Lab 5.1 Linked Lists

Project 5.1 More Linked Lists

Project 5.2 Application of Linked Lists: Hash Tables

NOTES TO THE INSTRUCTOR

The purpose of this lab exercise and projects is to introduce linked lists and construct a pointer-based `LinkedList` class. The lab exercise and the first project parallel closely the presentation of linked lists in Sections 6.4 and 6.5 of the text *ADTs, Data Structure, and Problem Solving with C++*, 2E. The second project introduces hash tables as described in Section 12.7.

The lab exercise develops a simple `LinkedList` class containing only the following basic operations:

- Constructor
- Size
- Insert
- Output

The first project adds four other important operations to the `LinkedList` class:

- Delete
- Destructor
- Copy constructor
- Assignment operator

This part should be assigned if a complete `LinkedList` class is to be built.

The second project is a an introduction to hash tables. The problem is to build a hash table class, using an array (or `vector`) of linked lists to store the elements. It uses the *chaining* method of resolving collisions as described in Section 12.7 of the text.

Notes:

1. Lab Exercise 5.1 and Project 5.1 use the file `LinkedList.h` containing the `LinkedList` class and the program `linktester.cpp` to test linked-list operations. (These files can be downloaded from the website whose URL is given in the preface.)

2. Project 5.2 uses an array (or `vector`) of linked lists to implement a hash table. You should tell the students if you have a preference for `vectors` over arrays.

3. Project 5.2 makes a good extra-credit project to be done independently.

Course Info: ________________ *Name:* Chris Dang

Lab 5.1 Linked Lists

Introduction

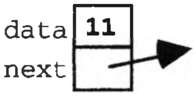
Arrays (either static or dynamic) provide a way to store and process lists. Usually, list elements are stored in consecutive array locations. Inserting a value into such a list, however, can involve extensive copying of values to make room for the new item. Deleting a value from such a list can also involve extensive copying of values to close the gap left by the removed item. (See Section 6.1 of the text *ADTs, Data Structures, and Problem Solving with C++*, 2E for a description of the inefficiency of these list operations in such an array-based implementation.)

Chapter 6 of the text describes an alternative way to implement lists that permits insertions and deletions without this expensive copying associated with the *sequential-storage implementation* using arrays—**linked lists**. This linked-list approach is the subject of this lab exercise.

The basic idea is to build a list of structures, each consisting of

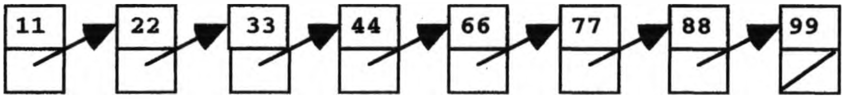
- a *data* member, in which a data value is stored; and
- a *next* member, in which the address of the same kind of structure can be stored.

These structures that contain both a value and a pointer are called **nodes**. We might visualize a node as follows:



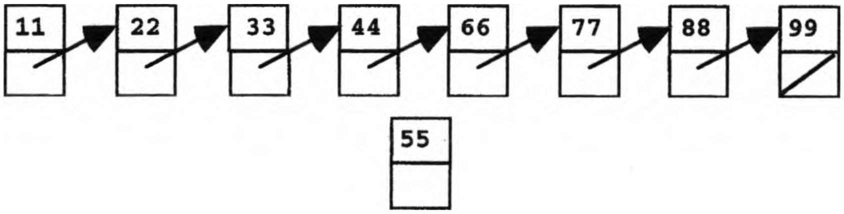
The *next* members in the nodes make it possible to link nodes together to form a list. This is accomplished by storing in the next member of a node the address of a node that contains the next list value.

This method of linking values together solves the efficiency problems of the array-based implementation of list when inserting elements into or removing elements from locations other than the end of the list. To illustrate, suppose that the list of values 11, 22, 33, 44, 66, 77, 88, 99 is stored as follows:

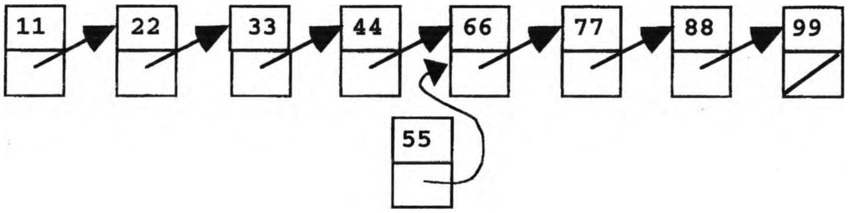


Then, to insert 55 after 44 we perform the following steps:

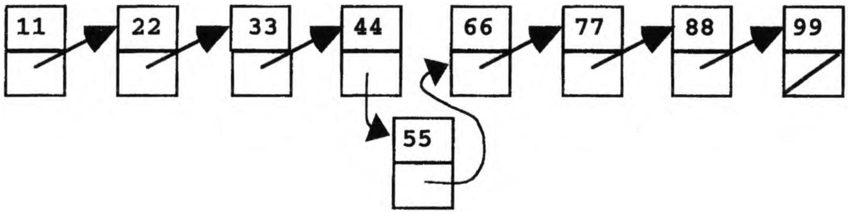
1. Allocate a new node to hold the new value:



2. Store the address of its successor node (the one containing 66) in its next member.

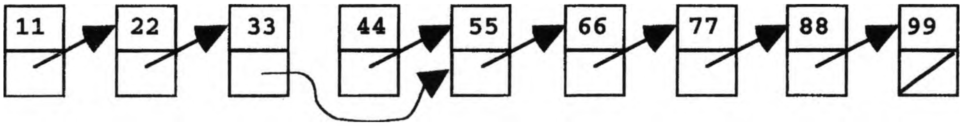


3. Store its address in the next member of the predecessor (the node containing 44).

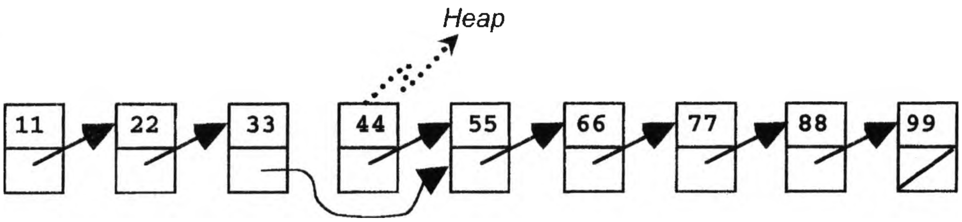


Notice that **no data values have been moved!** This is also true for removing elements. For example, suppose we want to remove the node containing 44 from this new list:

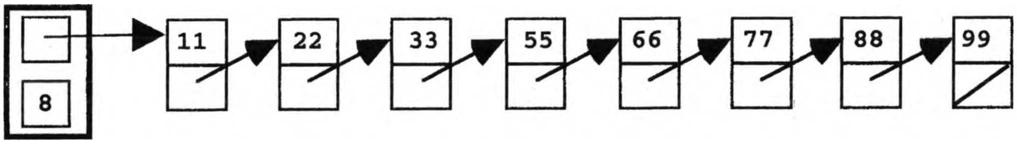
1. Perform a bypass from its predecessor (the node containing 33) by changing the address in the next member of the predecessor to that of the successor of the node to be deleted.



2. Return the removed node to the heap (free store).



Given the ability to construct linked nodes, we can then construct a linked structure called a **simply-linked list**, which stores the address of the first node in the list, and perhaps the number of nodes (i.e., the number of list elements it stores):



This is one form of linked list, and simplicity is its primary advantage. Other forms include **circular-linked lists** and **doubly-linked lists** described in Section 11.1 of the text.

The Linked-List Lab Exercise

In this lab exercise you will be using the two files `LinkedList.h` and `linktester.cpp`. You should get copies of these files using the procedure specified by your instructor. (They can be downloaded from the website whose URL is given in the preface.)

Design Issues

Since a linked list is made up of nodes, we have two different objects to consider: `LinkedList` objects and `Node` objects. However, if we implement `Node` and `LinkedList` as two separate classes, then the `LinkedList` operations will not be able to access the private members of the class `Node`.

To circumvent this difficulty, one of the easiest strategies is to declare `Node` as a *class* (or struct) *inside the class* `LinkedList`. We will use a "public class" rather than a struct because we will be adding a function member, and structs are commonly used only for C-style structures, which have no function members.

```
class LinkedList
{
  ...
  private:
    class Node
    {
      public:
        // members of class Node ...
    };
  ...
  // members of class LinkedList ...
};
```

This makes it possible for members (and friends) of `LinkedList` to access `Node` and its members because it is a part of class `LinkedList`. However, other classes, functions, and programs cannot access `Node` or its members because it is declared in a private section of class `LinkedList`. This is as it should be, because a `Node` is really an implementation detail for linked lists.

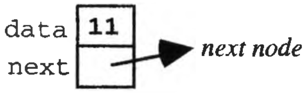
We will begin with the class `Node`.

The Data Members of Class **Node**

Based on the preceding discussion, we can see that the class `Node` needs two data members:

- `data`, whose type is the type of the values being stored in the list; and
- `next`, whose type is a pointer to a `Node`

In the context of the previous example, we can picture the `Node` object as:



[1] The first decision to make is what to use for the type of the data values. To make it possible to use the class with different types of elements, we can use a `typedef` statement of the form

```
typedef SpecificType ElementType
```

to make `ElementType` a synonym for *`SpecificType`* and use `ElementType` throughout the `LinkedList` and `Node` declarations and their implementations. Place a `typedef` statement before the class declaration that makes `ElementType` a synonym for `int` (unless your instructor prefers a different approach such as that in the note on the next page).

