Chapter 4-2, Lecture notes

Programming with Linked Lists

1. Linked list operations are the basis for many of the data structures in the book.
   1.1 This is essential material.
   1.2 If you needed to print the linked list, you would have a pointer to the head of the list, and go through the list. Like so:

```
Node *cur = head;

while (cur != NULL)
{
        cout << cur->item << endl;
        cur = cur->next;
}
```

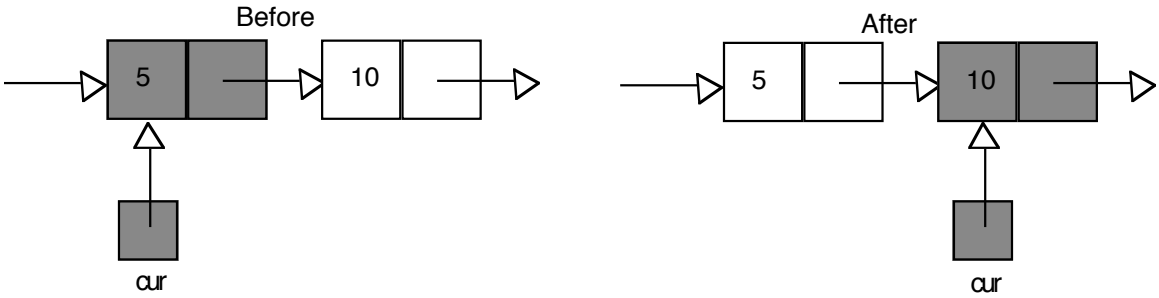
   1.3 The cur pointer points to the current node and, after the item is printed, moves to the next node. You could move down the list like so:

```
temp = cur->next;
cur = temp;
```

But you can also do:

```
cur = cur->next;
```

   1.4 The operation acts like so:



   1.5 You can also use a for loop to print the entire linked list, like so:

```
for (Node *cur = head; cur != NULL; cur = cur->next)
        cout << cur->item << endl;
```

      1.5.1 You will continue down the list (or "visit" each node) until you hit the NULL.

1.5.2 One common error would be to check cur->next instead of cur.
1.5.3 If the list is empty, then the "cur->next" will crash the program.
1.5.4 Printing each node in a linked list does not change the linked list.
1.5.5 Let's look at operations that insert and delete.

1.6 With insert and remove (or delete) operations, you need to point to the current node, called cur, and point to the previous node, called prev.
1.6.1 The reason for prev is to be able to either insert the new node or to remove a node.
1.6.2 So cur is the current pointer and prev is the "trailing" pointer.
1.6.3 On page 192 is the first attempt at the insert in pseudocode:

```
prev = NULL
cur = head
while (newValue > cur -> item) // causes a problem
{       prev = cur
        cur = cur->next
}
```

1.6.4 The while loop would have a serious problem if the new value is greater than ANY value in the list (or the list is empty) – cur would become NULL and cause the program to crash.
1.6.5 So while must be written to be:

```
prev = NULL
cur = head
while (cur != NULL and newValue > cur->item)
{       prev = cur
        cur = cur->next
}
```

1.6.6 If cur is NULL and prev is not, then you insert at the end of the list.
1.6.6.1 But the insert would be the same as inserting in the middle.
1.6.6.2 You can use the following code for middle or end, like so:

```
newPtr->next = cur;
prev->next = newPtr;
```

1.6.7 Yet, you do need to check to see if prev is NULL. If so, then you need to insert at the head of the list. You would do:

```
newPtr->next = cur;
head = newPtr;
```

1.6.7.1 Even with an empty list, it would work fine because cur would be NULL (because head is NULL in an empty list) and the newPtr->next link would point to NULL.

1.6.7.2 The above pseudocode can be done by a creative use of the for statement. Like so (page 194):

```
for (prev = NULL, cur = head;
     (cur != NULL)  && (newValue > cur->item);
    prev = cur, cur = cur->next);
```

1.6.7.3 The && operation has a "short-circuit" feature in which if the first part of the operation is false then it does NOT do the second operation.

