SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Linked List

swatimali@somaiya.edu

Somaiya TRUST

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Outline

- Linked List – concept
- Linked list representation
- Linked list vs Array
- Linked List types
- Linked List implementations
- Linked List applications
- Summary
- Queries?

Somaiya TRUST

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Linked List

- Random insertions and deletions anywhere in the data structure

- A linked list is kind of like a scavenger hunt. You have a clue, and that clue has a pointer to place to find the next clue. So you go to the next place and get another piece of data, and another pointer. To get something in the middle, or at the end, the only way to get to it is to follow this list from the beginning (or to cheat ;))[1]

Somaiya TRUST

1:Courtesy: https://stackoverflow.com/questions/644167/what-is-a-practical-real-world-example-of-the-linked-list

What is this good for ?

- Reservations/admissions and cancellations
- Todo/shopping lists
- Train cars are linked in a specific order so that they may be loaded, unloaded, transferred, dropped off, and picked up in the most efficient manner possible.
- message delivery on network (message is broken into packets and each packet has a key of the next one so that at the receiver's end , it will be easy to club them) –SLL
- Escalator
- Time sharing problem used by the scheduler during the scheduling of the processes in the Operating system.

A Linked List

- Definition:
  - An ordered collection of homogenous data items
  - Where elements can be added anywhere and removed from anywhere
- Operations:
  - Create an empty linked List
  - check if it is empty
  - Insert: add an element at the desired position
  - Delete: remove the desired element
  - Destroy : remove all the elements one by one and destroy the data structure

Linked List Representation

- Node representation
- Array representation

Index	Data	Address
0	S	5
1		
2	P	0
3	A	6
4		
5	M	-1
6	T	2

Head= 3

Linked List Vs Array

- Student assignment

Types of Linked Lists

- Singly linked list- can be traversed only in one direction
- Circular linked list- last node is connected to first node
- Doubly linked list – can be traversed in both directions

Singly linked list

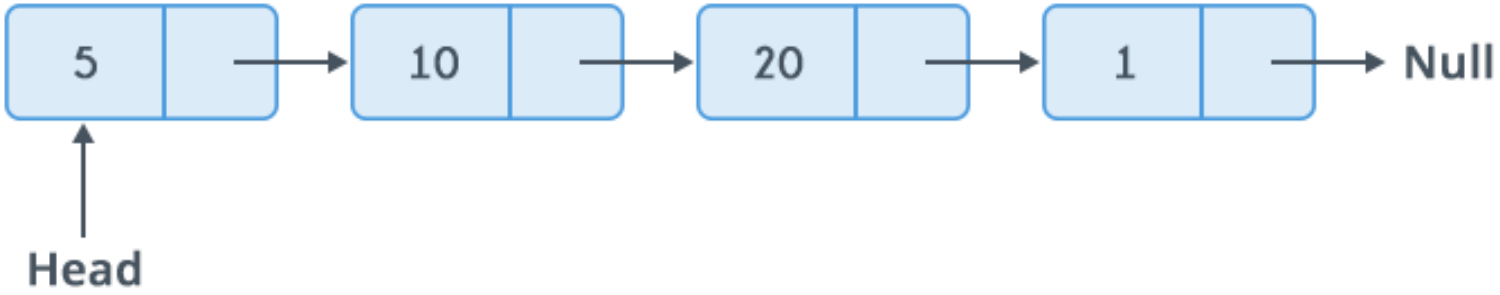
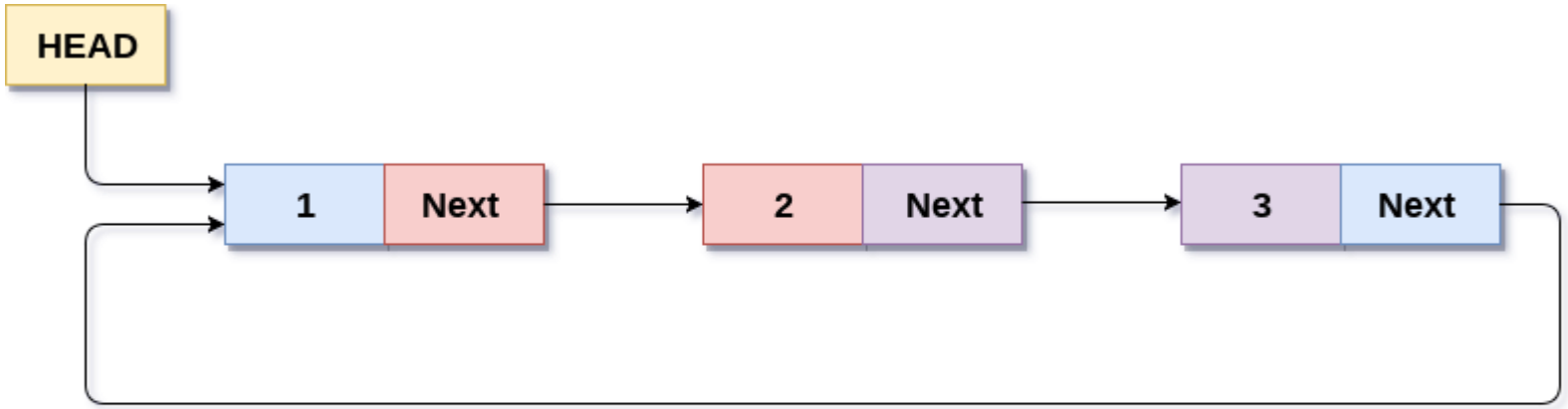


Image courtesy: GeeksforGeeks.org

Circular Linked list



Circular Singly Linked List

Image courtesy: GeeksforGeeks.org

Doubly Linked list

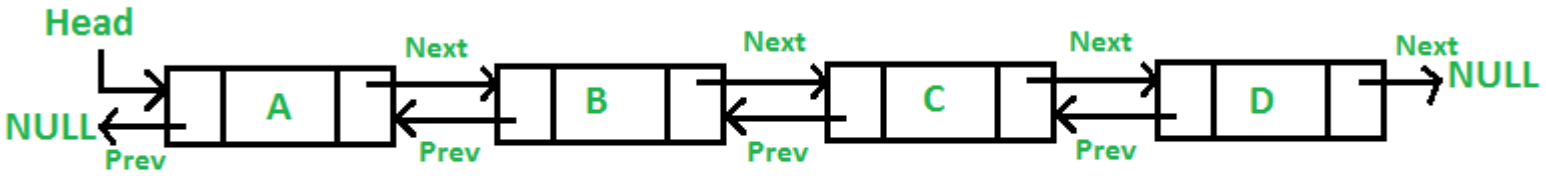


Image courtesy: GeeksforGeeks.org

Linked Lists

- Sorted/ordered linked list
  - Elements are inserted in sorted order
  - Deletions as per user requirements
- Unsorted/unordered linked list
  - Elements are inserted and deleted as per user requirements
  - E.g. insertion in the beginning, in the end, before an element, after an element, etc

