EE 441 Data Structures

Lecture 6:

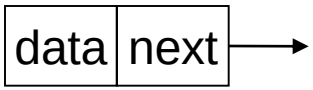
Linked Lists

Remember Problems of Arrays

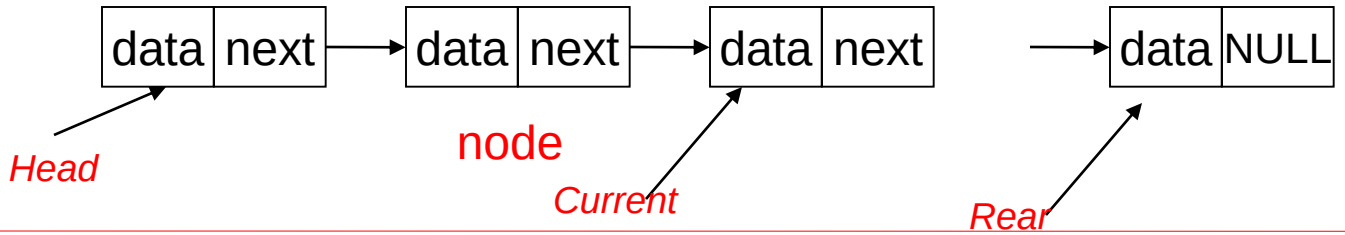
- Fixed Length: Even a dynamic array has fixed size after a resize operation
- Not efficient and flexible in dealing with problems such as:
  - Joining two arrays,
  - Insert an element at an arbitrary location.
  - Delete an element from an arbitrary location
  - Example: Maintaining a list of customers that stay at a hotel

Solution: Linked Lists

- Linked Lists:
  - A set of items usually of the same type (nodes)
  - Each item is linked to the next (with pointers)
- Nodes:
  - Data
  - Link (pointer to next node)



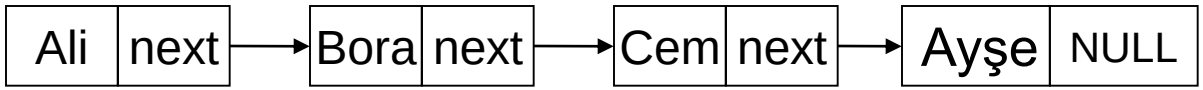
- Special pointers:
  - *List Head:* points to the first node in the linked list
  - *Rear (last node):* points to a NULL address
  - *Current Pointer:* points to the node currently being processed



