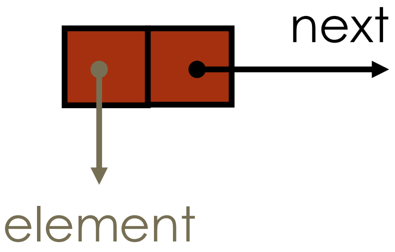
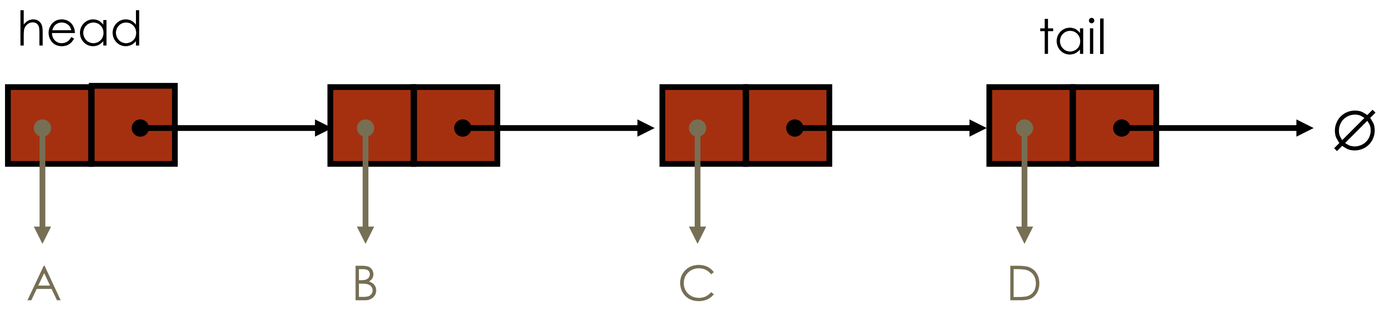
Linked list and Double Linked List

Singly Linked List

- A data structure consisting of a sequence of nodes.



- Each node stores an element and a link to the next node



- In a **linked list** we store items non-contiguously rather than in the usual contiguous array.

Array Vs Linked list

- Arrays are **index-based** data structure where each element associated with an index. Linked list relies on references where each node consists of the data and the references (link) to the next element.
- Basically, an array is a set of similar data objects stored in **sequential memory locations**. While a linked list is a data structure that store items **non-contiguously**.
- Arrays are of fixed size. In contrast, Linked lists are dynamic and flexible and can expand and contract its size.
- In an array, memory is assigned during compile time while in a Linked list it is allocated during execution or runtime.
- Inserting a new element into an array is expensive because a room has to be created for the new elements and to create room existing elements have to be shifted. While, elements can be inserted/deleted into a linked list in a fast and efficient way.

Array vs Linked list

ARRAY	LINKED LIST
The size has to be specified during declaration.	No need to specify the size; grow and shrink during execution.
Element location is allocated during compile time.	Element position is assigned during run time.
Stored consecutively	Stored randomly (non-contiguously)
Element can be accessed directly or randomly. All you need is to specify the array index.	Random access is not allowed. We have to access elements sequentially starting from the first node.
Insertion and deletion of elements are slow as shifting is required.	Ease of insertion/deletion
Memory required is less	Memory required is more
memory utilization is inefficient	memory utilization is efficient

What are the advantages of linked lists over arrays?

What are the drawbacks of linked lists over arrays?

List ADT

InsertFront(e):	Insert a new element e at the beginning of the list.
InsertBack(e):	Insert a new element e at the back of the list.
RemoveFront():	Remove the first element from the list.
RemoveBack():	Remove the last element from the list.
Search(e):	Search for the element e in the list.
InsertAfter(p, e):	Insert a new element e after the position p.
InsertBefore(p, e):	Insert a new element e before the position p.
Remove(p):	Remove element from the list at the position p.
RemoveAfter(p):	Remove the element after the position p.
RemoveBefore(p):	Remove the element before the position p.
ReplaceElement(p,e):	Replace the element at the position p with e.

