DATA Structures

Dr. Asmaa Gad El-Kareem

Lecture #3

Data Structures

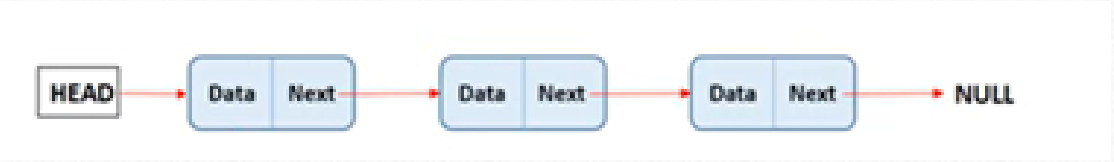
Lecture 3: Linked Lists

Overview

- **Linked lists**
  - Abstract data type (ADT)
- **Basic operations of linked lists**
  - Insert, find, delete, print, etc.
- **Variations of linked lists**
  - Circular linked lists
  - Doubly linked lists

Linked Lists

What is a Linked List?



- A linked list is a collection of nodes, where each node contains some data along with information about the next node.

How it works?

- A linked list uses **non-contiguous** memory locations and hence requires each node to remember where the **next node** is.

Node Organization

What is a Node?

- Node is a combination of Data and Link (Next).
- Data: is the part where the actual data is stored.
- Link: is the link (pointer) to the next element of the list.

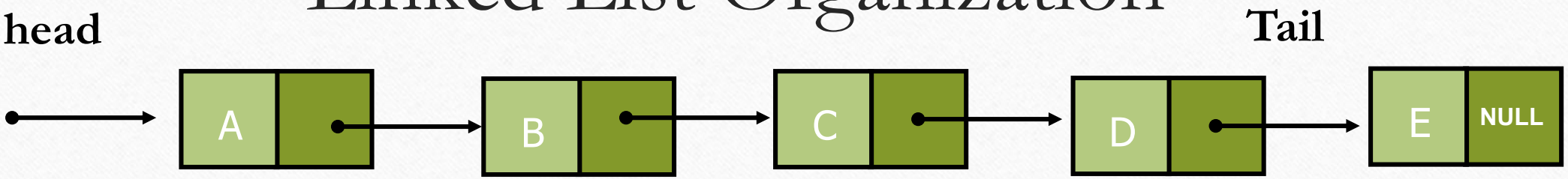
Two ways of implementing

- It could be either an **index** to an array element (array implementation)

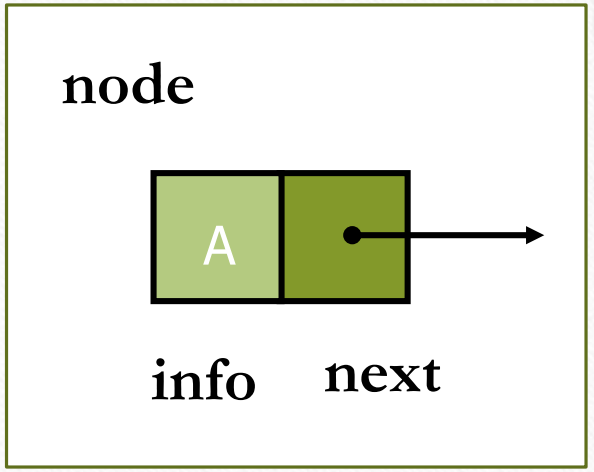
OR

- a **pointer variable** containing the address of the next element (pointer)

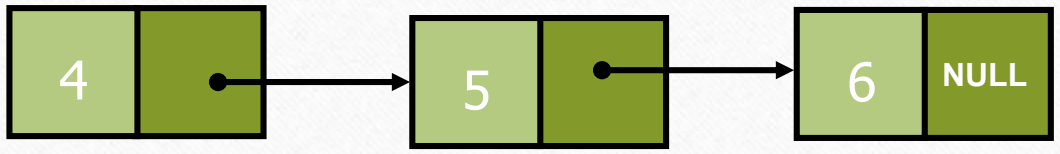
Linked List Organization



- A **linked list** is a series of connected nodes that form a linear sequence
- Each **node** contains two fields:
  - A piece of data being stored (any type) →info
  - Pointer to the next node in the list →next
- **Head**: pointer to the first node
- **Tail**: pointer to the last node
- The last node points to **`NULL(address 0)`**



Example: Representation

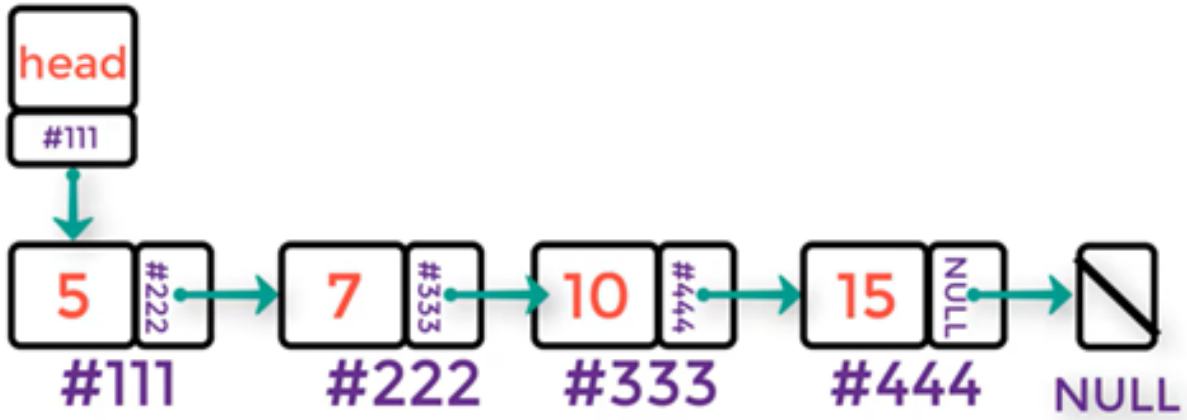


1) The above linked list contains 3 nodes located at different memory locations each node has a pointer that points to the next node.

