General Lists

DATA STRUCTURES

USING C++

Two Ways of Implementing General Lists

- Array-Based Lists (Using Contiguous storage) .

  an array in which the elements are physically next to one an another in adjacent memory locations

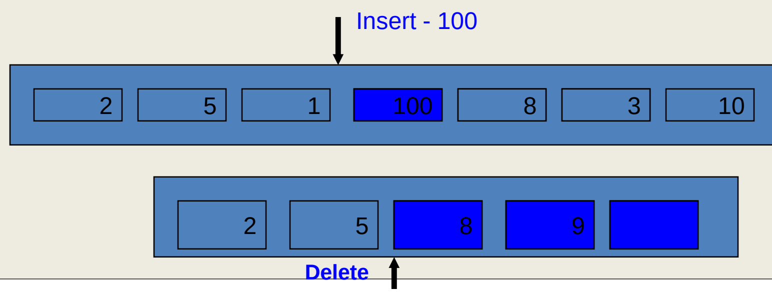
- **Linked-Based Lists** (Non Contiguous storage).

Disadvantage of Contiguous memory

Two Disadvantages,

1) Insertion in Position requires moving of elements 'DOWN' one position.

2) Deletion requires moving of elements 'UP' one position



Two Ways of Implementing General Lists

- Array-Based Lists .

  Static

- **Linked-Based Lists.**

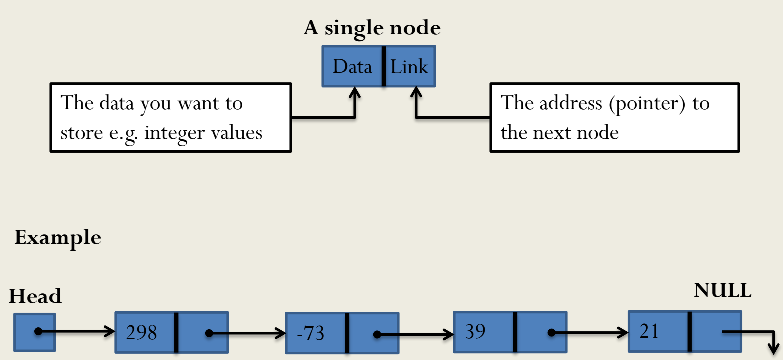
  Dynamic

Linked List Overview

- Definition
  - a collection of components, called nodes.
  - Every node (except the last node) contains some **data** along with **the address** of the next node.
- Every node in a linked list has two components:
  - one to store the relevant information (that is, **data**)
  - one to store **the address**, called **the link**, of the next node in the list.
- The address of the first node in the list is stored in a separate location, called the head or first.

How it works? A linked list uses non-contiguous memory locations and hence requires each node to remember where the next node is

Linked List in Pictures



Linked List: A list of items, called nodes, in which the order of the nodes is determined by the address, called the link, stored in each node.

Linked List Declaration

- Because each node of a linked list has two components, we need to declare each node as a *class* or *struct*
- The data type of each node depends on the specific application that is, what kind of data is being processed.
  - However, the link component of each node is always *a pointer*.
  - The data type of this pointer variable is the node type itself.

```
struct nodeType
{
      int info;
      nodeType *link;
};
```

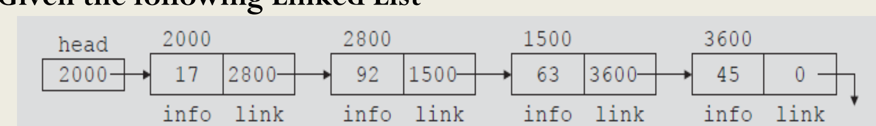
The variable declaration is as follows:

```
nodeType   *head;
```

Question: What are the differences between a class and a struct in C++?

Linked List Properties

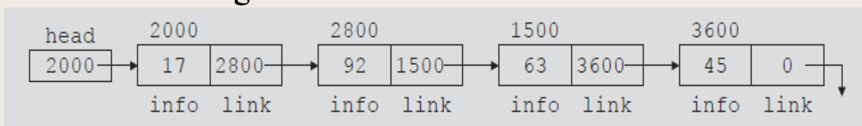
Given the following Linked List



	Value	Explanation
`head`	2000	
`head->info`	17	Because `head` is 2000 and the `info` of the node at location 2000 is 17
`head->link`	2800	
`head->link->info`	92	Because `head->link` is 2800 and the `info` of the node at location 2800 is 92

Linked List Properties

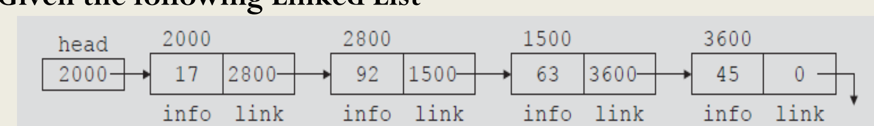
Given the following Linked List



head->link->link	?
head->link->link->info	?
head->link->link->link	?
head->link->link->link->info	?
head->link->link->link->link	?
head->link->link->link->link->info	?

Linked List Properties

Given the following Linked List



head->link->link	1500
head->link->link->info	63
head->link->link->link	3600
head->link->link->link->info	45
head->link->link->link->link	0 (NULL)
head->link->link->link->link->info	Does not exist (run time error)

Linked List Properties

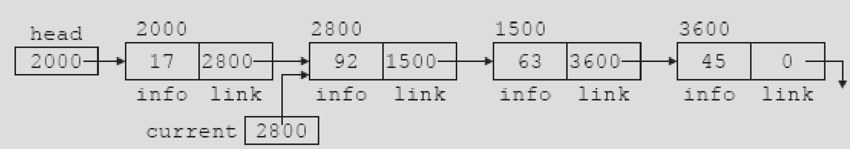
- Suppose that current is a pointer of the same type as head then

  current = head;

  Copies the address of head into current.