Check here when finished X

To change to a linked list of doubles or some other type, one need only replace the specific type (int) in this statement with the new type.

*Note: A better alternative is to use the **template mechanism** described in Chapter 9 of the text. It makes it possible to pass the type of the elements into the class from a declaration of an object of this class type, which means that no editing of the class's header file is needed to change element types. If your instructor prefers, you might study templates and use them instead of the* `typedef` *mechanism.*

[2] Next, inside the stub of class `Node`, declare data members:

- `data`, whose type is `ElementType`
- `next`, whose type is a pointer to a `Node`

Check here when finished __X__

Note that a `typedef` declaration has been added following the declaration of class `Node` in the file `LinkedList.h`:

```
typedef Node * NodePointer;
```

This defines the name `NodePointer` as a more readable alias for the type `Node *` so that we can declare variables of type `NodePointer` within the function members of class `LinkedList`. Note that the alias `NodePointer` cannot be used to declare the `next` data member in class `Node` because the declaration of `next` precedes the definition of `NodePointer`.

The Data Members of `LinkedList`

As described earlier, our simple implementation of a linked list has two data members, one to keep track of the first node in the list and the other to keep a count of the nodes.

[3] Add declarations of these two data members to class `LinkedList` (but outside class `Node`'s declaration).

`first` — whose type is a pointer to a `Node`
`mySize` — an integer

Check here when finished X___

Operations on a `LinkedList`

A linked list consists of a sequence of linked nodes. Because of this, some of the `LinkedList` operations will require implementing `Node` operations. It is difficult, however, to anticipate in advance all of the `Node` operations that will be needed, so we will add `Node` operations when they are needed for some `LinkedList` operation.

There are many `LinkedList` operations that could be put in the class. We will only implement a few of the basic ones in this lab exercise, enough to gain some familiarity with linked structures. (The `list` class template in STL provides a full slate of list operations.) For now, the operations that we will consider are:

- A class constructor, so that empty `LinkedList` objects can be declared and initialized
- A `size()` operation to retrieve the number of nodes in the list
- An `insert(index, dataValue)` operation to insert `dataValue` into a `LinkedList` at a position `index`
- An output operation, so that the contents of a `LinkedList` can be displayed

In Project 5.1, four other operations will be added:

- An `erase(index)` operation to remove the node at position `index`
- A destructor, so that the run-time storage in `LinkedList` objects can be reclaimed
- A copy constructor, so that `LinkedList` objects can be copied when needed
- Assignment, so that one `LinkedList` object can be copied and assigned to another

The Class Constructor for a `LinkedList`

This is the simplest of the operations to implement. The problem is specified as follows:

Precondition: A `LinkedList` object must be constructed.
Postcondition: Its data members have been initialized appropriately for an empty list.

To accomplish this, our constructor must:

1. Set the data member `first` of class `LinkedList` to the null pointer (0)
2. Set the `mySize` member to 0

🖥 [4] a. Put the prototype of the constructor inside the `LinkedList` class declaration at the designated place.

b. Unless your instructor prefers a different approach (such as that noted earlier), start building an implementation file `LinkedList.cpp` and put the defnition of the constructor there.

Note: Because the constructor is very simple, it really should be ***inlined*** *and your instructor may prefer this. One way to do this is to put the definition after the end of the class declaration and precede the heading of the function with the keyword* `inline`. *See the description of the purpose of inlining in Section 4.3 (p. 156) of the text.*

c. Test what you have done so far by compiling, linking, and executing `linktester.cpp`.

Check here when it compiles and executes correctly X

Finding the Size of a `LinkedList`

🖥 [5] The `size()` operation is simply an accessor operation that retrieves the value stored in the `mySize` data member.

What kind of function member should this be to ensure that it doesn't change this data member?
It needs to be a constant function.

Write its prototype below:
int size() ;

a. Put the prototype and documentation of the `size()` function inside the `LinkedList` class declaration at the designated place.

b. Put the definition of `size()` in the appropriate place. (See the comments in step 4.)

c. To test the size operator, comment out with `//` (or remove) the lines containing the comment delimiters (`/*` and */) in the section labeled `PART 1` in `linktester.cpp` and then compile, link, and execute the program.