2) The node that have a **NULL** indicates the it is the end of the list (**last node**).

In C/C++ programs either **NULL** or **0** can be used to indicate the end of the list.

Linked Lists



C++

```
print
head->item; 5
head->inext; #222
head->inext->item; 7
```

Java

```
print
head.item; 5
head.next; #222
head.next.item; 7
```

C#

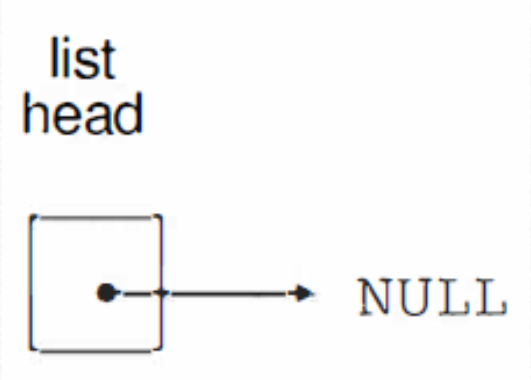
```
print
head.item; 5
head.next; #222
head.next.item; 7
```

Python

```
print
head.item 5
head.next #222
head.next.item 7
```

Empty List

- If a list currently contains 0 nodes, it is the empty list
- In this case the list head points to NULL



A singly linked list, or Simply a linked list Class

- We use two classes: **`Node`** and **`List`**
- Declare `Node` class for the nodes: No memory is allocated at this time
  - `info`: `int`-type data in this example
  - `next`: a pointer to the next node in the list

```
class Node {
public:
   int    info;     // data
   Node*    next;     // pointer to next
};
```

A Simple Linked List Class

Declare `List`, which contains

- `head`: a pointer to the first node in the list.
- `Tail`: a pointer to the last node in the list.

  Since the list is empty initially, `head and tail` is set to `NULL`

- Operations on `List`

A Simple Linked List Class

```
class List
{
private:
    Node* head;
public:
    list( ) // constructor
    {
        head=0; //NULL
    }
    ~list( ); //destructor
    int isEmpty( );
    void addToHead(int);
    void addToTail(int);
    void addNode(int, int);
    void printList();
    int deleteFromHead();
    int deleteFromTail();
    int deleteNode(int);
    int countList();
    int findNode(int);
};
```

A Simple Linked List Class

- Operations of `List`
  - **`List`**: `initialize head and tail to Null.` →**`constructor`**
  - **`~list`**: `destroy all nodes of the list.` →**`destructor`**
  - **`isEmpty`**: determine whether or not the list is empty.
  - **`addToHead`**: insert a new node at the front of the list.
  - **addToTail**: insert a new node at the end of the list.
  - **addNode**: insert a new node at a particular position.

A Simple Linked List Class

- Operations of `List`
  - **`findNode`**: find a node with a given value.
  - **`DeleteNode`**: delete a node with a given number.
  - **`deleteFromHead`**: delete the first node from the list.
  - **`deleteFromTail`**: delete the last node from the list.
  - **`printList`**: print all the nodes in the list.
  - **`countList`**: print the number of nodes in the list.

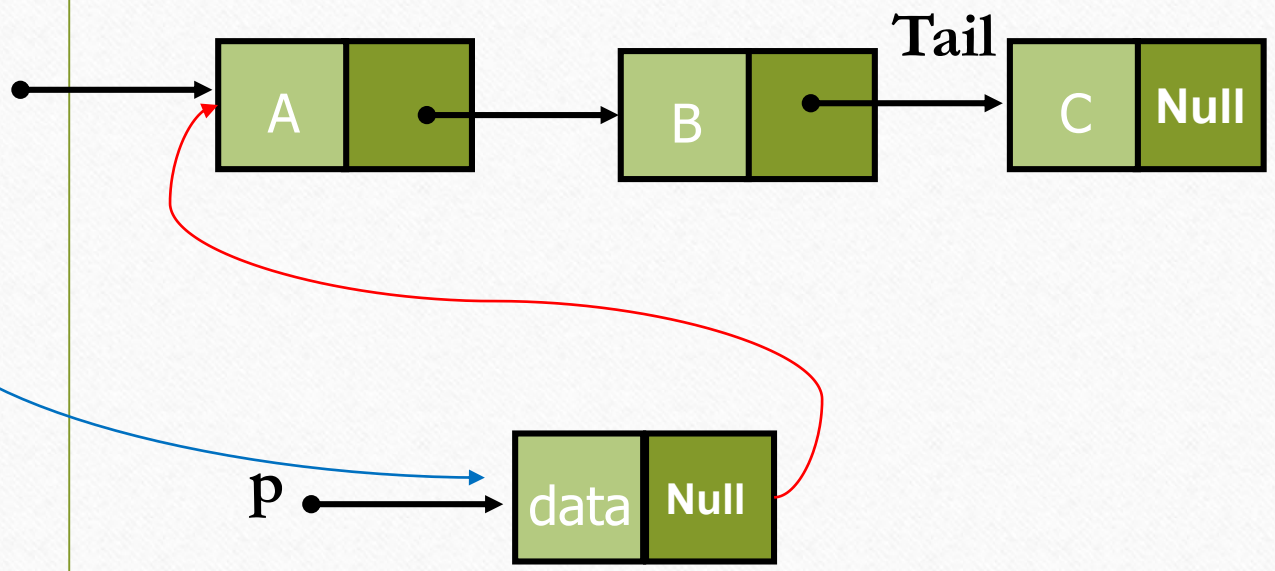
AddToHead Function

- To insert a node at the front of the list (this list is used to store integer data):