- Question: What is the meaning of the following statement?

  current = current-> link



Linked List Properties

- Suppose that current is a pointer of the same type as head then

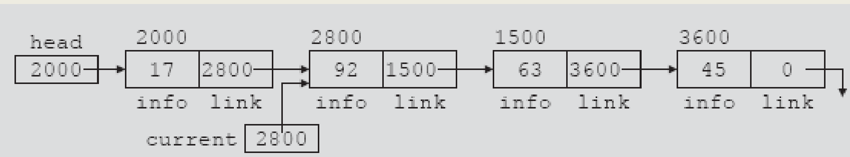
  current = head;

  Copies the address of head into current.

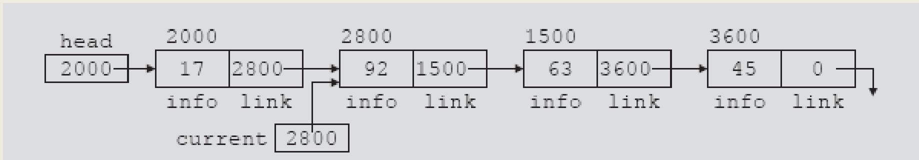
- Question: What is the meaning of the following statement?

  current = current-> link

This statement copies the value of current->link, which is 2800, into current. Therefore, after this statement executes, current points to the second node in the list.



Linked List Properties



	Value
current	2800
current->info	92
current->link	1500
current->link->info	63
head->link->link	1500
head->link->link->info	63
head->link->link->link	3600
current->link->link->link	0 (that is, NULL)
current->link->link->link->info	Does not exist (run-time error)

Basic Operations

- The basic operations of a linked list are :
- Search the list to determine whether a particular item is in the list
- Insert an item in the list
- Delete an item from the list.
- These operations require the list to be **traversed**. That is, given a pointer to the first node of the list, we must step through the nodes of the list.

Next: TRAVERSING A LINKED LIST

TRAVERSING A LINKED LIST

- **Traversing means:**
  - Given a pointer to the first node of the list, we must step through the nodes of the list.
- Suppose that the pointer ***head*** points to the first node in the list, and the link of the last node is ***NULL***
  - We cannot use the pointer ***head*** to traverse the list, **WHY?**

……………………………………………………….

WHY?

Suppose that the pointer head points to the first node in the list, and the link of the last node is NULL. We cannot use the pointer head to traverse the list because if we use the head to traverse the list, we would lose the nodes of the list. This problem occurs because the links are in only one direction. The pointer head contains the address of the first node, the first node contains the address of the second node, the second node contains the address of the third node, and so on.

If we move head to the second node, the first node is lost (unless we save a pointer to this node). If we keep advancing head to the next node, we will lose all the nodes of the list (unless we save a pointer to each node before advancing head, which is unworkable because it would require additional computer time and memory space to maintain the list).

TRAVERSING A LINKED LIST

- **Traversing means:**
  - Given a pointer to the first node of the list, we must step through the nodes of the list.
- Suppose that the pointer ***head*** points to the first node in the list, and the link of the last node is ***NULL***
  - We cannot use the pointer ***head*** to traverse the list, **WHY?**

Therefore, we always want head to point to the first node.

- It now follows that we must traverse the list using another pointer of the same type e.g. ***current***

TRAVERSING A LINKED LIST

Suppose that current is a pointer of the same type as head.

The following code traverses the list

```
current = head;
while (current != NULL)
{
      //Process current
      current = current->link;
};
```

Hint:
current = current->link;
Work as
Location++
in arrayBasedList

e.g.To dump (output, print) the contents of each node to the screen one can use the following code

```
current = head;
while (current != NULL)
{
       cout << current→info   << " ";
       current = current→link;
}
```

Item Insertion and deletion

This section discusses how to insert an item into, and delete an item from, a linked list.

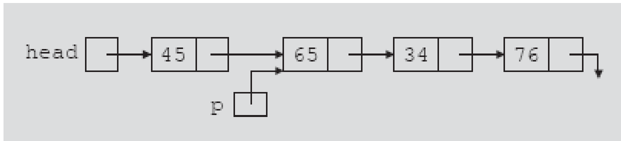
```
//Consider the following definition of a node.
struct nodeType{
   int info;
   nodeType *link;
};

//We will use the following variable declaration:
nodeType *head, *p, *q, *newNode;
```

Item Insertion

Item Insertion

Consider the linked list shown in the following Figure. and a new node with info <u>50</u> is to be created and inserted after *p*.



newNode = new nodeType; //create newNode

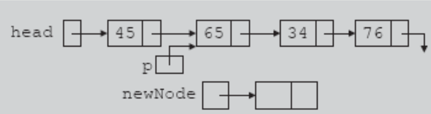
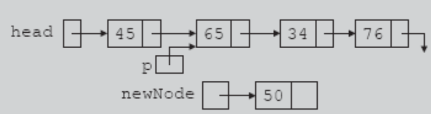
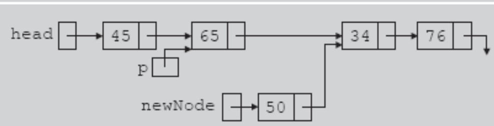
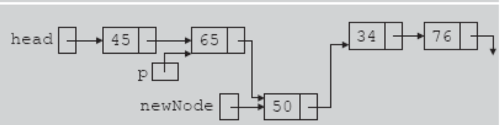
newNode->info = 50; //store 50 in the new node

newNode->link = p->link;

