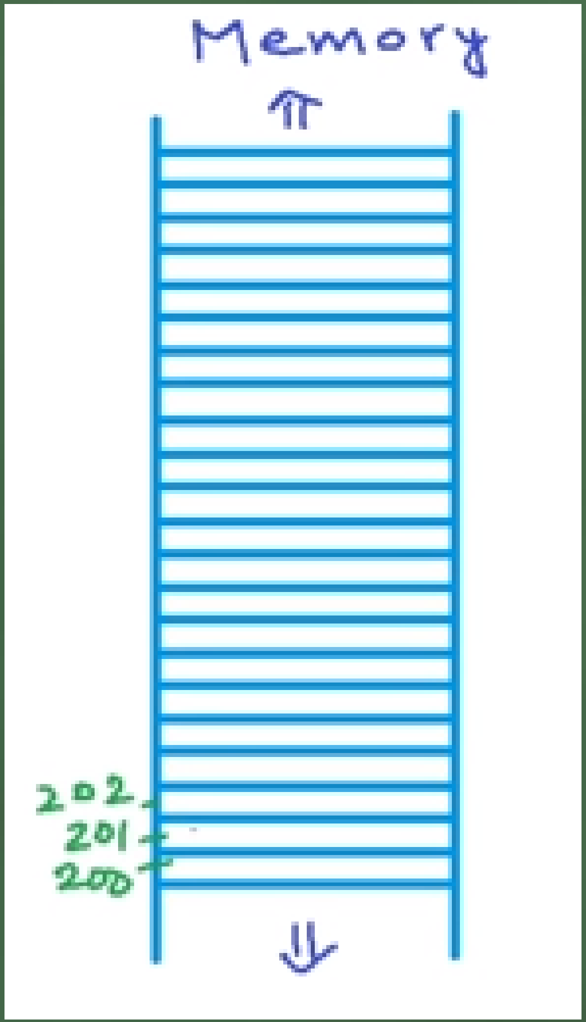
Data Structures and Algorithms

Shikha Mehrotra

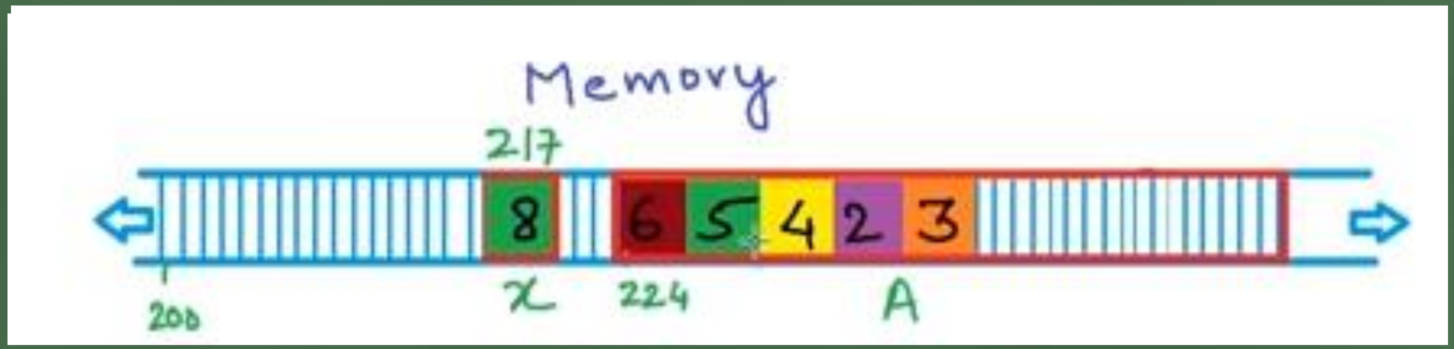
Linked List

Introduction of Linked list



Introduction of Linked list

Albert

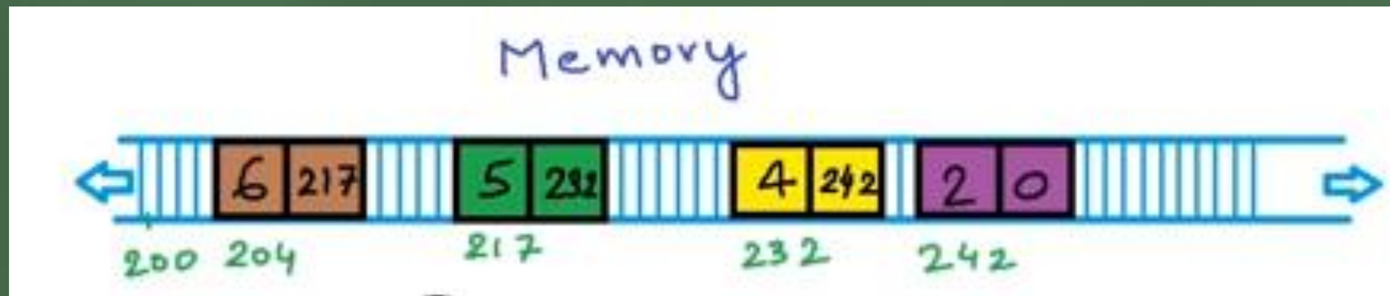


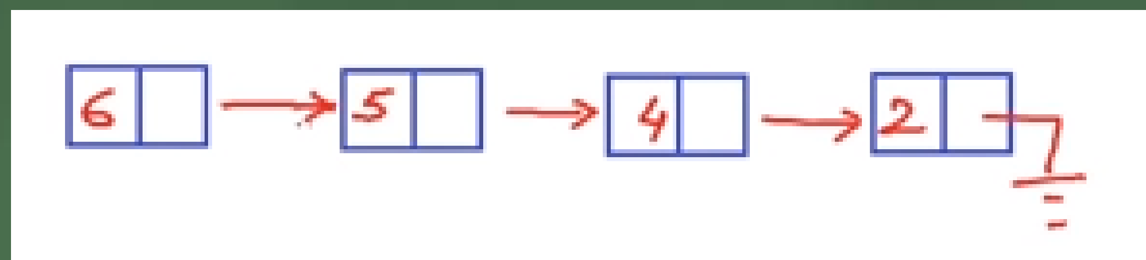

```
int x;
x = 8;
int A[4];
A[3] = 2;  //constant time
201 + 3x4 = 213
```

Memory Manager

Introduction of Linked list

Albert





6, 5, 4, 2

```
Struct node
{
    int data;
    node* next;
}
```

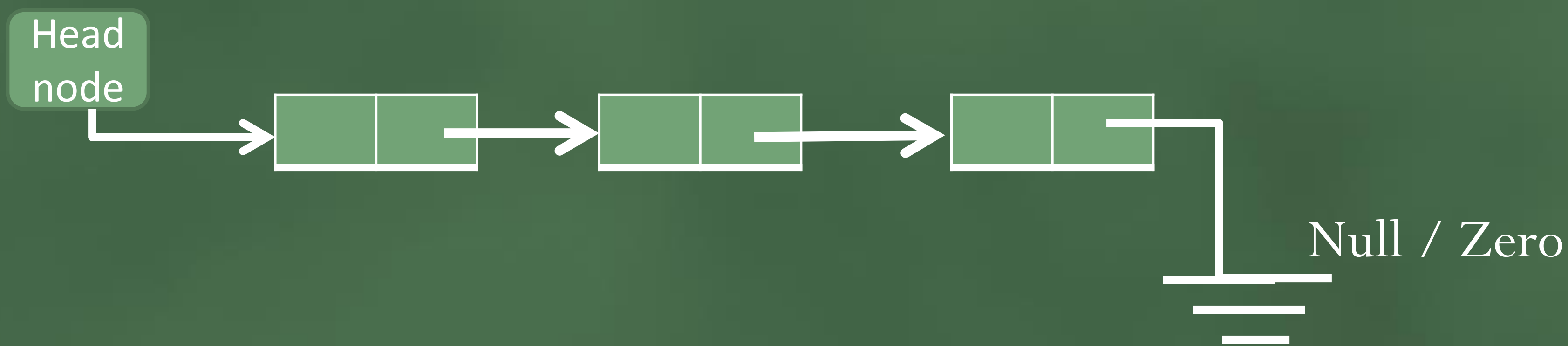
Memory Manager

Limitation of Arrays

- An array has a limited number of elements
  - routines inserting a new value have to check that there is room
- Can partially solve this problem by reallocating the array as needed (how much memory to add?)
  - adding one element at a time could be costly
  - one approach - double the current size of the array
- A better approach: use a *Linked List*

