BITS Pilani
Pilani Campus

Module 12 – part 1 – Linked Lists

Department of Computer Science & Information Systems

Module Overview

- Motivation
- Linked Lists Implementation

innovate achieve lead

BITS Pilani
Pilani Campus

Motivation

Random Access List vs Sequential Access List

Random Access List:

- Given a list of elements, you should be able to access any element of the list:
  - *quickly*
  - *easily*
  - *without traversing any other element of the list*
- Example: Using **arrays** to represent the list.

```
int arr[100] = {5,8,34,98,13,25,73,88,28,30};
```

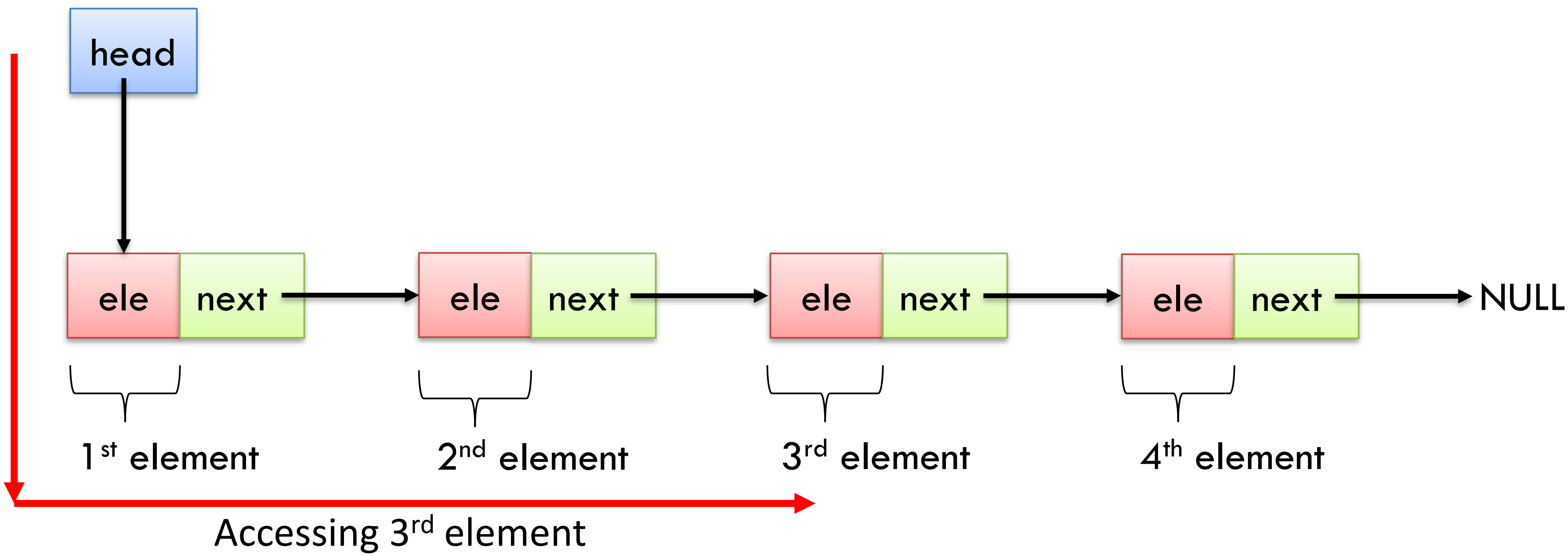
5	8	34	98	13	25	73	88	28	30

- You can access 3rd element of the array by `arr[2]`
  - This is quick, easy and doesn't need one to traverse the entire list to read the 3rd element