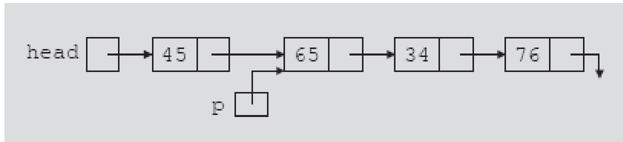
p->link = newNode;

//Note that the sequence of statements is very important

Item Insertion

Statement	Effect
`newNode = new nodeType;`	
`newNode->info = 50;`	
`newNode->link = p->link;`	
`p->link = newNode;`	

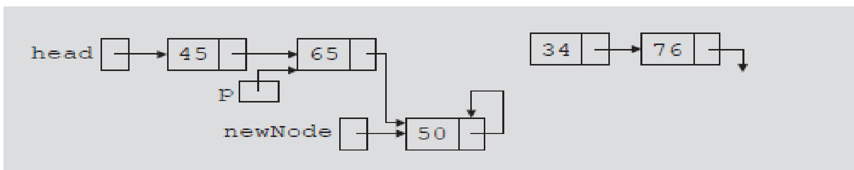
Item Insertion



// Changing the sequence of statements

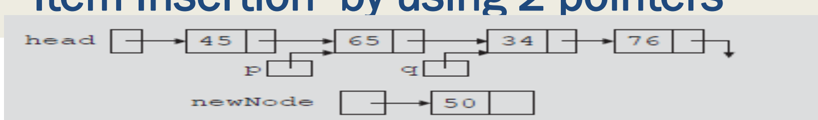
p->link = newNode;

newNode->link = p->link;



- newNode <u>points back to itself</u> and the <u>remainder of the list is lost</u>.

Item Insertion by using 2 pointers



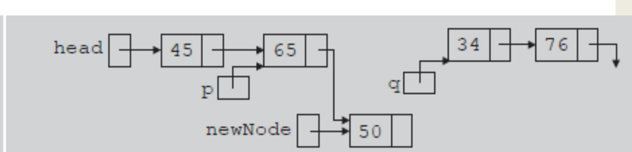
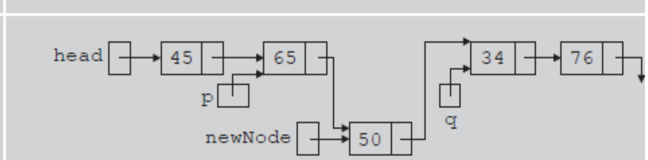
newNode->link = q;

p->link = newNode;

OR

p->link = newNode;

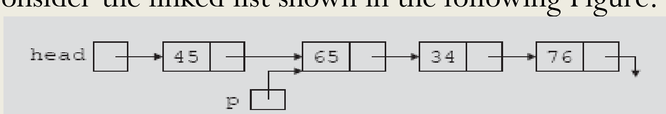
newNode->link = q;

`p->link = newNode;`	
`newNode->link = q;`	

Item Deletion

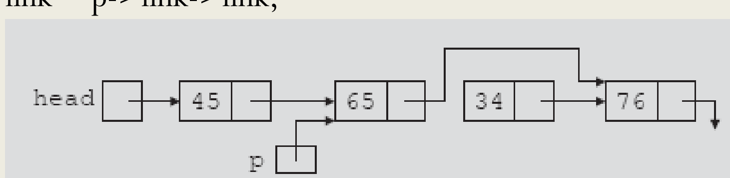
Item Deletion

- Consider the linked list shown in the following Figure:



- Suppose that the node with info 34 is to be deleted from the list. The following statement removes the node from the list:

  p->link = p->link->link;

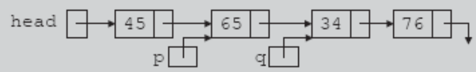
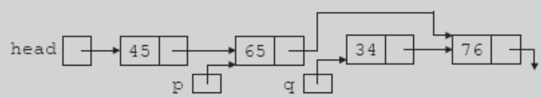
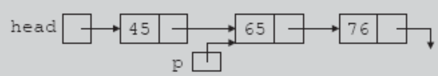


- It is clear that the node with info 34 is removed from the list. However, the memory is still occupied by this node and this memory is inaccessible.

Item Deletion

To deallocate the memory, we need a pointer to this node, e.g. **q.**

1. **q = p->link;**
2. **p->link = q->link;**
3. **delete q;**

Statement	Effect
`q = p->link;`	
`p->link = q->link;`	
`delete q;`	

Building A Linked List

Two manners

1) **Forward (**a new node is always inserted at the end of the linked list**).**

2) **Backward (**a new node is always inserted at the beginning of the list**).**

Building A Linked List: Forward

- We need three pointers to build the list:
- One to point to the first node in the list, which cannot be moved **(*first).**
- One to point to the last node in the list **(*last).**
- One to create the new node **(*newNode).**

nodeType *first, *last, *newNode;

int num; first = NULL; last = NULL;

Building A Linked List: Forward

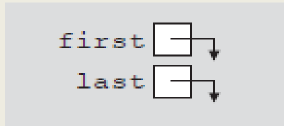
process the following data:
2 15 8 24 34

nodeType *first, *last, *newNode;