1. An empty node is created (create new node)
2. The node's info member is initialized to a particular integer (fill data part)
3. Because the node is being included at the front of the list, the next member becomes a pointer to the first node on the list; that is, the current value of head.
4. The new node precedes all the nodes on the list, but this fact has to be reflected in the value of head; otherwise, the new node is not accessible. Therefore, head is updated to become the pointer to the new node .

AddToHead Function

```
void List::addToHead(int data)
{
    Node* p= new Node;
    p->info=data;
    p->next=head;
    head=p;
}
```



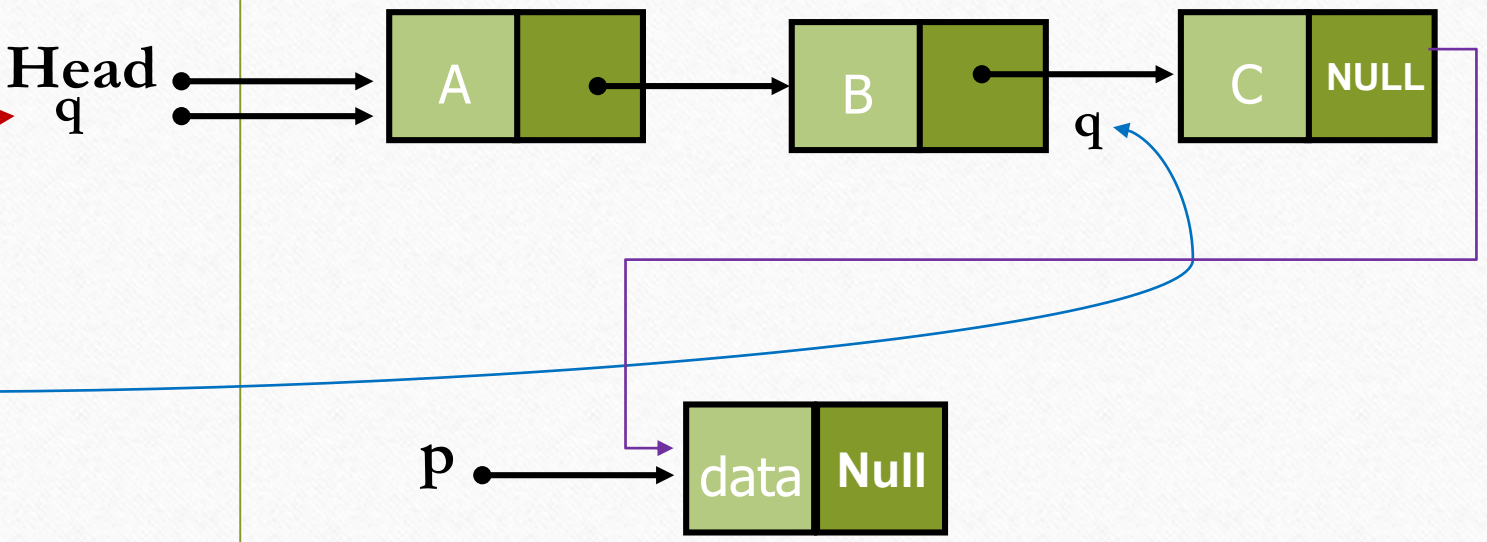
AddToTail Function

- To insert a node at the end of the list:

1. An empty node is created.
2. The node's info member is initialized to an integer.
3. Because the node is being included at the end of the list, the next member is set to NULL.
4. The node is now included in the list by making the next member of the last node of the list a pointer to the newly created node.

AddToTail Function

```
void list::addToTail(int data)
{
    Node* p= new node;
    p->info=data;
    node* q=head;
    while(q->next!=NULL)
        q=q->next;
    q->next=p;
    p->next=NULL;
}
```

