EE 441 – CH6

LINKED LISTS

Instructor: UĞUR HALICI

Arrays are not efficient in dealing with problems such as:

- Joining two arrays,
- Inserting an element at an arbitrary location in a sorted array
- Delete an element from an arbitrary location

To overcome these problems, another data structure called linked list can be used in programs.

Linked list is formed of a set of data items connected by link fields (pointers).

So, each node contains:

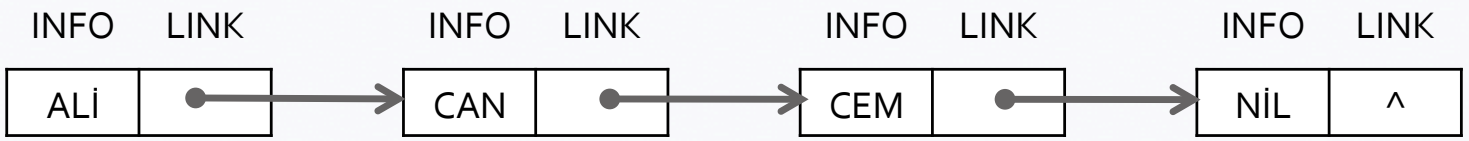
a) an info (data) part,

b) a link (pointer) part

Nodes do not have to follow each other physically in memory

The linked list ends with a node which has "^" (nil) in the link part, showing that it is the last element of the chain.

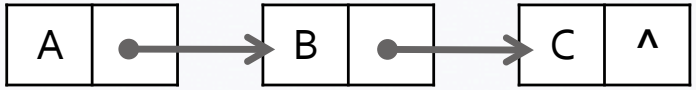
Example:



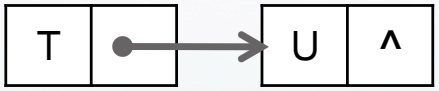
The physical ordering of this linked list in the memory may be

	INFO	LINK
1008	CAN	1128
1024	ALİ	1008
1062	NİL	^
1128	CEM	1062

LIST1:

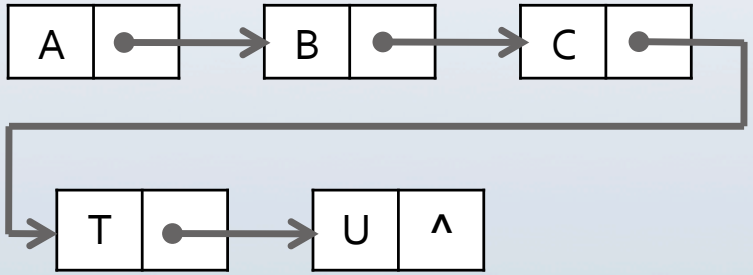


LIST2:

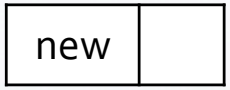


To join LIST1, LIST2: modify pointer of "C" to point to "T".

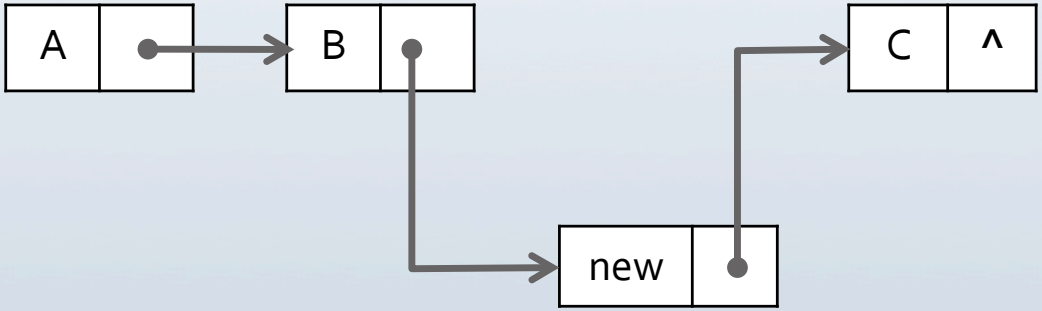
LIST1:



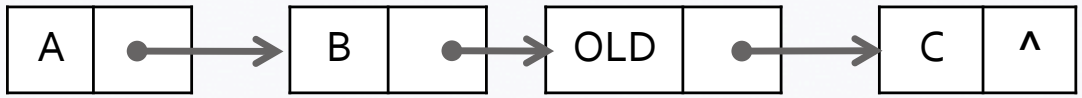
To insert a new item after B:



1) modify pointer field of NEW to point to C
2) modify pointer field of B to point to NEW

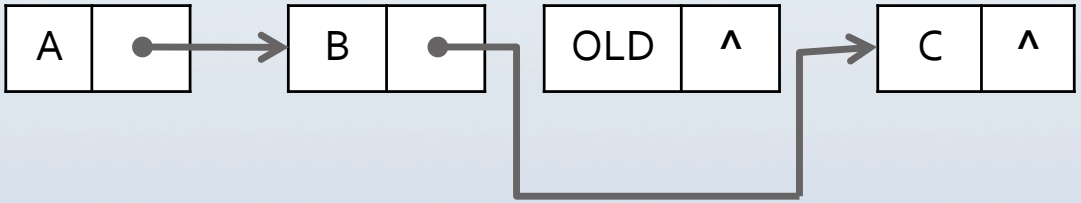


To delete an item coming after B,



1) modify pointer field of B, to point to the node pointed by pointer of OLD

2) modify pointer field of OLD as ^ (not to cause problem later on)

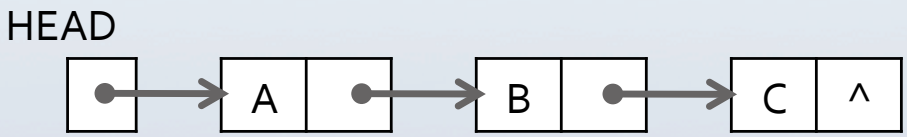


Some Problems with linked lists can be listed as follows:

1) They take up extra space because of pointer fields.

2)To reach the n'th element, we have to follow the pointers of (n-1) elements sequentially. So, we can't reach the n'th element directly.

For each list, let's use an element "list head" which is simply a pointer pointing at the first entry of the list:



Implementation Node Class in C++

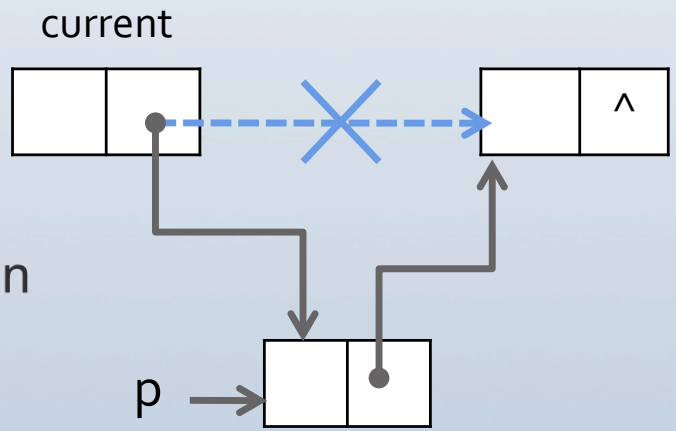
```
//declaration of Node Class
template <class T>
class Node
{private:
Node <T> *next; // next  part is a pointer to nodes of this type
public :
T data; // data part is public
// constructor
Node (const T &item, Node<T>*  ptrNext=0);
//list modification methods
void InsertAfter(Node<T> *p);
Node <T> *DeleteAfter(void);
//get address of next node
Node<T> *NextNode(void) const;
}
```

Note: that the pointer member is private while the data member is public. To avoid the need for function *NextNode, we could declare *next to be public.

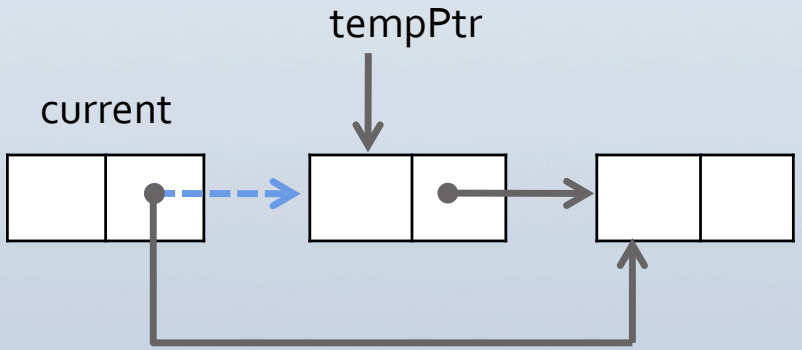
```
// Class Implementation
//constructor
template <class T>
Node <T>::Node(const T& item, Node<T>* ptrnext): data(item),
    next(ptrnext)
{}

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

//insert node pointed by p after the current one
template <class>
void Node<T>::InsertAfter(Node<T> *p)
{ // syntax      p-> ≡ *p and  p->next ≡ (*p).next
  p->next=next;
  next=p; //also note correct sequence of operation
}
```