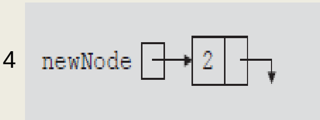
int num; first = NULL; last = NULL;

```
cin >> num;                //read and store a number in num
newNode = new nodeType;    //allocate memory of type nodeType
                           //and store the address of the
                           //allocated memory in newNode
newNode->info = num;       //copy the value of num into the
                           //info field of newNode
newNode->link = NULL;      //initialize the link field of
                           //newNode to NULL
if (first == NULL)         //if first is NULL, the list is empty;
                           //make first and last point to newNode
{
    first = newNode;
    last = newNode;
}
else                       //list is not empty
{
    last->link = newNode;  //insert newNode at the end of the list
    last = newNode;        //set last so that it points to the
                           //actual last node in the list
}
```

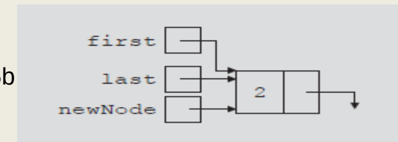
Declaration statement



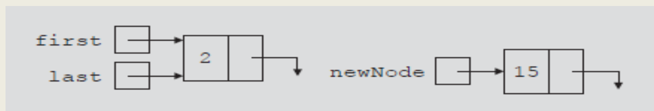
Statements 1 - 4



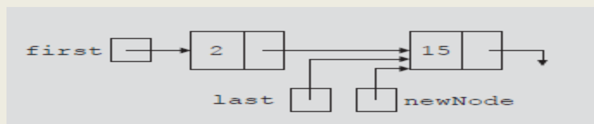
If TRUE
Statements 5, 5a , 5b



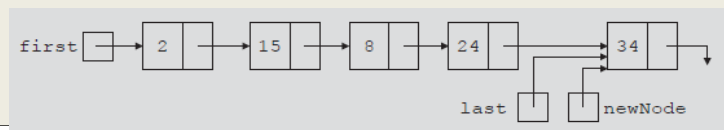
Repeat
Statements 1 - 4



If False
Else
Statements 6, 6a , 6b



Keep Repeating
Statements 1 - 6




```
nodeType* buildListForward()
{
    nodeType *first, *newNode, *last;
    int num;

    cout << "Enter a list of integers ending with -999."
         << endl;
    cin >> num;
    first = NULL;

    while (num != -999)
    {
        newNode = new nodeType;
        newNode->info = num;
        newNode->link = NULL;

        if (first == NULL)
        {
            first = newNode;
            last = newNode;
        }
        else
        {
            last->link = newNode;
            last = newNode;
        }
        cin >> num;
    } //end while

    return first;
} //end buildListForward
```

Put the previous code in while loop

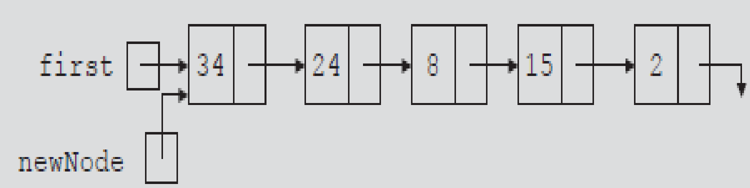
Building A Linked List: Backward

- Because the new node is always inserted at the beginning of the list, we do not need to know the end of the list, so the pointer last is not needed.
- Also, we need to update the value of the pointer first to correctly point to the first node in the list.
- we need only two pointers to build the linked list:
- One to point to the list **(*first).**
- One to create the new node **(*newNode).**

```
nodeType *first, *newNode;
int num;    first = NULL;
```

Building A Linked List: Backward

process the following data:
2 15 8 24 34



```
nodeType* buildListBackward()
{
    nodeType *first, *newNode;
    int num;

    cout << "Enter a list of integers ending with -999."
         << endl;
    cin >> num;
    first = NULL;

    while (num != -999)
    {
        newNode = new nodeType;   //create a node
        newNode->info = num;      //store the data in newNode
        newNode->link = first;    //put newNode at the beginning
                                  //of the list
        first = newNode;          //update the head pointer of
                                  //the list, that is, first
        cin >> num;               //read the next number
    }

   return first;
} //end buildListBackward
```

Linked Lists

Linked List Operations

1. Create the Linked List.
   - *The list is initialized to an empty state.*
2. Determine whether the list is empty.
3. Print the Linked List
4. Find the length of the Linked list.
5. Destroy the list.
6. Determine whether an item is the same as a given list element.
7. Retrieve the info contained in the first node
8. Retrieve the info contained in the last node
9. Search the Linked list for a given item
10. Insert an item in the linked list
11. Delete an item from the linked list
12. Make a copy of the Linked list

Linked List As A ADT: linkedListType

We should declare new user define data type using **struct** to define a **node** that consist of two component (**info** to store data from any type, ***link** to keep the address of next node).

Note: we can use class declaration to define a node (later).

In class declaration we need three private data members; two pointers to maintain a linked list:

- One to point to the first node in the list **(*first).**
- One to point to the last node in the list **(*last).**

And we need int variable (**count)** to store the number of nodes in the list.