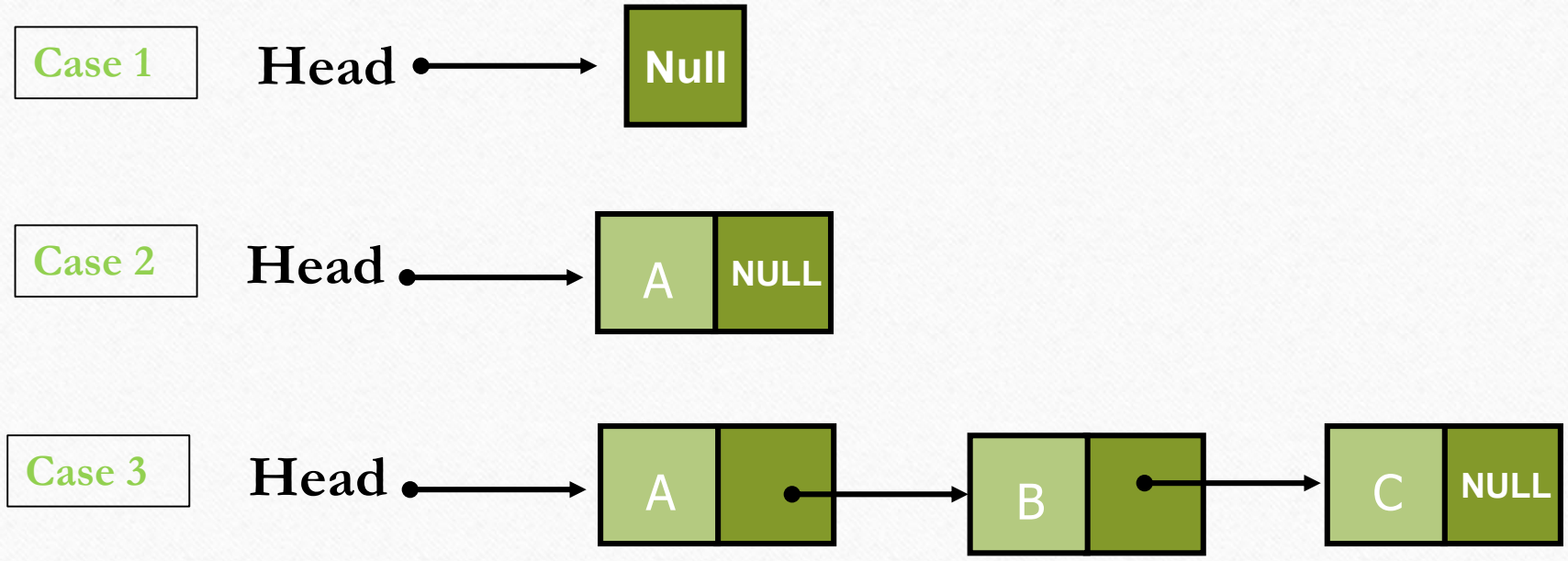


Deleting a node from a list

When deleting a node from a list, we need to consider all the following cases:

- Case 1: deleting a node from an empty list →the function is immediately exited.
- Case 2: deleting the only node from a one-node linked list →the head is set to null.
- Case 3: deleting the first node of the list with at least two nodes →requires updating head.
- Case 4: deleting a node with a number that is not in the list →do nothing.

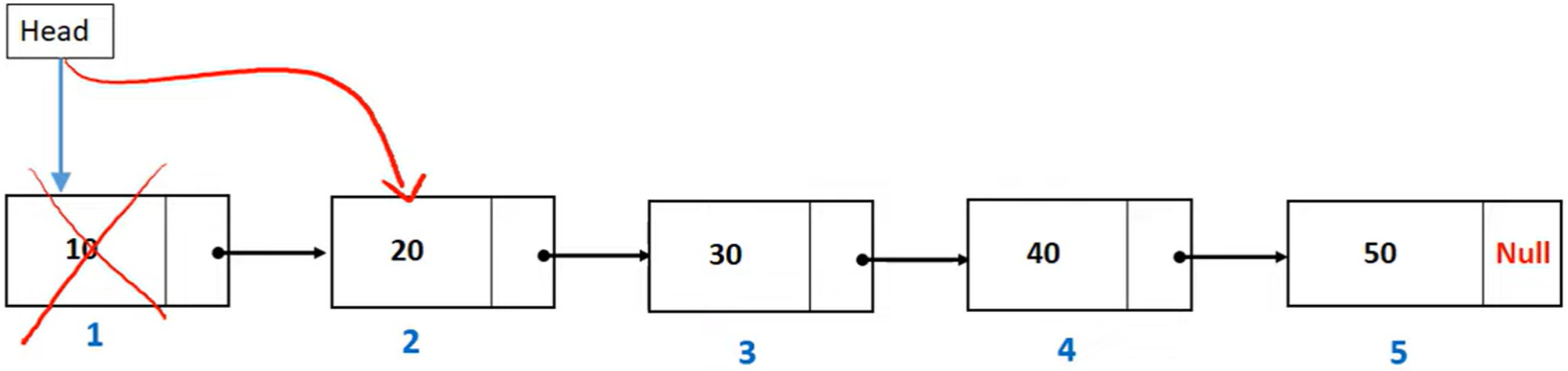
Deleting a node from a list



deleteFromHead Function

- In this operation, the information from the first node is temporarily stored in a local variable, the first node is deleted, and then head is reset so that what was the second node becomes the first node.
- The information of the deleted node is returned.