1.6.7.4 That prevents doing a deference of the operation of "cur->item" and accessing an NULL cur pointer.

2 So we can use pointers for implementing the ADT List.

2.1 Can also use indexes to place items in the list.

2.1.1 Will use a class for the implementation.

2.1.2 The prev and cur pointers will be local pointers only and not be included in the class definition.

2.2 Want to be able to specify the Ith place in the list.

2.2.1 Need to use a "find" operation to do so and have it return a pointer.

2.2.2 But you do not want the client to get a pointer as a return value since that would violate the "wall" and require the client to know how the implementation is done.

2.2.3 The find would need to be a private function to the class – that would allow the class functions to use the find operation but not the client.

2.3 The Listp.h file would be (with a print operation added):

```
/* @file ListP.h */

#include "ListException"
#include "ListIndexOutOfRangeException.h"
typedef int ListItemType;
/** ADT list - Pointer-based implementation */
class List
{
public:
// Constructors and destructors

  /** Default constructor */
  List();

  /** Copy constructor
```

```
   * @param aList   The list to copy */
  List(const List& aList);

  /** Destructor */
  ~List();

  // List operations;
  bool isEmpty() const;
  int getLength() const;
  void insert(int index, const ListItemType& newItem)
    throw(ListIndexOutOfRangeException, ListException);
  void remove(int index)
    throw(ListIndexOutOfRangeException);
  void retrieve(int index, ListItemType& dataItem) const
    throw(ListIndexOutOfRangeException);
  void print();

private:
  struct ListNode  // A node on the list
  {
    ListItemType item;  // data item on the list
    ListNode    *next;  // pointer to the next node
  }; // end ListNode

  int size;        // number of items in a linked list
  ListNode *head;  // pointer to linked list of items

  /* Locates a specified node in a linked list
   * @pre index is the number of desired node
   * @post None
   * @param index    the index of the node to locate
   * @return A pointer to the index-th node. If index < 1 or
   *         index > length of list it will return NULL */
  ListNode *find(int index) const;
}; // end List
```

2.3.1 Note that the above would use the ListException, ListExcept.h and the ListIndexOutOfRangeException.h files from chapter 3 to handle the try and catch for errors.

Those are:

```
// @file ListException

#include <stdexcept>
#include <string>

using namespace std;
```

```
class ListException : public logic_error
{
public :
	ListException(const string & message = "")
		: logic_error(message.c_str())
	{}
}; // end
```

And:

```
// @file ListExcept.h

#include <stdexcept>
#include <string>

using namespace std;

class ListException : public logic_error
{
public :
	ListException(const string & message = "")
		: logic_error(message.c_str())
	{}
}; // end
```

And finally:

```
// @file ListIndexOutOfRangeException.h
#include <stdexcept>
#include <string>

using namespace std;

class ListIndexOutOfRangeException : public out_of_range
{
public:
	ListIndexOutOfRangeException(const string & message = "")
		: out_of_range(message.c_str())
	{}
}; // end
```

2.3.2 The implementation of ListP.cpp would start out as:

```
/** @file ListP.cpp
 *  ADT list - Pointer-based implementation */

#include <cstddef>  // for NULL
#include <new>      // for bad_alloc
#include "ListP.h"  // header file
```

```
#include <iostream>
#include <string>

using namespace std;
```

2.3.3 The default constructor is:

```
List::List() : size(0), head(NULL)
{
} // end default constructor
```

2.3.4 The "size(0)" and "head(NULL)" is a C++ convention that is equivalent to "size = 0" and "head = NULL" except that it can be used to set the initial values of constants (very handy).

2.3.5 The other constructor will copy one list to the current list being created.

2.3.6 It could do a "shallow copy" in which the head of the current list could point to the start of the other list.

2.3.6.1 However, that is NOT a good idea!

2.3.6.2 And violates the idea of the "wall".

2.3.7 A "deep copy" would create another list and copy the contents of the passed listed to the currently created list.

2.3.7.1 Much better! That copy constructor would look like:

```
List::List(const List& aList) : size(aList.size)
{
  if (aList.head == NULL)
    head = NULL; // original list is empty

  else
  { // copy first node
    head = new ListNode;
    head->item = aList.head->item;

    // copy the rest of the list
    ListNode *newPtr = head; // new list pointer
    // newPtr points to the last node in the new list
    // origPtr points to nodes in original list
    for (ListNode *origPtr = aList.head->next;
         origPtr != NULL;
         origPtr = origPtr->next)
    {  newPtr->next = new ListNode;
       newPtr = newPtr->next;
       newPtr->item = origPtr->item;
    } // end for

    newPtr->next = NULL;
  } // end if
} // end copy constructor
```

2.3.7.2 Note that it has the “size(aList.size)” to do the operation of “size = aList.size” where aList is the passed list to copy from.