Random Access List vs Sequential Access List

Sequential Access List

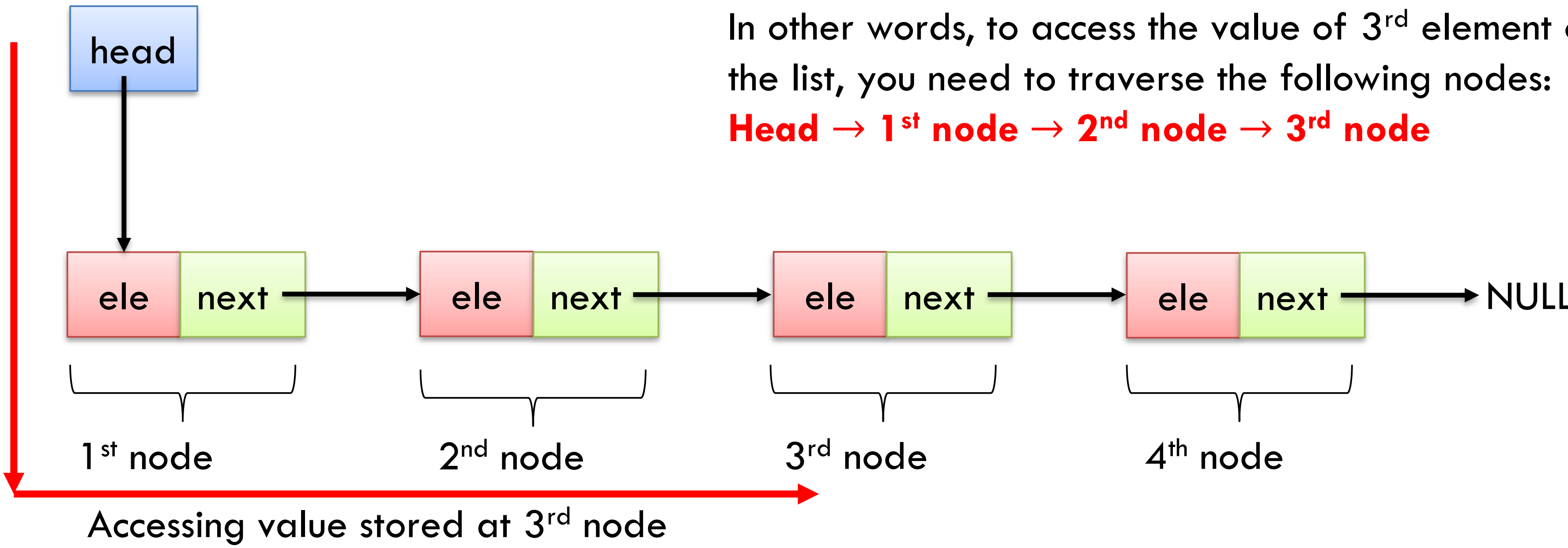
- There is another way of representing lists where you should traverse the list to reach any element of the list.



- To access 3rd element of the list you need to traverse 1st, 2nd elements

Linked Lists

In other words, to access the value of 3rd element of the list, you need to traverse the following nodes:
Head → 1st node → 2nd node → 3rd node

