Introduction to the Linked List ADT

- A linked list is a series of connected nodes, where each node is a data structure.
- A linked list can grow or shrink in size as the program runs.

![Linked List Diagram]

list head

NULL
Introduction to Linked List ADT

- References may be address or array indices
- Data structures can be added to or removed from the linked list during execution

Advantages of Linked Lists Over Arrays and Vectors

- A linked list can easily grow or shrink in size
- Insertion and deletion of nodes is quicker with linked lists than with vectors
The Composition of a Linked List

- Each node in a linked list contains one or more members that represent data

Empty List

- If a list currently contains no nodes, it is said to be an empty list
- In this case, the list head points to NULL
The Composition of a Linked List

- A linked list is called "linked" because each node in the series has a pointer that points to the next node in the list.
- The linked list has a pointer, **List Head**, that points to the first node.
- The last node points to **NULL**.

Declarations

- We will look at a linked list class.
  - This is different from the way the book does this.
```cpp
template < class T >
class LinkedList
{
  private:
    class ListNode
    {
      public:
        T value; // Node value
        ListNode *next; // Next Node
        ListNode ( T v, ListNode *n = NULL) : value( v ), next( n ) {} 
      }; // ListNode
      ListNode *head;
    public:
      LinkedList ( ListNode *ptr = NULL ) { head = ptr; }
      ~LinkedList ( void );
      void appendNode ( T );
      void insertNode ( T );
      void deleteNode ( T );
      void displayList ( void );
}; // LinkedList
```

---

**Declarations**

- The next step is to declare a pointer to serve as the list head, as shown below:

  ```
  ListNode *head = NULL;
  ```

- Once you have declared a node data structure and have created a NULL head pointer, you have an empty linked list
- The next step is to implement operations with the list
**NULL Pointer**

- The null pointer is used to indicate end-of-list
- It should always be tested before using a pointer
  
  ```
  ListNode *p;
  for ( ... ; p != NULL; ... ) ...
  ```
- It can also be tested itself
  
  ```
  for ( ; !p; ) ... // same as above
  ```

---

**Linked List Operations**

- **Basic operations:**
  - Append a node to the end of the list
  - Insert a node into the list
  - Traverse the list
  - Delete a node from the list
  - Delete/destroy the list
Create a New Node

- Allocate memory for the new node:
  ```
  newNode = new ListNode;
  ```

- Initialize the contents of the node:
  ```
  newNode->value = num;
  ```

Appending a Node

- Add a node to the end of the list
- Basic process:
  - Create a new node (as already described)
  - Add the node to the end of the list:
    - If the list is empty, set head pointer to this node
    - Else,
      - Traverse the list to the end
      - Set the pointer of the last node to point to the new node
Appending a Node

New node created, end of list located

New node added to end of list
Inserting a Node Into a Linked List

- Used to maintain a linked list in order
- Requires two pointers to traverse the list:
  - A pointer to locate the node with a data value greater than that of the node to be inserted
  - A pointer to ‘trail behind’ one node, to point to the node before the point of insertion
- A new node is inserted between the nodes pointed at by these pointers

New node created, correct position located
Inserting a Node Into a Linked List

New node inserted in order in the linked list

Traversing a Linked List

- Visit each node in a linked list and display the data in each node
- Basic process:
  - Set a pointer to the contents of the head pointer
  - While the pointer is not NULL
    - Process the data
    - Go to the next node by setting the pointer to the pointer field of the current node in the list
  - End while
Traversing a Linked List

- `nodePtr` points to the node containing 5, then the node containing 13, then the node containing 19, then points to `NULL`, and the list traversal stops.