Linked List As A ADT: linkedListType

//Definition of the node

template <class Type>

struct nodeType

{

Type info;

nodeType<Type> *link;

};

Member Variables of the class **linkedListType**

private:

int count; //variable to store the number of elements in the list

nodeType<Type> *first; //pointer to the first node of the list

nodeType<Type> *last; //pointer to the last node of the list

```
template <class Type>
struct nodeType
{
    Type info;
    nodeType<Type> *link;
};
```

```
template<class Type>
class linkedListType
{
private:
    nodeType<Type> *first; //pointer to the first node of the list
    nodeType<Type> *last;  //pointer to the last node of the list
    int count; //variable to store the number of elements in the list
```

```
public:
    void initializeList();
      /* ... */
    bool isEmptyList();
      /* ... */
    bool isFullList();
      /* ... */
    void print();
      /* ... */
    int length();
      //Return the number of elements in the list
    void destroyList();
      /* ... */
    void retrieveFirst(Type& firstElement);
      /* ... */
    void retrieveLast(Type& lastElement); /* ... */
    void search(const Type& searchItem);
      /* ... */
    void insertFirst(const Type& newItem);
      /* ... */
    void insertLast(const Type& newItem);
      /* ... */
    void deleteNode(const Type& deleteItem);
      /* ... */
    linkedListType();
      /* ... */

     ~linkedListType();
```

Linked List Operations

1- Default Constructor

It simply initializes the list to an empty state. Recall that when an object of the linkedListType type is declared and no value is passed

```
template<class Type>
linkedListType<Type>::linkedListType() // default constructor
{
    first = NULL;
    last = NULL;
    count = 0;
}
```

What is the time complexity to create a linked list?
O(1)

2- Destroy a Linked List

The function destroyList deallocates the memory occupied by each node.

We traverse the list starting from the first node and deallocate the memory by calling the operator delete.

Traversing the linked list need a temporary pointer to move from node to next one and deallocate the memory. Hint: we need while loop for traversing.

Once the entire list is destroyed, we must set the pointers first and last to NULL and count to 0.

2- Destroy a Linked List

```
template<class Type>
void linkedListType<Type>::destroyList()
{
    nodeType<Type> *temp;   //pointer to deallocate the memory
                            //occupied by the node
    while(first != NULL)    //while there are nodes in the list
    {
       temp = first;        //set temp to the current node
       first = first->link; //advance first to the next node
       delete temp;         //deallocate memory occupied by temp
    }
    last = NULL;    //initialize last to NULL; first has already
                   //been set to NULL by the while loop
    count = 0;
}
```

What is the time complexity to destroy a linked list? O(n)

3- Initialize a Linked list to an empty state

The function initializeList reinitializes the list to an empty state. This task can be accomplished by using the destroyList operation

```
template<class Type>
void linkedListType<Type>::initializeList()
{
    destroyList(); //if the list has any nodes, delete them
}
```

What is the time complexity to initialize a linked list?

4- isEmptyList Operation

```
template<class Type>
bool linkedListType<Type>::isEmptyList()
{
    return(first == NULL);
}
```

- If first is null, this will guarantee that the linked list is empty

5- isFullList Operation

?

5- isFullList Operation

```
template<class Type>
bool linkedListType<Type>::isFullList()
{
    return false;
}
```

6- Print the Linked List

```
template<class Type>
void linkedListType<Type>::print()
{
    nodeType<Type> *current; //pointer to traverse the list

    current = first;    //set current so that it points to
                        //the first node
    while(current != NULL) //while more data to print
    {
       cout<<current->info<<" ";
       current = current->link;
    }
}//end print
```

What is the time complexity to print a linked list? O(n)

7- Length of a Linked List

Returns how many nodes are in the linked list so far

```
template<class Type>
int linkedListType<Type>::length()
{
    int count = 0;
    nodeType<Type> *current; //pointer to traverse the list

    current = first;

    while (current!= NULL)
      {
       count++;
       current = current->link;
    }

    return count;
}  // end length
```

What is the time complexity to get the length a linked list? O(n)

8- Retrieve the Data of the First Node

```
template<class Type>
void linkedListType<Type>::retrieveFirst(Type& firstElement)
{
    assert(first != NULL);
    firstElement = first->info; //copy the info of the first node
}//end retrieveFirst
```

assert terminates the program if the condition inside the parenthesis is wrong.
Note: add header file #include <assert.h>

What is the time complexity of this function? O(1)

9- Retrieve the Data of the Last Node

```
template<class Type>
void linkedListType<Type>::retrieveLast(Type& lastElement)
{
    assert(last != NULL);
    lastElement = last->info; //copy the info of the last node
}//end retrieveLast
```

What is the time complexity of this function? O(1)

10- Destructor

```
template<class Type>
linkedListType<Type>::~linkedListType() // destructor
{
    nodeType<Type> *temp;

    while(first != NULL)  //while there are nodes left in the list
    {
        temp = first;          //set temp point to the current node
        first = first->link;   //advance first to the next node
        delete temp;           //deallocate memory occupied by temp
    }//end while

    last = NULL; //initialize last to NULL; first is already null
}//end destructor
```

```
template <class Type>
linkedListType<Type>::~linkedListType() //destructor
{
    destroyList();
}
```

What is the time complexity of this function? O(n)

```
int main()
{
    linkedListType<int> list1, list2;
    int num;
    cout<<" Enter numbers ending with -999" <<endl;
    cin>>num;

    while(num != -999)
    {
        list1.insertLast(num);
        cin>>num;
    }

    cout<<endl;
    cout<<" List 1: ";
    list1.print();
    cout<<endl;
    cout<<" Length List 1: "<<list1.length()
        <<endl;

    int retfirst;    // Add by J
    list1.retrieveFirst(retfirst);  // Add by J
    cout<< "list1.retrieveFirst; retfirst = " << retfirst<<endl; // Add by J

    int retlast;    // Add by J
    list1.retrieveLast(retlast);  // Add by J
    cout<< "list1.retrieveLast; retlast = " << retlast<<endl; // Add by J
```