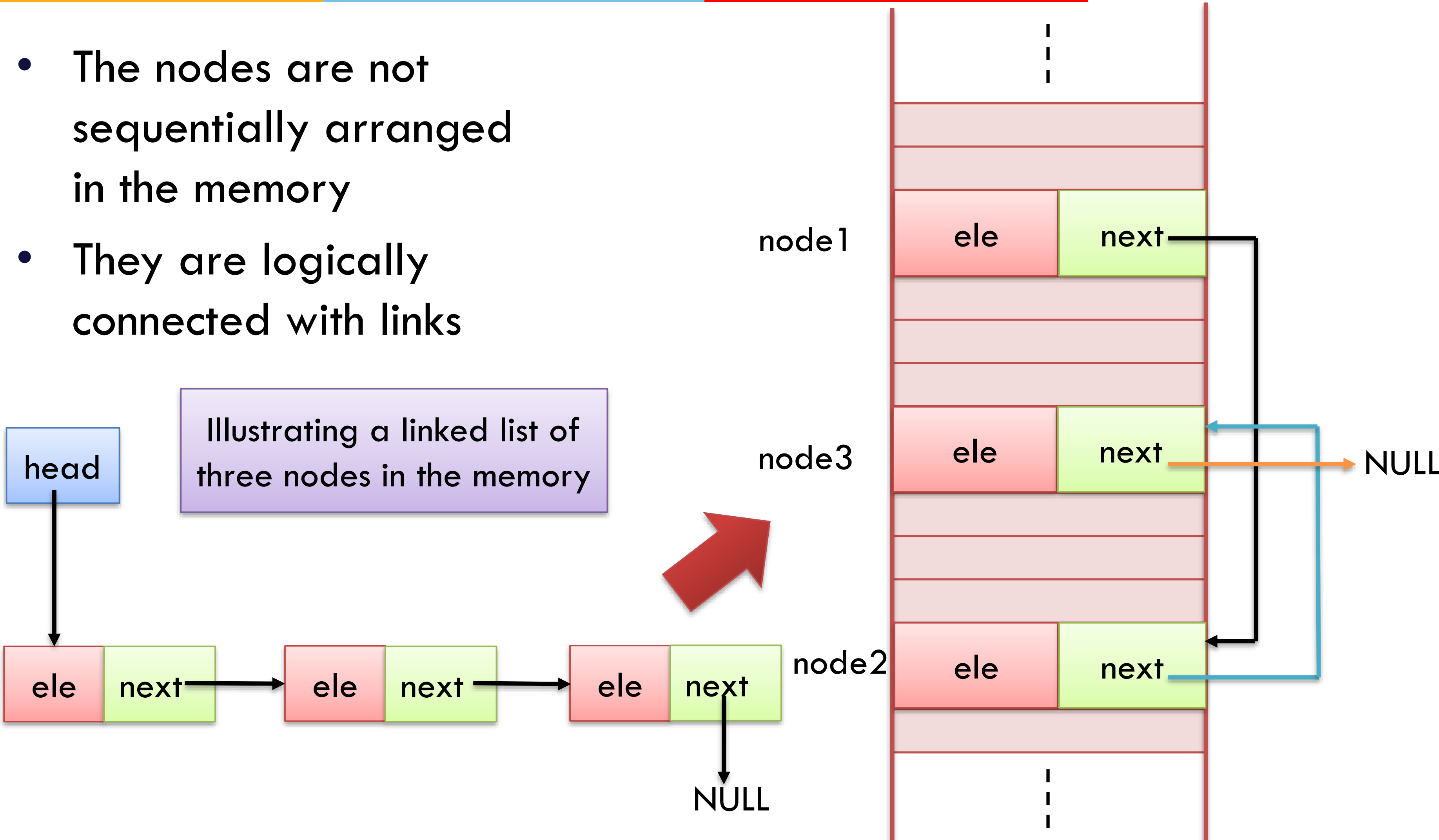


- Lists are now organized as a sequence of nodes, each containing a
  - *value of the element stored at that node:* ***ele***
  - *address of the next node:* ***next***
- *The* **head node** *contains the* ***address of the first node***
- *The* ***last node points*** *to* **NULL**, *meaning end of the list*

All the nodes together represent a list or a sequence

Linked List in Memory

- The nodes are not sequentially arranged in the memory
- They are logically connected with links



Sequential Access Lists - Uses

Where are Sequential Access Lists useful?

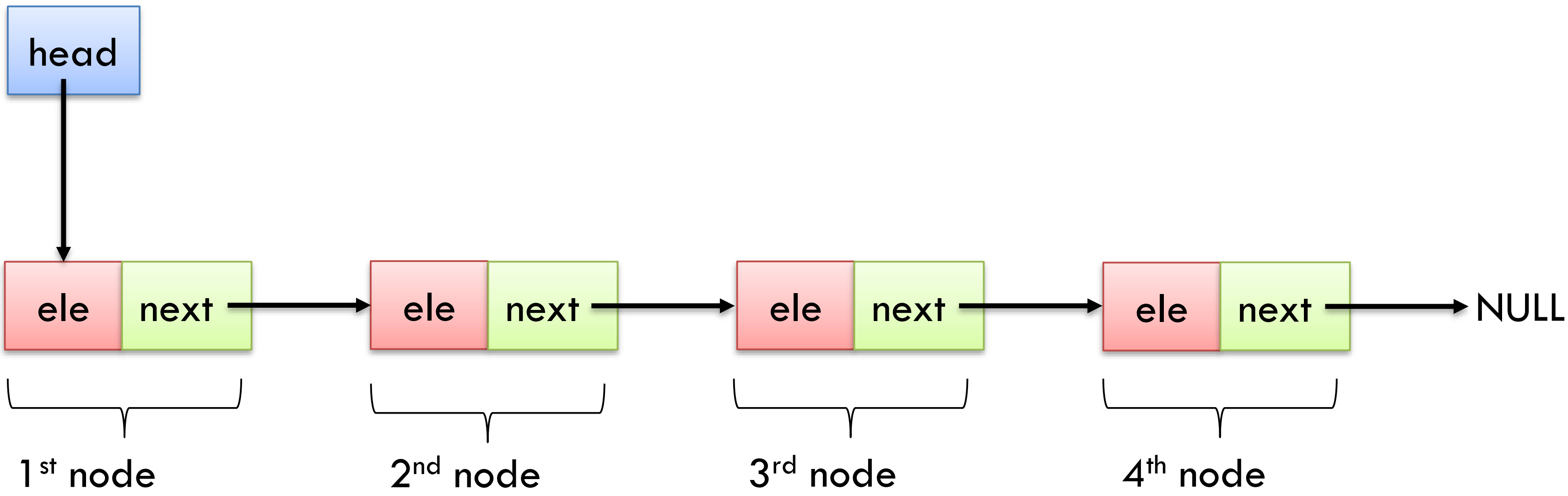
- Create dynamic lists on run-time
  - *you can keep on adding nodes to the list, without bothering about resizing the list, like in arrays when the number of elements exceed their size*
- Efficient insertion and deletion
  - *Without any shift operations*
- Used to implement
  - *Stacks*
  - *Queues*
  - *Other user-defined data types*