What value does it give for the size of `intList`? 0

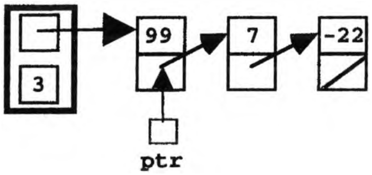
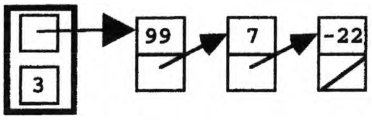
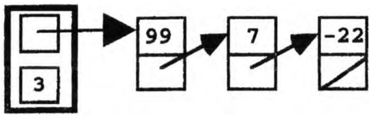
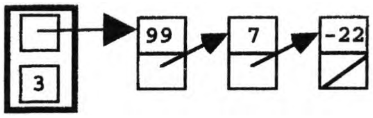
Displaying a **`LinkedList`**

To output a linked list, we perform a *traversal* of the list, visiting each node in the list exactly once, beginning with the first node, displaying the value in the `data` member of that node, and then moving to the next node. When the end of the list is reached, we stop. These observations lead to the following algorithm:

```
1. Set a NodePointer named ptr to point to the first node in the list.
2. While ptr  is not null:
     a. Display the data value of the Node pointed to by ptr.
     b. Make ptr point to the next Node in the list.
   End while.
```

Note that because we have used a pretest loop, this algorithm will also work correctly for an empty list.

✍ 6.1 Fill in the following table, showing the output produced and the position of *`ptr`* on each pass through the loop:

Position of `ptr`	Output
	out: 99 ptr at first
	out: 7 ptr at second
	out: -22 ptr at third
	ptr at null

Step 2a of the algorithm requires accessing the data member of a class object through a pointer. For this, we could use an expression of the form `(*ptr).`*`dataMember`* to dereference *`ptr`* and then access *`dataMember`*. (The parentheses are needed because the dot operator has higher priority than the * operator—see Table C.2 in Appendix C of the text.) However, because this notation is clumsy and such accesses are needed so frequently, C++ provides an equivalent but more readable expression:

```
ptr->dataMember
```

Think of `->` as an arrow from a pointer variable to a data member in the object to which it points. This **arrow operator** is the standard way to access data members via pointers.

This means that if `ptr` contains the address of a node in a linked list, we can use

```
ptr->data
```

to access the list item stored in the `data` part of the node pointed to by `ptr` as required in step 2a of the algorithm. Similarly, we can use the assignment statement

```
ptr = ptr->next;
```

to make `ptr` point to the next node in the linked list. This statement changes the address in `ptr` to that of the node that follows the one to which it was pointing.

6.2 Using these ideas, overload the output operator << for class `LinkedList` in the same manner as in Lab 4.1.

a. Put the prototype and documentation of the `display()` function inside the `LinkedList` class declaration at the designated place.

b. Put the definition of `display()` at the appropriate place. (See the comments in Step 4).)

Note: You will need a pointer of type `NodePointer` to traverse the linked list. Here it is important to remember that:

> To use a type (or constant) name defined inside a class declaration outside the class declaration, the type (or constant) name must be qualified with the name of the class and the scoping operator (`::`):
>
> *`ClassName::TypeName`*

Thus, to use the type `NodePointer` outside of class `LinkedList`, use its fully-qualified name `LinkedList::NodePointer`.

c. To test `display()`, comment out with `//` (or remove) the lines containing the comment delimiters (`/*` and `*/`) in the section labeled `PART 2A` in `linktester.cpp` and then compile, link, and execute the program.

d. Now add a prototype and documentation for a nonmember function `operator<<()` in `LinkedList.h` *after the end of the class declaration.*

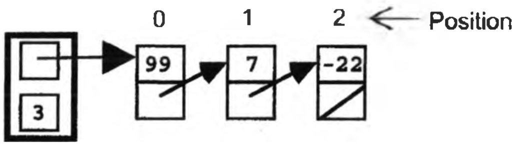
e. Put the definition of `operator<<()` at the appropriate place. (See the comments in step 4.)

f. To test your output operator, comment out with `//` (or remove) the lines containing the comment delimiters (`/*` and `*/`) in the section labeled `PART 2B` in `linktester.cpp` and then compile, link, and execute the program.

Check here when it compiles and executes correctly __X__

Inserting a Value into a **`LinkedList`**

When inserting a data value into a linked list, we need to specify the insertion point. So how do we go about doing this? Here we will simply number each value in the list and then specify the position at which we want the value to be inserted. We will use the same numbering convention as used for C++ arrays: the first value in the list will be numbered 0, the second value in the list numbered 1, and so on.



Assuming this convention, we can specify the problem in the following way:

Preconditions: A data value *dataValue* is to be inserted into this `LinkedList` object at a specified position *index*, which is an integer with 0 ≤ *index*

Postcondition: *dataValue* has been stored in node numbered *index*; and `mySize` has been incremented by 1, provided that *index* satisfied the precondition; otherwise, an error message was displayed and the `LinkedList` object was unchanged

7.1 Using this specification, add a prototype for `insert()` at the designated place in the declaration of class `LinkedList`, but don't write a definition for `insert()` yet, because there is another issue to address. The first parameter should be a integer `index`, which specifies the position where a value is to be inserted, and the second parameter, `dataValue`, is that value.