Linked Lists

- Sorted linked list
- Unsorted linked list

Insert(26), insert(54), insert(10), delete(26), insert(98), insert(76), delete(10), delete(54), insert(2)

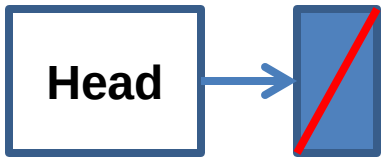
Implementing Linked List

```
Struct NodeType{
                ElementType Element;
                struct NodeType *Next;
        }
```

1. Algorithm LLType CreateLinkedList()

//This Algorithm creates and returns an empty Linked List, pointed by a pointer - head

```
{ createNode(head);
head=NULL;
}
```



Implementing Queue: unordered Linked List

2. LLType Insert(LLType Head, NodeType NewNode)

// This Algorithm adds a NewNode at the desired position in the linked list. Head is the pointer that points to the first node in the linked list

```
{ if (head==NULL) // first element in Queue
        NewNode->Next = NULL;
        head=NewNode;
 Else // General case: insertion before head, in the end, in between


}
```

Insertion in Linked List

- Unsorted list
- Sorted list

..nserting a new node

- Possible cases of `InsertNode`
  1. Insert into an empty list
  2. Insert in front
  3. Insert at back
  4. Insert in middle
- But, in fact, only need to handle two cases
  - Insert as the first node (Case 1 and Case 2)
  - Insert in the middle or at the end of the list (Case 3 and Case 4)

..nsertion Description

- Insertion at the top of the list
- Insertion at the end of the list
- Insertion in the middle of the list

..nsertion Description

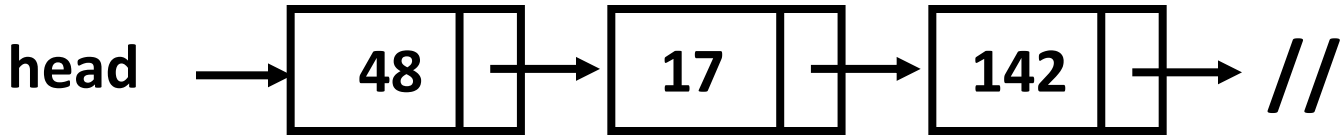
- Insertion at the top of the list
- Insertion at the end of the list
- Insertion in the middle of the list

.nsertion at the top

Steps:

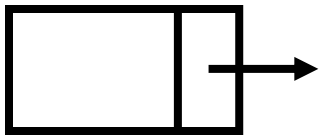
- Create a Node
- Set the node data Values
- Connect the pointers

..nsertion Description

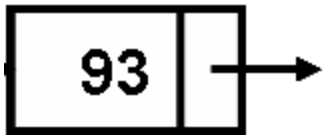


- Follow the previous steps and we get

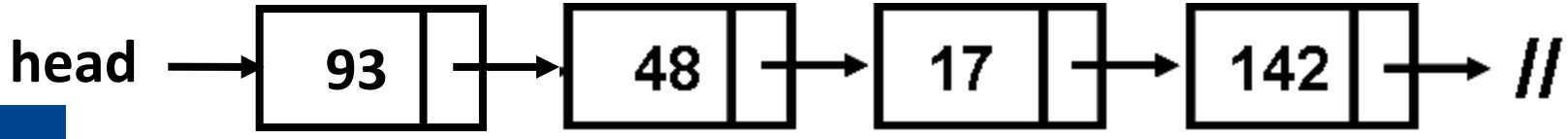
Step 1



Step 2



Step 3



..nsertion Description

- Insertion at the top of the list
- Insertion at the end of the list
- Insertion in the middle of the list

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

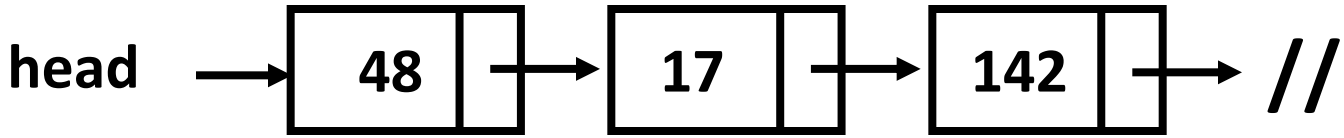
.nsertion at the end

Steps:

- Create a Node
- Set the node data Values
- Connect the pointers

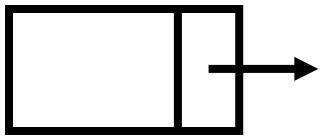
Somaiya TRUST

..nsertion Description

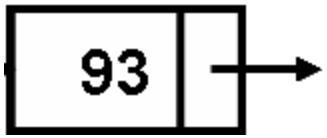


- Follow the previous steps and we get

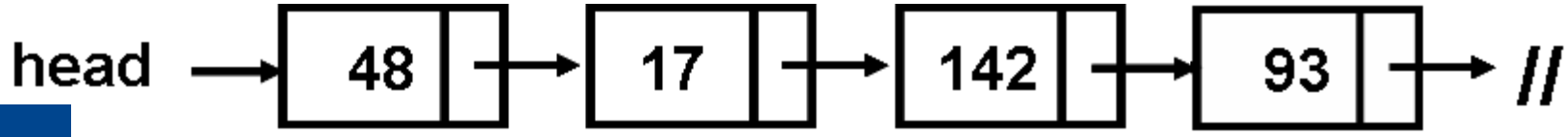
Step 1



Step 2



Step 3



..nsertion Description

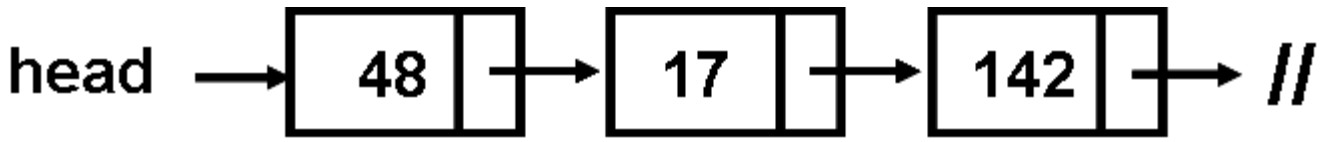
- Insertion at the top of the list
- Insertion at the end of the list
- Insertion in the middle of the list

...sertion in the middle

Steps:

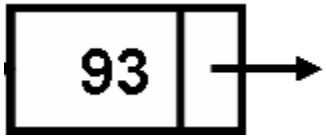
- Create a Node
- Set the node data Values
- Break pointer connection
- Re-connect the pointers