•Anatomy of a linked list

- A linked list consists of:
  - A sequence of nodes



Each node contains a value
and a link (pointer or reference) to some other node

The last node contains a null link

The list must have a header

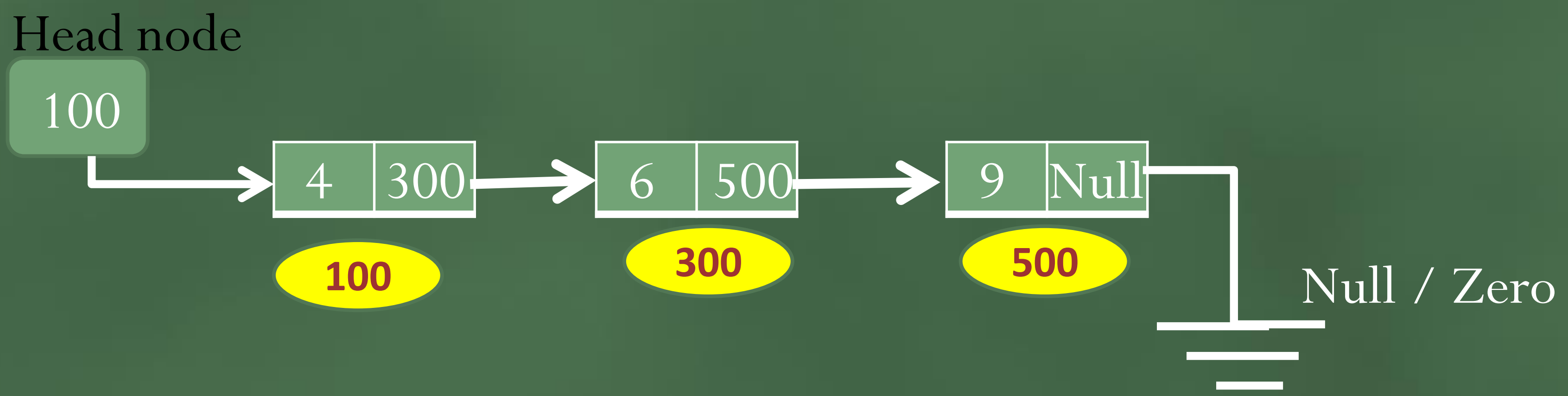
Terminology

- **Head (front, first node):**
  - The node without predecessor, the node that starts the lists.
- **Tail (end, last node):**
  - The node that has no successor, the last node in the list.
- **Current node:** The node being processed.
  - From the current node we can access the next node.
- **Empty list:** No nodes exist

•More terminology

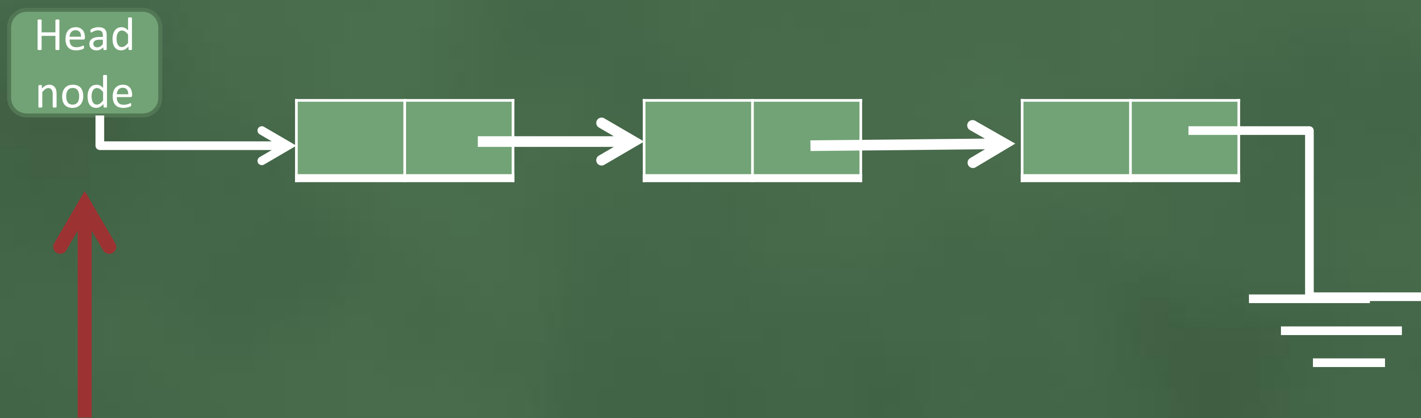
- ➢A node’s successor is the next node in the sequence
  - ✓The last node has no successor
- ➢A node’s predecessor is the previous node in the sequence
  - ✓The first node has no predecessor
- ➢A list’s length is the number of elements in it
  - ✓A list may be empty (contain no elements)

- **pointers recap**
- Int *p;
- P = (int *)malloc(sizeof(int));
- *p=10;