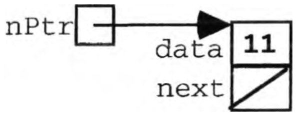
Check here when finished X

A `Node` *Constructor*

Inserting a data value requires, first of all, that it be stored in a `Node`. Thus, `insert()` must allocate a `Node` (using the `new` operation) and then put the data value in the `data` member of this node. It would be convenient if we could simply write

```
nPtr = new Node(dataValue);
```

to obtain a new `Node` pointed to by `nPtr`, where a `Node` constructor initializes the `data` member of the `Node` to `dataValue` and the `next` member to the null pointer. For example, if `dataValue` is 11, this statement should produce the result shown at the right.



7.2. a. Put the prototype for the `Node` constructor inside the `Node` class declaration at the designated place.

b. For convenience, you might put the definition of the `Node` constructor below the `Node` class declaration.

(Note: Your instructor may prefer that you inline this definition below the `LinkedList` class declaration.)

c. To test what you have written, start a definition of `insert()`at an appropriate place—see the comments in step 4—by putting in its body a declaration of `nPtr` and the assignment statement

```
nPtr = new(nothrow) Node(dataValue);
```

(or use it as an initializer in the declaration statement). Be sure to `#include <new>`. Before proceeding to the next part to complete the definition of `insert()`, it is probably a good idea to make sure that everything compiles correctly.

Check here when finished ______

Back to the Insertion Operation for `LinkedList`

Now that we can construct a `Node` in which a data value is stored, we are ready to actually insert this new node into the appropriate place in the linked list.

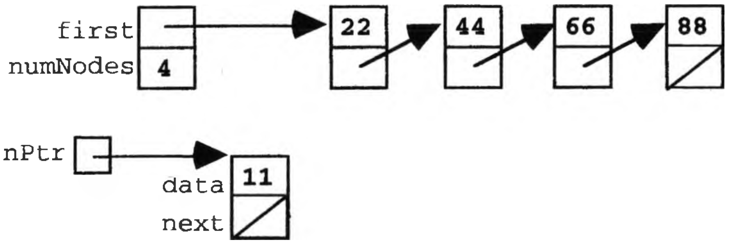
<u>*Note*</u>: ***Visualization*** *is an important part of programming linked lists. In developing a linked-list operation, it is always helpful to draw pictures to determine what must be done and to test the algorithm for the operation. You must also consider the various cases for the operation, especially the* ***boundary conditions****—i.e., at the beginning and at the end of the list.*

As described in Section 6.4 of the text, there are two cases to consider.

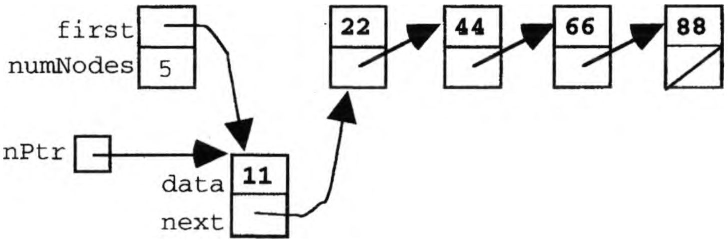
Case 1*:** A new value is to be inserted at the ***beginning of the list (i.e., `index = 0`).

For example, suppose we want to insert 11 at the beginning of a linked list containing 22, 44, 66, and 88.

We can picture the situation as follows:



We need to change this so that the new node's next pointer points at the node which contains 22 and `first` points to the new node:

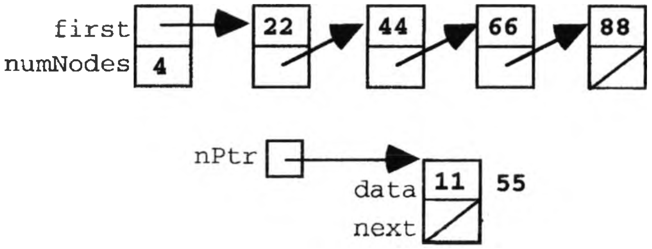


We can do this the following three operations (<u>but be sure to do the first two in the order shown</u>):

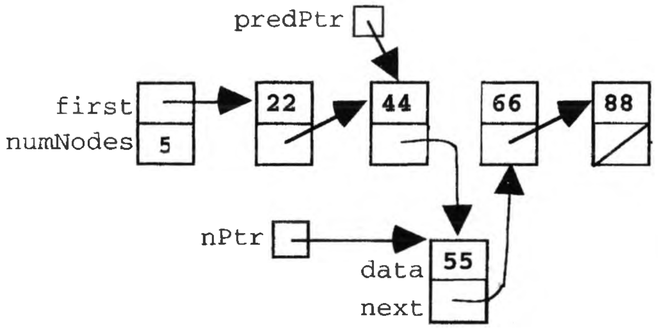
1. Set `nPtr->next` to `first`. // `first` points to the successor of the new node
2. Set `first = nPtr`. // Now set `first` to point to the new first node
3. Increment `mySize`. // Note that if we reverse the order of 1 and 2 we will lose
   // access to all nodes except the new one.