Case 3: deleting the first node of the list

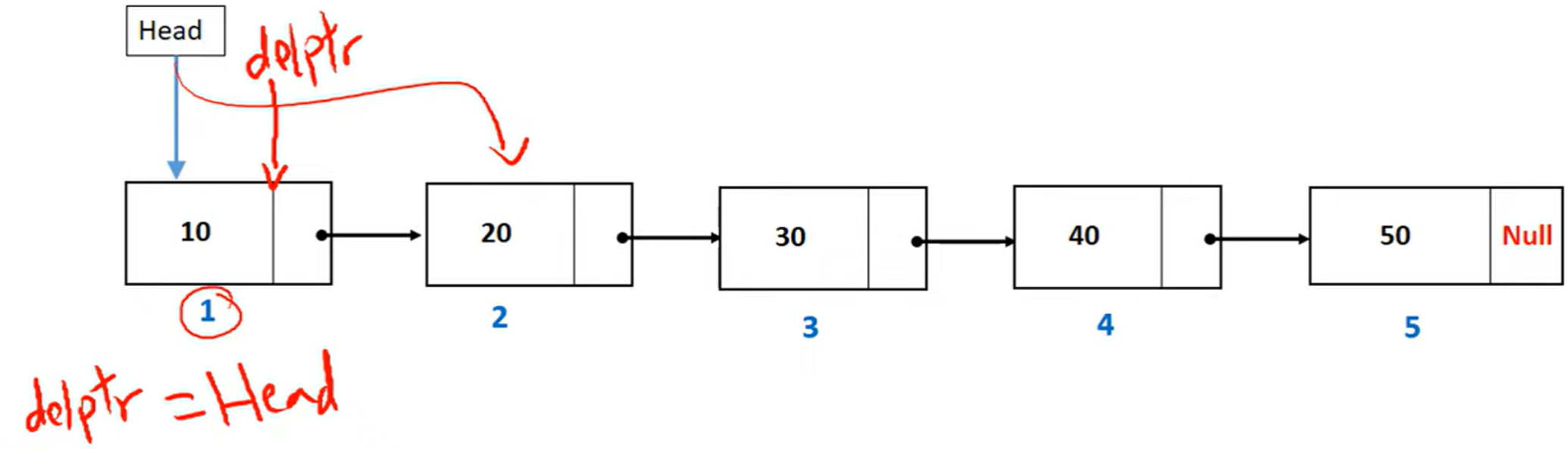


caution!

this will remove the node from list but it still in memory

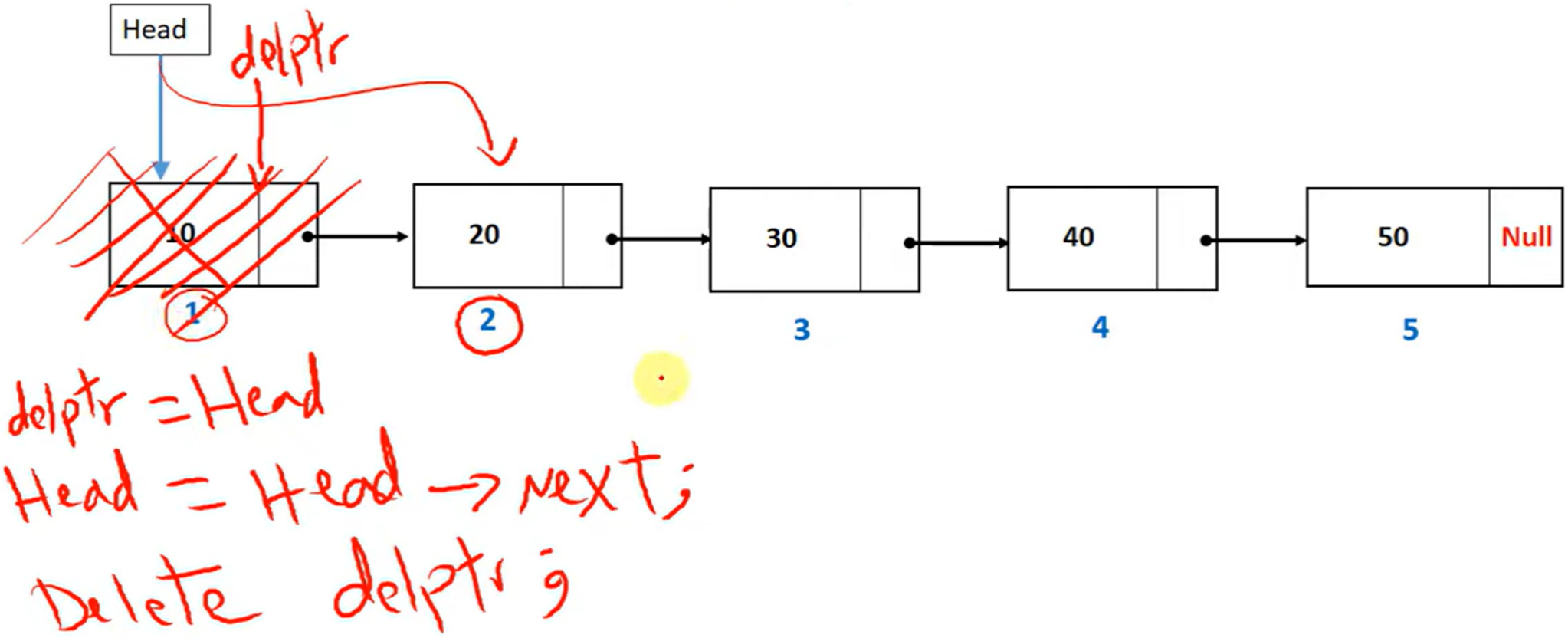
Case 3: deleting the first node of the list

solution



Case 3: deleting the first node of the list

solution



deleteFromHead Function

1.deleting from an empty list

```
int list::deleteFromHead()
{
        int x;
        if(head==NULL)
        {
                cout<<"list Empty";
                exit(0);
        }
```

Case 1

2.deleting the only node

```
else if(head->next==NULL)
{
        x=head->info;
        delete head;
        return x;
}
```

Case 2

3.deleting the first node

```
else
{
        Node* p=head;
        head=head->next;
        x=p->info;
        delete p;
        return x;
}
}
```

Case 3

deleteFromTail Function

- In this operation, the information from the last node is temporarily stored in a local variable, the last node is deleted and then the next of the new last node is set to NULL.
- The information of the deleted node is returned.

deleteFromTail Function

```
int list::deleteFromTail()
{
        int x;
        if(head==NULL)
        {
                printf("list Empty");
                exit(0);
        }
       else if(head->next==NULL)
        {
                x=head->info;
                delete head;
                return x;
        }
```