..nsertion Description

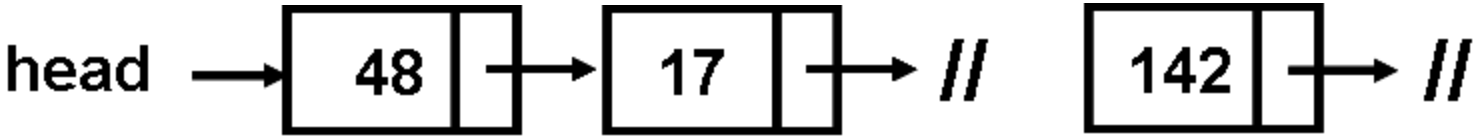


Step 1

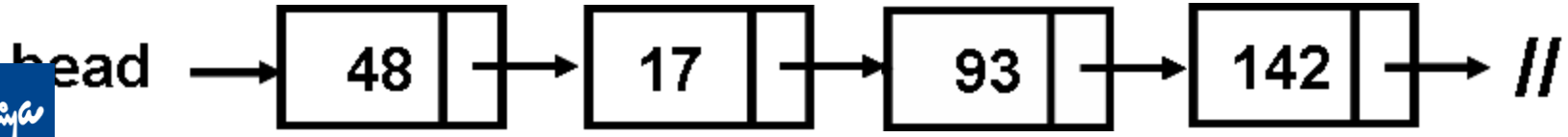
Step 2



Step 3



Step 4



Implementing unordered Linked List.. contd

3. Algorithm LLType Insert(LLType Head, NodeType NewNode)

Case 1: insertion before the head
Newnode->next=Head;
Head=Newnode ;

Case 2: insertion after the last
Struct nodeType *temp;
Temp=head;
While(temp->next!=NULL)
Temp=temp->next;

Temp->next = Newnode;
}

```
if (LL==Null) //inserting the first element
LL=Newnode; exit(0)
temp=LL, prev=null
while((newnode->data > temp->data) && temp!=Null)
{ prev=temp;
temp=temp->next;
}
if(temp==Null) //insertion in the end
Prev->next=Newnode;
if( temp=LL && newnode->data < temp->data)
{ Newnode ->next=temp; LL=Newnode }
Else // general case
Newnode->next =temp;
prev->next = Newnode
```

Implementing unordered Linked List.. contd

3. Algorithm LLType Insert(LLType Head, NodeType NewNode)

Case 3: insertion before some node (Key)

```
struct NodeType *temp;
Temp=Head; Prev = Null;
While(temp->data!=key && temp!=Null)
  prev=temp; temp=temp->next;
If(temp==Null) “Key not found”
Else
        newnode->next=temp;
        Prev->next=temp;
```

Case 2: insertion after some node(Key)

```
Struct nodeType *temp;
While(temp->data!=key && temp!=Null)
temp=temp->next;
If(temp==Null) “Key not found”
Else
Newnode->next =temp->next;
Temp->next = Newnode;
}
```

Implementing Linked List

3. Algorithm ElementType Delete(LLType Head, ElementType ele)

//This algorithm returns ElementType ele if it exists in the List, an error message otherwise. Temp and current are the a temproary nodes used in the delete process.

```
{ if (Head==NULL)
        Print "Underflow"
        exit;
 Else
        {0. search for element
        1. element doesn't exist in list
        2. deletion in unsorted list
        3. deletion in sorted list
        }
```

Deleting a node

- `Operation DeleteNode`
  - Delete a node with the value equal to `x` from the list.
  - If such a node is found, return its position. Otherwise, return 0.
- Steps
  - Find the desirable node (similar to `FindNode`)
  - Release the memory occupied by the found node
  - Set the pointer of the predecessor of the found node to the successor of the found node
- Like `InsertNode`, there are two special cases
  - Delete first node
  - Delete the node in middle or at the end of the list
  - Delete the last remaining node in the list

Deletion Description

- Deleting from the top of the list
- Deleting from the end of the list
- Deleting from the middle of the list

Deletion Description

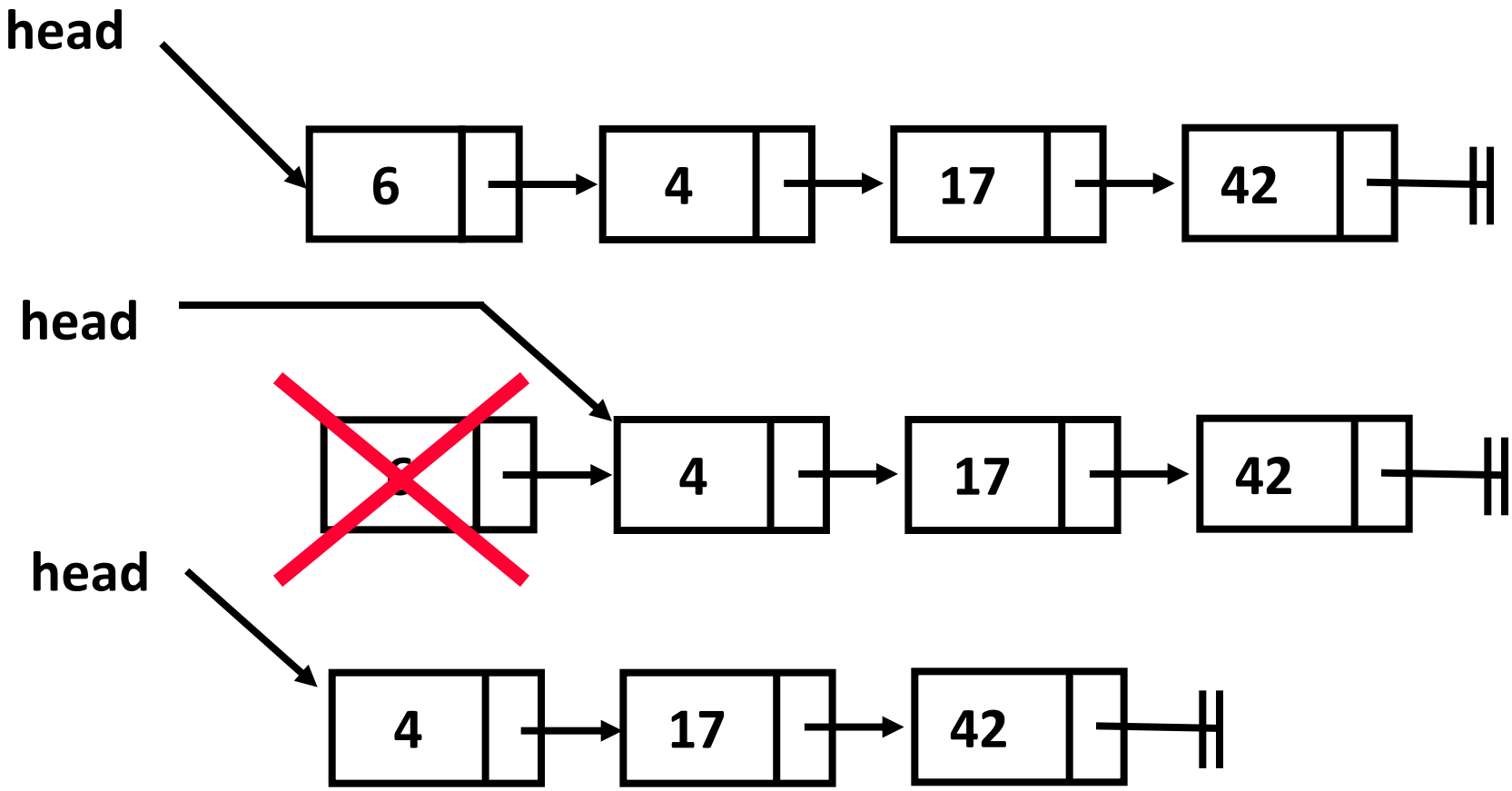
- Deleting from the top of the list
- Deleting from the end of the list
- Deleting from the middle of the list