Two Types of Linked Lists

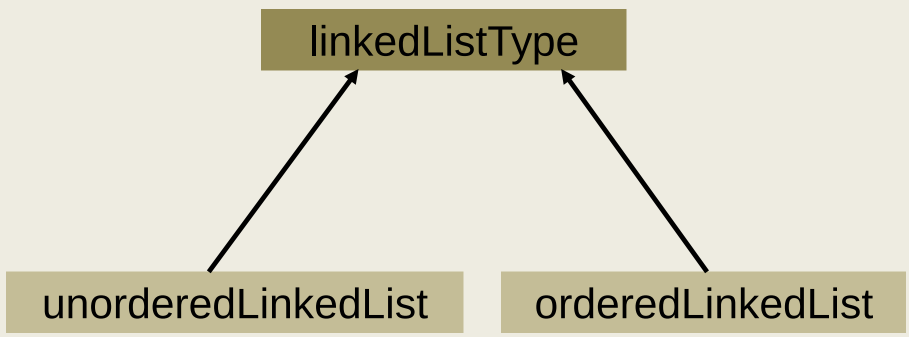
In general, there are two types of linked lists:-

- Sorted lists (ordered), whose elements are arranged according to some criteria. e.g. ascending order.
- Unsorted lists (unordered), whose elements are in no particular order.
- The algorithms to implement the operations **search**, **insert**, and **remove** slightly differ for sorted and unsorted lists.
- Therefore, the definition of class linkedListType is implement the basic operations on a linked list as an abstract class. (the common operations between ordered & unordered linked lists).

Inheritance

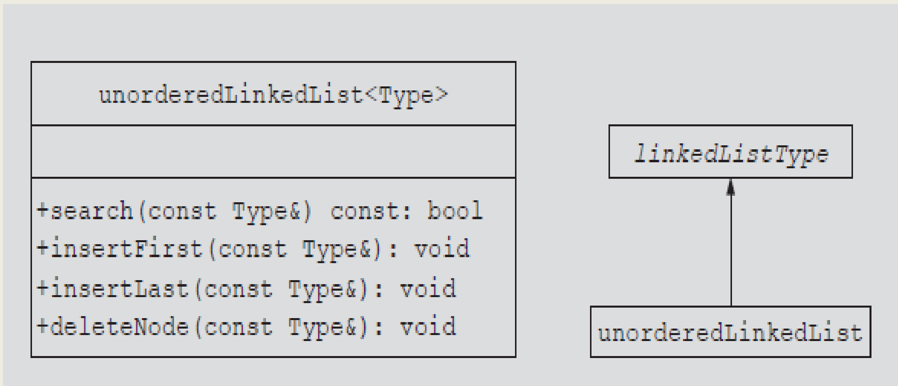
Using the principal of inheritance, we derive two classes:

- unorderedLinkedList and orderedLinkedList from the class linkedListType.



Unordered Linked Lists

Unordered Linked List



This class is created by inheriting class linkedListType.

Search Unordered Linked List

- The member function **search** searches the list for a given item. If the item is found, it returns true; otherwise, it returns false.

- Because a linked list is not a random access data structure, we must sequentially search the list starting from the first node.

- We use ***current*** pointer and ***while loop*** to traverse the linked list.

Search Unordered Linked List

This function has the following steps:

- Compare the search item with the current node in the list. If the info of the current node is the same as the search item, stop the search; otherwise, make the next node the current node.
- Repeat Step 1 until either the item is found or no more data is left in the list to compare with the search item.

Search Unordered Linked List as A void function

```
template<class Type>
void linkedListType<Type>::search(const Type& item)
{
    nodeType<Type> *current; //pointer to traverse the list
    bool found;

    if(first == NULL)  //list is empty
        cout<<"Cannot search an empty list. "<<endl;
    else
    {
        current = first;  //set current pointing to the first
                          //node in the list

        found = false;    //set found to false

        while(!found && current != NULL) //search the list
            if(current->info == item)      //item is found
                found = true;
            else
                current = current->link; //make current point to
                                         //the next node

        if(found)
            cout<<"Item is found in the list."<<endl;
        else
            cout<<"Item is not in the list."<<endl;
    } //end else
}//end search
```

What is the complexity of this function? O(n)

Search Unordered Linked List as A Boolean Function

```
template <class Type>
bool unorderedLinkedList<Type>::
                    search(const Type& searchItem) const
{
    nodeType<Type> *current; //pointer to traverse the list
    bool found = false;

    current = first; //set current to point to the first
                     //node in the list

    while (current != NULL && !found)    //search the list
        if (current->info == searchItem) //searchItem is found
            found = true;

        else
            current = current->link; //make current point to
                                     //the next node
    return found;
}//end search
```

What is the complexity of this function? O(n)

Insert in an Unordered Linked List insertFirst (Backward)

- The function **insertFirst** inserts the new item at the beginning of the list that is, before the node pointed to by first.
- The steps needed to implement this function are as follows:
  1. Create a new node.
  2. If unable to create the node, terminate the program.
  3. Store the new item in the new node.
  4. Insert the node before first.
  5. Increment count by 1.