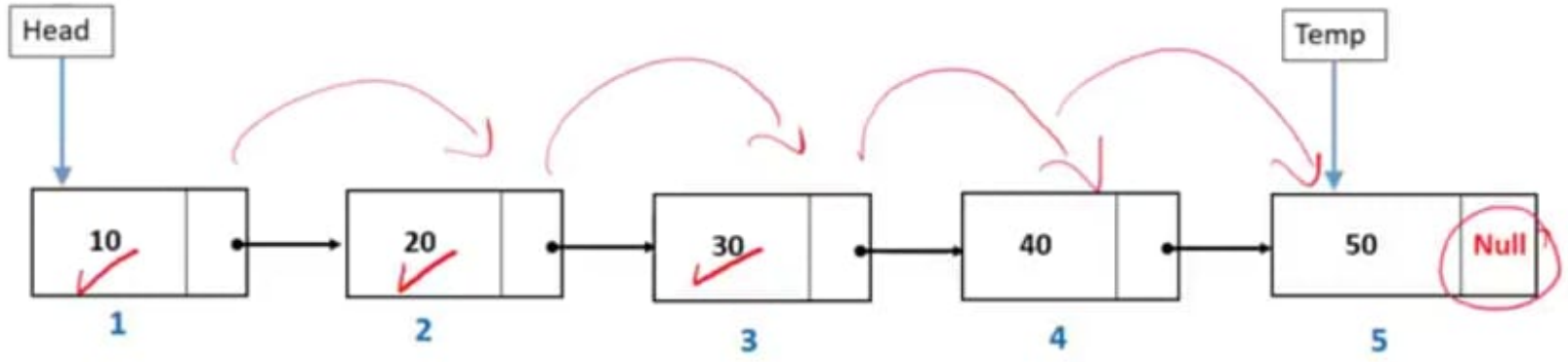
Case 1

Case 2

```
else
{
        node* p=head,*q;
        while(p->next!=NULL)
        {
                q=p;
                p=p->next;
        }
        x=p->info;
        delete p;
        q->next=0;
        return x;
}
}
```

Case 3

print list Items Example



Node* temp = Head;

Temp = Temp->next

Temp = Null

Temp	Printed Output
1	10
2	20
3	30
4	40
5	50

printList Function

- This function is used to print the information of all nodes in the list.
- If the list is empty, it prints List Empty.

```
void list::printList()
{
        if(head==NULL)
                cout<<“List Empty”;
        else
        {
                Node *p=head;
                while(p!=0){
                        printf("%d\t",p->info);
                        p=p->next;}
                printf("\n");
        }
}
```

countList Function

- This function is used to print the number of nodes in the list.

```
int list::countList()
{
        int c=0;
        Node* p=head;
        while(p!=NULL)
        {
                c++;
                p=p->next;
        }
        return c;
}
```

Finding a node

- Search for a node with the value equal to x in the list.
- If such a node is found, return its position. Otherwise, return 0.

```
int list::findNode(int x) {
Node* p   =  head;
int c  =  1;
while (p!=NULL && p->info!= x) {
   p  =  p->next;
   c++;
}
if (p->info==x) return c;
return 0;
}
```

Destructor ~List( )

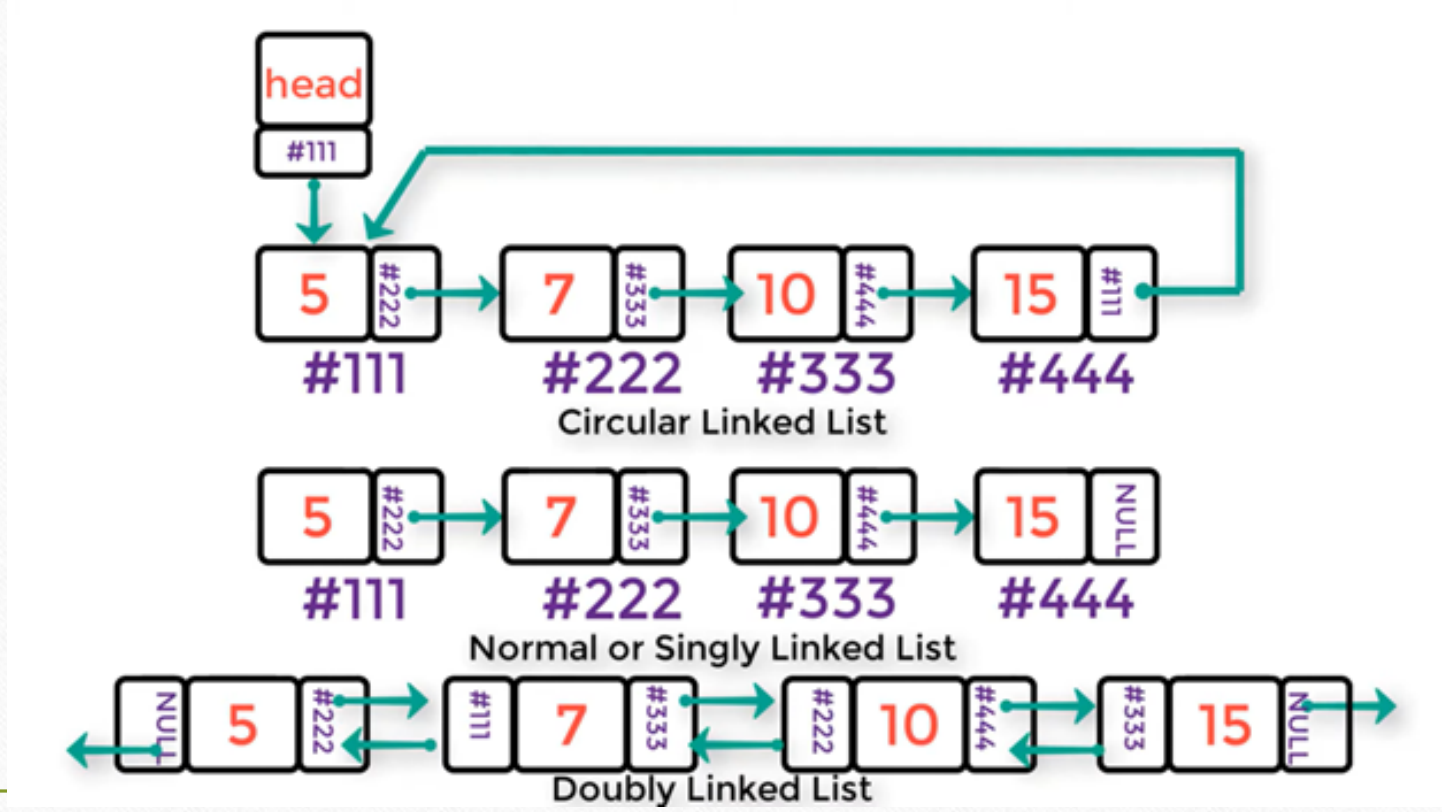
- Use the destructor to release all the memory used by the list.
- Step through the list and delete each node one by one.

```
list::~list(void) {
    node* p = head, *q = NULL;
    while (p != NULL)
    {
        q   = p->next;
        delete p;
        p   =   q;
    }
}
```