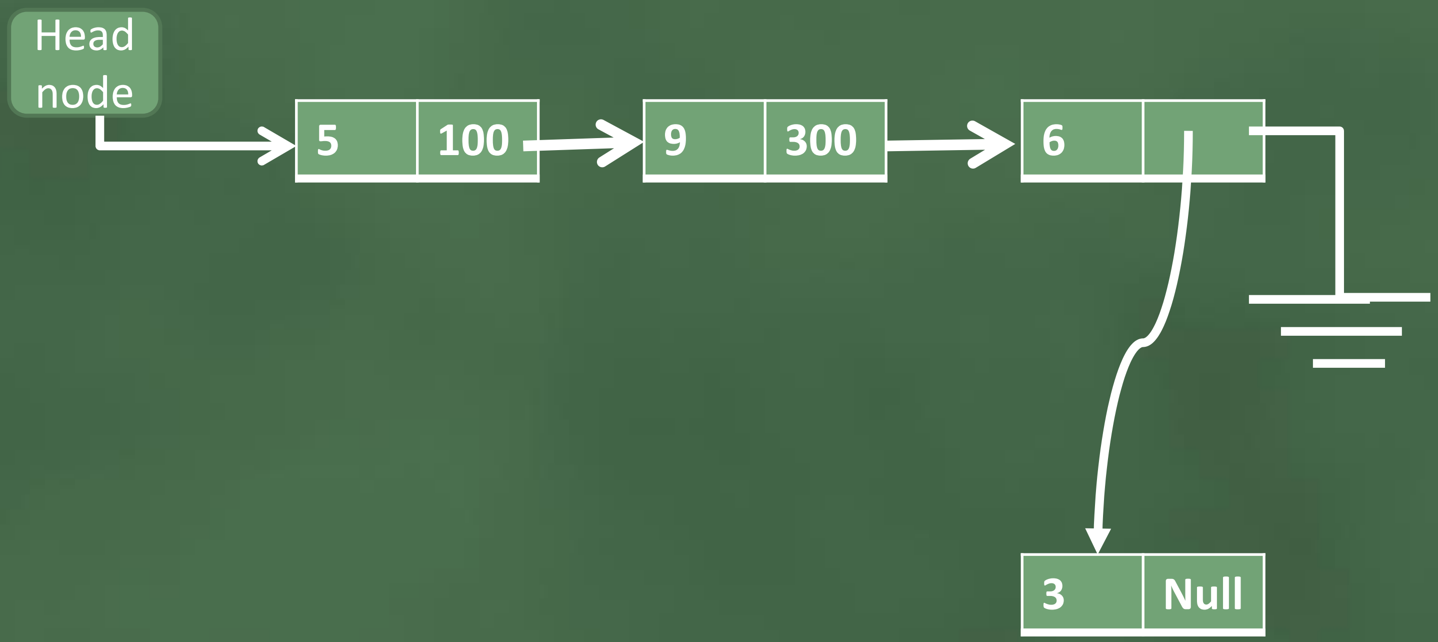


Head node
100
4
300
6
500
9
Null
100
300
500
Null / Zero

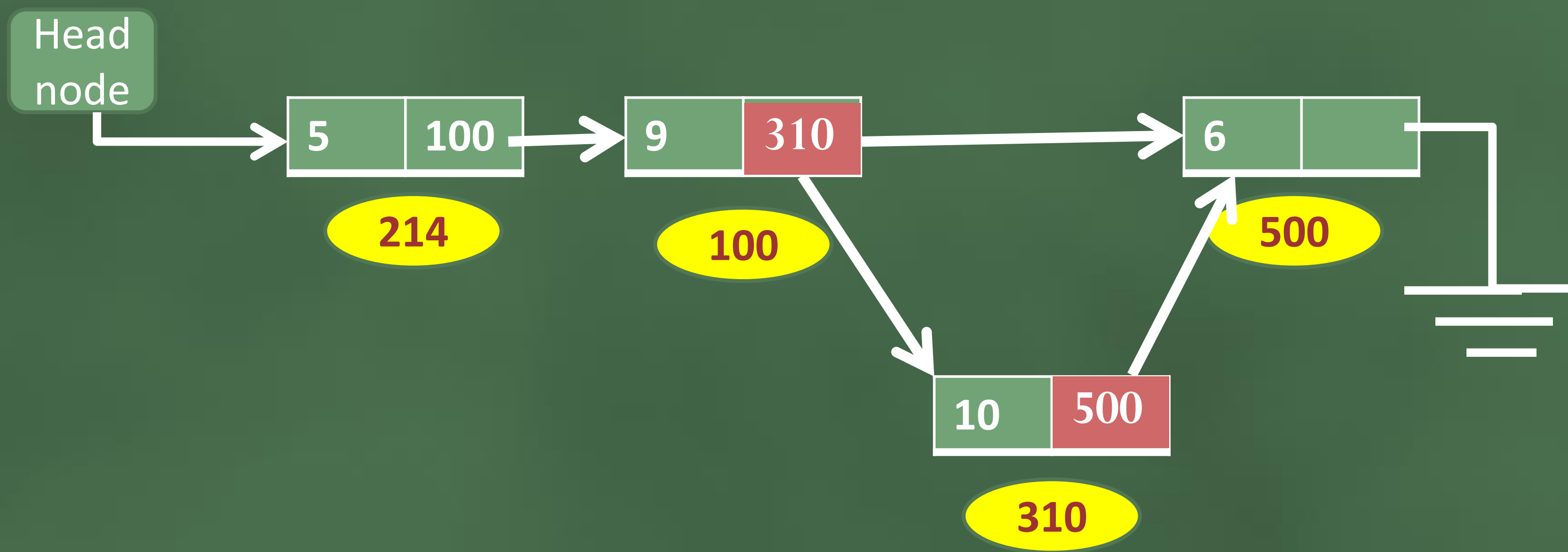
- **Traversal**



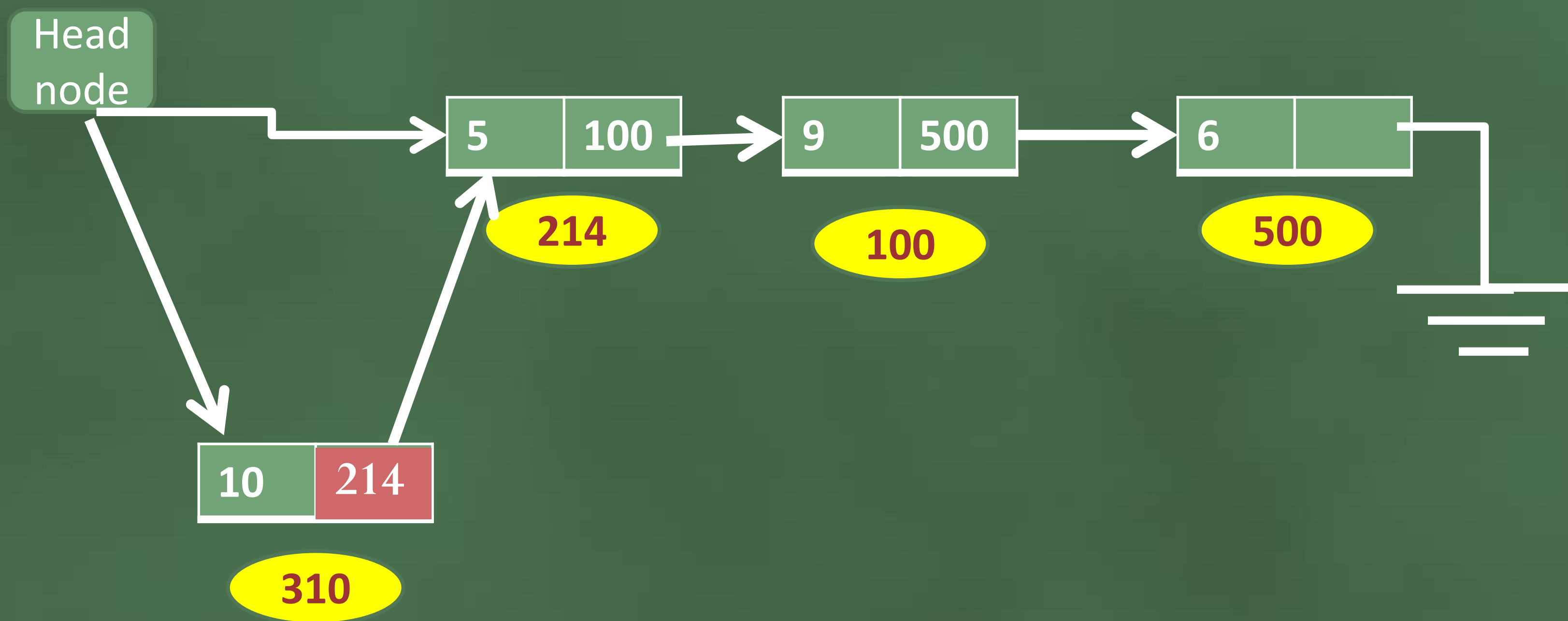
- **Insertion – at the end of list**



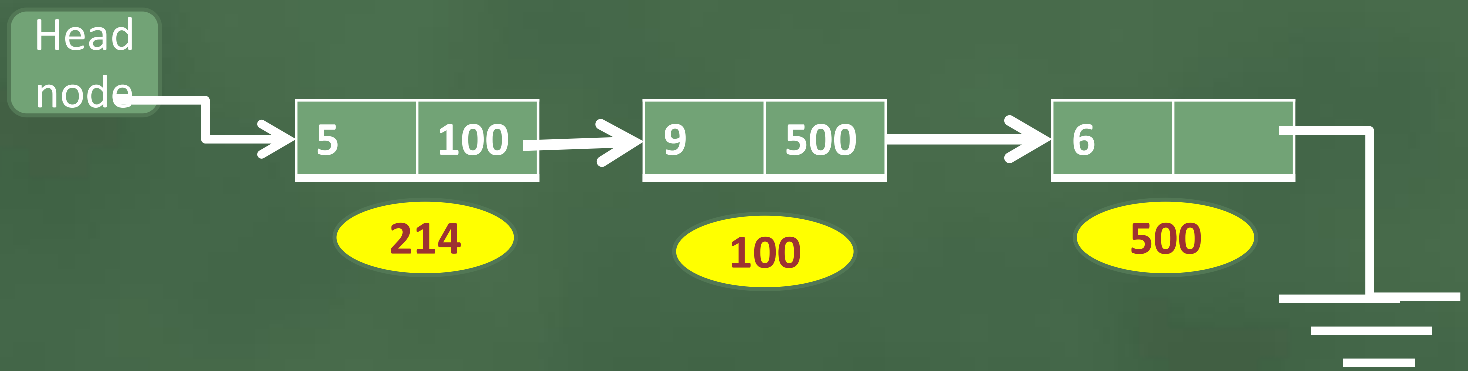
• Insertion – between two nodes



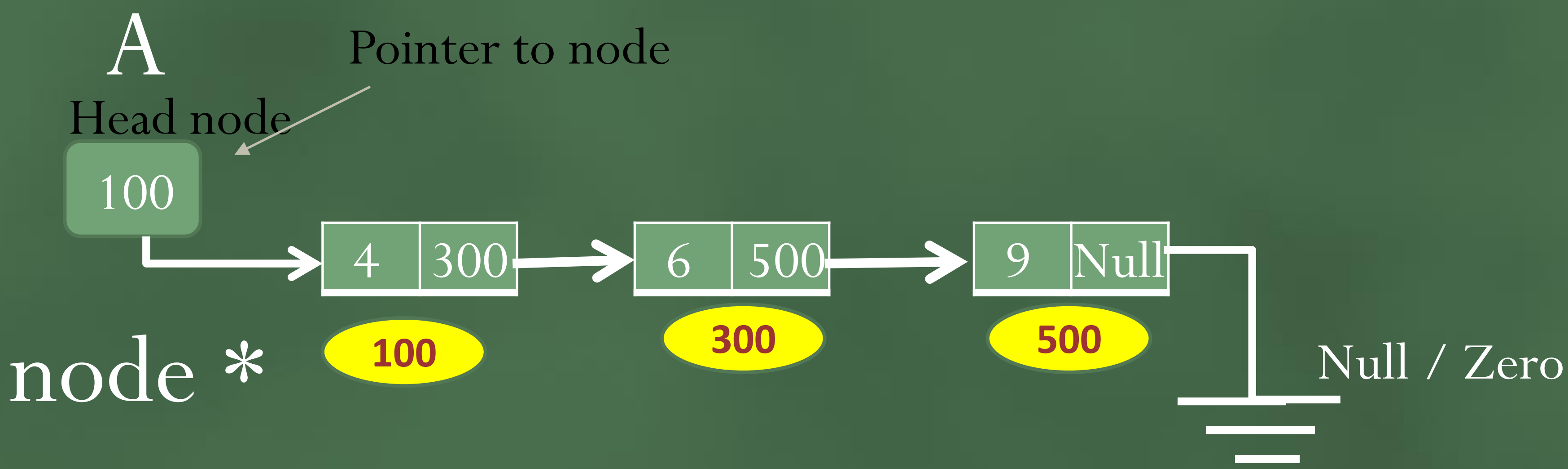
- **Insertion - beginning of linked list**