2.3.7.3 The rest of the code will, if the aList is not NULL, allocate a node for the head of the current list, copy the first data of the first node to the first node of the current list, then loop and create nodes, link them into the current list, and copy the data from the aList to the current list.

2.3.8 The Destructor looks like:

```
List::~List()
{
  while (!isEmpty())
    remove(1);
} // end destructor
```

2.3.8.1 That is interesting in that the destructor used the “isEmpty” operation to drive deleting nodes via the “remove(1)” – which removes the node at location 1 or the beginning.

2.3.8.2 Eventually the list is emptied and the memory is released.

2.3.9 The operation of isEmpty and getLength are straightforward, like so:

```
bool List::isEmpty() const
{
  return size == 0;
} // end isEmpty

int List::getLength() const
{
  return size;
} // end getLength
```

3 A linked list does not provide direct access to the specified position (unlike arrays)

3.1 Need to have the retrieval, inserting, and deletion operation traverse the list from the start to the correct position.

3.1.1 The “find” does the traversal like so:

```
List::ListNode *List::find(int index) const
{
  if (  (index < 1) || (index > getLength())  )
     return NULL;
  else // count from the beginning of the list
  {  ListNode *cur = head;
     for (int skip = 1; skip < index; ++skip)
       cur = cur->next;
     return cur;
```

```
   } // end if
} // end find
```

3.1.2 If the "find" returns a NULL that means that it could not find the proper location (bad "index").

3.1.3 The retrieve would use the find function and would do a "throw" if the correct node is not found, like so:

```
void List::retrieve(int index, ListItemType& dataItem) const
  throw(ListIndexOutOfRangeException)
{
  if ((index < 1) || (index > getLength()))
    throw ListIndexOutOfRangeException(
    "ListIndexOutOfRangeException:retrieve index out of range");
  else
  { // get pointer to node, then data in node
    ListNode *cur = find(index);
    dataItem = cur->item;
  } // end if
} // end retrieve
```

3.1.4 If the node is found the "dataItem = cur->item" will set the dataItem which is a reference and thus the value at the position of "index" is passed back.

3.2 With the insert, the first position insert is a special case.

3.2.1 The insert will also throw and exception for a bad index (less than 1 or greater than the length + 1 or the expanded list). It will also use a "try" block to allocate a node.

3.2.2 If it can't then the corresponding "catch" block is called with an object of "bad_alloc e" (where more information could be obtained from) and another exception is thrown (most of the time the program will have more than enough memory to operate with).

3.2.3 The function looks like:

```
void List::insert(int index, const ListItemType& newItem)
  throw(ListIndexOutOfRangeException, ListException)
{
  int newLength = getLength() + 1;
cout << index << " " << newLength << endl;

  if (  (index < 1) || (index > newLength)  )
     throw ListIndexOutOfRangeException(
        "ListIndexOutOfRangeException: insert index out of range");
  else
  { // Create new node and place newItem in it
    try // Try to allocate memory
    {
```

```
      ListNode *newPtr = new ListNode;
      size = newLength;
      newPtr->item = newItem;

      // Attach new node to list
      if (index == 1)
      { // Insert new node at beginning of list
        newPtr->next = head;
        head = newPtr;
      }
      else
      { ListNode *prev = find(index - 1);
        // Insert new node after node to which prev points
        newPtr->next = prev->next;
        prev->next = newPtr;
      } // end if
    } // end try
    catch (bad_alloc e)
    {
      throw ListException(
              "ListException: insert cannot allocate memory");
    }
  } // end if
} // end insert
```