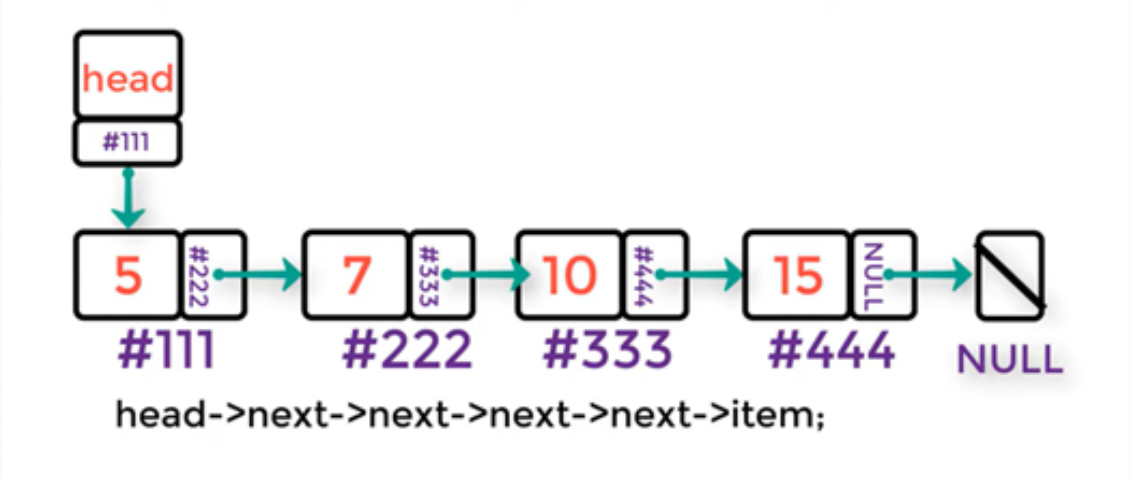
Using the List

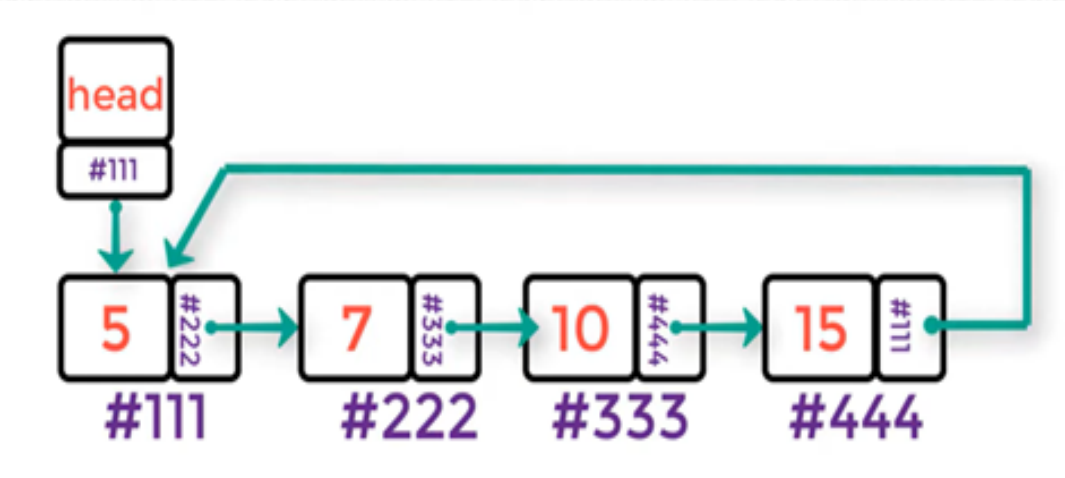
```
int main()
{
	list l;
	l.addToHead(5);
	l.addToTail(2);
	l.printList();
	printf("No. list nodes: %d\n",l.countList());
	l.deleteFromHead();
	l.deleteFromTail();
	l.printList();
	return 0;
}
```

Variations of Linked Lists



Why Circular linked lists?