Deleting a Node

- Used to remove a node from a linked list
- If the list uses dynamic memory, then delete the node from memory
- Requires two pointers:
  - One to locate the node to be deleted
  - One to point to the node before the node to be deleted
Deleting a Node

Locating the node containing 13

Adjusting pointer around the node to be deleted
Deleting a Node

Linked list after deleting the node containing 13

Destroying a Linked List

- Must remove all nodes used in the list
- To do this, use the list traversal to visit each node
- For each node:
  - Unlink the node from the list
  - If the list uses dynamic memory, then free the node’s memory
- Set the list head to NULL
ListNode.h – Program

```cpp
#ifndef LISTNODE_H
#define LISTNODE_H

class ListNode
{
    public:
    double value;  // Node value
    ListNode *next;  // Next Node

    ListNode ( int v, ListNode *n = NULL ) : value( v ), next( n ) {}  
}; // ListNode

#endif
```

LinkedList.h

```cpp
#ifndef LINKEDLIST_H
#define LINKEDLIST_H
#include <iostream>
using namespace std;
template < class T >
class LinkedList
{
    private:
    class ListNode
    {
        public:
        T value;  // Node value
        ListNode *next;  // Next Node

        ListNode ( T v, ListNode *n = NULL ) : value( v ), next( n ) {}  
    }; // ListNode
    ListNode *head;

    public:
    LinkedList ( ListNode *ptr = NULL ) { head = ptr; }
    ~LinkedList ( void );
    void appendNode ( T );
    void insertNode ( T );
    void deleteNode ( T );
    void displayList ( void );
}; // LinkedList
#endif
```
displayList – Program

template < class T >
void LinkedList<T>::displayList( void )
    for ( ListNode *nodePtr = head; nodePtr; nodePtr = nodePtr->next )
        cout << nodePtr->value << endl;
} // LinkedList::displayList

appendNode

template < class T >
void LinkedList<T>::appendNode( T num )
{ ListNode *newNode = new ListNode( num ), *nodePtr;
    if ( !head )
        head = newNode;
    else
    { for ( nodePtr = head; nodePtr->next; nodePtr = nodePtr->next );
        nodePtr->next = newNode;
    } // else
} // LinkedList::appendNode
**insertNode**

```cpp
template < class T >
void LinkedList<T>::insertNode( T num )
{
    ListNode *newNode = new ListNode( num );
    ListNode *nodePtr, *previousNode = NULL;
    if ( !head )
        head = newNode;
    else
    { for ( nodePtr = head; nodePtr && nodePtr->value < num;
        previousNode = nodePtr, nodePtr = nodePtr->next );
        if ( previousNode )
            previousNode->next = newNode;
        else
            head = newNode;
    newNode->next = nodePtr;
} // else
} // LinkedList::insertNode
```

**deleteNode**

```cpp
template < class T >
void LinkedList<T>::deleteNode ( T num )
{
    ListNode *nodePtr, *previousNode = NULL;
    if ( !head )
        return;
    if ( head->value == num )
    { nodePtr = head->next;
        delete head;
        head = nodePtr;
    } // if
    else
    { for ( nodePtr = head; nodePtr && nodePtr->value != num;
        previousNode = nodePtr, nodePtr = nodePtr->next );
        previousNode->next = nodePtr->next;
        delete nodePtr;
    } // else
} // LinkedList::deleteNode
```
~LinkedList

template < class T >
LinkedList<T>::~LinkedList( void )
{ ListNode *nextNode;
  for ( ListNode *nodePtr = head; nodePtr; nodePtr = nextNode )
  { nextNode = nodePtr->next;
    delete nodePtr;
  } // for
} // LinkedList::~LinkedList
#endif

Stepping Through the Program

- In the declaration, head pointer is initialized to NULL, that indicates the list is empty
- The first call to appendNode passes 2.5 as the argument.
  - In the following statements, a new node is allocated in memory
  - 2.5 is copied into its value member
appendNode( 2.5 );

ListNode *newNode = new Node ( num );

The next statement to execute is the following if statement

if ( !head )
    head = newNode;

There are no more statements to execute, so control returns to function main
appendNode( 7.9 );

In the second call to appendNode, 7.9 is passed as the argument. Once again, the first two statements in the function create a new node and store the argument in the node's value member.