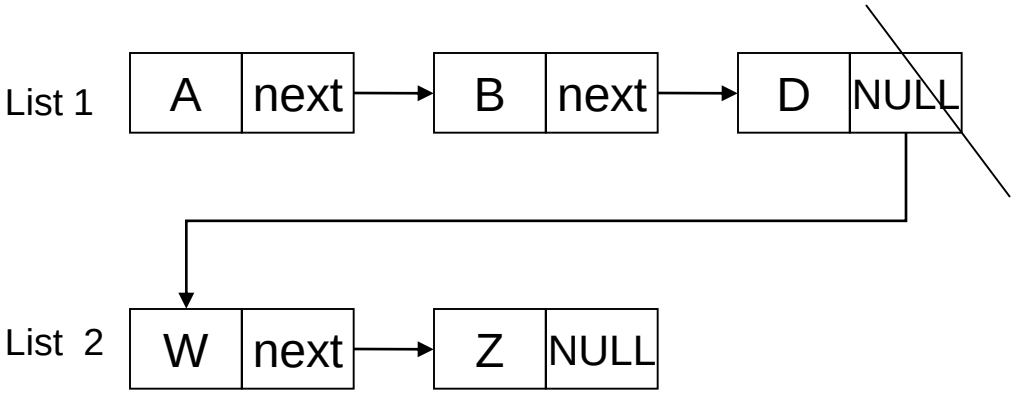
Linked Lists

- Nodes do not have to follow each other physically in memory

Addr	data	next
1	Bora	3
2	Ali	1
3	Cem	5
4		
5	Ayşe	Null

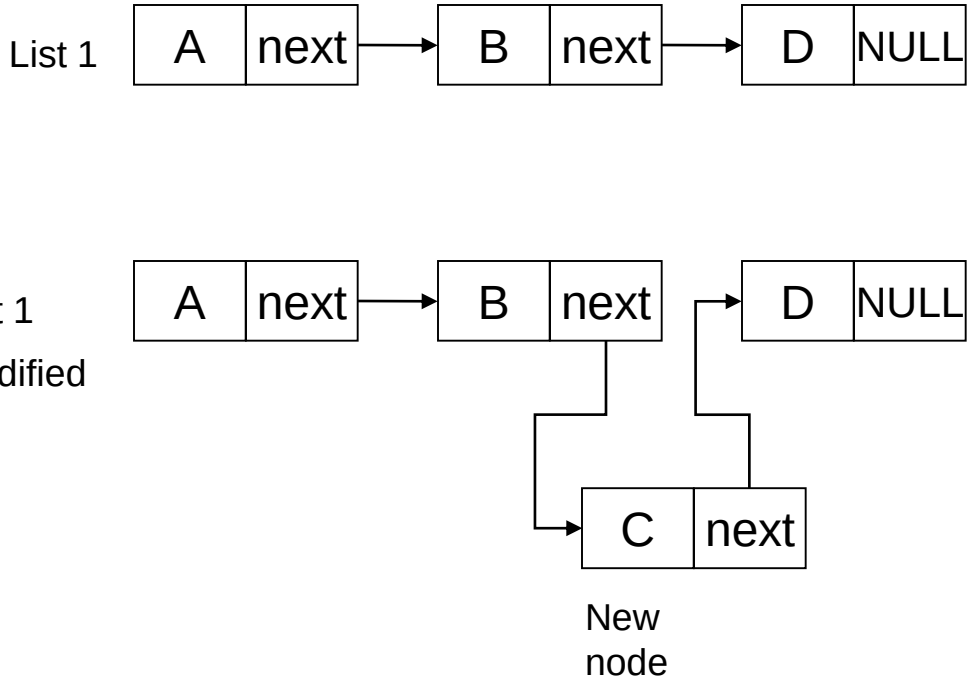


Operations: Combining



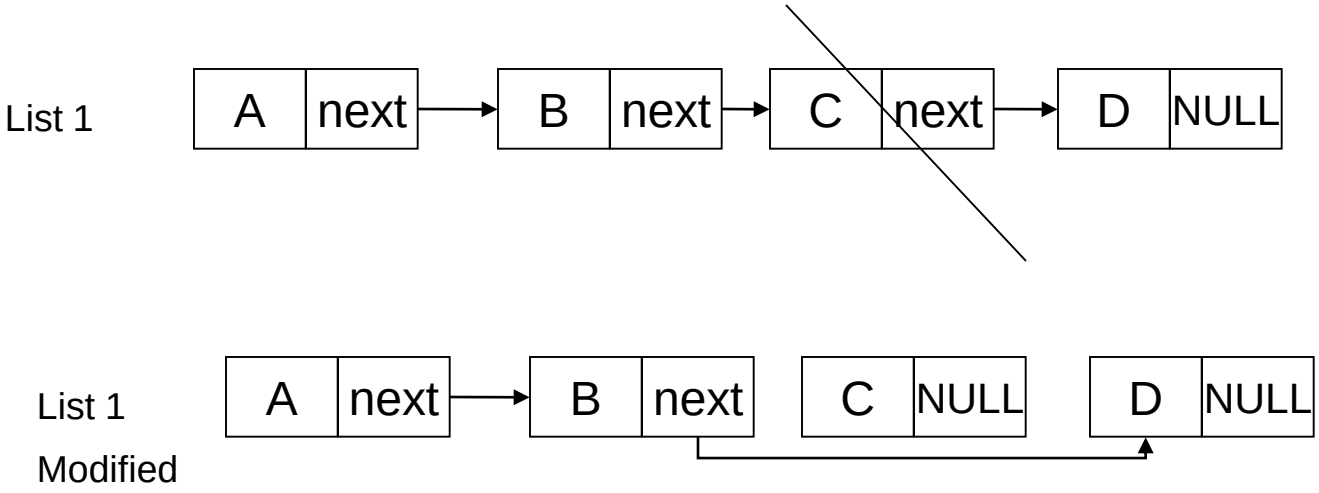
- modify pointer of D to point at W

Operations: Inserting nodes



- Inserting nodes at any position:
  - modify pointer field of C to point to D
  - modify pointer field of B to point to C

Operations: Deleting nodes



- Deleting nodes:
  - modify pointer field of B, to point to the node pointed by pointer of C
  - modify pointer field of OLD as NULL (not to cause problem later on)

What are the problems

- Linked Lists take up extra space because of pointer fields.
- We can't reach the n'th element directly: To reach the nth element, we have to follow the pointers of (n-1) elements sequentially.