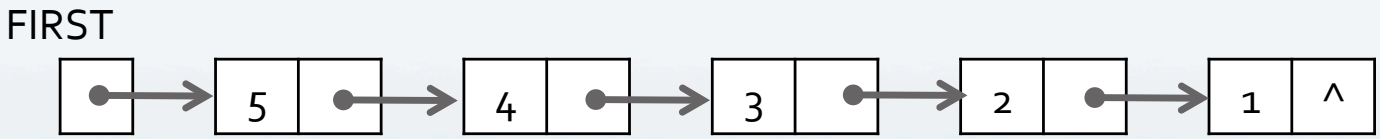
```
//delete node following the current node and return its address
Node<T> *Node<T>::DeleteAfter(void)
{
//save address of node to be deleted
Node <T> *tempPtr=next;
//if no successor, return NULL
if (next==NULL)
    return NULL;
//delete next node by copying its nextptr to the
//nextptr of current node
    next=tempPtr->next;
//return pointer to deleted node
    return tempPtr;
}
```



Now, let's define a template-function GetNode that dynamically allocates a node and initializes it

```
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;
}
```

```
Node <int> *first=null;
for (i=1; i<=5; i++)
  first = getNode(i, first);
```



```
//function to insert a new item at the front of a 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);
}
```

Execises:

1) write a function to insert a new item at the end of a list
2) write a function to find and delete the first occurrence of "key" in a list
3) repeat (2) for all occurences
4) write a function to reverse the order of a list
5) Write a procedure Add(Node <T> * p1, *p2) that will add two polynomials of variable x represented by doubly linked lists whose head nodes are pointed by P1 & P2
6) repeat (5) for multiplication of two polinomials
7) Consider $p(x,y,z)=2xy^2z^3+3x^2yz^2+4xy^3z+5xy^3z+5x^2y^2$

   rewrite so that terms are ordered lexicographically , that is x in ascending order, for erqual x powers y in a.o., then z in a.o
8) Write a procedure to count the no. of nodes in a list

Example: Function to delete the first occurrence of "key" in a list

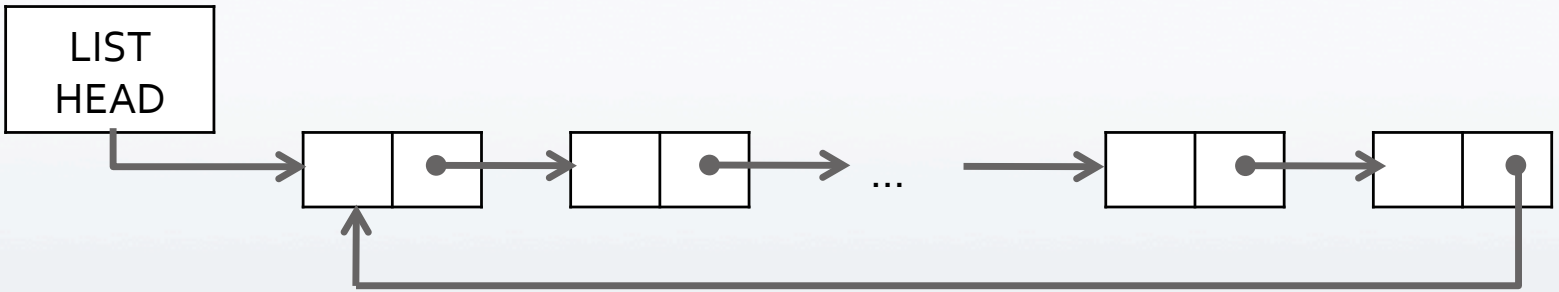
```
template <class T>
void Delete(Node <T>* &head, T key)
{Node<T> *currPtr=head, *prevPtr=NULL
//return if listempty
if (currPtr==NULL)
return;
while(currPtr !=NULL&&curPtr->data!=key)
{
     prevPtr=currPtr; //keep prev item for being able to apply deleteAfter
     currPtr=currPtr->NextNode();
}
if (currPtr!=NULL) //i.e. keyfound
{
      if (prevPtr==NULL) //i.e key found at first entry
            {head=head->NextNode();}
      else
            {prevPtr->DeleteAfter(); }
 delete currPtr; //remove memory space to memory manager
  }
}
```