Since head no longer points to NULL, the else part of the if statement executes:

```c
else
{
    for ( nodePtr = head; nodePtr->next;
         nodePtr = nodePtr->next );
    nodePtr->next = newNode;
} // else
```
appendNode( 7.9 );

nodePtr is already at the end of the list, so the for loop immediately terminates. The last statement, nodePtr->next = newNode; causes nodePtr->next to point to the new node. This inserts newNode at the end of the list.

appendNode( 12.6 );

The third time appendNode is called, 12.6 is passed as the argument. Once again, the first two statements create a node with the argument stored in the value member.
Next, the else part of the if statement executes. As before, nodePtr is made to point to the same node as head

Since nodePtr->next is not NULL, the for loop executes. After its first iteration, nodePtr points to the second node in the list
appendNode(12.6);

The for loop's conditional test fails after the first iteration because nodePtr->next now points to NULL. The last statement, nodePtr->next = newNode; causes nodePtr->next to point to the new node. This inserts newNode at the end of the list.

The figure above depicts the final state of the linked list.

Understanding insertNode

- Let's examine how insertNode puts something in the linked list
- Assume that we are putting 10.5 in the list
**insertNode( 10.5 );**

In `insertNode`, a new node is created. Since the list already has nodes stored in it, the else part of the if statement executes. It begins by assigning `nodePtr` to `head`.

```
insertNode( 10.5 );
```

Since `nodePtr` is not `NULL` and `nodePtr->value` is less than `num`, the for loop iterates. During the iteration, `previousNode` is made to point to the node that `nodePtr` is pointing to. `nodePtr` is then advanced to point to the next node.

```
insertNode( 10.5 );
```

Since `nodePtr` is not `NULL` and `nodePtr->value` is less than `num`, the for loop iterates. During the iteration, `previousNode` is made to point to the node that `nodePtr` is pointing to. `nodePtr` is then advanced to point to the next node.
insertNode(10.5);

Once again, the loop performs its test. Since `nodePtr` is not NULL and `nodePtr->value` is less than `num`, the loop iterates a second time. During the second iteration, both `previousNode` and `nodePtr` are advanced by one node in the list.

This time, the loop’s test fails because `nodePtr` is not less than `num`. The statements after the loop executes, which causes `previousNode->next` to point to `newNode`, and `newNode->next` to point to `nodePtr`.

If you follow the links, from the head pointer to the end of the list, you see that the nodes are stored in the order of their value members.
Understanding deleteNode

- Deleting a node from a linked list requires two steps:
  - Remove the node from the list without breaking the links created by the next pointers
  - Deleting the node from memory

look at the else part of the second if statement. This is where the function performs its action since the list is not empty, and the first node does not contain the value 7.9. Just like insertNode, this function uses nodePtr and previousNode to traverse the list. The for loop terminates when the value 7.9 is located. At this point, the list and the other pointers are in the state depicted in the figure below
deleteNode(7.9);

Next, the statement `previousNode->next = nodePtr->next;` executes. The statement above causes the links in the list to bypass the node that `nodePtr` points to. Although the node still exists in memory, this removes it from the list.

The last statement uses the delete operator to complete the total deletion of the node.

---

**A Linked List Template**

- When declaring a linked list, you must specify the type of data to be held in each node.
- Using templates, you can declare a linked list that can hold a data type determined at the list definition time.
### LinkedList.h – Program

```cpp
#ifndef LINKEDLIST_H
#define LINKEDLIST_H
#include <iostream>
using namespace std;

// LinkedList class

template <class T>
class LinkedList
{
  private:
    ListNode *head;
  public:
    LinkedList(); { head = NULL; }
    ~LinkedList();
    void appendNode(T);
    void insertNode(T);
    void deleteNode(T);
    void displayList(T);
}; // LinkedList

#endif
```

### LinkedList.h – Program

- The rest of the code is similar to what we’ve seen before
- You can look at it in the book
Problems for Linked Lists

- Count the number of nodes in a linked list
- Are the nodes in order?
- Insert an element after some node
- Number of occurrences of an element
- Are there adjacent duplicates?