Nodes :Constructor

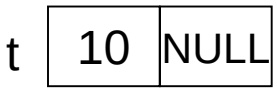
- Constructor for the node:

```
Node (const T& item, Node<T>* ptrnext = NULL);
```

✂ Next pointer is a default argument, if it is not supplied it is NULL

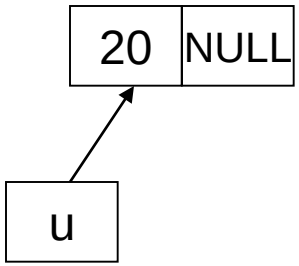
Create a node t with data value 10:

```
Node<int> t(10);
```



Dynamically create a node u with data value 20:

```
Node<int> *u=new Node<int>(20);
```



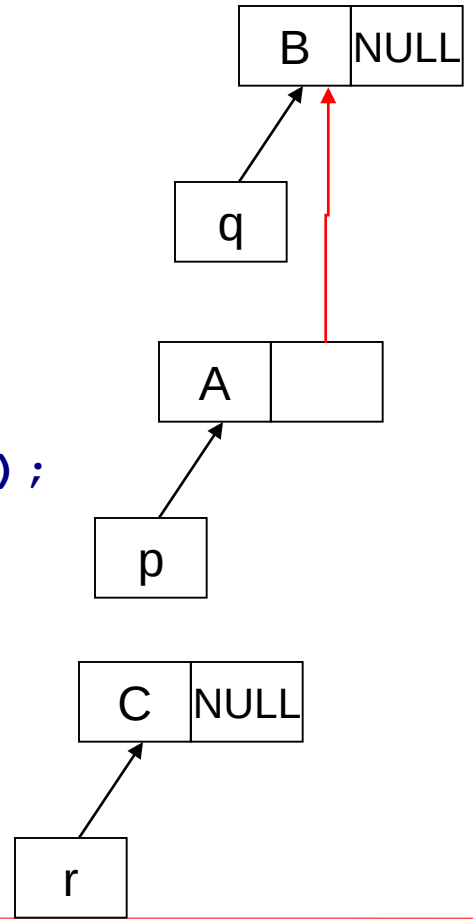
Node Class in C++

```
//declaration of Node Class

template <class T>
class Node
{

  private:     // Data members
     Node <T> *next;
  public :
     T data;

// Constructor

Node (const T &item,
   Node<T>* ptrNext=0);

  // List modification

void InsertAfter(Node<T> *p);
Node <T> *DeleteAfter(void);

 // Access to pointers

Node<T> *NextNode(void)
   const;
}
```

- `data` is public
- `next` is private: next pointers keep the list together, we need to protect them

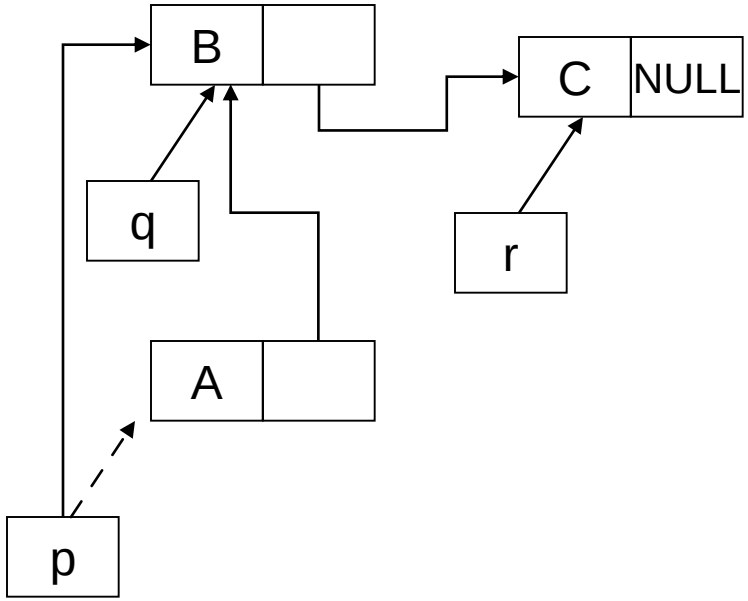
Nodes

```
Node<char> *p, *q, *r;

q=new Node<char>('B');

p=new Node<char>('A',q);

r=new Node<char>('C');
```