Example: a function to reverse the order of a list

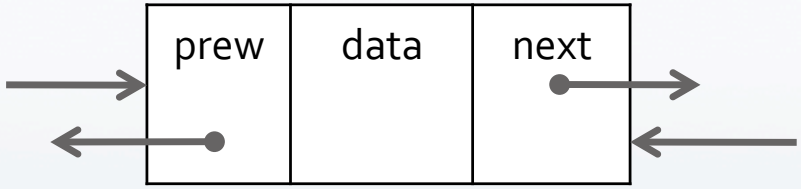
```
template <class T>
void Reverse(Node<T>* &first)
{Node<T> *first1, *temp;
 first1=first;
first=first->next;
first1->next=null;
while (first!=null)
{temp=first;
 first=first->next;
 temp->next=first1;
first1=temp;
}
```

Circular Lists

The last node points to the first

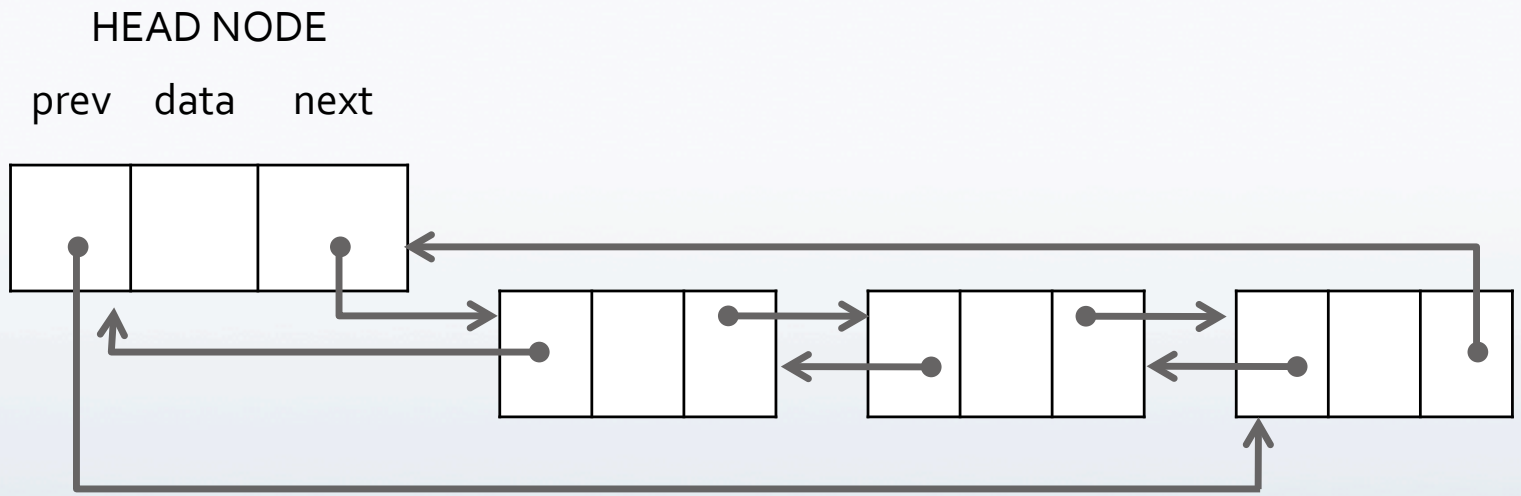


Doubly Linked Lists



1) Easy to traverse both ways
2) Easy to delete the node which is pointed at (rather than the one following it, as in the case of simply linked lists)

Example: A doubly linked circular list:

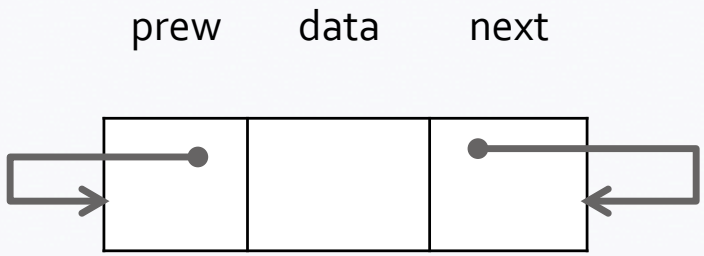


Conventions:

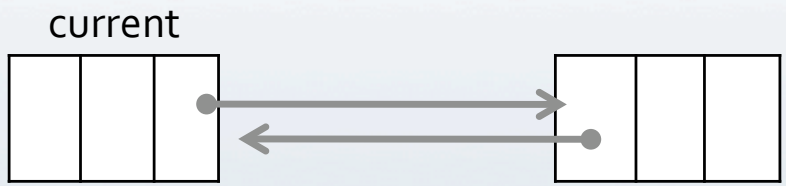
HEAD NODE: does not carry data, it simply points to the first node (NEXT) and the last node (PREV)

NEXT pointer of the last node & the PREV pointer of the first node point to the HEAD NODE

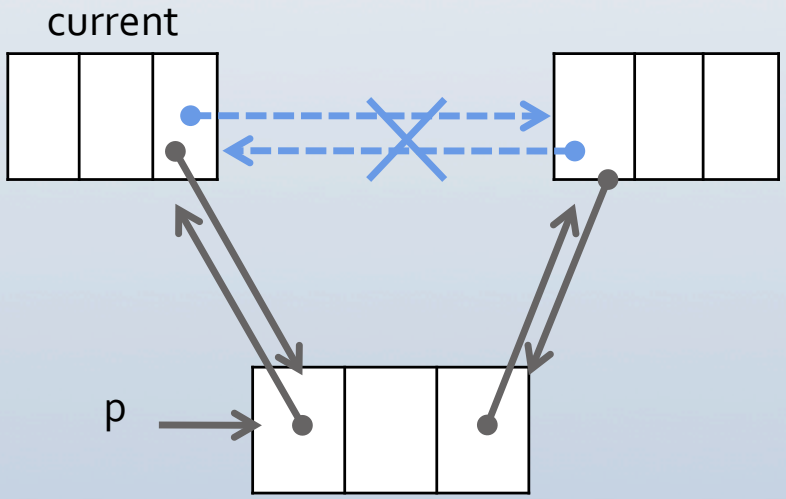
empty doubly linked list



Insertion into a doubly linked list:




```
void  dinsert(Node *p)
/*insert node pointed by p to the right of current node*/
current
{ p->next=next;
 p->prev=this;
p->next->prev=p;
next=p;
};
```



Implementation of a linked list as a class

Note: all list processing functions can be implemented using only the Node class and node operations, however this makes it more object oriented.

```
# include <iostream.h>
#include <stdlib.h>
#include "node.h"
//assuming that "node.h" contains a complete definition of the node class
template <class T>
class LinkedList
{Private:
//pointers to access front and rear
Node <T>  *front, *rear;
// pointers for traversal
Node <T> *PrevPtr, *currPtr;
//count of elements in list
int size;
//relative position of current
int position;

//private methods to allocate and deallocate nodes
Node<T> *GetNode(const T& item, Node<T>* ptrNext=NULL);
void FreeNode(Node<T> *p);
// copy list L to current list
void CopyList(const LinkedList<T>& L);
public:
//constructor
LinkedList(void);
LinkedList(const LinkedList<T>& L);
// Destructor
~LinkedList(void)
```

```
// assignment
LinkedList<T>&operator=(const LinkedList<T>&L);
//check list status
int ListSize(void) const;
int List Empty(void) const;
//Traversal
void Reset (int pos=0);
//sets prevPtr to currPtr and currPtr to the address corresponding to given pos
// if pos==0 the method sets prevPtr to null and currPtr to the front of the list
void Next(void); // advance both pointers by one node
int EndOfList(void) const; // indicate whether currPtr is pointing to the last node
int CurrentPosition(void) const;
//returns current location so that it can be stored and given to Reset for Later processing

// Insertion methods
void InsertFront(const T& item); // i.e. newNode=GetNode(item); front=newNode;
void InsertRear(constT& item);
void InsertAt(const T& item); // after the node currently pointed by prevptr
void InsertAfter(const T& item); // i.e. after the node currently pointed by currPtr
//Deletion
T Deletefront(void);
void DeleteAt(void);
void DeleteAfter(void);
//Data retrieval and/or modification
T& Data(void); //return data of the current node
// e.g. L.Data()=L.Data()+8;
void clearList(void); //remove allnodes and  mark list as empty
} // end of class linked List
```

Eg. To scan and process whole list L:

```
for (L.Reset();!L.EndOfList();L.Next())
{//process current location}
```

Example: print the content of a list

```
void PrintList (const LinkedList<T> &L)
       L.Reset()
       if (L.ListEmpty())
    cout<<"EmptyList" \n";
else
      while (!L.EndOfList())
      {    cout<<L.Data()<<endl;
           L.Next();
       }
```

Exercise: Implement all methods of LinkedList