3.3 The deletion operation is like the insertion operation:

3.3.1 Must find the item,

3.3.2 Make sure that prev and cur pointers are set correctly,

3.3.3 But remove and do not insert a node.

3.4 Deletion from the front is, of course, a special operation. The remove (deletion) operation is like so:

```
void List::remove(int index) throw(ListIndexOutOfRangeException)
{
  ListNode *cur;

  if (  (index < 1) || (index > getLength())  )
     throw ListIndexOutOfRangeException(
        "ListIndexOutOfRangeException: remove index out of range");
  else
  {  --size;
     if (index == 1)
     { // delete the first node from the list
       cur = head;  // save pointer to node
       head = head->next;
     }
```

```
      else
      { ListNode *prev = find(index-1);
        // delete the node after the node to which prev points
        cur = prev->next;  // save pointer to node
        prev->next = cur->next;
      } // end if

     // return node to system
     cur->next = NULL;
     delete cur;
     cur = NULL;
   } // end if
} // end remove
```

3.5 The print operation is just like the following:

```
void List::print()
{
  for (ListNode *cur = head; cur != NULL; cur = cur->next)
    cout << cur->item << endl;
}
```

4 Every ADT implementation would have advantages and disadvantages – i.e. there is no such thing as the proverbial "free lunch".

4.1 With array-based implementation, the array behaves like a list and is easy to use.

4.1.1 However, it is of fixed size and it is possible to overflow it.

4.1.2 If you use arrays, are you confident on how many items are in the list?

4.1.3 If not, then the array-based implementation can be problematic.

4.2 Now with a dynamically allocated array, you could double the array each time you need to have more room.

4.2.1 The problem is that you need to copy the old values into the new array.

4.2.2 That could be time consuming if you have to reallocate the array several times.

4.2.3 But if the number of items would be small, then the array-based implementation is reasonable.

4.3 With the pointer-based implementation, you can use the "new" operation to allocate as many nodes as you need to – no wasted space.

4.3.1 However, with the array-based list, the next item or successor in the list is "implicit" or, rather, the node after "index" is implied to be "index+1" but with pointer-based, you need the pointer of "next" in the current node to tell you where the successor node is.

4.3.2 And with pointer-based, you need to traverse the list to insert, delete, and/or access a node.

4.3.3 With array-based, you have "direct access" with pointer-based you have an "access time" to get to the proper node (the longer the list the large the access time will be).

4.3.4 But, with array-based list that is ordered, you need to shift the nodes with an insert and a delete (or remove) – can take up to "size-1" moves. The pointer-bases does not have to shift any nodes whatsoever.

4.3.5 Chapter 9 we'll talk about a more formal way to discuss the efficiency of algorithms.

5 One problem that we have is how to read in a LOT of nodes without laboriously typing them in all the time and be able to save LOTS of nodes to future retrieval.

5.1 The easiest thing would be to use a text file to store the data into and read the data from – i.e. write the "item" part of the nodes into a file and read the data from the file to store into the "item" part of the nodes (recreate the linked list).

5.2 Can add to the ADTList definition the following:

```
void writeFile(string fileName);
void readFile(string fileName);
```

5.2.1 For writing to and reading from the file.

5.2.2 The "fileName" is the string that holds the name of the file.

5.2.3 With both files, you need to include fstream (file I/O stream) like so:

```
#include <fstream>
```

5.2.4 The writeFile would look like:

```
void List::writeFile(string fileName)
{
 // Saves a linked list's data in a text file of integers; head points
  //   to the linked list. The fileName is a string that names an
  //   external text file to be created

  ofstream outFile(fileName.c_str());

  // traverse the list to the end, writing each item
  for (ListNode *cur = head; cur != NULL; cur = cur->next)
    outFile << cur->item << endl;

  outFile.close();
}
```

5.3 That is the same as in the textbook but with a few changes.

5.3.1 For instance, the statement "ofstream outFile(fileName.c_str())" both defines the output file stream (ofstream) and also opens the file in the fileName at the same time! However, the textbook made a mistake in that the fileName would need to be a c string or character array (which "string" is NOT).