Variations of the Linked List

- Doubly linked list:
Variations of the Linked List

- Circular linked list:

The STL list Container

- The list container, found in the STL, is a template version of a doubly linked list
- STL lists can insert elements, or add elements to their front quicker than vectors can, because lists do not have to shift the other elements
- Lists are also efficient at adding elements at their back because they have a built-in pointer to the last element in the list (no traversal required)
### STL list Member Functions

#### Member Function Examples & Description

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>back</code></td>
<td><code>cout &lt;&lt; list.back() &lt;&lt; endl;</code>&lt;br&gt;The back member function returns a reference to the last element in the list.</td>
</tr>
<tr>
<td><code>erase</code></td>
<td><code>list.erase(iter);</code>&lt;br&gt;<code>list.erase(firstIter, lastIter)</code>&lt;br&gt;The first example causes the list element pointed to by the iterator <code>iter</code> to be removed. The second example causes all of the list elements from <code>firstIter</code> to <code>lastIter</code> to be removed.</td>
</tr>
<tr>
<td><code>empty</code></td>
<td><code>if ( list.empty() )</code>&lt;br&gt;The empty member function returns true if the list is empty. If the list has elements, it returns false.</td>
</tr>
<tr>
<td><code>end</code></td>
<td><code>iter = list.end();</code>&lt;br&gt;end returns a bi-directional iterator to the end of the list.</td>
</tr>
<tr>
<td><code>front</code></td>
<td><code>cout &lt;&lt; list.front() &lt;&lt; endl;</code>&lt;br&gt;front returns a reference to the first element of the list.</td>
</tr>
<tr>
<td><code>insert</code></td>
<td><code>list.insert(iter, x)</code>&lt;br&gt;The insert member function inserts an element into the list. The example shown above inserts an element with the value <code>x</code>, just before the element pointed to by <code>iter</code>.</td>
</tr>
<tr>
<td><code>merge</code></td>
<td><code>list1.merge(list2);</code>&lt;br&gt;merge inserts all the items in <code>list2</code> into <code>list1</code>. <code>list1</code> is expanded to accommodate the new elements plus any elements already stored in <code>list1</code>. <code>merge</code> expects both lists to be sorted. When <code>list2</code> is inserted into <code>list1</code>, the elements are inserted into their correct position, so the resulting list is also sorted.</td>
</tr>
</tbody>
</table>
## STL list Member Functions

**Member Function Examples & Description**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pop_back</td>
<td>List.pop_back(); pop_back removes the last element of the list.</td>
</tr>
<tr>
<td>pop_front</td>
<td>List.pop_front(); pop_front removes the first element of the list.</td>
</tr>
<tr>
<td>push_back</td>
<td>List.push_back(x); push_back inserts an element with value x at the end of the list.</td>
</tr>
<tr>
<td>push_front</td>
<td>List.push_front(x); push_front inserts an element with value x at the beginning of the list.</td>
</tr>
<tr>
<td>reverse</td>
<td>List.reverse(); reverse reverses the order in which the elements appear in the list.</td>
</tr>
</tbody>
</table>

---

## STL list Member Functions

**Member Function Examples & Description**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size()</td>
<td>Returns the number of elements in the list.</td>
</tr>
</tbody>
</table>
| swap     | List1.swap(List2)
The swap member function swaps the elements stored in two lists. For example, assuming list1 and list2 are lists, the statement shown above will exchange the values in the two. |
| unique   | List.unique(); unique removes any element that has the same value as the element before it. |
Using STL list Container – Program

```cpp
#include <iostream>
#include <list>
using namespace std;

int main ( void )
{   list<int> myList;
    list<int>::iterator iter;
    for ( int x = 0; x < 100; x += 10 )
        myList.push_back( x );
    for ( iter = myList.begin(); iter != myList.end(); iter++ )
        cout << *iter << " ";
    cout << endl;
    myList.reverse();
    for ( iter = myList.begin(); iter != myList.end(); iter++ )
        cout << *iter << " ";
    cout << endl;
    return 0;
} // main
```

Using STL list Container – Output

```
0 10 20 30 40 50 60 70 80 90
90 80 70 60 50 40 30 20 10 0
```