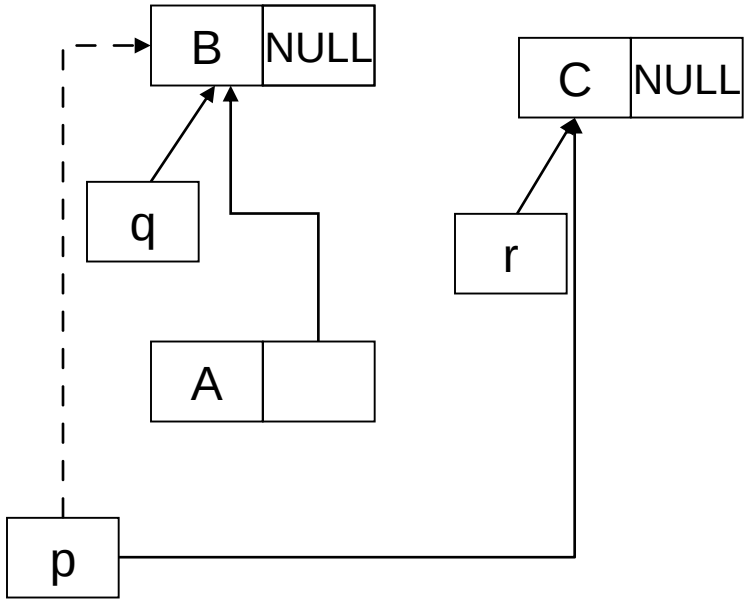
Nodes: Operations with Pointers

```
//Insert node pointed by r
after the node pointed by q
q->InsertAfter(r);

cout<<p->data;    //A

//move p to next node
p=p->NextNode( );

cout<<p->data;    //B
```



Nodes: Operations with Pointers

```
/* Delete node after q copy
the address of the deleted
node to p */
p=q->DeleteAfter( );
```



Node Class in C++

- Constructor

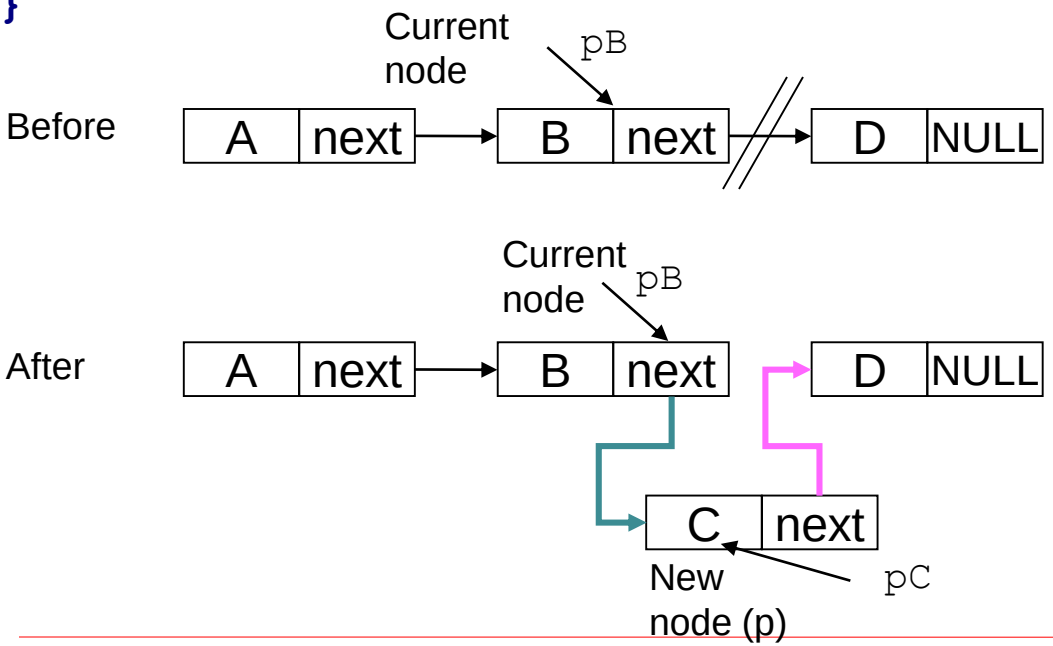
```
template <class T>
Node<T>::Node(const T& item, Node<T>* ptrnext) :
                data(item), next(ptrnext)
{}
```

✂List Traversal

```
template <class T>
Node<T> *Node<T>::NextNode(void) const
{
    return next;
}
```

Node Class in C++:Building the List

```
template <class>
void Node<T>::InsertAfter(Node<T> *p)
{
        p->next=next;//notice access to private part of
                     // member of same class
        next=p; //also note correct sequence of operation
}
```