5.3.2 The operation of ".c_str()" will take a string and generate a character string.

5.3.3 Then the for loop will merely traverse the linked list and output the data items to the text file.

5.4 To read the values in, it is assumed that the list is empty.

5.4.1 The readFile function is like the textbook but with the addition of a throw of an exception is the list is NOT empty (i.e. “head != NULL”). Like so:

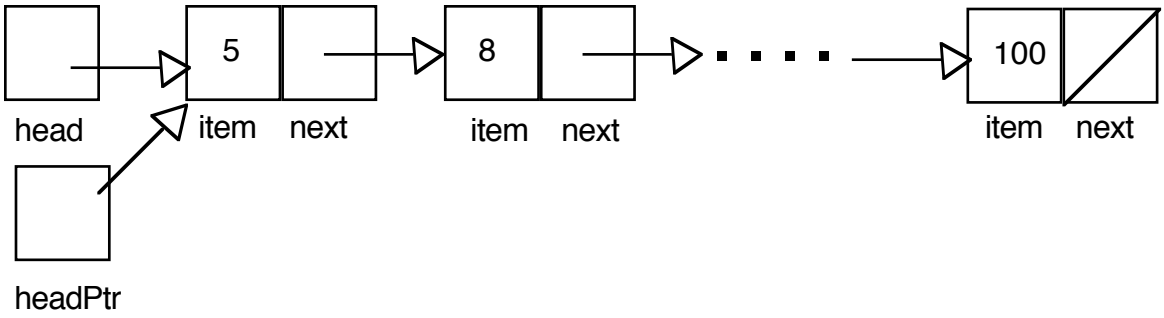
```
void List::readFile(string fileName)
{
  // Creates a linked list from the data in a text file. The pointer
  //   variables head and tail are initially NULL. The fileName is a
  //   string that names an existing external text file

  ifstream inFile(fileName.c_str());
  int nextItem;

  if ( head != NULL  )
    throw ListIndexOutOfRangeException(
"ListIndexOutOfRangeException: list must be empty to read from file");

  if (inFile >> nextItem) // Is file empty?
  {  // File not empty
     try
     { head = new ListNode;
       // Add the first integer to the list
       head->item = nextItem;
       head->next = NULL;
       size ++;
       ListNode *tail = head;

       // Add remaining integers to linked list
       while (inFile >> nextItem)
       { tail->next = new ListNode;
         tail = tail->next;
         tail->item = nextItem;
         tail->next = NULL;
         size ++;
       } // end while
     } // end try
     catch (bad_alloc e)
     { throw ListException(
          "ListException: restore cannot allocate memory.");
     } // end catch
  } // end if

  inFile.close();
}
```

5.4.2 One interesting feature of reading from a file is that if the operation fails, it returns a false, which can be tested.
5.4.2.1 Therefore if the rather odd "if (inFIle >> nextItem)" statement attempts to read and if the operation fails (empty file, for instance), then you get a false and the function ends.
5.4.2.2 Otherwise, you have read in a value – all in one "if"!
5.4.2.3 Rather clever – but also a bit odd.
5.4.2.4 The "while (inFile >> nextItem)" does the same thing: attempts to read from the file into the nextItem.
5.4.2.5 If it fails, you get a false and the loop ends.
5.4.2.6 Otherwise, you have read into nextItem and you can insert into the linked list.
5.4.3 The function used the pointer of "tail" rather than "cur" or "prev" to keep track of the end of the list to be able to read in the next value.
5.4.3.1 Note that the last node added always had its "next" pointer set to NULL just in case the data items from the file runs out.
5.4.3.2 And the operation of "tail->next = new ListNode" allows to allocate AND link the next new node very efficiently.
5.4.3.3 You need the next statement of "tail = tail->next" to have tail point to the new node to be able to set the node values.

6 You can pass a pointer to a function, such as head, to a function and thus giving the function access to the linked list.
6.1 Not a good idea unless the function is private and part of the "List" class (don't want the client code to muck up the linked list).
6.2 But passing the head pointer would make sense for a recursive traversal of a linked list.
6.3 For instance if you wanted to print recursively, you would print the head of a list of n items, then call yourself to print the list of n-1 items (i.e. the "rest" of the list).
6.4 With the print function, it is not changing anything so you can do a call by value to the function. Like so:

```
void print(ListNode *headPtr);
```