innovate achieve lead

BITS Pilani
Pilani Campus

Linked lists Implementation

Linked Lists



To create linked lists we need two kinds of structures:

- *One for storing the* **head**
- *The other to represent each* **node** *in the list*

Let us see how each of these can be defined…

Self referential structures

Before we see the structure definition of linked lists, let us see what self-referential structures are:

"Self-referential structures contain a pointer member that points to a structure of the same structure type"

Wrong Declaration

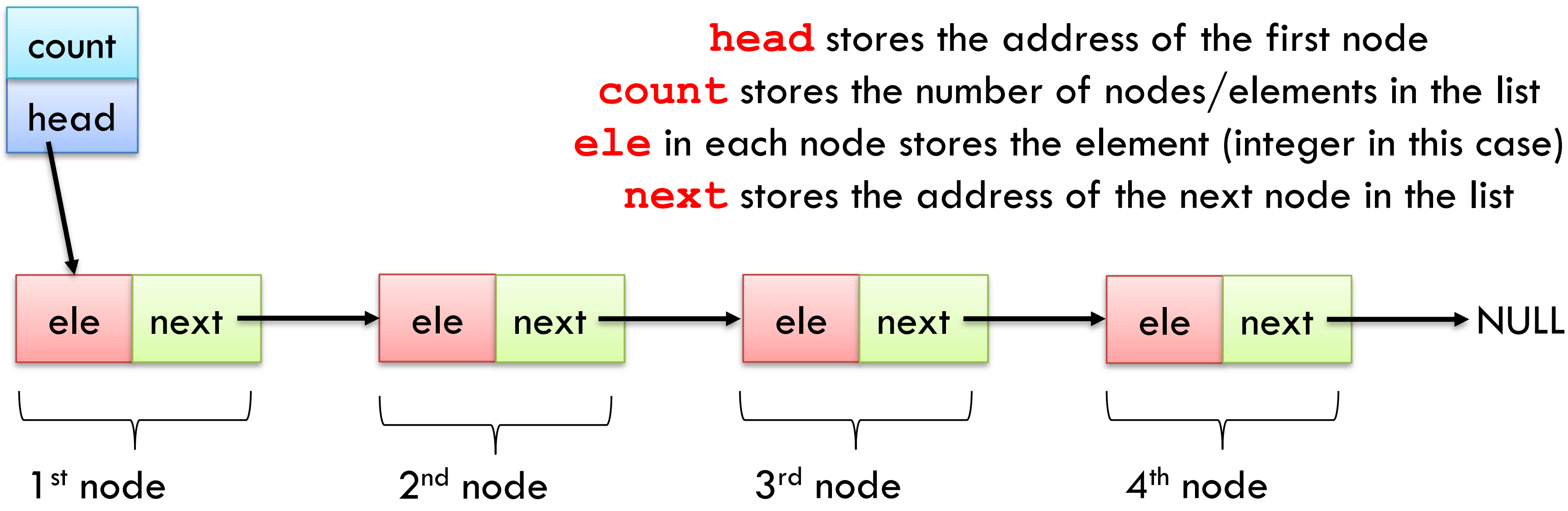
```
struct self_ref
{
  int data;
  struct self_ref b;
};
```

Correct Declaration

```
struct self_ref
{
  int data;
  struct self_ref *b;
};
```

Self-referential structures essentially store a pointer variable to of its own type to reference to another structure variable of its kind.