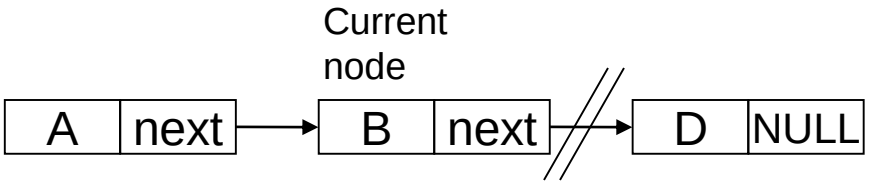


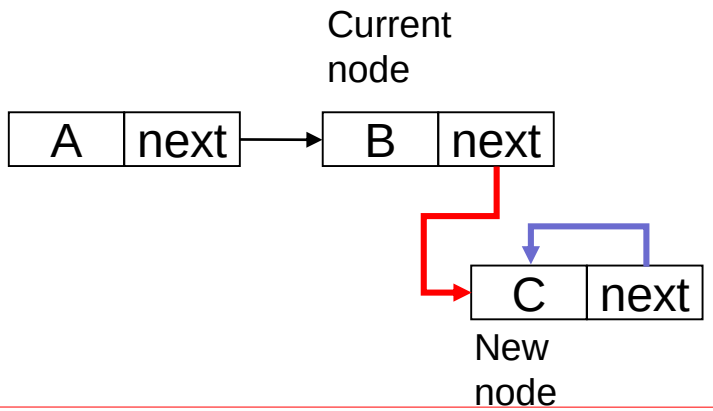

```
pB->InsertAfter(pC);
```

- Two pointers are updated
  - next of the current node
  - next of the new node

Node Class in C++:Order of pointers

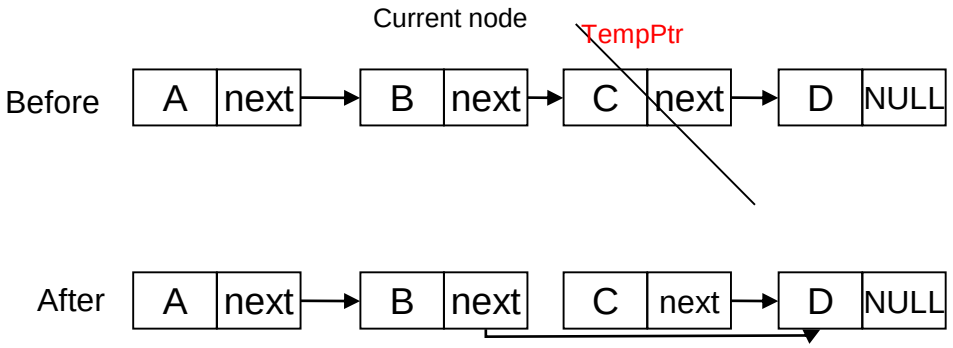
```
template <class>
void Node<T>::InsertAfter(Node<T> *p)
{
        next=p;
        p->next=next
}
```





- Rest of the list is lost!!

Node Class in C++:Deleting nodes




```
Node<T> *Node<T>::DeleteAfter(void)
{
    //save address of node to be
    //deleted
    Node <T> *tempPtr=next;

    //if no successor, return NULL
    if (next==NULL)
        return NULL;

    //delete next node by copying
    //its nextptr to the nextptr of
    //current node
    next=tempPtr->next;

    //return pointer to deleted node
    return tempPtr;
}
```

- Returns the address of the deleted node if the programmer wants to deallocate the memory

Linked List Ideas

- Dynamically created nodes put together
- Unnecessary nodes can be deleted
- List dynamically changes
- Maintaining the list:
  - head pointer points at the beginning of the list
  - Initially the value of the head pointer is NULL to indicate an empty list
  - If you lose the head, you lose the list

Linked List Operations

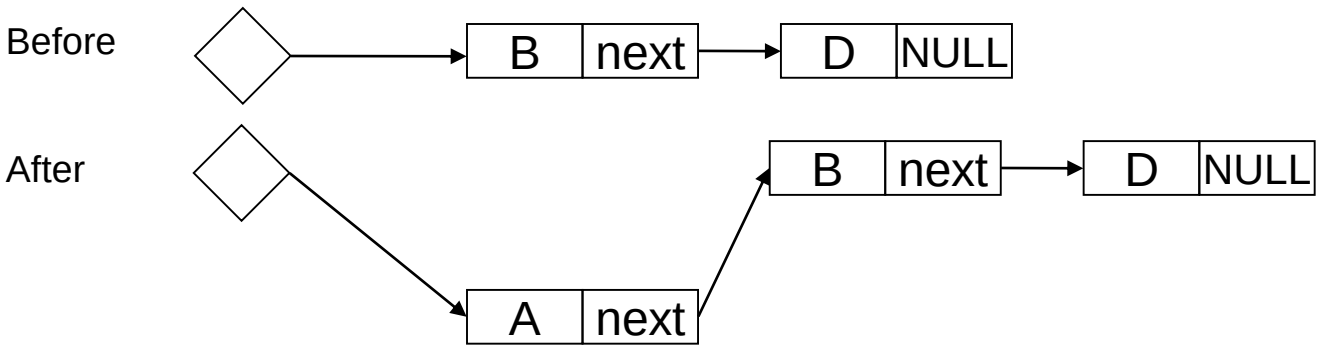
- We will build a list and modify it using:
  - Node class
  - Global (non-member) functions
- Operations:
  - Dynamically creating a node
  - Inserting a node into a linked list
  - Deleting a node from a linked list