Deleting from the top

Steps

- Break the pointer connection
- Re-connect the nodes
- Delete the node

Deletion Description



Deletion Description

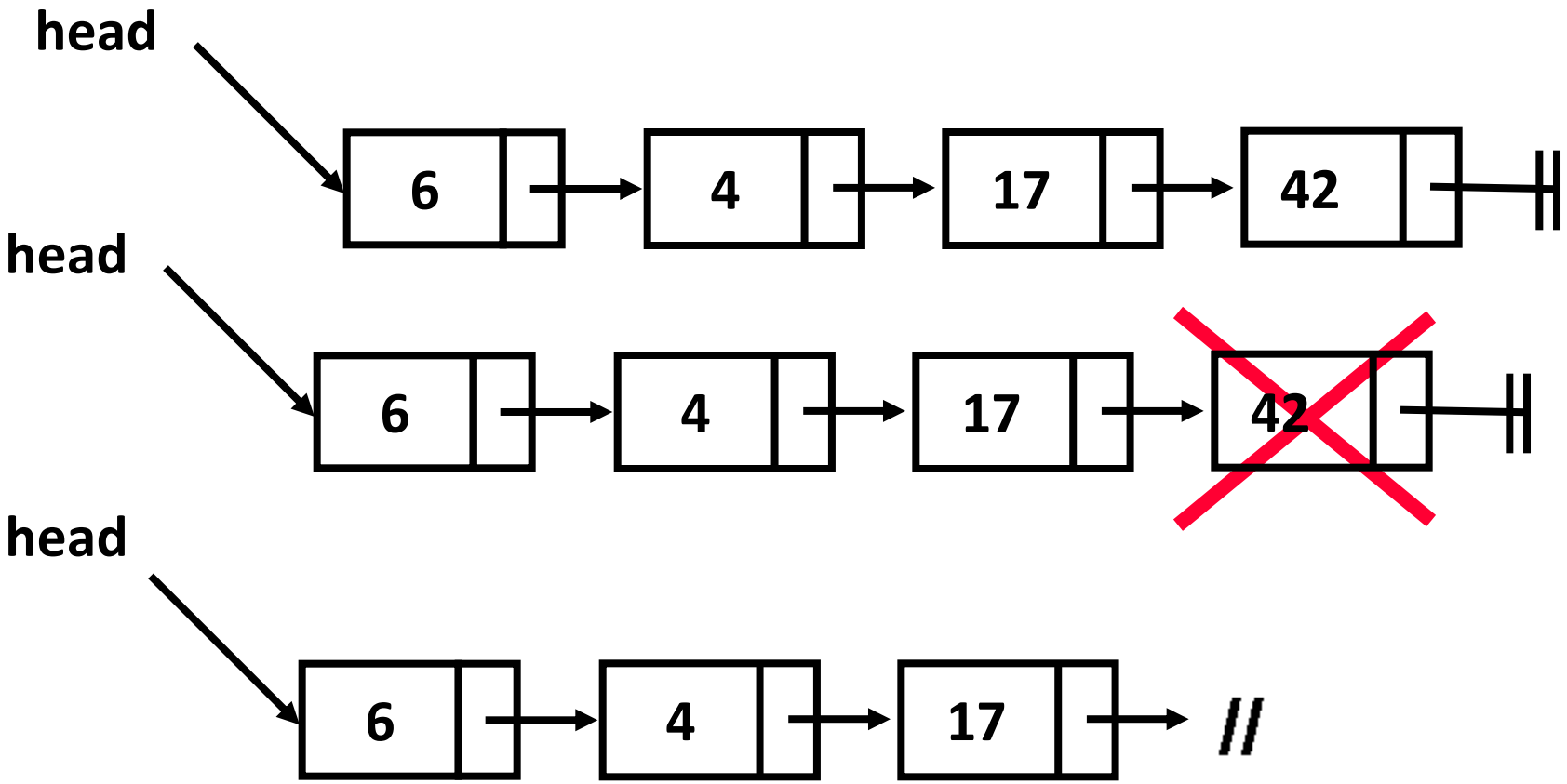
- Deleting from the top of the list
- Deleting from the end of the list
- Deleting from the middle of the list

Deleting from the end

Steps

- Break the pointer connection
- Set previous node pointer to NULL
- Delete the node