Linked Lists



head stores the address of the first node
count stores the number of nodes/elements in the list
ele in each node stores the element (integer in this case)
next stores the address of the next node in the list

Consider that our linked list stores integer elements.

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

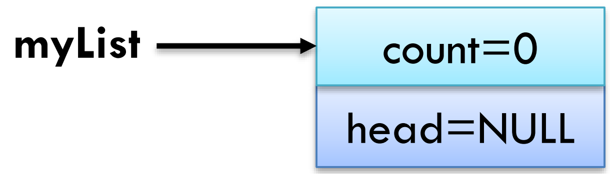
```
struct linked_list{
  int count;
  struct node * head;
};
```

Creating linked list using malloc() (on heap segment)

```c
typedef struct node * NODE;
struct node{
   int ele;
   NODE next;
};
```

```c
typedef struct linked_list * LIST;
struct linked_list{
   int count;
   NODE head;
};
```

```c
LIST createNewList(){
   LIST myList;
   myList = (LIST) malloc(sizeof(struct linked_list));
   // myList = (LIST) malloc(sizeof(*myList));
   myList->count=0;
   myList->head=NULL;
   return myList;
}
```

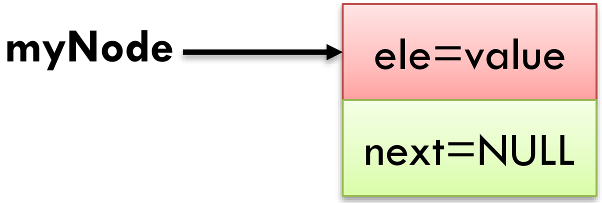


Creating new node

```
typedef struct node * NODE;
struct node{
   int ele;
   NODE next;
};
```

```
typedef struct linked_list * LIST;
struct linked_list{
   int count;
   NODE head;
};
```

```
NODE createNewNode(int value){
   NODE myNode;
   myNode = (NODE) malloc(sizeof(struct node));
   // myList = (NODE) malloc(sizeof(*myNode));
   myNode->ele=value;
   myNode->next=NULL;
   return myNode;
}
```