Creating a node with dynamic memory allocation

```
template <class T>
Node<T> *GetNode(const T& item, Node<T> *nextPtr=NULL)
{

    Node<T> *newNode; //declare pointer

    newNode=new Node<T>(item, nextPtr);

        /*allocate memory and pass item and nextptr to the constructor which
creates the object*/
        //terminate program if allocation not successful

        if (newNode==NULL)
        {
          cerr<<"Memory allocation failed"<<endl;
          exit(1);
        }
        return newNode;
}
```

Inserting a node at the front of a linked list

```
template <class T>

void InsertFront(Node<T>* & head, T item)
/*we are passing in the address of the head pointer by &head so that it can
be modified*/

{
        //allocate new node so that it points to the first item in the
       //original list, and updated head pointer to point to the new node
        head=GetNode(item,head);
}
```



Inserting a item at the rear of a linked list

```
template <class T>
void InsertRear(Node<T>* & head, const T& item)
{
    Node<T>  *newNode, *currPtr = head;

    // if list is empty, insert item at the front
    if (currPtr == NULL)
        InsertFront(head,item);
    else
    {
        // find the node whose next pointer is NULL
        while(currPtr->NextNode() != NULL)
                currPtr = currPtr->NextNode();

        // allocate memory for the new node and insert at rear (after currPtr)
        newNode = GetNode(item);
        currPtr->InsertAfter(newNode);
        /*
        Remember: Insert After:
        newNode->next = currPtr->next;
        currPtr->next= newNode;
        */
    }
}
```

Deleting a node at the front of a linked list

```
template <class T>
void DeleteFront(Node<T>* & head)
{
    // save the address of node to be deleted
    Node<T> *p = head;

    // make sure list is not empty
    if (head != NULL)
    {
        // move head to second node and delete original
        head = head->NextNode();
        delete p;
    }
}
```

Why do we care

- Operations:
  - Insert at front
  - Insert at rear
  - Delete front
- These are all basic operations of stacks and queues
- We can build stacks and queues using linked lists instead of the arrays

Traversing a linked list

```
template <class T>
void ShowList(Node<T>*&head)
{
    int pos=0;
    Node<T>*currPtr=head;
    while(currPtr!=NULL)
    {
        cout<<"current list position: "<<pos<<" - data: "
                                        <<currPtr->data<<endl;
        //what is the potential problem to print out the data?
        currPtr = currPtr->NextNode();
        pos++;
    }
}
```

Example

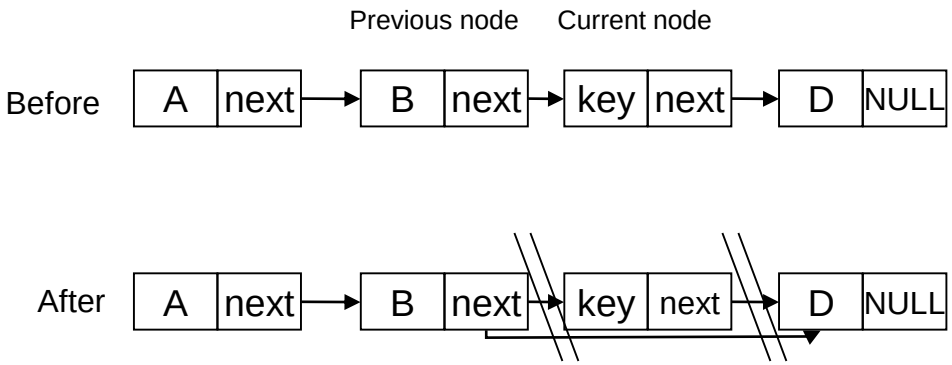
- Write a function to find the first occurrence of "key" in a key and delete it
- Problem:
  - When we find the item it is the current item
  - We defined the node member functions based on the next pointer:

```
void InsertAfter(Node<T> *p);
Node<T> *DeleteAfter(void);
```