Deletion Description



Deletion Description

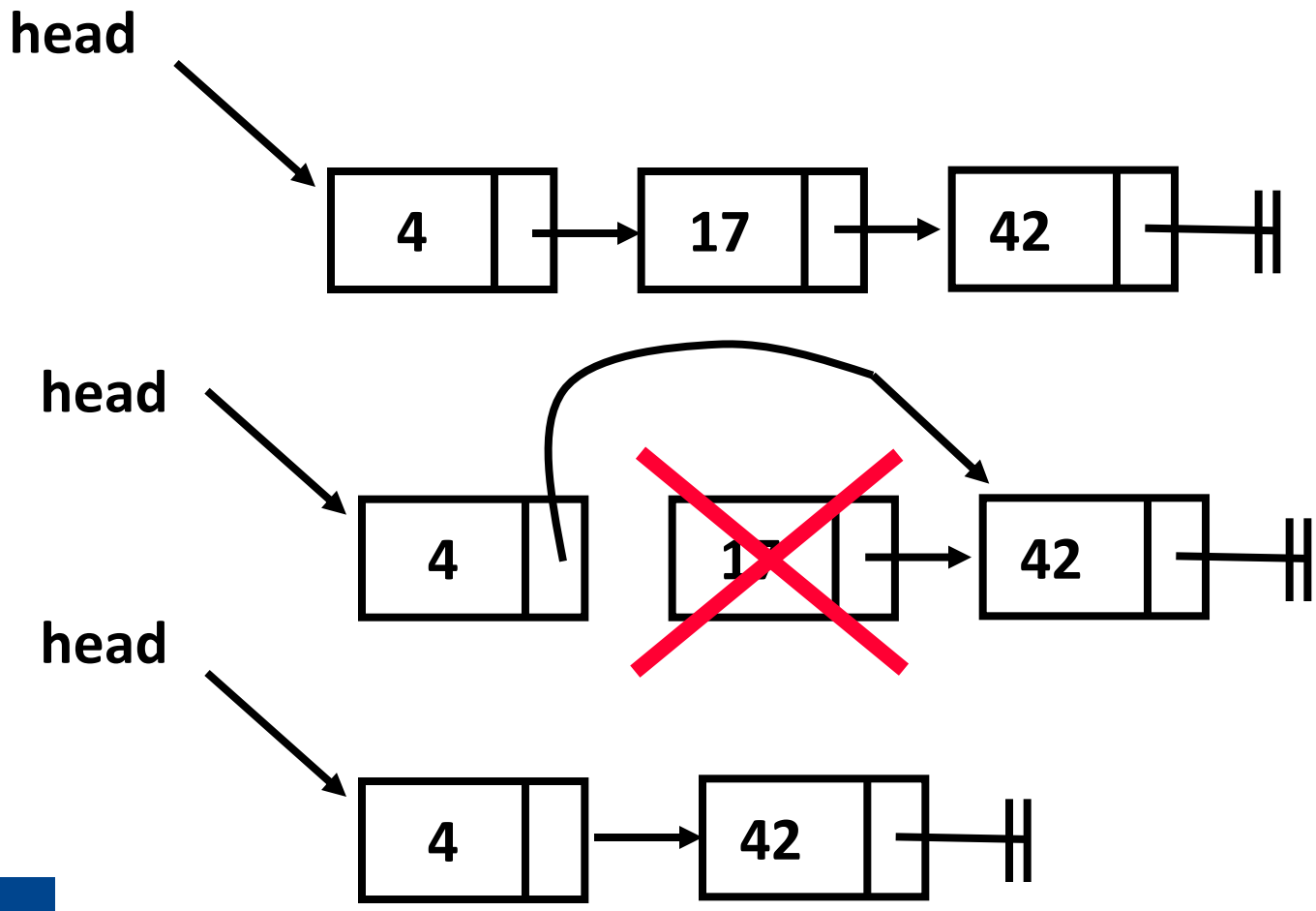
- Deleting from the top of the list
- Deleting from the end of the list
- Deleting from the middle of the list

Deleting from the Middle

Steps

- Set previous Node pointer to next node
- Break Node pointer connection
- Delete the node

Deletion Description



Implementing unordered Linked List

3. Algorithm NodeType Search(LLType Head, ElementType Key)

//This algorithm returns NodeType node which contains the ‘keyvalue” being searched.

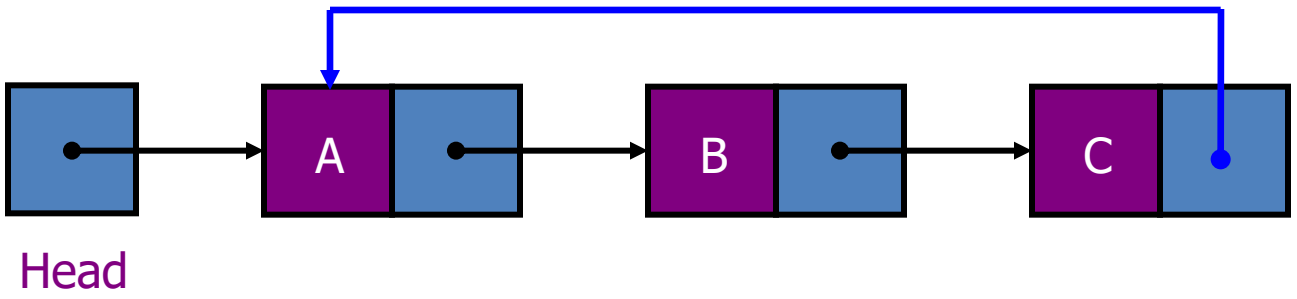
```
{ if (Head==NULL)
        Print “element doesn’t exist”
        exit;
 Else
```

```
Insertion into ordred LL
{
//case 1: insertion into empty LL
if (Head==Null)
Head=Newnode
Else
{Temp=Head, prev=Null
while(Temp->data < Newnode->data && temp!=Null)
            Prev=Temp; Temp=Temp->next
  //Case 2: insertion before first node
 if(Prev==Null)
            { Newnode->next=Head;
        Head=Newnode;
            }
  else  //General case
     {
            Newnode->next=temp;
            Prev->next=Newnode
     }
}//InsertOrdredLL
```

```
DeleteOrdredLL
{
If (Head==Null)
  { Print Underflow..; Exit();
Else
  { //Search the node
    Temp=Head, Prev=Null
    While(Temp->data <Key)
         Prev=Temp; Temp=Temp->next
    if(Temp->data) != Key) //Value not found
     Print Error, Exit(0)
    Elseif(temp->data ==Key && Prev=Null)//Deleting first element
        Head=Head->next
    Elseif(Temp->data==key && Temp->next==NULL)// Deleting only element
in LL
        Head=Null
    Else //General case, including deleting last element
        Prev->next=Temp->next
    }//Else
Return Temp;
 }
```