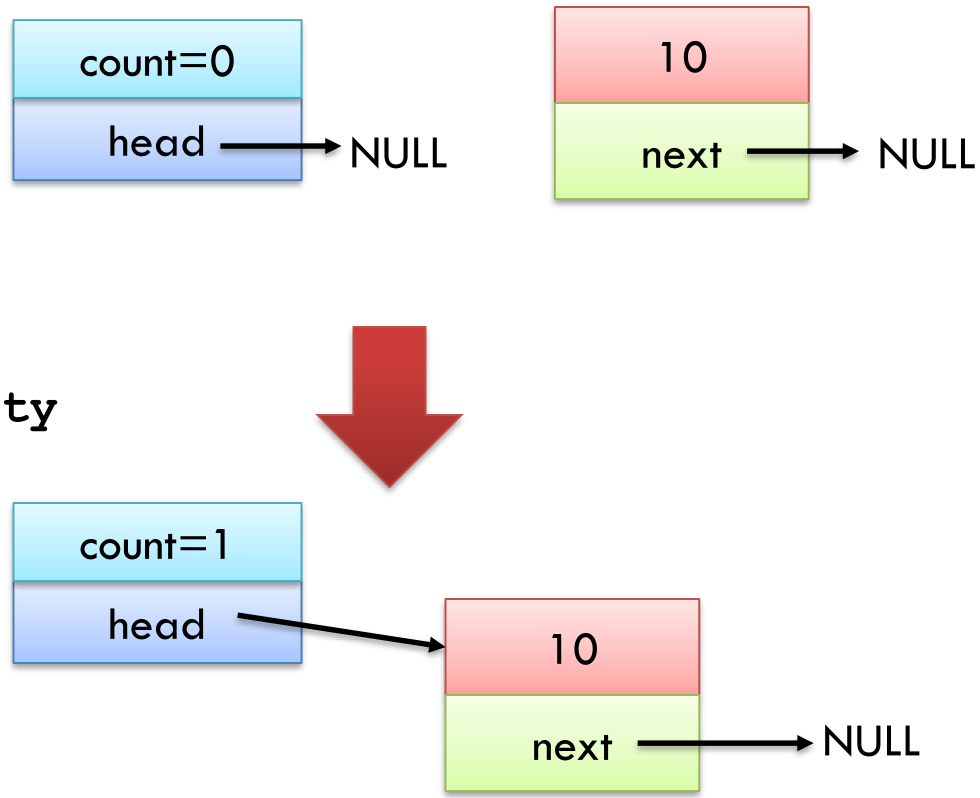


main()

```
int main(){
    LIST newList = createNewList();
    NODE n1 = createNewNode(10);
    NODE n2 = createNewNode(20);
    NODE n3 = createNewNode(30);

    insertNodeAtBeginning(n1,newList);
    insertNodeAtBeginning(n2,newList);
    insertNodeAtEnd(n3,newList);

    NODE n4 = createNewNode(40);
    insertAfter(10,n4,newList);

    removeFirstNode(newList);
    removeLastNode(newList);
    return 0;
}
```

Inserting a node at the beginning of the list

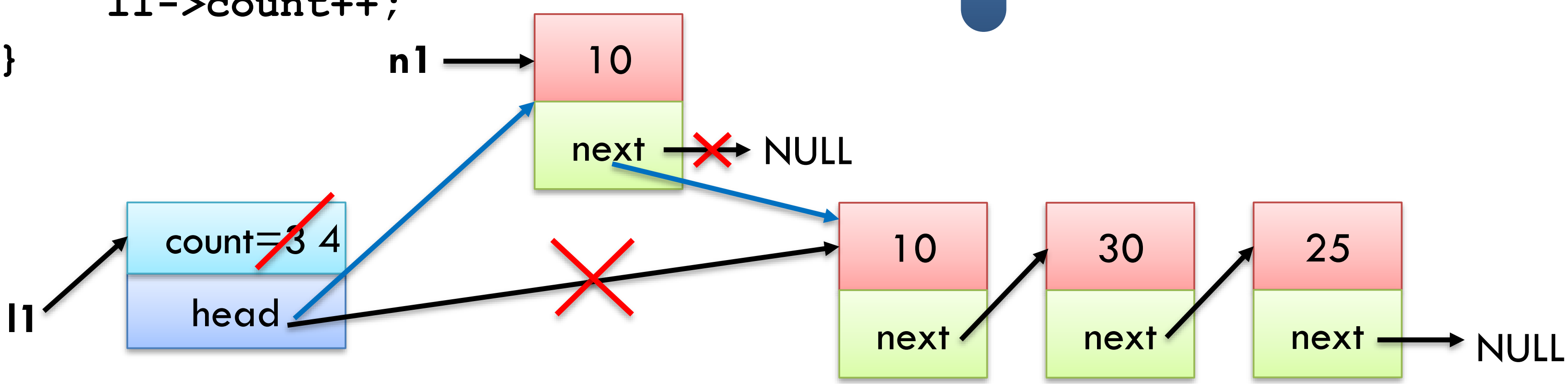
```
void insertNodeAtBeginning(NODE n1, LIST l1){
   // case when list is empty
   if(l1->count == 0) {
        l1->head = n1;
        n1->next = NULL;
        l1->count++;
   }
   // case when list is non empty
   else {
        ... ...
   }
}
```



Inserting a node at the beginning of the list (contd.)

```
void insertNodeAtBeginning(NODE n1, LIST l1){
   // case when list is empty
   if(l1->count == 0) {
        ... ...
   }
   // case when list is non empty
   else {
        n1->next = l1->head;
        l1->head = n1;
        l1->count++;
   }
}
```

Insertion is usually done at the beginning of the list. It is very fast. Doesn't require any traversal or shifting of elements