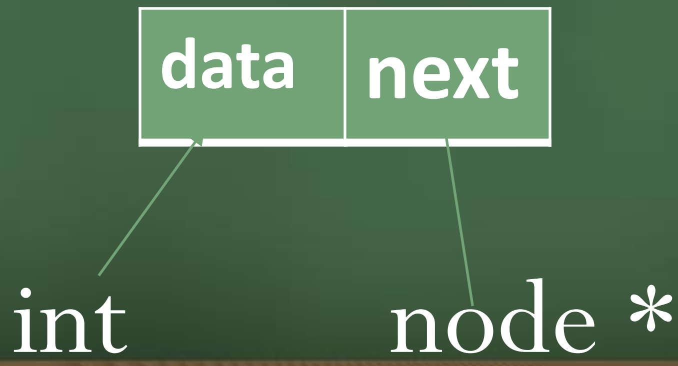
Deletion - last node



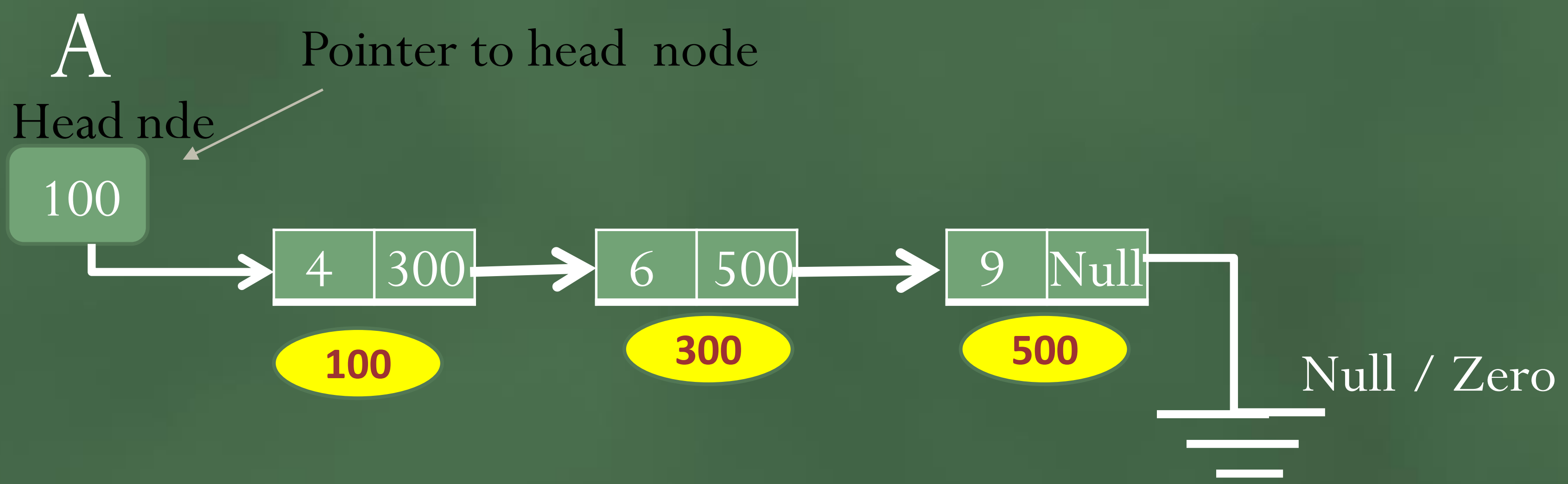
Implementation of linked list




```
Struct node
{
    int data;
    node* next;
}
```



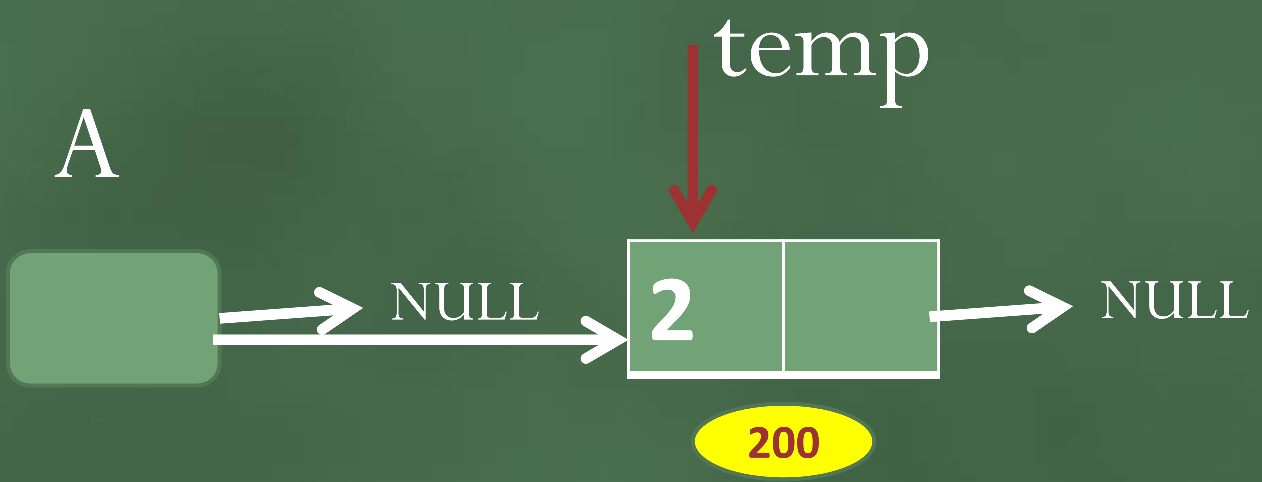
Implementation of linked list




```
Struct node
{
    int data;
    node* next;
}
```

```
Node* A
A = NULL
```

Implementation of linked list – inserting first node




```
Struct node
{
    int data;
    node* next;
}
```

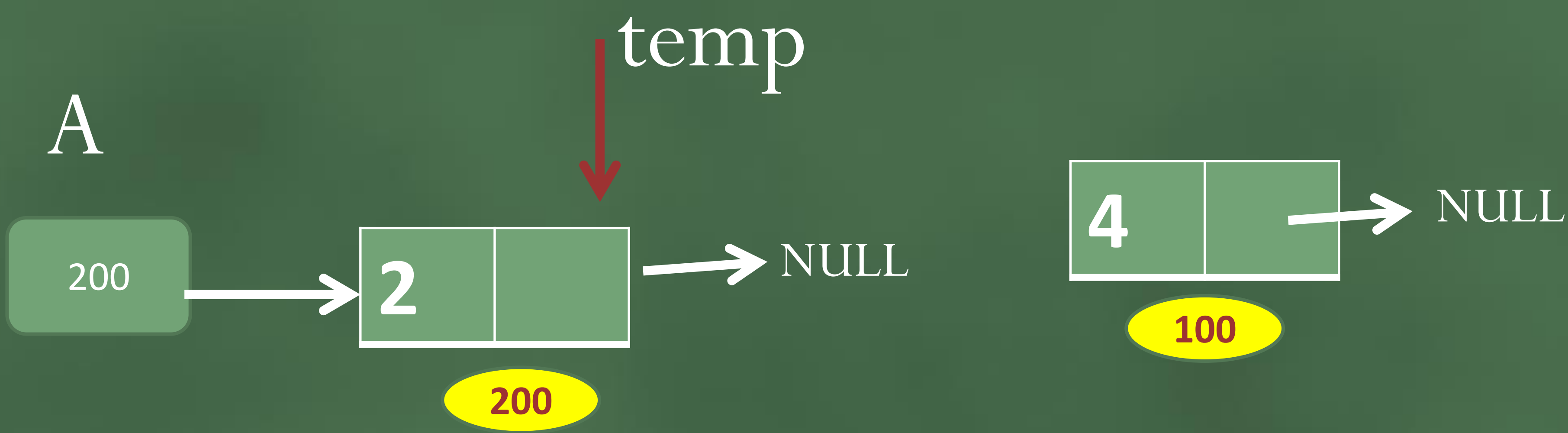
```
Node* A
A = NULL
```

```
Node* temp =(Node*)
malloc (sizeOf(Node)

temp -> data = 2
temp -> next = NULL

A=temp
```

Insertion at the end



Node* temp =(Node*)
malloc (sizeOf(Node)

temp -> data = 2

temp -> next = NULL

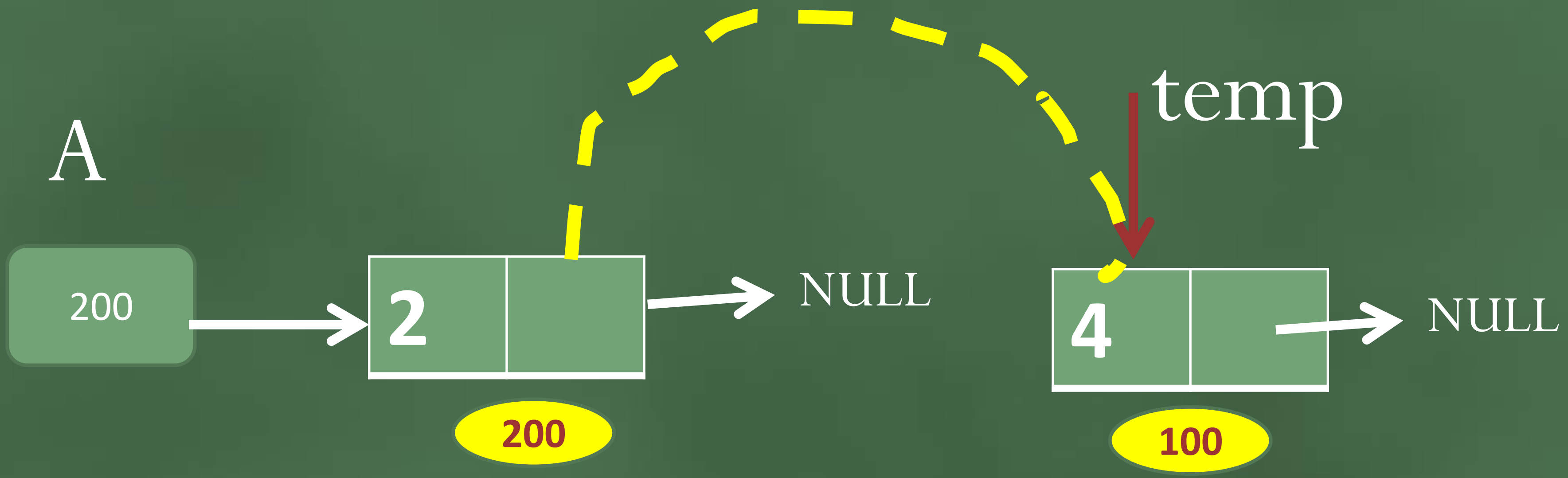
A=temp

Temp = (node*)malloc(sizeOf(Node)