Variations of Linked Lists

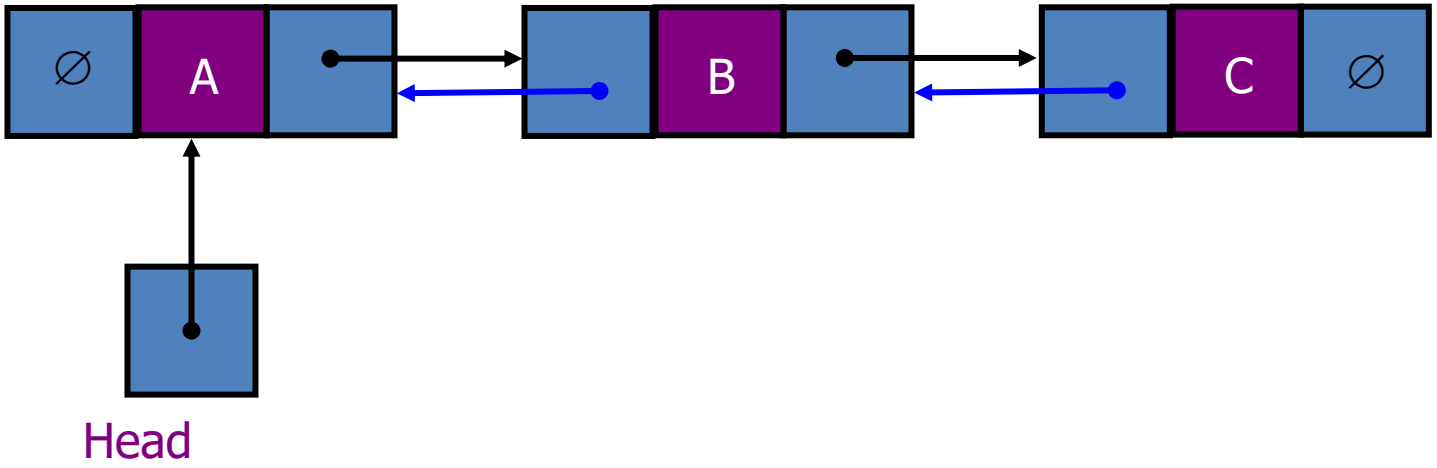
- *Circular linked lists*
  - The last node points to the first node of the list



  - How do we know when we have finished traversing the list? (Tip: check if the pointer of the current node is equal to the head.)

Variations of Linked Lists

- *Doubly linked lists*
  - Each node points to not only successor but the predecessor
  - There are two `NULL:` at the first and last nodes in the list
  - Advantage: given a node, it is easy to visit its predecessor. Convenient to traverse lists backwards



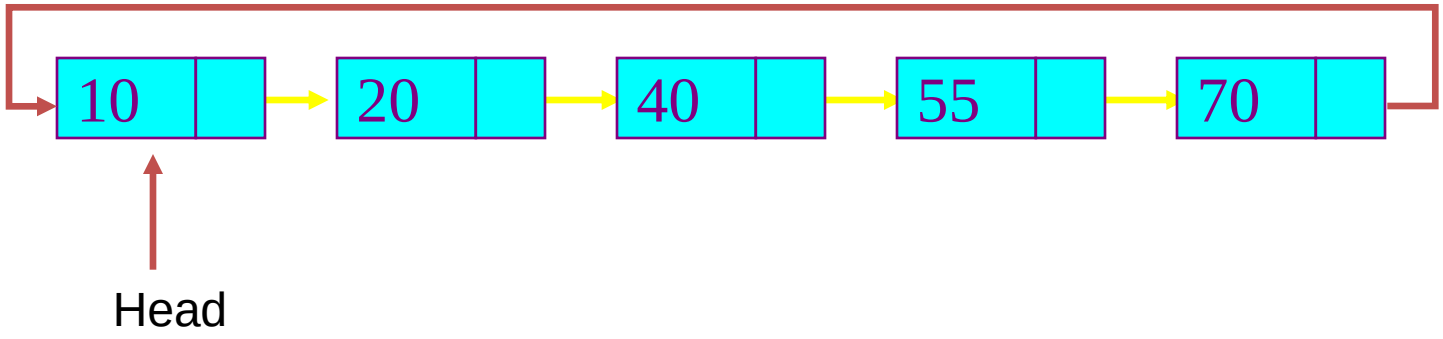
Array versus Linked Lists

- 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.

Circular Linked List

Circular Linked Lists

- A Circular Linked List is a special type of Linked List
- It supports traversing from the end of the list to the beginning by making the last node point back to the head of the list
- A Rear pointer is often used instead of a Head pointer



Motivation

- Circular linked lists are usually sorted
- Circular linked lists are useful for playing video and sound files in “looping” mode
- They are also a stepping stone to implementing graphs

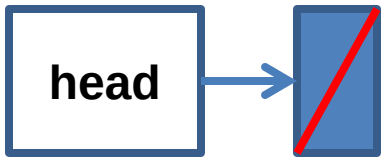
Implementing Circular Linked List

```
Struct NodeType{
                ElementType Element;
                NodeType *Next;
        }
```

1. Algorithm CLLType CreateCLL()

//This Algorithm creates and returns an empty Circular Linked List, pointed by a pointers- Head

```
{ createNode(Head);
Head =NULL;
}
```



Insertion Description

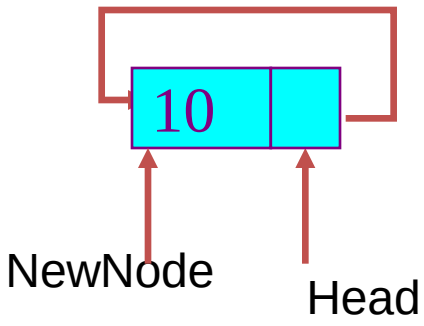
- Insertion at the top of the list
- Insertion at the end of the list
- Insertion in the middle of the list

Insert Node

- Insert into an empty list

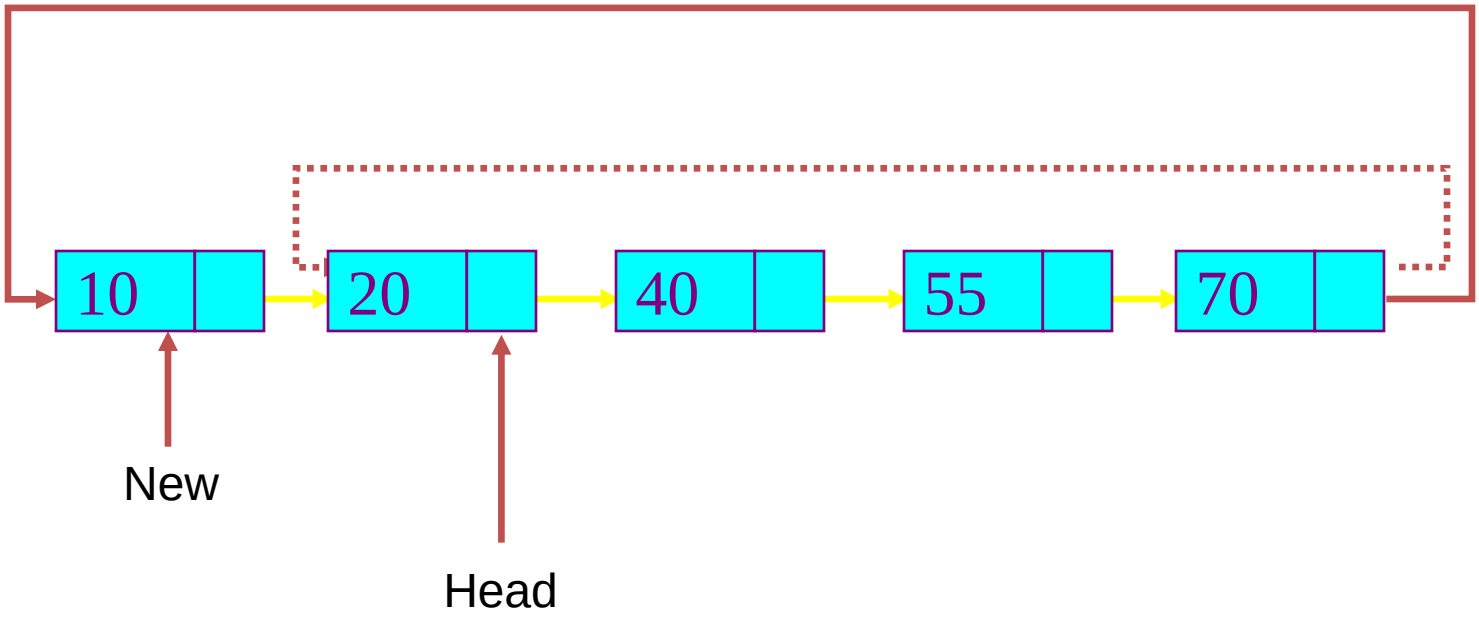
```
CreateNode(NewNode)

If(Head==Null)
{
Head = NewNode;
Head->next = Head;
}
```