6.5 Would create a temporary pointer of headPtr that would point to the same linked list as head. Like so:



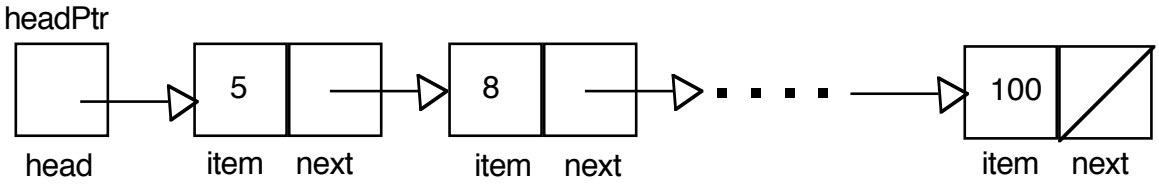
6.6 However, print is a public function and thus the client would need to know the variable head.

6.6.1 With recursion, it would be better to have a "printRecur" private function and have the public print function call it and pass the head pointer to it. Like so:

```
void LinkedList::print()
{
  printRecur(head);
}

void LinkedList::printRecur(Node *headPtr)
{
  if (headPtr != NULL)
  {
    cout << headPtr->item << endl;
    printRecur(headPtr->next);
  }
}
```

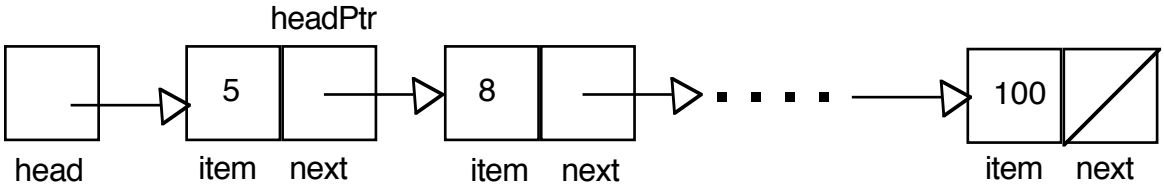
6.6.2 The printRecur function has the base case of headPtr becoming NULL.

6.6.3 Now if you wanted to change the linked list, you need to pass the pointer as a reference so when you change headPtr inside the function, it changes the pointer on the outside, like so:



6.6.4 On the first call, headPtr is the same thing as head.

6.6.5 However, when you move down, the headPtr also moves, like so:



6.6.5.1 The headPtr now is the same as the "next" in the node.
6.6.5.2 If you change headPtr, you also change that next pointer (it is a little mind-bending but it does make sense).
6.6.5.3 So all you really need to insert into a linked list recursively is to insert at the "beginning" (beginning is where the headPtr is currently at).
6.6.5.4 You would have an insert function that allocates a new node, places the data into item, THEN calls the insertRecur function (easier). The code looks like:

```
void LinkedList::insert(ItemType val)
{
  Node *temp = new Node;
  temp->item = val;
  temp->next = NULL;
  insertRecur(head, temp);
}


void LinkedList::insertRecur(Node *& headPtr, Node *newItem)
{
  if (headPtr == NULL)
    headPtr = newItem;
  else if (headPtr->item > newItem->item)
  {
    newItem->next = headPtr;
    headPtr = newItem;
  }
  else
    insertRecur(headPtr->next, newItem);
}
```

6.6.6 Notice that the headPtr is a reference with the "&".
6.6.6.1 When headPtr is changed, the calling pointer is also changed.
6.6.6.2 The only special case is if the list is NULL.
6.6.6.3 Other than that, the inserting is done at the "beginning" of the list (which, in recursive terms, could actually be in the middle or the end) – a difficult concept.
6.6.6.4 Remove would be similar (except it does not need to allocate a node). That code is:

```
void LinkedList::remove(ItemType val)
{
  if (removeRecur(head, val))
    cout << val << " removed\n";
  else
    cout << val << " not there\n";
}

bool LinkedList::removeRecur(Node *& headPtr, ItemType val)
{
  if (headPtr == NULL)
    return false;
  else if (headPtr->item != val)
    return removeRecur(headPtr->next, val);
  else
  {
    Node *temp = headPtr;
    headPtr = headPtr->next;
    delete temp;
  }
}
```

6.6.7 Notice that the remove function also would print out whether or not node has been removes – that is arguably not needed.

6.6.8 Again, the most conceptually difficult thing is that the statement in removeRecur of "headPtr = headPtr->next" is changing the pointer on the outside of the function to remove a node.