A Simple Linked List Class

- We use two classes: **Node** and **List**
- Declare a Node class for the nodes
  - `data`: `int` data type in this example.
  - `next`: a pointer to the next node in the list.

```
class Node {
Public:
      Node()
Private:
      int          data          // data
      Node*        next          // pointer to next node
};
```

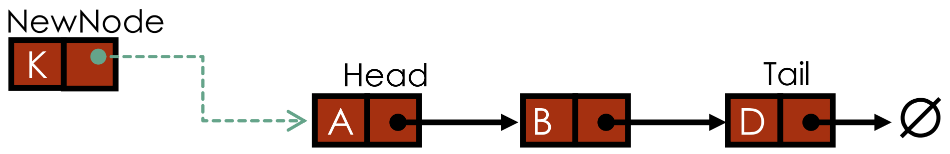
A Simple Linked List Class

- Declare List, which contains
  - head: a pointer to the first node in the list. Since the list is empty initially, head is set to NULL
  - tail: a pointer to the last node in the list.
  - Operations on List

```
class List {
 public:
   List() { head=tail=NULL } // Default constructor
   ~List()                    // destructor
   void   InsertFront(int e)
   void   InsertFront(int e)
   int    RemoveFront()
   int    RemoveBack()
   void   InsertAfter(int p, int e)
   int    RemoveAfter(int p)
   bool   IsEmpty() { return head == NULL}
   int    size()
   void   DisplayList()
 private:
   Node*  head
   Node*  tail
}
```

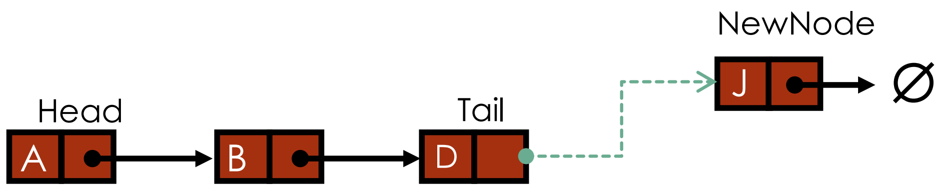
Insert at the beginning

```
void InsertFront(int e)
{
        Node*  NewNode= new Node()
        NewNode -> data= e
        // Empty or not?
        if (head==NULL)
                tail = NewNode
        else
                NewNode -> next=head
        head=NewNode
}
```



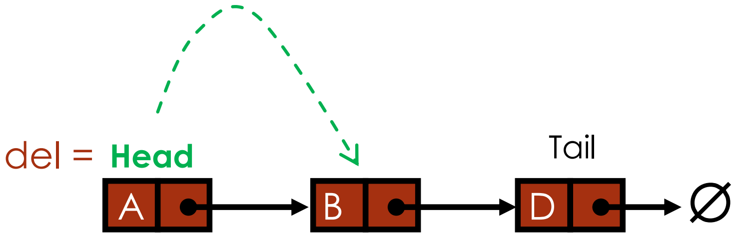
Insert at the end

```
void InsertBack(int e)
{
        Node*  NewNode= new Node()
        NewNode -> next=NULL
        NewNode -> data= e
        // Empty or not?
        if (head== NULL)
                head = NewNode
        else
                tail->next=NewNode
        tail=NewNode
}
```



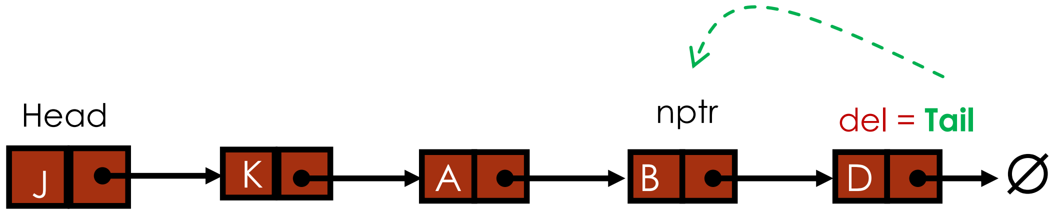
Remove from the beginning

```
int RemoveFront()
{
    // Save a pointer to Node that will be deleted
    Node* del = head
    int e=del->data
    // Adjust head to the next node
    head = head->next
    // If head is null then make tail to be null too. Empty list.
    if (head==NULL)
        tail = 0;
    // Free the deleted Node
    delete del
    return e
}
```



Remove from the End

```
int RemoveBack()
{
    // Save a pointer to the Node that will be deleted
    Node* del = tail
    int e=del->data
    if (head == tail) // One Node
        head = tail = 0
    else
    {
        // More than one Node
        // Find the previous node to the last Node
        Node *nptr = head
        while (nptr->next != tail)
            nptr = nptr->next
        // nptr now points to the next-to-last Node
        tail = nptr
        tail->_next = 0
    }
    // Delete the Node
    delete del
    return e
}
```



InsertAfter(p,e)

```
InsertAfter(int p, int e)
{
    // Save a pointer to the head
    Node* nptr=head
    // Move nptr to the position p
    For(i=1; i<p; i++)
        nptr=nptr->next
    //Make a new node
    Node*  NewNode= new Node()
    NewNode -> data= e
    NewNode->next=nptr->next
    nptr->next=NewNode
}
```