***Case 2*:** *dataValue* is to be inserted somewhere within the list.

To illustrate, suppose you wish to insert 55 in the linked list containing 22, 44, 66, and 88 between 44 and 66 (i.e., `index = 2`). The situation can be pictured as:



We want to change the linked list to



Follow this rather closely. It is always dangerous to change a pointer, since we may lose track of the thing that it was originally pointing to. So we have to be careful.

1. First traverse the list to position a pointer named `predPtr` at the node containing the predecessor (44) of the new data value; that is, `predPtr` is positioned at the node numbered `index` – 1.
2. Set `nPtr->next` to `predPtr->next`.
3. Set `predPtr->next` to `nPtr`.
4. Increment `mySize`.

7.3

~~7.4~~ Check that these steps will also work correctly when the new node is being inserted at the end of the list by drawing similar "before and after" pictures below for inserting 99 at the end of the linked list containing 22, 44, 66, and 88 (i.e., `index` = 4).

We can handle both cases for insertion with the following algorithm:

```
1. Construct a new Node pointed to by nPtr that contains the data value to be inserted.
2. If index is zero:
   // Insert at the beginning of the list
      a. Set nPtr->next to first.
      b. Set first to nPtr.
   Else
      // First position a pointer predPtr at the predecessor of the new node.
      a. Declare another pointer predPtr and initialize it to first.
      b. For each integer i in the range 1 through index – 1:
            Set predPtr to predPtr->next.
         End for.
      //  Now connect the new node into the list.
      c. Set nPtr->next to predPtr->next.
      d. Set predPtr->next to nPtr.
3. Increment mySize.
```

7.4

🖳 [redacted] Using this algorithm, complete the definition of `insert()` for the class `LinkedList`. Note that you've already done step 1. To test your definition, comment out with `//` (or remove) the lines containing the comment delimiters (`/*` and `*/`) in the section labeled `PART 3` in `linktester.cpp` and then compile and execute the program.

When it compiles and executes correctly, you are finished. Hand in the items listed in the grade sheet on the next page.

Course Info: _____________ ***Name:*** ____

Lab 5.1 Grade Sheet

Hand in:

1. This lab handout with the answers filled in.
2. Attach printouts showing:
    a. The complete `LinkedList` class
    b. A demonstration that the driver program `linktester.cpp` compiles and links okay
    c. At least one execution trace of `linktester.cpp`

Category	Points Possible	Points Received
Lab Exercise Answers	10	____________
Driver program: Sample run—adequate testing	20	____________
`LinkedList` class	(110)	
Correctness of declarations	20	____________
Correctness of functions	50	____________
Structure/efficiency of functions	20	____________
Documentation	10	____________
Style/readability	10	____________
Total	**140**	____________

Project 5.1 More Linked Lists

Completing the **`LinkedList`** *Class*

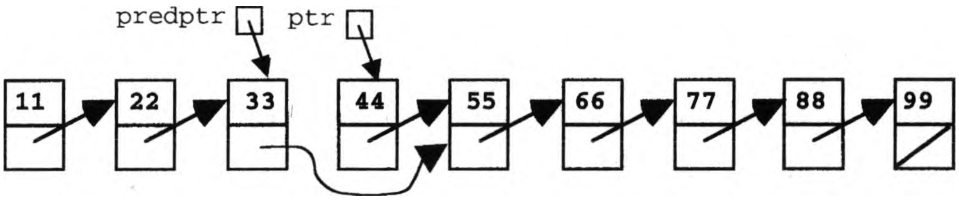
You will continue to flesh out the `LinkedList` class started in Lab 5.1. As noted there, you will add the following operations:

- An erase operation to remove the node at a specified position
- A destructor to prevent memory leaks by deallocating dynamically allocated memory
- A copy constructor to make distinct copies of `LinkedList` objects
- An assignment operator to assign a distinct copy of a `LinkedList` object to another `LinkedList` object

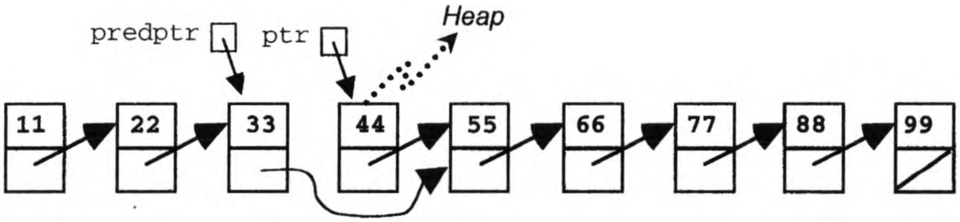
You will continue the development of `LinkedList` from the lab exercise, and you will also be using `linktester.cpp` to check that your class is working properly.