temp -> data = 4

temp -> next = NULL

Traversal

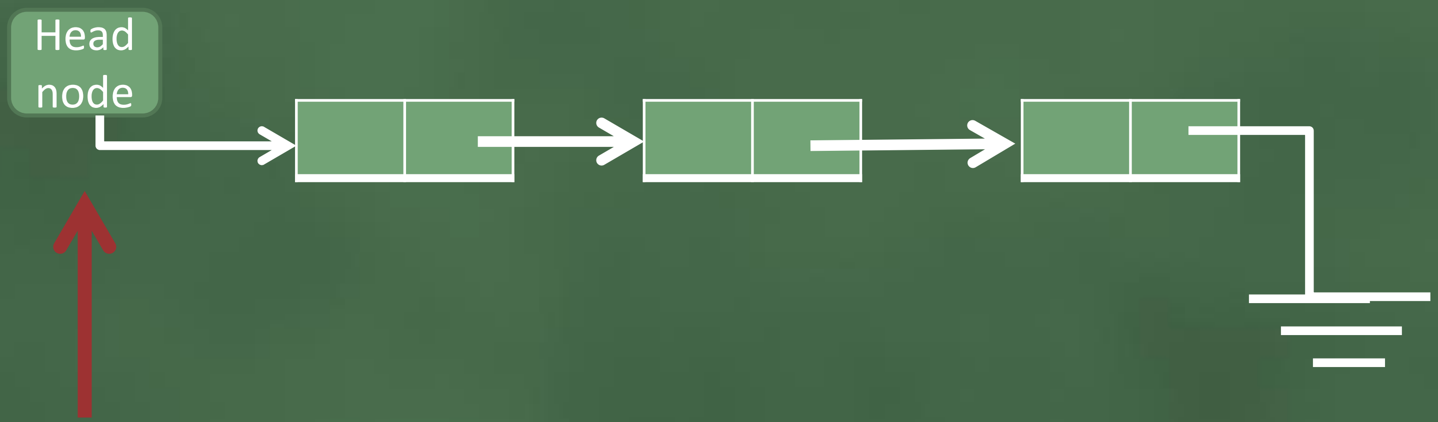



```
Temp =
(node*)malloc(sizeOf(Node)

temp -> data = 4
temp -> next = NULL
```

```
Node* temp1 = A
While (temp1->next !=NULL)
{
        temp1=temp1->next
}
```

Traversal

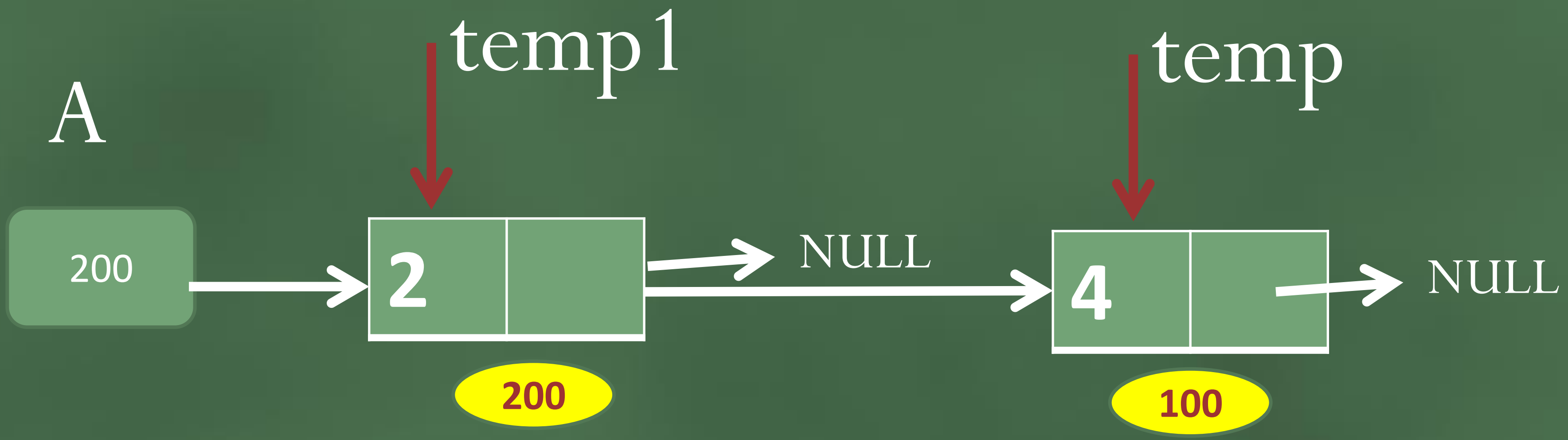



```
Temp =
(node*)malloc(sizeOf(Node)

temp -> data = 4
temp -> next = NULL
```

```
Node* temp1 = A
While (temp1->next !=NULL)
{
        temp1=temp1->next
}
```

Insertion at the end



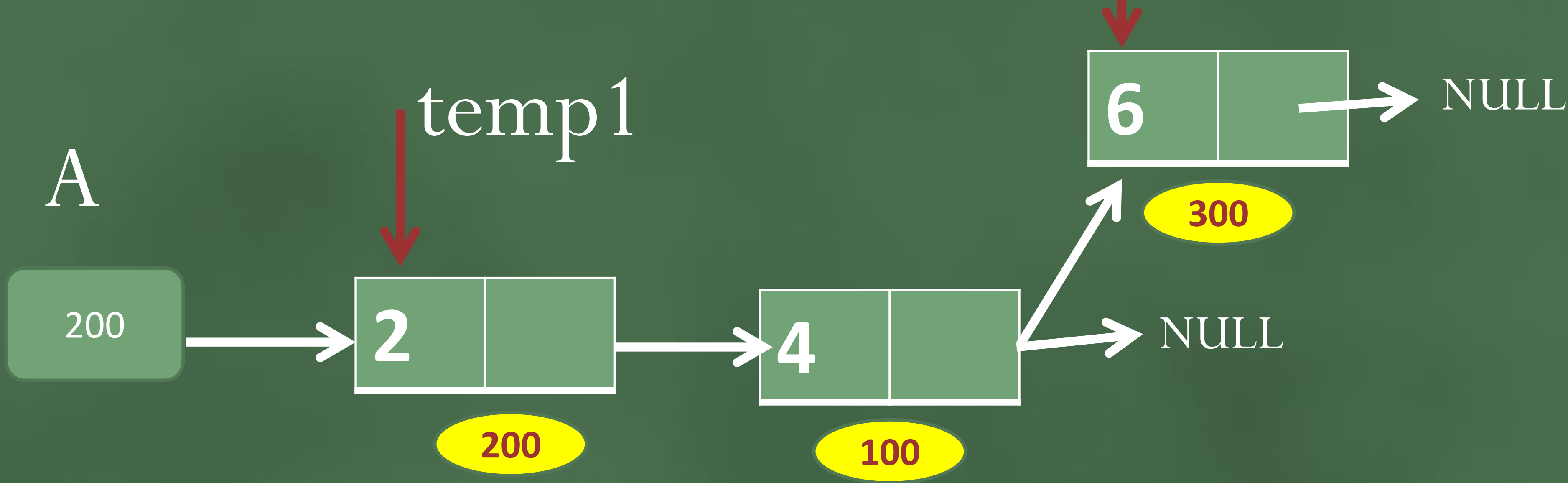

```
Temp =
(node*)malloc(sizeOf(Node)

temp -> data = 4
temp -> next = NULL
```

```
Node* temp1 = A
While (temp1->next !=NULL)
{
        temp1=temp1->next
}

Temp1-> next = temp
```

Insertion at the end



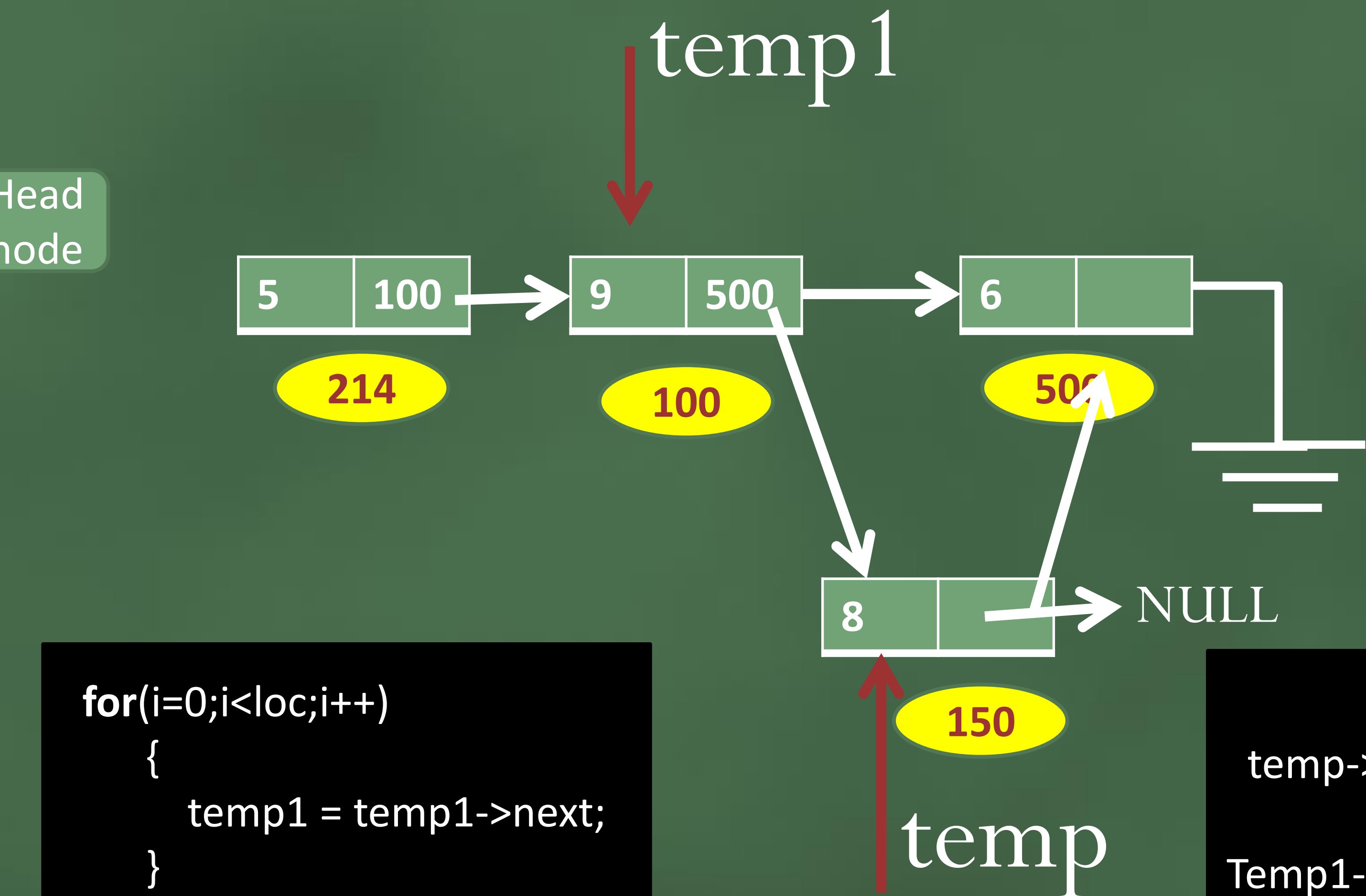
Temp =
(node*)malloc(sizeOf(Node)