Inserting a node at the end of the list

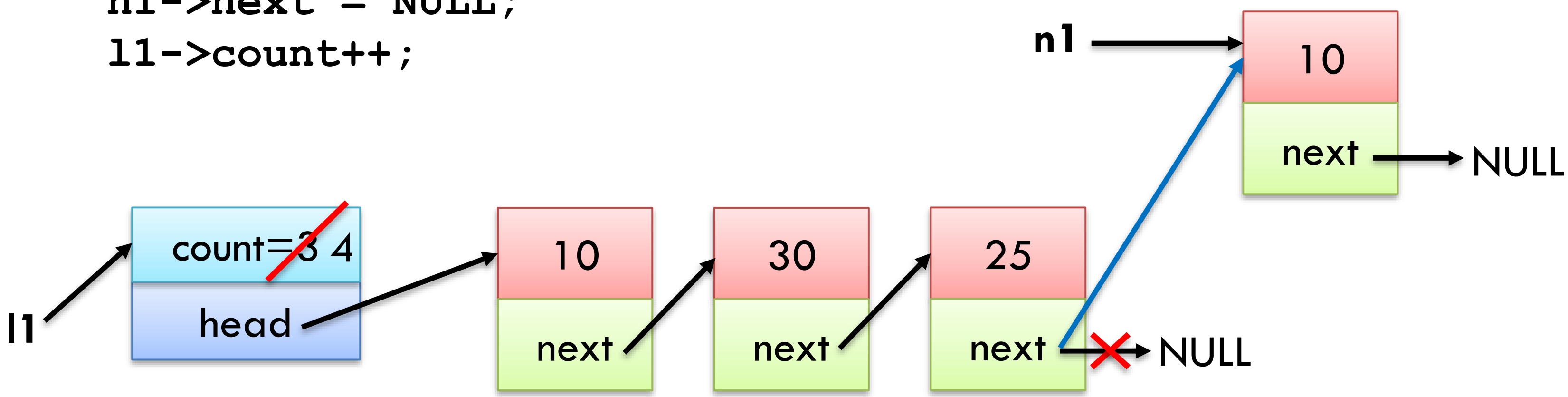
```
void insertNodeAtEnd(NODE n1, LIST l1){
     // case when list is empty
   if(l1->count == 0) {
        l1->head = n1;
        n1->next = NULL;
        l1->count++;
   }
   // case when list is non empty
   else {
          ... ...

   }
}
```

This case is same as insert at the beginning of an empty list.

Inserting a node at the end of the list

```
void insertNodeAtEnd(NODE n1, LIST l1){
        ...  ...
   // case when list is non empty
   else {
         NODE temp = l1->head;
         while(temp->next!=NULL)
         {
             temp = temp->next;
         }
         temp->next = n1;
         n1->next = NULL;
         l1->count++;

   }
}
```