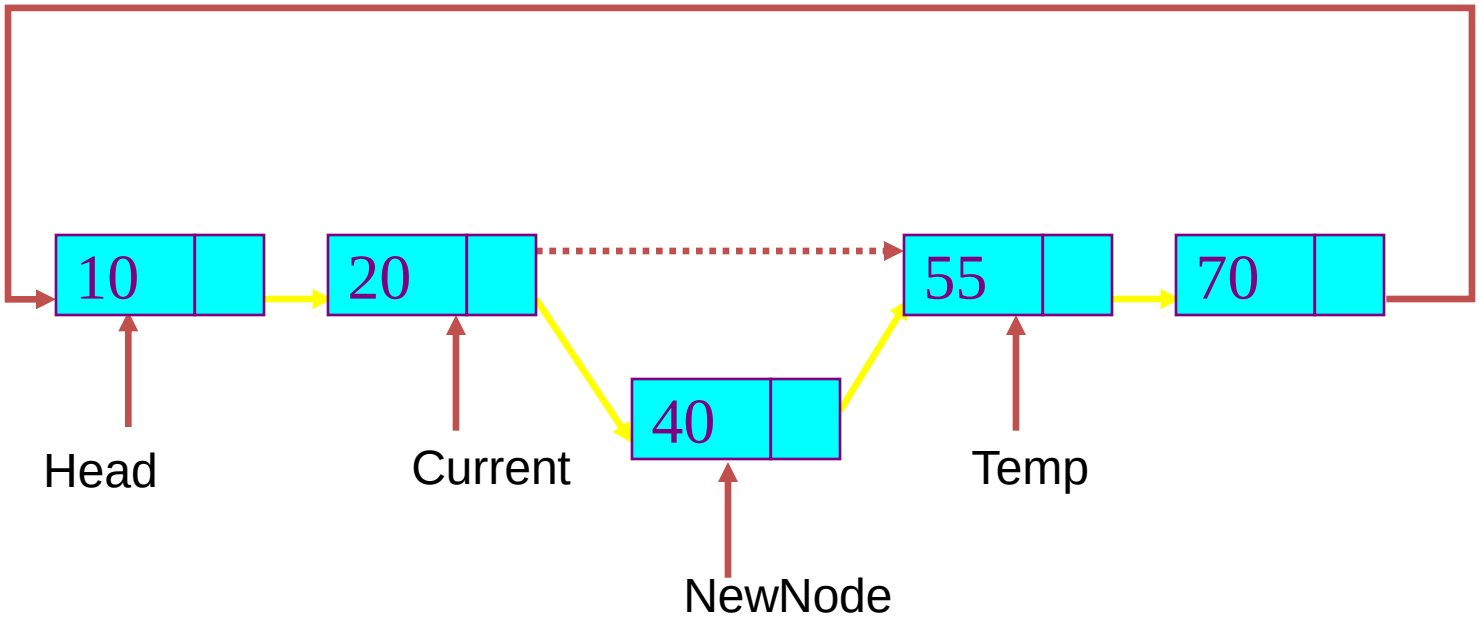


- Insert to head of a Circular Linked List

1. Add NewNode before Head
2. Locate last node and make its next point to new node

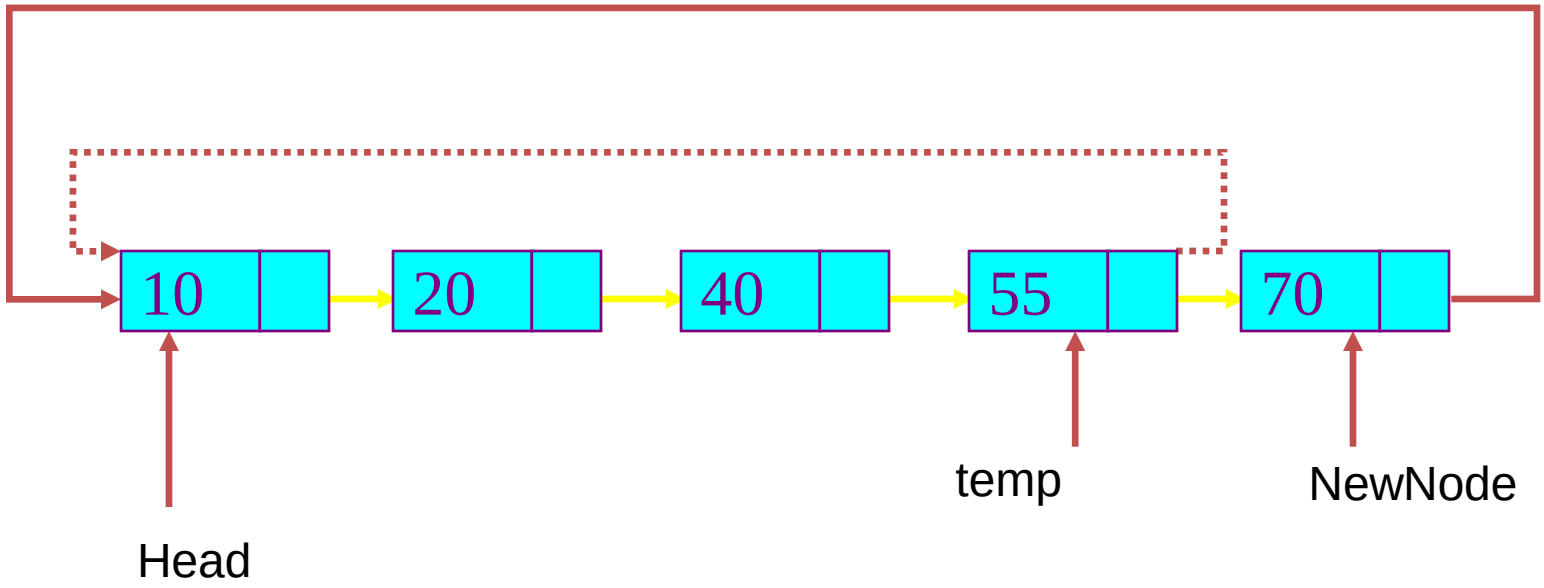


- Insert to middle of a Circular Linked List between `Temp` and `Current`



- Insert to end of a Circular Linked List

1. Find the position for NewNode
2. Check if its after last node
3. Connect NewNode's next field to Head

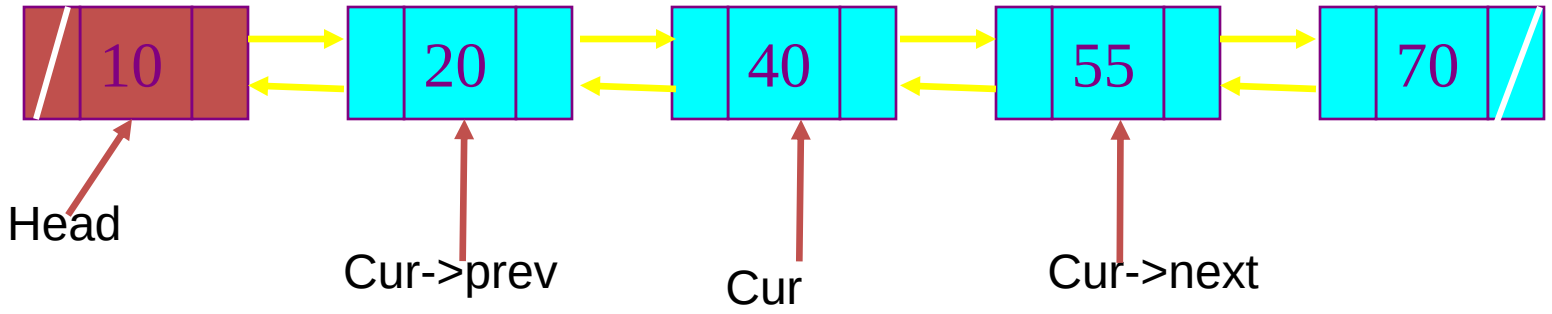


Deletion Description

- Deleting from the top of the list
- Deleting from the end of the list
- Deleting from the middle of the list

Doubly Linked Lists

- In a Doubly Linked List each item points to both its predecessor and successor
  - `prev` points to the predecessor
  - `next` points to the successor



Motivation

- Doubly linked lists are useful for playing video and sound files with “rewind” and “instant replay”
- They are also useful for other linked data which require “rewind” and “fast forward” of the data

Doubly Linked List Definition

```
struct Node{
        int data;
        Node* next;
        Node* prev;
};
Node* Head, List1, List2;
```

Insertion into DLL

1. initialize data structure
CreateNode(Head);
Head=NULL

Search // searching in sorted list for insertion operation
temp=Head
while(NewNode->data > temp->data && temp->next!=Null)
    temp=temp->next;

2. Insertion
createNode(NewNode) // CreateNode creates a node, adds data
 to it and assigns both addresses NULL value
2A. insertion of first node

if(Head==NULL)
head=NewNode;

2B. insertion at the end

```
if(NewNode->data > temp->data && temp->next==Null)
{ NewNode->prev = temp
  temp->next= NewNode
}
```

2C. Insert before the head node

```
if(NewNode->data < temp->data && temp==Head)
{ NewNode->next = temp;
  temp->prev = NewNode
  Head= Newnode;
}
```