temp -> data = 4
temp -> next = NULL

```
Node* temp1 = A
While (temp1->next !=NULL)
{
        temp1=temp1->next
}

Temp1-> next = temp
```

Insert node at nth position




```
for(i=0;i<loc;i++)
    {
       temp1 = temp1->next;
    }
```

```
temp->next = temp1 ->next;

Temp1->next= temp
```

```
#include<stdlib.h>
struct node
{
    int data;
    struct node *next;
};
struct node *head, *ptr;
void beginsert ();
void lastinsert ();
void randominsert();
void begin_delete();
void last_delete();
void random_delete();
void display();
void search();

ptr = (struct node *)malloc(siz
Int main()
{
 int choice =0;
    while(choice != 9)
    {
        printf("\nEnter your choice?\n");
        scanf("\n%d",&choice);
eof(struct node *));
```

```
    {
        case 1:
            beginsert();    break;
        case 2:
            lastinsert();     break;
        case 3
           randominsert();    break;
        case 4:
            begin_delete();    break;
        case 5:
          last_delete();       break;
        case 6:
           random_delete();       break;
        case 7:
            search();       break;
        case 8:
               display();     break;
        case 9:
                 exit(0);   break;
        default:
           printf("Please enter valid choice.
    }
  }
}
```

```
void beginsert()
{
   struct node *ptr;
   int item;
   ptr = (struct node *) malloc(sizeof(struct node *));
   if(ptr == NULL)
   {
      printf("\nOVERFLOW");
   }
   else
   {
      printf("\nEnter value\n");
      scanf("%d",&item);
      ptr->data = item;
      ptr->next = head;
      head = ptr;
      printf("\nNode inserted");
   }

}
```

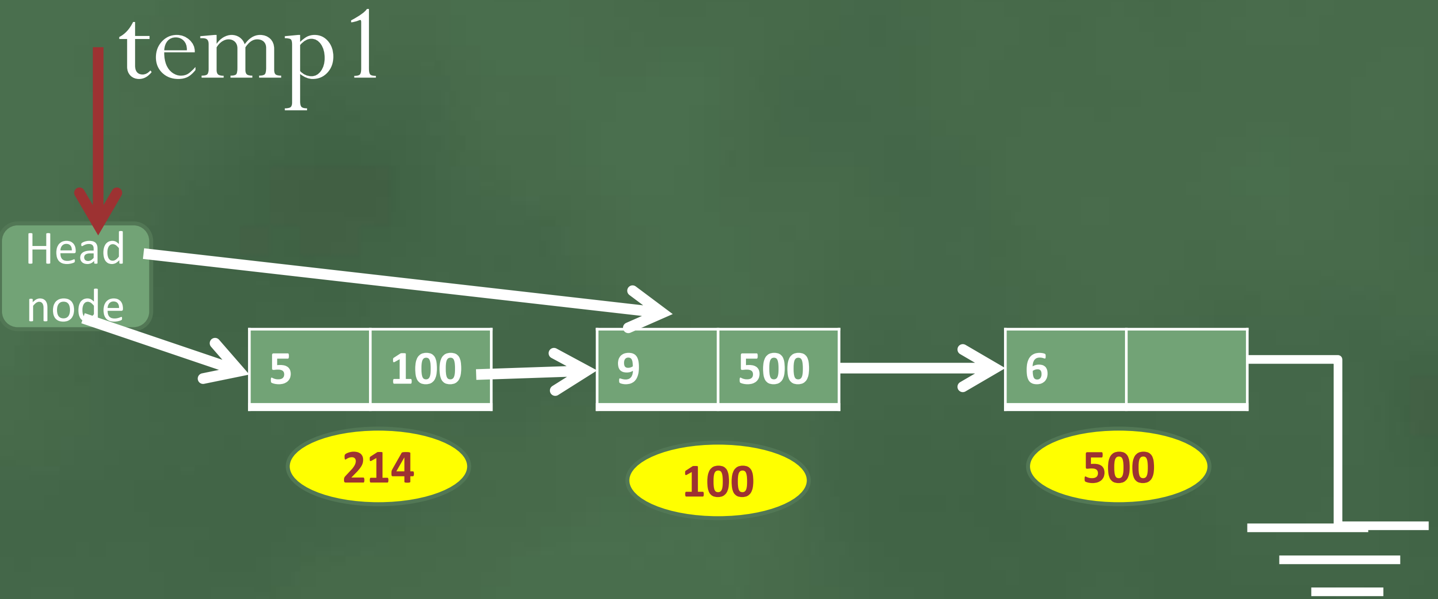
```
void lastinsert()
{      struct node *ptr,*temp;
   int item;
   ptr = (struct node*)malloc(sizeof(struct
   if(ptr == NU {
      printf("\nOVERFLOW");  }
   else  {
      printf("\nEnter value?\n");
      scanf("%d",&item);
      ptr->data = item;
      if(head == NULL {
         ptr -> next = NULL;
         head = ptr;
         printf("\nNode inserted");
      }
      else  {
         temp = head;
         while (temp -> next != NULL) {
            temp = temp -> next;
         }
         temp->next = ptr;
         ptr->next = NULL;
         printf("\nNode inserted");
      }    }  }
```

```
void randominsert()
{
    int i,loc,item;
    struct node *ptr, *temp;
    ptr = (struct node *) malloc (sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter element value");
        scanf("%d",&item);
        ptr->data = item;
        printf("\nEnter the location after which you want t
        scanf("\n%d",&loc);
        temp=head;
```

```
 for(i=0;i<loc;i++)
        {
            temp = temp->next;
            if(temp == NULL)
            {
                printf("\ncan't insert\n");
                return;
            }

        }
        ptr ->next = temp ->next;
        temp ->next = ptr;
        printf("\nNode inserted");
    }
}
```

Delete the first node

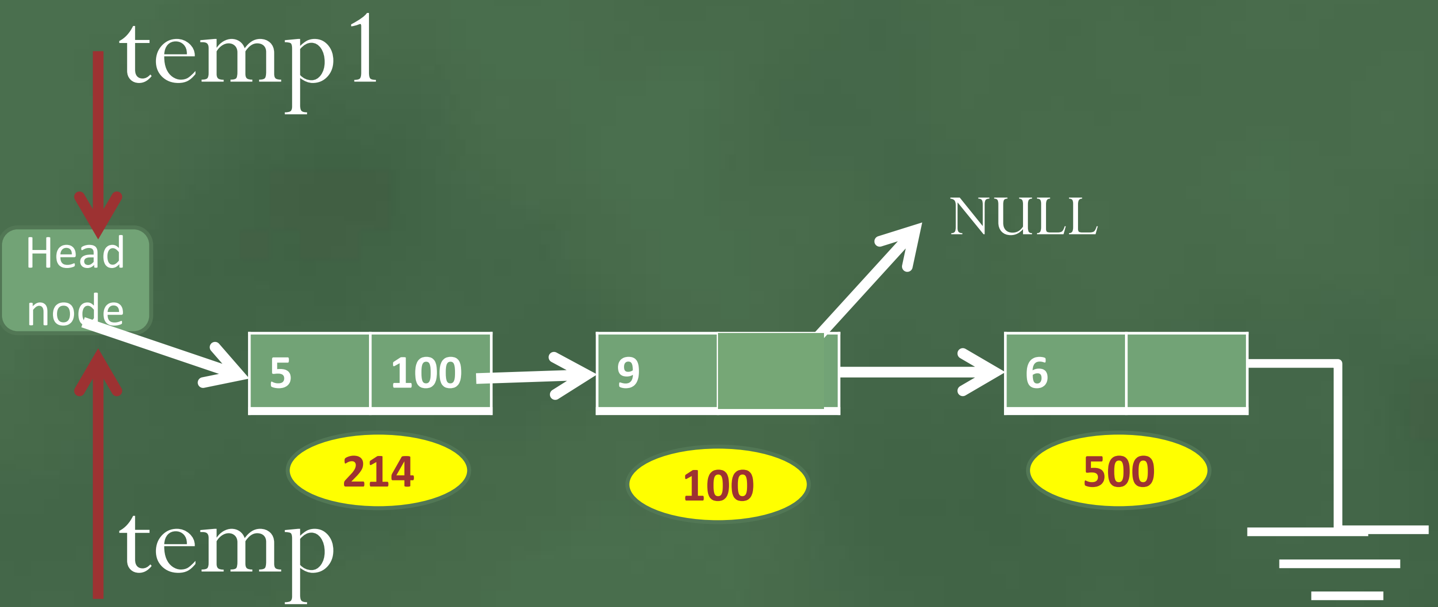



```
temp1 = head

head = temp1 -> next

Free(temp1)
```