How do we keep track of the operations on the current item

Example

- Idea: Move 2 sequential pointers in a pair
  - Previous pointer
  - Current Pointer
  - `next` of previous pointer is the current pointer
  - We can apply `DeleteAfter` on the previous pointer to delete the current pointer



Solution

```
template <class T>
void Delete(Node <T>* &head, T key)
{
    // currPtr moves through list, trailed by prevPtr
    Node<T> *currPtr=head, *prevPtr=NULL;
    // return if the list is empty
    if (currPtr==NULL)
        return;
    // cycle list until key is located or come to end
    while(currPtr !=NULL && currPtr->data!=key)
    {   // advance currPtr so prevPtr trails it
        prevPtr=currPtr; //keep prev item to delete next
        currPtr=currPtr->NextNode();
    }
    if (currPtr!=NULL) //i.e. keyfound
    {
        if (prevPtr==NULL) //i.e key found at first entry
            head=head->NextNode();
        else
            // match occurred at 2nd or subsequent node
            // prevPtr->DeleteAfter() unlinks the node
            prevPtr->DeleteAfter();//note that we return address of the deleted
                                   //node but no delete operation
        delete currPtr; //remove memory space to memory manager
    }
}
```

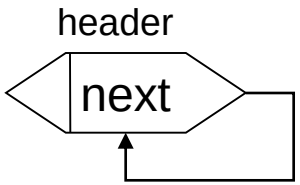
Example

insert item into the ordered list

```
template <class T>
void InsertOrder(Node<T>* & head, T item)
{// currPtr moves through list, trailed by
 //prevPtr
    Node<T> *currPtr, *prevPtr, *newNode;

    // prevPtr == NULL signals match at front
    prevPtr = NULL;
    currPtr = head;

//cycle through the list, find insertion point
    while (currPtr != NULL)
    {
        // found insertion point if item <
        //current data
        if (item < currPtr->data)
            break;

       // advance currPtr so prevPtr trails it
        prevPtr = currPtr;
        currPtr = currPtr->NextNode();
    }

    // make the insertion
    if (prevPtr == NULL)
        // if prevPtr == NULL, insert at front
        InsertFront(head,item);
    else
    {
        // insert new node after previous
        newNode = GetNode(item);
        prevPtr->InsertAfter(newNode);
    }
}
```

Circular Linked List

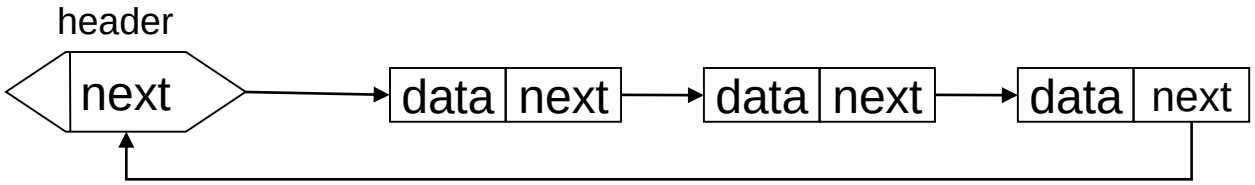
Empty circular list



Tests for an empty list:

Linear: `head==NULL`

Circular: `header->next==header`



- More efficient design for certain applications
- An empty list has one node (header)
- Header (sentinel) node:
  - Not a real node, it does not store data
  - It points at the first real node
- NULL is never used

Circular Linked List

```
template <class T>
class CNode
{
        private:
        // circular link to the next node
                        CNode<T> *next;          // Data members
        public:
        // data is public
                        T data;

        CNode(void);
        CNode(const T& item);                    // Constructors

        void InsertAfter(CNode<T> *p);
        CNode<T> *DeleteAfter(void);             // List Modification

        CNode<T> *NextNode(void) const;          // Access to pointers
};
```

Circular Linked List

- Circular linked list constructors

```
// constructor that creates an empty list and
// leaves the data uninitialized. use for header
template <class T>
CNode<T>::CNode(void)
{
    // initialize the node so it points to itself
    next = this;
}

// constructor that creates an empty list and initializes data
template <class T>
CNode<T>::CNode(const T& item)
{
        // set node to point to itself and initialize data
    next = this;
    data = item;
}
```

Circular Linked List

```
// return pointer to the next node
template <class T>
CNode<T> *CNode<T>::NextNode(void) const
{
    return next;
}

// insert a node p after the current one
template <class T>
void CNode<T>::InsertAfter(CNode<T> *p)
{
    // p points to successor of the
    //current node, and current node
    // points to p.
    p->next = next;
    next = p;
}
```

