {
        node *temp=new node;
        temp->data=value;
        temp->link=NULL;
        if(head==NULL)
        {
            head=temp;
            tail=temp;
            temp=NULL;
        }
        else
        {
            tail->link=temp;
            tail=temp;
        }
    }
/*---------------------------------------
```

```
void insert_at_end()
{
cout<<"\n Insertion method" ;
//if full
if(si==0 && li==size-1)
{
    cout<<"\n Array is full ";
}
else if(si==-1)
{
    cout<<"\n Array is empty till now and we changed si and li to 0";
    si=li=0;
    cout<<"\n Enter value at "<<li <<" index: ";
    cin>>arr[li];
}
else if(li<size-1){
    li++;
    cout<<"\n Enter value at "<<li <<" index: ";
    cin>>arr[li];
}
else{
        shift_left(si, li);
        cout<<"\n Enter value at "<<li<<" index: ";
        cin>>arr[li];
}

}
```

Deletion operation

Dynamic List(Linked List	Static List(Array Based)
Delete from start: we simply insert the new node after the current node. So 'add' is a one-step operation.	**Delete from start:** Suppose if we have to insert the element in the start of the array, all the elements to the right one spot are shifted.
Delete from end: We insert a new node after the current node in the chain.	**Delete from end:** Suppose if we have to insert the element in the start of the array, all the elements to the left one spot are shifted.
Delete from specific position: For this, we have to change two or three pointers while changing the values of some pointer variables.	**Delete from specific position:** if we have to add an element in the center of the array, the space for it is created at first. For this, all the elements that are after the current pointer in the array, should be shifted one place to the right.

Deletion operation

```
/*-----------------------------------------
 Member function that delete a node from
         start of  Linked List
-----------------------------------------*/
    void delete_first()
    {
        node *temp=new node;
        temp=head;
        head=head->link;
        delete temp;
    }
/*-----------------------------------------
```

Deletion operation

```
/*-----------------------------------------------
Member function that delete a node from
    specific location of  Linked List
-----------------------------------------------*/
        void delete_position(int pos)
        {
            node *current=new node;
            node *previous=new node;
            current=head;
            for(int i=1;i<pos;i++)
            {
                previous=current;
                current=current->link;
            }
            previous->link=current->link;
        }
-----------------------------------------------*/
```

Deletion operation

```
/*------------------------------------------
 Member function that delete a node from
          last of  Linked List
------------------------------------------*/
     void delete_last()
     {
         node *current=new node;
         node *previous=new node;
         current=head;
         while(current->link!=NULL)
         {
             previous=current;
             current=current->link;
         }
         tail=previous;
         previous->link=NULL;
         delete current;
     }
/*------------------------------------------
```

Display Function

```
/*---------------------------------------------
 Member function to show elements of a Linked List
-----------------------------------------------*/
        void display()
        {
            node *temp=new node;
            temp=head;
            while(temp!=NULL)
            {
                cout<<temp->data<<"\t";
                temp=temp->link;
            }
        }
/*---------------------------------------------
```

CONCLUSION

Array versus Linked List

- Linked lists are more complex to code and manage than arrays, but they have some distinct advantages.
  - **Dynamic**: a linked list can easily grow and shrink in size.
    - We don't need to know how many nodes will be in the list. They are created in memory as needed.
    - In contrast, the size of a C++ array is fixed at compilation time.
  - **Easy and fast insertions and deletions**
    - To insert or delete an element in an array, we need to copy to temporary variables to make room for new elements or close the gap caused by deleted elements.
    - With a linked list, no need to move other nodes. Only need to reset some pointers.