Delete the last node

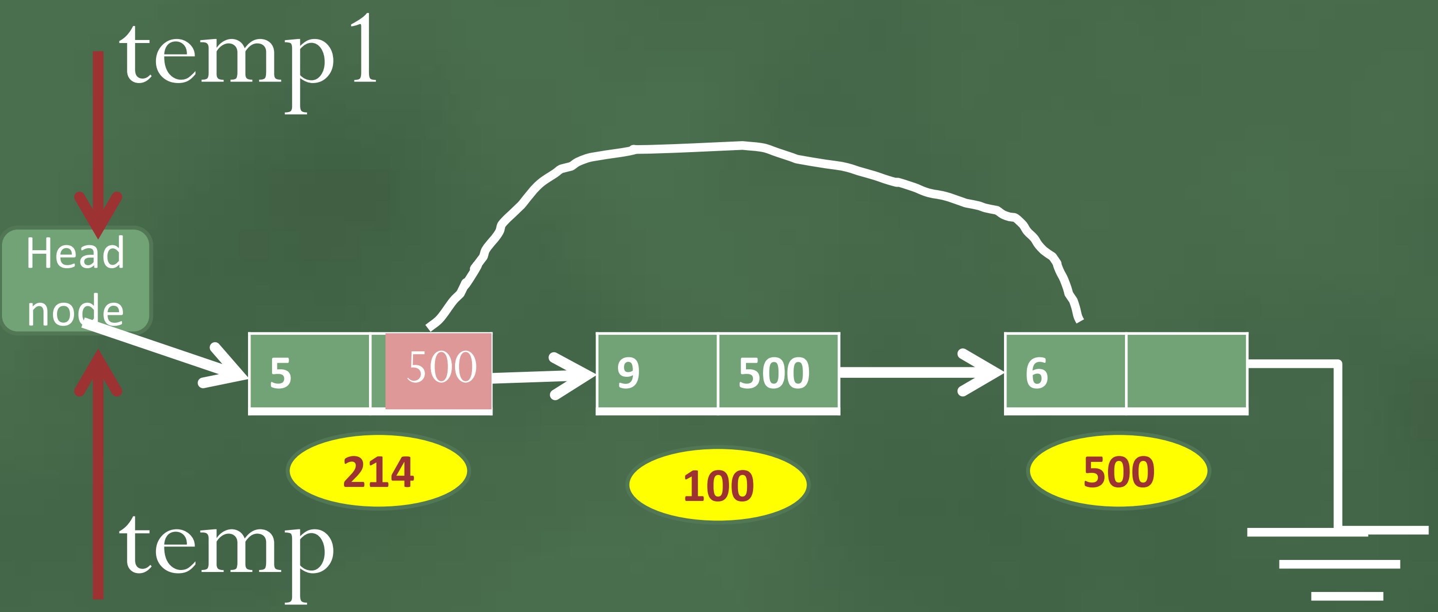



```
  temp1 = head
While( temp1->next!=NULL)
{
Temp = temp1
Temp1= temp1 -> next
}

Temp -> next   = NULL

Free (temp1)
```

Delete the nth node




```
  temp1 = head
For ( I =0; i<n; i++)
{
Temp = temp1;
Temp1=temp1->next
}
Temp -> next   =  temp1-> next

Free (temp1)
```

Delete the first node

```
  void begin_delete()
{
   struct node *ptr;
   if(head == NULL)
   {
      printf("\nList is empty\n");
   }
   else
   {
      ptr = head;
      head = ptr->next;
      free(ptr);
      printf("\nNode deleted from the begining ...\n");
   }
}
```

Delete the last node

```
void last_delete()
{
    struct node *ptr,*ptr1;
    if(head == NULL)
    {
        printf("\nlist is empty");
    }
    else if(head -> next == NULL)
    {
        head = NULL;
        free(head);
        printf("\nOnly node of the list deleted ...\n");
    }
else
    {
        ptr = head;
        while(ptr->next != NULL)
        {
            ptr1 = ptr;
            ptr = ptr ->next;
        }
        ptr1->next = NULL;
        free(ptr);
        printf("\nDeleted Node from the last ...\n");
    }
}
```

- **Deletion**

Deletion

To delete a node **X** between **A** and **B**:

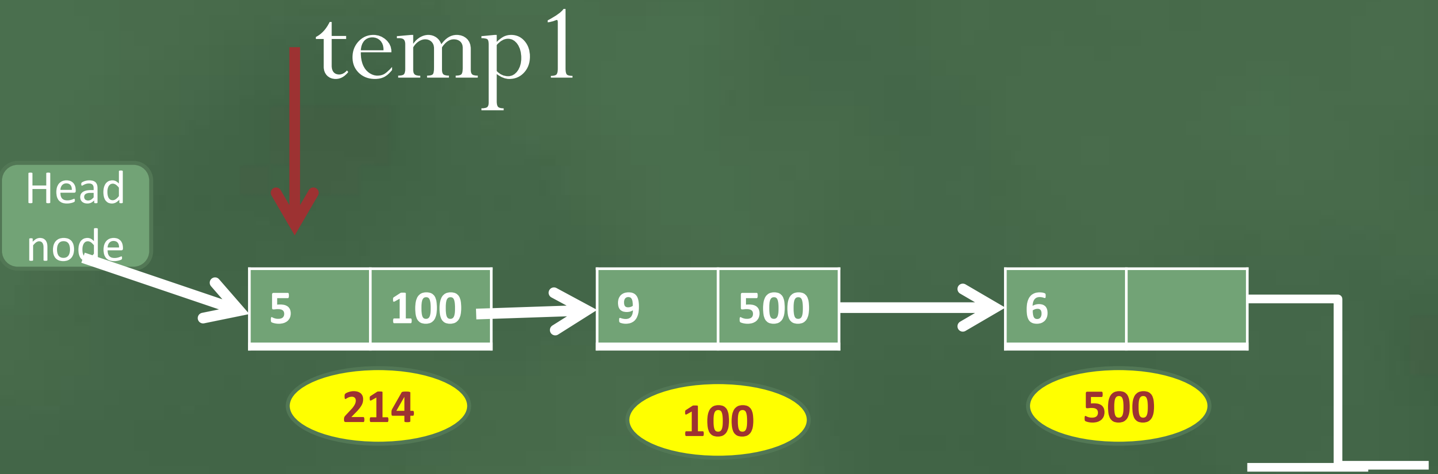
- Create a link from **A** to **B**,
- Remove node **X**

Random delete

```
void random_delete()
{
    struct node *ptr,*ptr1;
    int loc,i;
    printf("\n Enter the location of the node after which you want to perform deletion \n");
    scanf("%d",&loc);
    ptr=head;
    for(i=0;i<loc;i++)
    {
        ptr1 = ptr;
        ptr = ptr->next;

        if(ptr == NULL)
        {
            printf("\nCan't delete");
            return;
        }
    }
    ptr1 ->next = ptr ->next;
    free(ptr);
    printf("\nDeleted node %d ",loc+1);
}
```

Search an element in linked list




```
  temp1 = head
While(temp1!=NULL)
{
If(temp1->data == item)
{
            print(item found at location %d", i+1)
            flag =1
}
Else
            flag =1;

}
i++
Temp1=temp1->next
```