2D. Insertion in between

```
{
NewNode-> Next = temp;
NewNode -> prev = temp->prev
temp-> prev->next= NewNode
temp->prev = NewNode
}
```

Deletion in DLL

- 1. Deletion from empty DLL
- 1A. Element doesn't exist
- 2. Deletion of last/only node in DLL
- 3. Deletion of Head node
- 4. Deletion of last node
- 5. General case

Deletion in DLL

1. if (Head==Null)

  Print" Underflow.. Error“

//Search

temp=Head

while(temp!=Null && Temp->data <SearchKey)

temp=temp-> next

Deletion in DLL

1A. // unsuccessful search e.g. SearchKey ==1 or 30 or 500
if(Temp->data > SearchKey || (temp-> next ==Null && temp->data <SearchKey)))
print "Element does not exist"

2. Deletion of only node in list
if(temp->data == SearchKey && temp=Head && temp->next ==NULL)
{ Head=NULL;
return(Temp)
}

4. //e.g. SearchKey= 100 Deletion of last node
if(temp->data == SearchKey && temp->next ==NULL)
{ temp->prev->next = NULL
return(temp)
}

3. Deletion in general

```
{ temp->next->prev= temp->prev
temp->prev->next= temp->next
}
```

5. Deletion of first node in list

```
if(temp==head && temp->data == SearchKey)
{ head=head->next
temp->next->prev= NULL

return(temp)
}
```

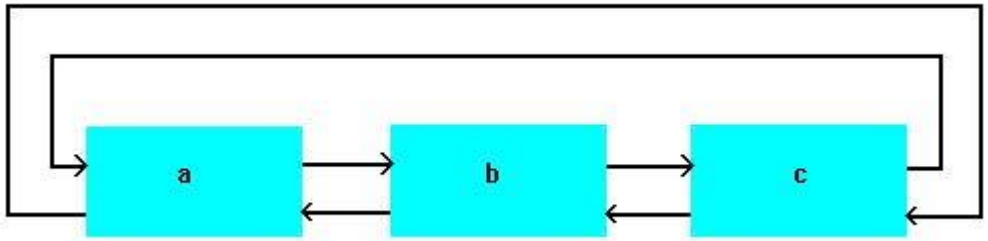
Advantages of Linked Lists

- The advantages of linked lists include:
  - Overflow can never occur unless the memory is actually full.
  - Insertions and deletions are *easier* than for contiguous (array) lists.
  - With large records, moving pointers is easier and faster than moving the items themselves.

Disadvantage of Linked Lists

- The disadvantages of linked lists include:
  - The pointers require extra space.
  - Linked lists do not allow random access.
  - Time must be spent traversing and changing the pointers.
  - Programming is typically trickier with pointers.

Homework: Circular doubly linked list



Queries?

Thank you!