RemoveAfter(p)

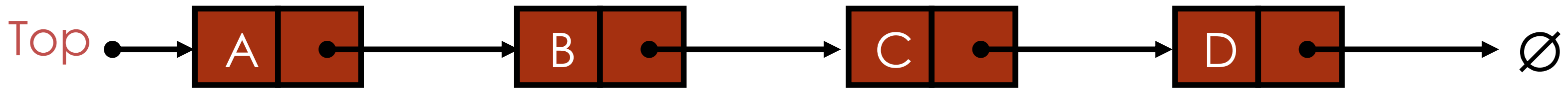
```
int RemoveAfter(p)
{
    // Save a pointer to the head
    Node* nptr=head
    // Move nptr to the position p
    For(i=1; i<p; i++)
        nptr=nptr->next
    Node* del=nptr->next
    int e=del->data
    nptr->next=del->next
    // Delete the Node
    delete del
    return e
}
```

Lab Assignment

- Implement a single linked list.

Stack with a Singly Linked List

- Singly Linked List implementation
  - top is stored at the first node



- Space used is ***O***(***n***) and each operation takes ***O***(1) time.

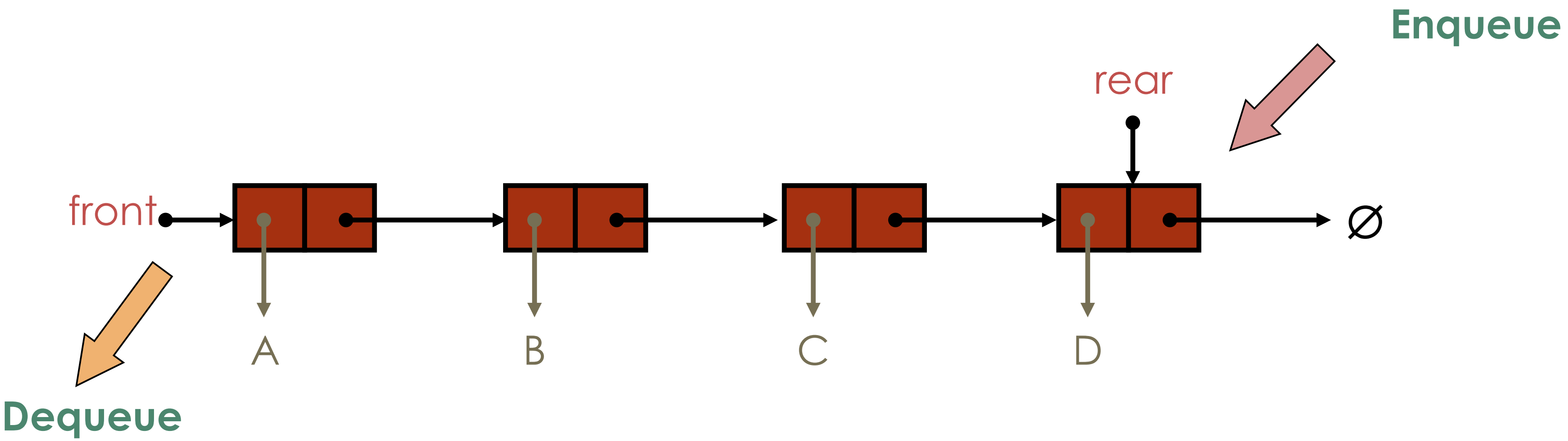
Push and Pop operations

```
Void Push(int e)
{
        Node*  NewNode= new Node()
        NewNode -> next=top
        NewNode -> data= e
        top=NewNode
}
```

```
Int Pop()
{
        if (top==NULL)
                throw an error “Stack is Empty”
        else
        {
                // Save a pointer to Node that will be deleted.
                Node<T>* del = top
                int e=del->data
                // Adjust top to the next node
                top = top->next
                // Free the deleted Node
                delete del
        }
        return e
}
```

Queue with a Singly Linked List

- Singly Linked List implementation
  - front is stored at the first node
  - rear is stored at the last node



- Space used is $O(n)$ and each operation takes $O(1)$ time

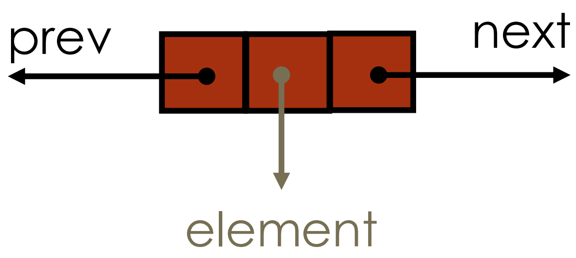
Enqueue and Dequeue operations

```
void Enqueue(int e)
{
    Node*  NewNode= new Node()
     NewNode -> next=NULL
    NewNode -> data= e
    // Empty or not?
    if (front== NULL)
        front = NewNode
    else
        rear>next=NewNode
    rear=NewNode
}
```