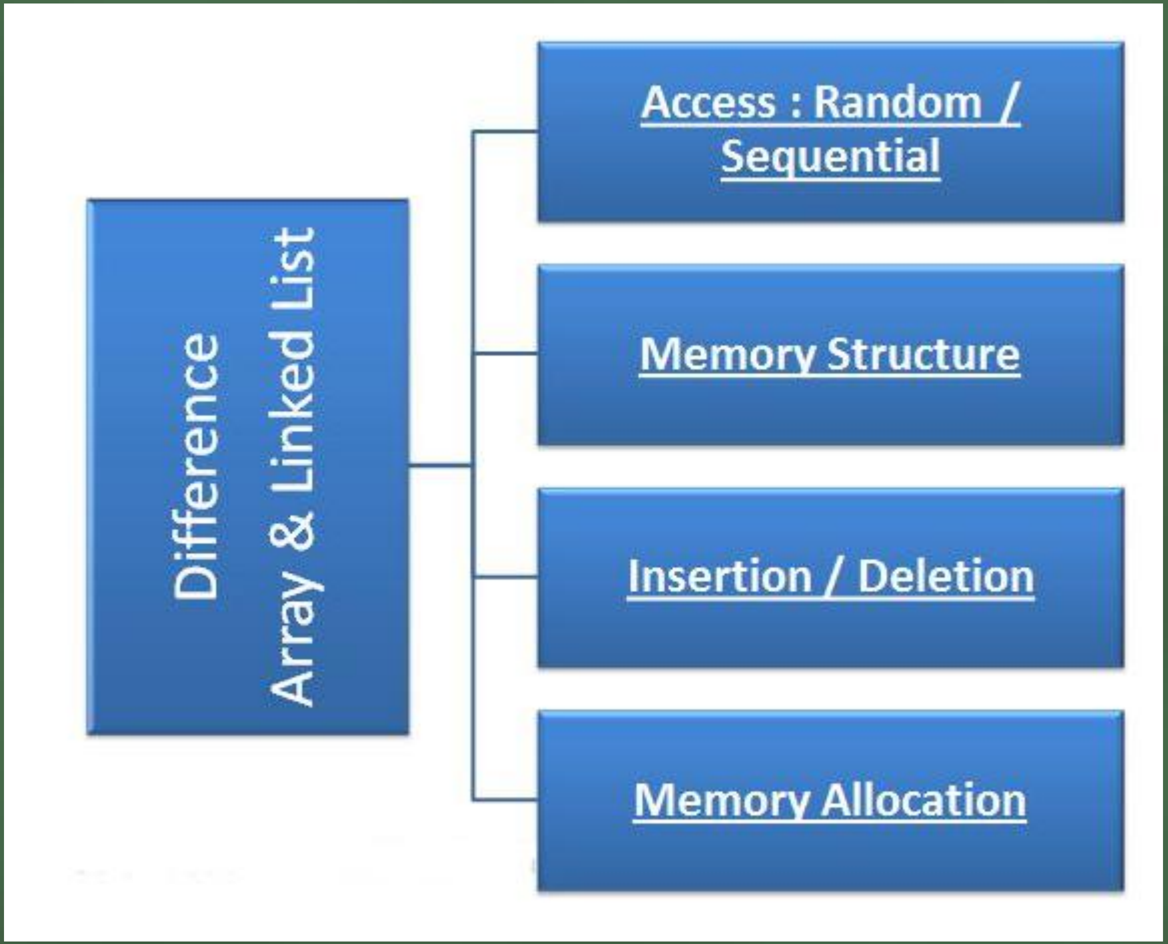
Search

```
void search()
{
   struct node *ptr;
   int item,i=0,flag;
   ptr = head;
   if(ptr == NULL)
   {
      printf("\nEmpty List\n");
   }
   else
   {
      printf("\nEnter item which you want to search?\n");
      scanf("%d",&item);
      while (ptr!=NULL)
      {
         if(ptr->data == item)
         {
            printf("item found at location %d ",i+1);
            flag=0;
         }
         else
         {
            flag=1;
         }
         i++;
         ptr = ptr -> next;
      }
      if(flag==1)
      {
         printf("Item not found\n");
      }
   }

}
```

- Limitations of a singly-linked list:
  - Insertion at the front is O(1)
  - insertion at other positions is **O(*n*)**
  - Insertion is convenient only after a referenced node
  - Removing a node requires a reference to the previous node
  - We can traverse the list only in the **forward direction**

- **Array vs Linked list**



•The List ADT

- Summary
  - running time coomparion

	Array List	(Single) Linked List
findKth	O(1)	O(n)
insert	O(n)	O(1)
delete	O(n)	O(1)

  - when to use Array list or Linked list?
    - **Array list: numerous findKth operations + seldom delete/insert operations**
    - **Linked list: numerous delete/insert operations + seldom findKth operations**

- Time complexity

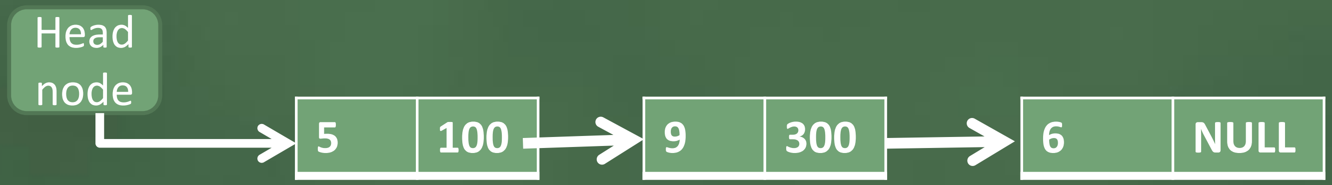
Stack (using Array)	Best Time Complexity	Worst time complexity
Push()	O(1)	O(N)
Pop()	O(1)	O(1)
Peek()	O(1)	O(1)

Queue (using Array)	Best Time Complexity	Worst time complexity
Enque()	O(1)	O(N)
Deque()	O(1)	O(N)
front()	O(1)	O(1)

- **Stack using linked list**
- Insert / delete
  - At the end of list (tail)
  - At beginning (head)

- What if array gets filled?
  - Error : “its full”
  - Create a new larger array and copy data
  - Unused array

•Stack – using linked list



Stack – using linked list

```
struct Node {
    int data;
    struct Node* link;
};
struct Node* top = NULL;
```

```
void Push(int x) {
    struct Node* temp =
      (struct Node*)malloc(sizeof(struct Node*));
    temp->data = x;
    temp->link = top;
    top = temp;
}
```

Stack – using linked list

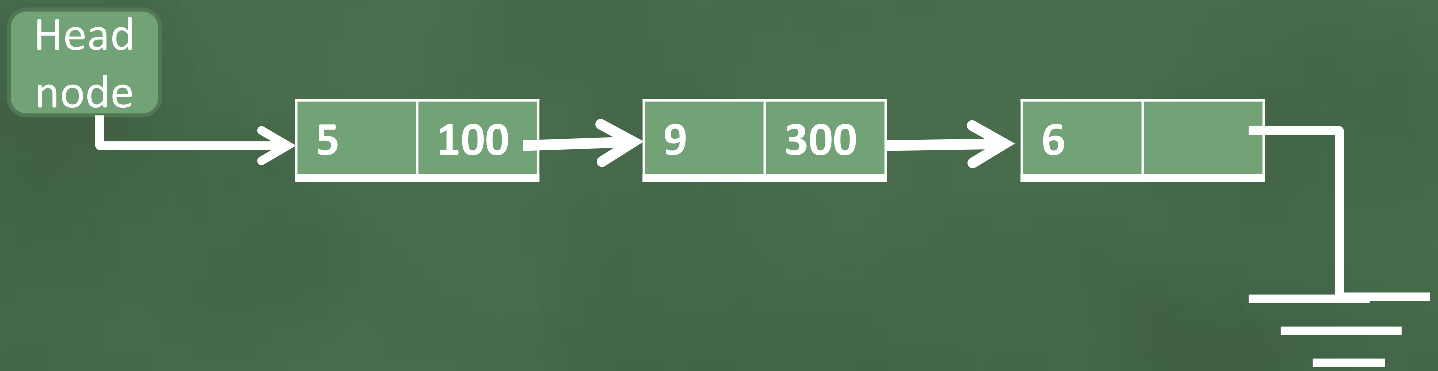
```
void Pop() {
    struct Node *temp;
    if(top == NULL) return;
    temp = top;
    top = top->link;
    free(temp);
}
```

- Time complexity

Stack (using Array)	Best Time Complexity	Worst time complexity
Push()	O(1)	O(N)
Pop()	O(1)	O(1)
Peek()	O(1)	O(1)

Stack (using Linked List)	Best Time Complexity	Worst time complexity
Push()	O(1)	O(1)
Pop()	O(1)	O(1)
Peek()	O(1)	O(1)

Queue – using linked list



Queue – using linked list

```c
/*Queue - Linked List implementation*/
#include<stdio.h>
#include<stdlib.h>
struct Node {
        int data;
        struct Node* next;
};
// Two glboal variables to store address of front and rear nodes.
struct Node* front = NULL;
struct Node* rear = NULL;

// To Enqueue an integer
```

Queue – using linked list

```c
// To Enqueue an integer
void Enqueue(int x) {
        struct Node* temp =
                (struct Node*)malloc(sizeof(struct Node));
        temp->data =x;
        temp->next = NULL;
        if(front == NULL && rear == NULL){
                front = rear = temp;
                return;
        }
        rear->next = temp;
        rear = temp;
}
```

Queue – using linked list

```c
// To Dequeue an integer.
void Dequeue() {
        struct Node* temp = front;
        if(front == NULL) {
                printf("Queue is Empty\n");
                return;
        }
        if(front == rear) {
                front = rear = NULL;
        }
        else {
                front = front->next;
        }
        free(temp);
}
```

Queue – using linked list

```
int Front() {
        if(front == NULL) {
                printf("Queue is empty\n");
                return;
        }
        return front->data;
}

void Print() {
        struct Node* temp = front;
        while(temp != NULL) {
                printf("%d ",temp->data);
                temp = temp->next;
        }
        printf("\n");
}
```

- **Time complexity**

Queue (using Array)	Best Time Complexity	Worst time complexity
Enque()	O(1)	O(N)
Deque()	O(1)	O(N)
front()	O(1)	O(1)

Queue (using Linked List)	Best Time Complexity	Worst time complexity
Enque()	O(1)	O(N) / O(1)
Deque()	O(1)	O(N) / O(1)
front()	O(1)	O(1) / O(1)