Insert in an Unordered Linked List insertFirst

```
template<class Type>
void linkedListType<Type>::insertFirst(const Type& newItem)
{
    nodeType<Type> *newNode;        //pointer to create the new node

    newNode = new nodeType<Type>;   //create the new node
    newNode->info = newItem;        //store the new item in the node
    newNode->link = first;          //insert newNode before first
    first = newNode;                //make first point to the
                                    //actual first node

    if(last == NULL)    //if the list was empty, newNode is also
                        //the last node in the list
        last = newNode;
}
```

What is the complexity of this function? O(1)

Insert in an Unordered Linked List insertLast (Forward)

- The function **insertLast** inserts the new item at the end of the list that is, after the node pointed to by last.

- The steps needed to implement this function are as follows:
  1. Create a new node.
  2. Store the new item in the new node.
  3. Insert the node after first (if first == NULL).
  4. Else; Insert the node after last.
  5. Increment count by 1.

Insert in an Unordered Linked List insertLast

```
template<class Type>
void linkedListType<Type>::insertLast(const Type& newItem)
{
    nodeType<Type> *newNode; //pointer to create the new node

    newNode = new nodeType<Type>; //create the new node
    newNode->info = newItem;      //store the new item in the node
    newNode->link = NULL;         //set the link field of new node
                                  //to NULL

    if(first == NULL)  //if the list is empty, newNode is
                //both the first and last node
    {
        first = newNode;
        last = newNode;
    }
    else     //if the list is not empty, insert newNnode after last
    {
        last->link = newNode; //insert newNode after last
        last = newNode; //make last point to the actual last node
    }
}//end insertLast
```

What is the complexity of this function? O(1)

Delete from an Unordered Linked List

- We need to consider the following cases:

1. The list is empty.
2. The list is nonempty and the node to be deleted is the first node.
3. The list is nonempty and the node to be deleted is not the first node, it is somewhere in the list.
4. The node to be deleted is not in the list.

Delete from an Unordered Linked List

To deallocate the memory of deleted node we need two pointers.

1. ***current .This pointer used to traverse the linked list**
2. ***trailCurrent (temp). This pointer just before current**

Delete from an Unordered Linked List

```
template<class Type>
void linkedListType<Type>::deleteNode(const Type& deleteItem)
{
    nodeType<Type> *current; //pointer to traverse the list
    nodeType<Type> *trailCurrent; //pointer just before current
    bool found;

    if(first == NULL)    //Case 1; list is empty.
        cout<<"Can not delete from an empty list.\n";
    else
    {
        if(first->info == deleteItem) //Case 2
        {
            current = first;
            first = first ->link;
            if(first == NULL)    //list had only one node
                last = NULL;
            delete current;
        }
        else  //search the list for the node with the given info
        {
            found = false;
            trailCurrent = first;   //set trailCurrent to point to
                                    //the first node
            current = first->link;  //set current to point to the
                                    //second node
```

Delete from an Unordered Linked List

```
            while((!found) && (current != NULL))
            {
                if(current->info != deleteItem)
                {
                    trailCurrent = current;
                    current = current-> link;
                }
                else
                    found = true;
            } // end while

            if(found) //Case 3; if found, delete the node
            {
                trailCurrent->link = current->link;

                if(last == current)          //node to be deleted was
                                             //the last node
                    last = trailCurrent;     //update the value of last
                delete current;              //delete the node from the list
            }
            else
                cout<<"Item to be deleted is not in the list."<<endl;
        } //end else
    } //end else
} //end deleteNode
```

Delete from an Unordered Linked List

What is the complexity of this function?
O(n)

Ordered Linked Lists

Ordered Linked Lists

- The elements of an ordered linked list are arranged using some ordering criteria. e.g. ascending order.

Search in ordered Linked List

This function has the following steps:

- Compare the search item with the current node in the list. If the info of the current node is greater than or equal to the search item; otherwise, make the next node the current node.

- Repeat Step 1 until either an item in the list that is greater than or equal to the search item is found, or no more data is left in the list to compare with the search item.

Search in ordered Linked List as a Void

```
template<class Type>
void orderedLinkedListType<Type>::search(const Type& item)
{
    bool found;
    nodeType<Type> *current; //pointer to traverse the list

    found = false;       //initialize found to false
    current = first;     //start the search at the first node

    if(first == NULL)
       cout<<"Cannot search an empty list."<<endl;
    else
    {
        while(current != NULL && !found)
            if(current->info >= item)
                found = true;
            else
                current = current->link;

        if(current == NULL)          //item is not in the list
            cout<<"Item is not in the list"<<endl;
        else
            if(current->info == item) //test for equality
                cout<<"Item is found in the list"<<endl;
            else
                cout<<"Item is not in the list"<<endl;
    }//end else
}//end search
```

Search in ordered Linked List as a Boolean

```
template <class Type>
bool orderedLinkedList<Type>::search(const Type& searchItem) const
{
    bool found = false;
    nodeType<Type> *current; //pointer to traverse the list

    current = first; //start the search at the first node

    while (current != NULL && !found)

        if (current->info >= searchItem)

            found = true;

        else

            current = current->link;

    if (found)

        found = (current->info == searchItem); //test for equality

    return found;
}//end search
```

Search in ordered Linked List

What is the complexity of this function?
O(n)

Insert a Node in Ordered Linked List

- Here we use two pointers, current and trailCurrent,to **search** the list. *trailCurrent points to the node just before current*.