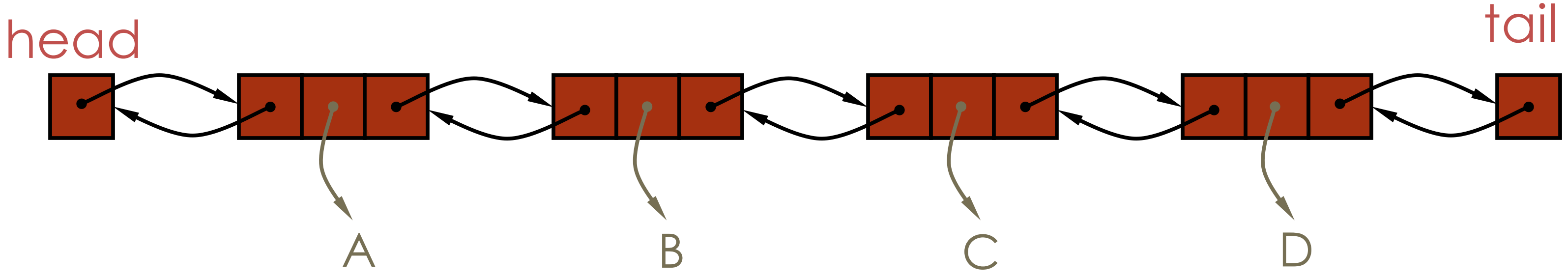
```
int Dequeue()
{
    // Save a pointer to Node that will be deleted
    Node<T>* del = front
    int e=del->data
    // Adjust front to the next node
    front = front>next
    // If front is null then make rear to be null too.
    if (front==NULL)
        rear = 0
    // Free the deleted Node
    delete del
    return e
}
```

Doubly Linked List

- Provides a natural implementation of List ADT
- Nodes store
  - element
  - link to previous node
  - Link to next node



- Special head and tail nodes



Double linked List ADT

InsertFront(e):	Insert a new element e at the beginning of the list.
InsertBack(e):	Insert a new element e at the back of the list.
RemoveFront():	Remove the first element from the list.
RemoveBack():	Remove the last element from the list.
Search(e):	Search for the element e in the list.
InsertAfter(p, e):	Insert a new element e after the position p.
InsertBefore(p, e):	Insert a new element e before the position p.
Remove(p):	Remove element from the list at the position p.
RemoveAfter(p):	Remove the element after the position p.
RemoveBefore(p):	Remove the element before the position p.
ReplaceElement(p,e):	Replace the element at the position p with e.

- We use two classes: **`Node`** and `Dlist`
- Declare a Node class for the nodes
  - `data`: `int` data type in this example.
  - `next`: a pointer to the next node in the list.
  - prev: a pointer to the previous node in the list.

```
class Node
{
public:
	Node()
private:
	int	data
	Node*	prev
	Node*	next

};
```

A Simple Double Linked List Class

- Declare List, which contains
  - head: a pointer to the first node in the list. Since the list is empty initially, head is set to NULL
  - tail: a pointer to the last node in the list.
  - Operations on Double linked List

```
class List {
 public:
   List() { head=tail=NULL } // Default constructor
   ~List()                    // destructor
   void   InsertFront(int e)
   void   InsertFront(int e)
   int    RemoveFront()
   int    RemoveBack()
   void   InsertAfter(int p, int e)
   int    RemoveAfter(int p)
   bool   IsEmpty() { return head == NULL}
   int    size()
   void   DisplayList()
 private:
   Node*  head
   Node*  tail
}
```

Insert at the beginning

```
void InsertFront(int e)
{
        Node*  NewNode= new Node()
        NewNode -> prev=NULL
        NewNode -> data= e
        // Empty or not?
        if (head==NULL)
                tail = NewNode
        else {
                NewNode -> next = head
                head -> prev = NewNode }
        head=NewNode
}
```

Insert at the end

```
void InsertBack(int e)
{
        Node*  NewNode= new Node()
        NewNode -> next=NULL
        NewNode -> data= e
        // Empty or not?
        if (tail== NULL)
                head = NewNode
        else {
                NewNode ->prev = tail
                tail->next=NewNode}
        tail=NewNode
}
```