- Traverse the list until the end.
- Insert new node at the end

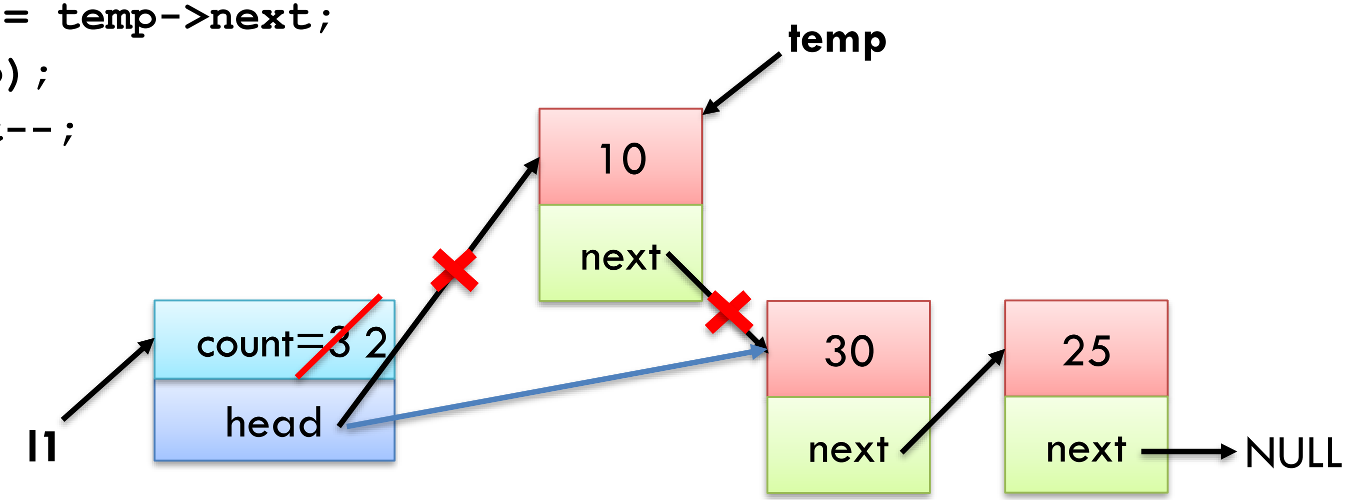


Inserting a node after a given node

```
void insertAfter(int searchEle, NODE n1, LIST l1){
    // case when list is empty
    … …
    // case when list is non-empty
     else  {
        NODE temp = l1->head;
        NODE prev = temp;
        while(temp!=NULL) {
           if (temp->ele == searchEle)
                break;
            prev = temp;
            temp = temp->next;
        }
        if(temp==NULL)  {
            printf("Element not found\n");
            return;
        }
        else{
            if(temp->next == NULL) {
                temp->next = n1;
                n1->next = NULL;
                l1->count++;
            }
            else {
                prev = temp;
                temp = temp->next;
                prev->next = n1;
                n1->next = temp;
                l1->count++;
            }
            return;
        }
    }
    return;
}
```

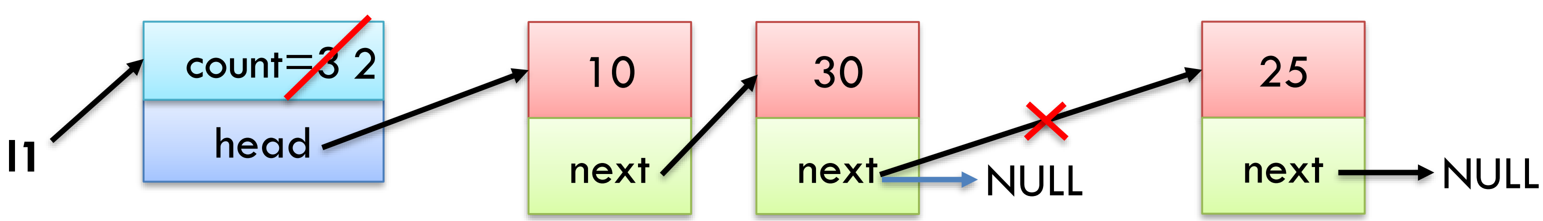
Removing a node from the beginning of the list

```
void removeFirstNode(LIST l1)
{
    if (l1->count == 0)
    {
      printf("List is empty. Nothing to remove\n");
    }
    else
    {
        NODE temp = l1->head;
        l1->head = temp->next;
        free(temp);
        l1->count--;
    }
    return;
}
```



Removing a node from the end of the list

```
void removeLastNode(LIST l1)
{
    if (l1->count == 0)
    {
       printf("List is empty\n");
    }
    else if(l1->count == 1)
    {
        l1->count--;
        free(l1->head);
        l1->head = NULL;
    }
    else
     {
       NODE temp = l1->head;
       NODE prev = temp;
       while((temp->next) != NULL)
       {
           prev=temp;
           temp = temp->next;
       }
       prev->next = NULL;
       l1->count--;
       free(temp);
    }
    return;}
```



Other functions

Exercise: Implement the following functions for a linked list:

- `search(int data, LIST mylist)`: returns the node that contains its ele=data
- `printList(LIST mylist)`: prints the elements present in the entire list in a sequential fashion
- `removeElement(int data, LIST mylist)`: removes the node that has its ele=data
- `isEmpty(LIST mylist)`: checks if the list is empty or not
- Modify the insert/delete functions to first check whether the list is empty using `isEmpty()` function.

In each of the above, you must have to decide which one is an appropriate datatype for the same.

innovate achieve lead

BITS Pilani

Pilani Campus

Thank you

Q & A