The **`erase()`** *Operation*

We saw in the lab exercise that we can remove a node from a linked list by performing a bypass from its predecessor,



and then returning the removed node to the heap (free store):



As with the insert operation, removing the first node in the list requires a special case.

1.1 In the space indicated on the grade sheet for this project, write an algorithm for the remove operation, similar to that given in the lab exercise for insert, that handles both cases—removal at the beginning of the list (position 0) and removal at some other position in the list.

1.2 a. Now add a prototype for `erase()` at the specified place in the `LinkedList` class declaration. It should have one parameter, namely, the position of the node to be deleted.

b. Then, using the algorithm you developed, define `erase()`.

c. To test your function, comment out with `//` (or remove) the lines containing the comment delimiters (`/*` and `*/`) in the section labeled `PART 4` in `linktester.cpp` and compile, link, and execute the program.

The **LinkedList** *Destructor*

Because all nodes in a linked list are dynamically allocated, the LinkedList destructor must deallocate each node in the linked list. A straightforward way to do this is to simply traverse the list, deallocating nodes as we go:

1. Declare a NodePointer pointer ptr and set it to point to the first node in the list.
2. Traverse the list and for each node do the following:
   a. Set first to the next node after that pointed to by ptr (i.e., first = ptr ->next).
   b. Use delete to deallocate the node pointed to by ptr (i.e., delete ptr).
   c. Set ptr equal to first.

[2] a. Add a prototype and documentation for the destructor at an appropriate place.

b. Then using the preceding algorithm, define the LinkedList destructor at ~~the designated place in LinkedList.cpp~~.

c. To test it, add the following temporary statement at the end of the destructor (after the while loop)

```
if (first == 0) cout << "List destroyed\n";
else cout << "List not destroyed\n";
```

and then comment out with // (or remove) the lines containing the comment delimiters (/* and */) in the section labeled PART 5 in linktester.cpp and compile, link, and execute the program.

You should see two "List destroyed" displays. Make sure you understand why this is.

The **LinkedList** *Copy Constructor*

The same basic approach that you used in the output operation in Lab 5.1 and in the destructor can be used to write the copy constructor. Traverse the list, copying the nodes as we go.

```
// To make a copy of LinkedList origList
1. Set mySize to origList.mySize.
2. If origList is empty:
   Set first to the null pointer.
3. Otherwise
   a. Declare pointer origPtr, which will run through origList, and lastPtr, which will keep track of the
      last node in the copy.
   b. Set origPtr to point to the first node in origList.
   c. Construct a node containing the data in the node pointed to by origPtr and have lastPtr point to it.
   d. Set first equal to lastPtr.
   e. Now let origPtr run through origList, starting with the next node, and for each node:
      i. Construct a node containing a copy of the data in the node pointed to by origPtr and set
         lastPtr->next to point to it.
      ii. Change lastPtr to lastPtr->next so it points to the last node in this copy of the list.
```

[3] a. Add a prototype and documentation for the copy constructor in the designated place in the LinkedList class declaration. Don't forget to make the LinkedList being copied a const reference parameter.

b. Then, using the preceding algorithm, define the LinkedList copy constructor at an appropriate place. You might want to make some diagrams to figure out what the algorithm is doing so that you can approach your coding with confidence.

c. To test your new copy constructor, comment out with // (or remove) the lines containing the comment delimiters (/* and */) in the two sections labeled PART 6 in linktester.cpp and compile, link, and execute the program.

The **LinkedList** *Assignment Operator*

From earlier classes we have built, we know that overloading the assignment operator is very much like the copy constructor. The differences are as follows: For an assignment expression of the form *var = origList*:

- Assignment must check first for self-assignment (i.e, of the form *x* = *x*). We have seen that we can check this by checking whether `this` (the address of *var*) is equal to the address of *origList* (&*origList*). In this case, nothing needs to be done.
- If not self-assignment, it should first destroy the value of *var* so there is no memory leak. This can be done by invoking its destructor with an expression of the form `this->~`*`ClassName`*`()`.
- It then makes a copy of *origList* in the same manner as with the copy constructor.
- It returns a const reference to the list that was assigned a value; that is, the return type of

  ```
  operator=()
  ```

  is `const LinkedList &` and it returns `*this`.

[4] a. Add a prototype and documentation for the assignment operator in the designated place in the `LinkedList` class declaration in `LinkedList.h`.

b. Then define the `LinkedList` assignment operator at an appropriate place.

c. To test your assignment operator, see the section labeled `PART 7` in `linktester.cpp`.