- Case 1: The list is initially **empty**. The node containing the new item added as the first node in the list.

  // Start searching for proper place

- Case 2: **The new item is smaller than the smallest item in the list.** The new item goes at the beginning of the list.

Insert a Node in Ordered Linked List

- Case 3: **The item is to be inserted somewhere in the list.**

  1) Case 3a: **The new item is larger than all the items in the list.** In this case, the new item is inserted at the end of the list.

  2) Case 3b: **The new item is to be inserted somewhere in the middle of the list.** In this case, the new item is inserted between trailCurrent and current.

Insert a Node in Ordered Linked List

- Here we use two pointers, current and trailCurrent,to search the list. *trailCurrent points to the node just before current.*

```
template<class Type>
void orderedLinkedListType<Type>::insertNode(const Type& newitem)
{
    nodeType<Type> *current;        //pointer to traverse the list
    nodeType<Type> *trailCurrent;   //pointer just before current
    nodeType<Type> *newNode;        //pointer to create a node

    bool  found;

    newNode = new nodeType<Type>;   //create the node
    newNode->info = newitem;        //store newitem in the node
    newNode->link = NULL;           //set the link field of the node
                                    //to NULL
```

```
    if(first == NULL)   //Case 1
    {    first = newNode;
         last = newNode;
         /* count++;*/    }
    else
    {
         current = first;
         found = false;

         while(current != NULL && !found)  //search the list
              if(current->info >= newitem)
                   found = true;
              else
              {
                   trailCurrent = current;
                   current = current->link;
              }
         if(current == first)     //Case 2
           {  newNode->link = first;
              first = newNode;   // count++;
           }
         else                     //Case 3
           {
              trailCurrent->link = newNode;   //Case 3b
              newNode->link = current;

              if (current == NULL)  //Case 3a
                   last = newNode;
              //count++;
           }
    }//end else
}//end insertNode
```

Insert a Node in Ordered Linked List

What is the complexity of this function?
O(n)

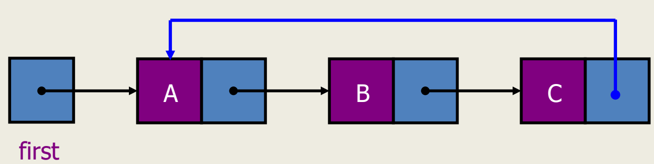
Delete from an Ordered Linked List

- To delete a given item from an ordered linked list, first we search the list to see whether the item to be deleted is in the list.
- The function to implement this operation is the same as the delete operation on general linked lists.
- Because the list is sorted, we can somewhat improve the algorithm for ordered linked lists.

Circular linked lists

Variations of Linked Lists

- *Circular linked lists*
  - The last node points to the first node of the list



  - How do we know when we have finished traversing the list?
    (Tip: check if the pointer of the current node is equal to the first.)

Circular linked lists

Advantages:

- **Traverse** the entire linked list **from any given node**. (When we revisit the given node, we know we have traversed the entire list).
- A circular linked list is often **used as a buffer** where one portion of the program produces data and another consumes it, such as in communications.
- It **saves time** when we have to go to the first node from the last node.

Circular linked lists

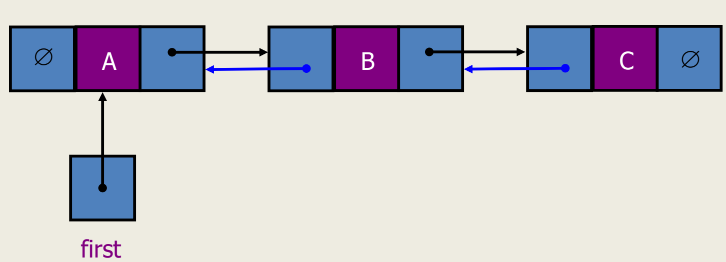
Disadvantages:

- It is not easy to **reverse** the linked list.
- If proper care is not taken, then an **infinite looping** can be caused while traversing it.

Doubly linked lists

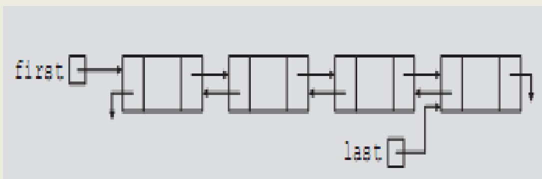
Variations of Linked Lists

- ***Doubly linked lists***
  - Each node points to not only successor but the predecessor
  - There are two `NULL:` at the first and last nodes in the list
  - Advantage: given a node, it is easy to visit its predecessor. Convenient to traverse lists backwards



Doubly Linked Lists

- A doubly linked list is a linked list in which every node has a next pointer and a back pointer.
- Every node contains the address of the next node (except the lastnode), and every node contains the address of the previous node (except the first node).




```
template <class Type>
struct nodeType
{
    Type info;
    nodeType<Type> *next;
    nodeType<Type> *back;
};
```

- A doubly linked list can be traversed in either direction.

Array versus Linked Lists

- Linked lists are more complex to code and manage than arrays, but they have some distinct advantages.
  - **Dynamic**: a linked list can easily grow and shrink in size.
    - We don't need to know how many nodes will be in the list. They are created in memory as needed.
    - In contrast, the size of a C++ array is fixed at compilation time.
  - **Easy and fast insertions and deletions**
    - To insert or delete an element in an array, we need to copy to temporary variables to make room (space) for new elements or close the gap caused by deleted elements.
    - With a linked list, no need to move other nodes. Only need to reset some pointers.