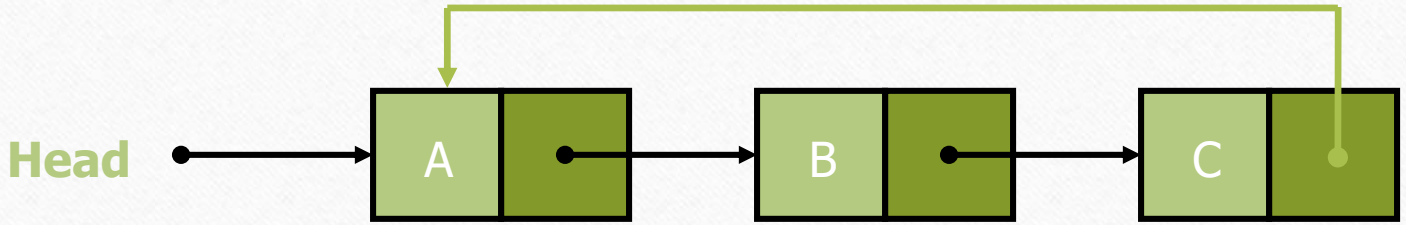


run time error!!

Circular linked lists

- **Circular Linked Lists**
  -In single linked list, every node points to its next node in the sequence and the last node points NULL.
  -In circular linked list, every node points to its next node in the sequence but the last node points to the first node in the list.

Circular linked lists



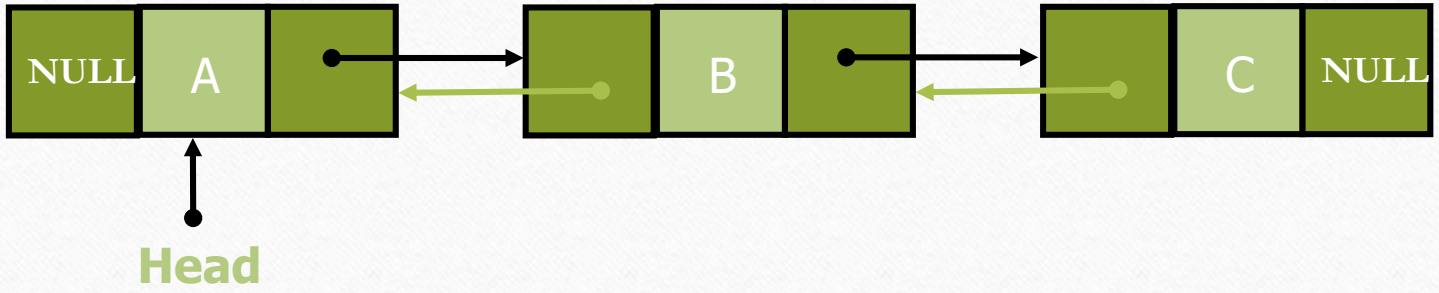
- The last node points to the first node of the list
- How do we know when we have finished traversing the list?
  - check if the pointer of the current node is equal to the head.

Doubly linked lists

- A singly linked list has the disadvantage that we can only traverse it in one direction.
- Many applications require searching backwards and forwards through sections of a list. A useful refinement that can be made to the singly linked list is to create a doubly linked list.
- The distinction made between the two list types is that while singly linked list have pointers going in one direction, doubly linked list have pointer both to the next and to the previous element.

Doubly linked lists

In this list each node contains three fields.



- Each node points to not only the successor but also 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 for traversing or searching of the list in both directions (backwards or forward)

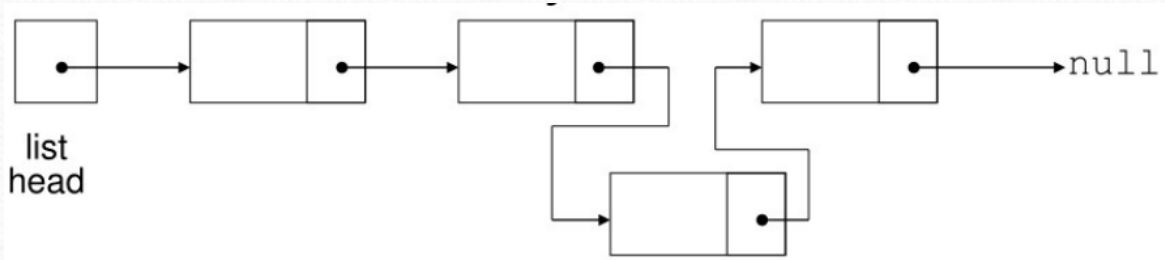
Linked Lists versus Arrays

- Arrays are great for storing things of similar types, but arrays have the following drawbacks

  - The capacity must be **fixed while creating (i.e, at compilation time)**
  - Programming languages have **limitations** on the maximum size of the array
  - To define an array, needed memory must be contiguous **not scattered**
  - In ordered arrays, insertions/deletions can be **time consuming and expensive** if many elements must be shifted because we need to copy to temporary variables to make room for new elements or close the gap caused by deleted elements.
- **Linked Lists overcome these shortcomings**

Advantages of Linked Lists

- Linked lists are more complex to code and manage than arrays, but they have some distinct advantages
  - **Dynamic size**: 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.
  - **Easy and fast insertions and deletions**
    - With a linked list, no need to move other nodes. Only need to reset some pointers.



Drawbacks of Linked Lists

- Random access is not allowed
  - we have to access elements sequentially starting from the first node.
- Extra memory space is requied for the pointer of each element of the list.

End of Lecture