Hand in the items listed on the grade sheet.

Course Info: ______________ ***Name:*** ____

Project 5.1 Grade Sheet

Hand in:

1. In the space below, write your algorithm for the erase operation or attach a printed copy of your algorithm.

Erase function:

Case 1: Erase at beginning of list

```
1 set ptr to first
2 set first to first -> next
3 delete nodeptr -> next
4 decrement mySize
```

Case 2: Erase node that is not first in list

```
1 traverse list with ptr ( followed by predPtr) until index is found
2 set predPtr to ptr -> next
3 delete ptr -> next
4 decrement mySize
```

2. A listing of the prototypes, documentation, and definitions of the new `LinkedList` operations (the source code for the entire `LinkedList` class if you like)

3. A sample run of `linktester.cpp`

Attach this grade sheet to your printouts.

Category	Points Possible	Points Received
Correctness of operations (including following instructions)	60	______________
Algorithm for the erase operation	10	______________
Design and structure of the functions	10	______________
Documentation (including specifications of functions)	10	______________
Style (white space, alignment, etc.)	10	______________
Total	**100**	______________

Project 5.2 Application of Linked Lists: Hash Tables (see Section 12.7)

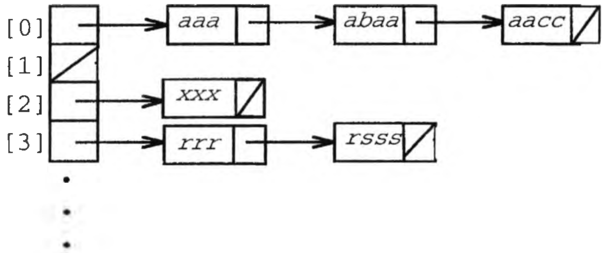
Note: In this project, you will be using an array (or `vector`) of linked lists to implement a hash table. To process the linked lists, you will use your `LinkedList` from Lab Exercise 5.1 and Project 5.1 (or your instructor may have you use STL's `list`.)

A *hash table* is a data structure in which the location of an item is determined directly as a function of the item rather than by a sequence of trial-and-error comparisons; it is used when fast searching is needed. Ideally, search time is O(1); i.e., it is constant—independent of the number of items to be searched. Section 12.7 of the text describes some of the ways that hash tables can be implemented. Here we will use the method known as *chaining*, in which the hash table is implemented using an array (or `vector`) of linked lists.

When an item is to be inserted into the table, a *hashing function h* is applied to the item to determine where it is to be placed in the table; for example, a common one is

$$h(item) = item \ \% \ table\text{-}size$$

where the table size is usually taken to be a prime number in order to scatter ("hash") the items throughout the table. For nonnumeric items, numeric codes—e.g., ASCII—are used. The linked list at location *h*(*item*) is searched for the item. If it is not found, the item is inserted into the linked list at this location.



A. Design a `HashTable` class for processing hash tables. You may use a small prime number such as 11 or 13 for the table size unless your instructor specifies otherwise. Your class should provide at least the following operations:

1. Constructor
2. Destructor
3. Insert an item into the hash table as just described. It doesn't matter where in the linked list it is inserted.
4. Display each index in the hash table and a list of all the items stored at that location.

For this particular problem, the hash table is to store strings and use the following hash function:

`h(str)` = (sum of the ASCII codes of the first three characters of `str`) % *table_size*.

For strings with fewer than three characters, just use whatever characters there are in the string.

B. Write a driver program to test your class. It should read several strings, storing each in the hash table. After all the strings are input, it should display the hash table by listing the indices and, for each index, a list of all the words in the linked list at that location (sort of like the diagram above).

C. After you have thoroughly tested your class, use it to process the strings in the piece of text:

```
DEAR MARLIN
   THE AARDVARKS AND THE CAMELS WERE MISTAKENLY SHIPPED TO THE
     AZORES
   SORRY ABOUT THAT
   SINCERELY  JIM
   PS  ANTS AND BATS AND COWS AND CATS ARE ANIMALS
   COWS ARE BIG BUT ANTS ARE SMALL AND BATS AND CATS ARE IN BETWEEN
```

Hand in the items specified on the grade sheet on the next page.

Course Info: ______________ ***Name:*** ______________________________________

Project 5.2 Grade Sheet

Hand in:

1. A listing of the source file(s) for class `HashTable`
2. A listing of the driver program used in (A) to test the class `HashTable`
3. At least two executions with your own test data
4. An execution with the text given in (C)

Attach this grade sheet to your printouts.

Category	Points Possible	Points Received
Correctness of class (including following instructions)	70	____________
Design and structure of the class	15	____________
Documentation (including specifications of functions)	10	____________
Style (white space, alignment, etc.)	5	____________
Driver program: (checks all the operations)	25	____________
Total	**125**	____________