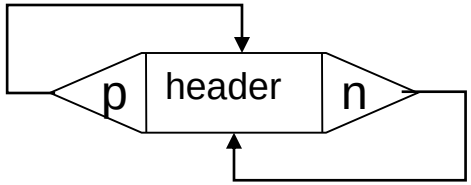
```
// delete the node following current and
return its address
template <class T>
CNode<T> *CNode<T>::DeleteAfter(void)
{
    // save address of node to be deleted
    CNode<T> *tempPtr = next;

    // if next is the address of current
    //object (this), we are
    //pointing to ourself. We don't
    //delete ourself! return NULL
    if (next == this)
        return NULL;

    // current node points to successor
    //of tempPtr.
    next = tempPtr->next;

    // return the pointer to the unlinked
    //node
    return tempPtr;
}
```

List Scan

- Linear list: start from list head, scan prev to next
- Circular list: start from any position in the list, scan to next
- Can be made more Flexible: start from any position in the list, scan in any direction ➔ Circular Doubly linked list
- Note: The new idea here is doubly linked list. Does not have to be circular!

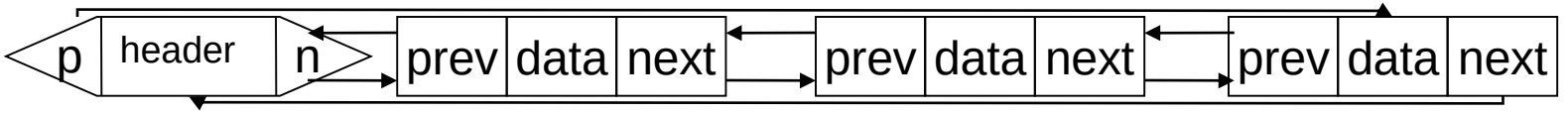
Circular Doubly Linked List

Empty doubly linked list: both pointers point at the node itself



- Used to access nodes in either direction
- Now we write the node such that we can delete the current node without going to the previous node
- Node: two pointers and a data field
- Header node:
  - Not a real node, it does not store data
  - It points at the first and last real nodes

Circular Doubly Linked List



- What do we need:
  - Insert and delete ops in both directions
  - Traverse methods in both directions

Doubly Linked List Class

```
template <class T>
class DNode
{
    private:
        DNode<T> *prev;
        DNode<T> *next;                          // Data members
    public:
        T data;

        DNode(void);                             // Constructors
        DNode(const T& item);

        void InsertNext(DNode<T> *p);
        void InsertPrev(DNode<T> *p);            // List modification
        DNode<T> *DeleteNode(void);

        DNode<T> *NextNodeNext(void) const;      // Access to pointers
        DNode<T> *NextNodePrev(void) const;
};
```

Doubly Linked List Class

- Circular doubly linked list constructors

```
// constructor that creates an empty list and
// leaves the data uninitialized. use for header
template <class T>
DNode<T>::DNode(void)
{
    // initialize the node so it points to itself
    prev = next = this;
}

// constructor that creates an empty list and initializes data
template <class T>
DNode<T>::DNode(const T& item)
{
    // set node to point to itself and initialize data
    prev = next = this;
    data = item;
}
```

Doubly Linked List Class

```
// insert a node p to the next of current node
template <class T>
void DNode<T>::InsertNext(DNode<T> *p)
{
    // link p to its successor on the next
    p->next = next;
    next->prev = p;
    // link p to the current node on its prev
    p->prev = this;
    next = p;
}

// insert a node p to the prev of current node
template <class T>
void DNode<T>::InsertPrev(DNode<T> *p)
{
    // link p to its successor on the prev
    p->prev = prev;
    prev->next = p;
    // link p to the current node on its next
    p->next = this;
    prev = p;
}
```

Doubly Linked List Class

```
// unlink the current node from the list and return its address
template <class T>
DNode<T> *DNode<T>::DeleteNode(void)
{
    // node to the prev must be linked to current node's next
    prev->next = next;
    // node to the next must be linked to current node's prev
    next->prev = prev;
    // return the address of the current node
    return this;
}

// return pointer to the next node on the next (right)
template <class T>
DNode<T> *DNode<T>::NextNodeNext(void) const
{
    return next;
}

// return pointer to the next node on the prev (left)
template <class T>
DNode<T> *DNode<T>::NextNodePrev(void) const
{
    return prev;
}
```

EE 441 Data Structures

Lecture 6:

Linked Lists