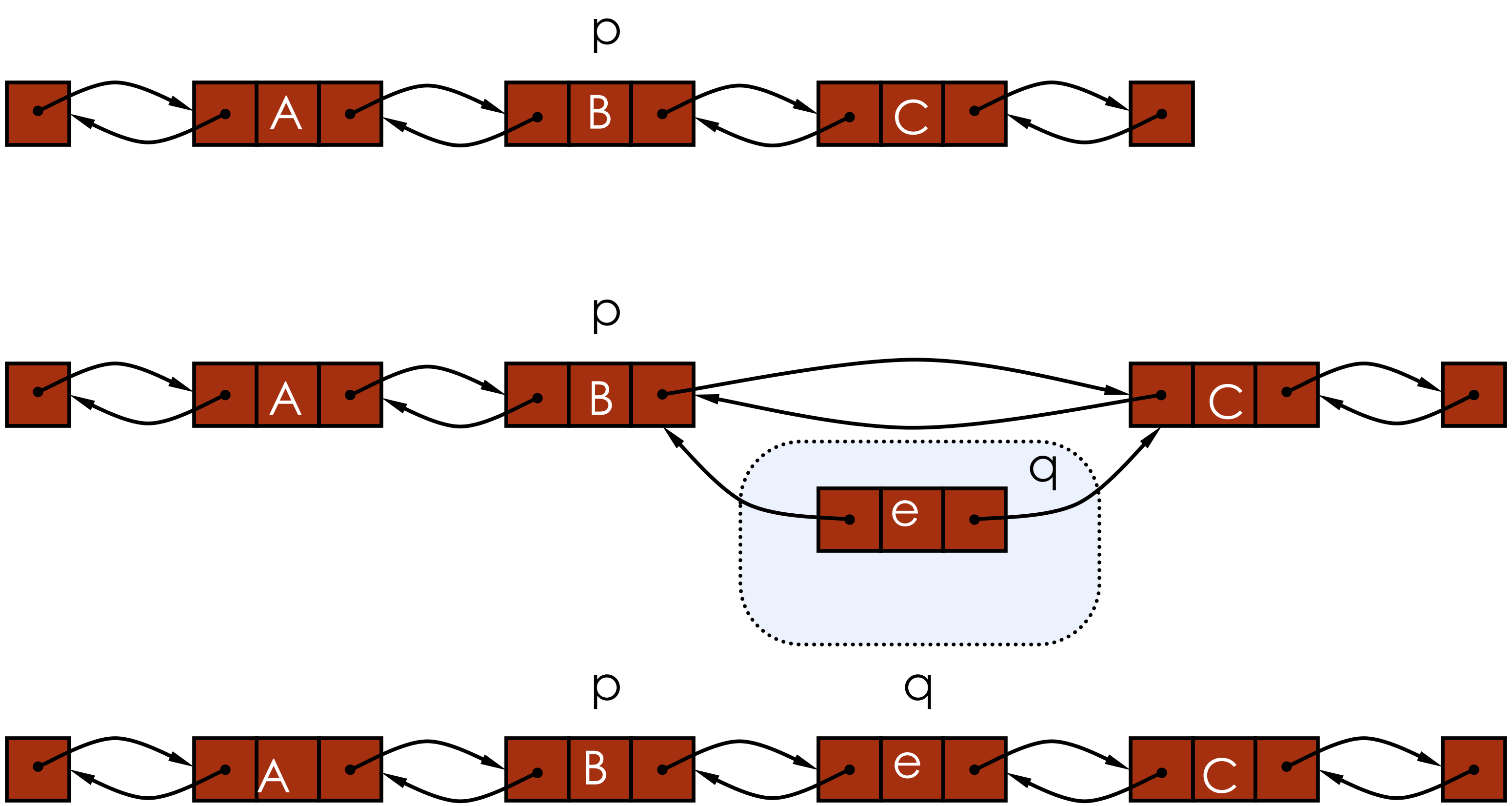
Remove from the beginning

```
int RemoveFront()
{
    // Save a pointer to Node that will be deleted
    Node<T>* del = head
    int e=del->data
    // Adjust head to the next node
    head = head->next
    // If head is null then make tail to be null too. Empty list.
    if (head==NULL)
        tail = NULL;
    else
        head->prev=NULL
    // deleted the Node
    delete del
    return e
}
```

Remove from the End

```
int RemoveBack()
{
    // Save a pointer to the Node that will be deleted
    Node* del = tail
    tail=tail->prev
    if (tail == NULL)
        head  = NULL
    else
        tail->_next = NULL
    // Delete the Node
    delete del
    return e
}
```

Insertion: insertAfter(*p*, *e*)



Deletion: remove(*p*)

- We visualize remove(p), where p = last()