Data Structures and Algorithms

Alice E. Fischer

Lecture 3 – 2016

1. Linear Data Structures
   - Arrays
   - Growing Arrays
   - vectors
2. Linked Lists
3. Linked List Operations
   - Empty Lists
   - Inserting Data - Unsorted
   - Inserting Data - Sorted
4. Implementing a List Class.

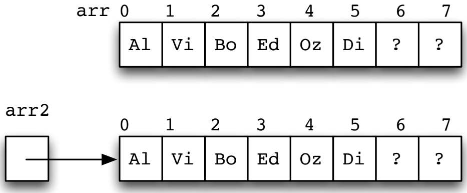
The Choices

- Array: limited in size but simple and efficient for random access. Can be sorted.
- Growing Array: able to grow as needed but with a performance penalty.
- Linked list: able to grow as needed, sequential access only. Takes extra space to store pointers and for allocation overhead.

Diagram of an array of Items

The first data structure would result from declaring an array variable then storing six strings in it.

```
    string ar[ 8 ] ;
```



The second diagram illustrates the result of dynamically allocating a similar array.

```
    string* ar2 = new string[ 8 ] ;
```

Implementation of a growing array

This is a resizeable array that expands when needed. It is intended for storing input data when the amount of data is not known ahead of time. New items are added at the end of the array, which is reallocated and enlarged when it is full.

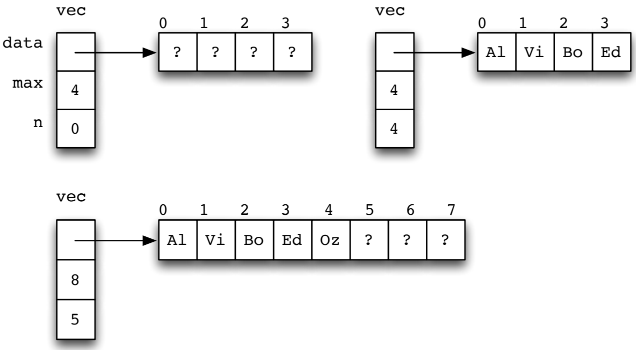
Any growing array class has these members:

- private: max, n (ints) and data (a pointer to a dynamically allocated array)
- public: void insert(item T): Grow if necessary, then add the item to the end of the array.
- private: void grow(): Double the max, allocate an array using the new value of max. Copy the data into it.

The growing array class that is part of C++ is `vector`.

Diagram of a Growing Array

This is a `vector<string>` that is currently big enough to contain 8 strings and does contain 5 strings. It just doubled in size because the original container of length 4 was too small to hold the fifth string. Any type or class could be substituted for string.



vector Constructors and basics

Here is a list of the most important `vector::iterator` functions. In this chart, *BT* stands for the base type of the array, bto stands for an object of that type, and vc stands for the vector object.

vector< *BT* > vc;	Construct empty vector.
vector< *BT* >:: iterator p1, pos;	Create iterators to point into vector.
vc.push_back(bto)	Insert object at end; grow if necessary.
vc.pop_back()	Remove and discard last object.
vc.clear()	Discard all elements.
k = (int) vc.size()	Return # of data objects in vc.

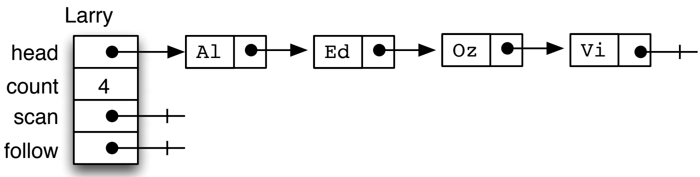
Vector and Iterators.

In this chart, *BT* stands for the base type of the array, bto stands for an object of that type, and vc stands for the vector object.

bto = vc[k]	Return the contents of the kth slot.
bto = vc.front()	Return the first element in the vector.
bto = vc.back()	Return the last element in the vector.
p1 = vc.begin();	Point to first element in vector.
p2 = vc.end();	Point to first unused address.
p1++ , ++p2	Increment iterator.
−−p2 , p2−−	Decrement iterator.
pos = find(p1, p2, bto);	Return position of bto or *vc.end*()
if (pos == vc.end())	has iterator reached end of vector?
vc.sort (p1, p2)	Sort the elements of vc

Diagram of a Linked List

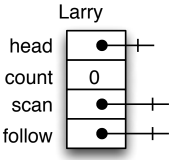
A linked list is composed of cells and pointers.
This list contains 4 Items. The last Cell in the list points to `null`.
The member named `count` is useful, but optional.
scan and follow are used for searching the list.



A linked list can grow easily.
Items can be added/removed in the middle of the list efficiently.
However, random access is impossible and sequential access is slow.

Construct an empty linked list

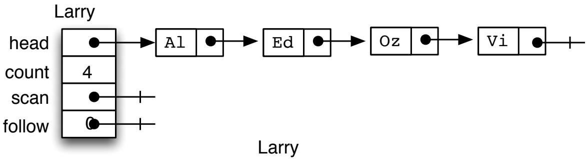
The List constructor should set the head pointer to null (since there is no data yet) and the count to 0. The utility pointers, scan and follow, may also be set to null, for completetness.



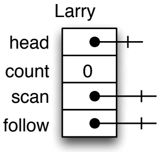
Clear all Data out of a List.

This operation would be done after using all the data in an existing list and prior to re-using a List object.

Before: Larry has one or more entries:



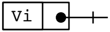
After: The entries are gone, Larry is empty. The programmer must delete each of the four cells.



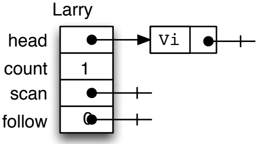
Add Data to an Unsorted List.

Since the list is not sorted, data is always entered at the head of the list because that is fastest.

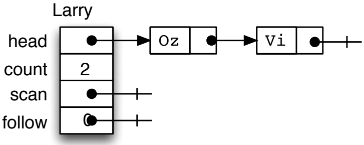
Step 1: Call Entry constructor with the value of the new data object and the current contents of head.



Step 2: Make Larry's head point at the new object and increment Larry's count.



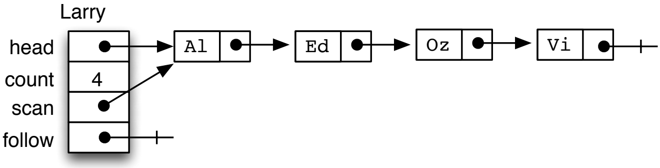
Step 3: Add another.



Add Data to a Sorted List.

Since the list is sorted, data must be inserted in its proper place. Thus, the insertion process has two steps: a) find the right place and b) insert the cell. Let us focus first on finding the right place.

Step 1: Let scan = head and follow = nullptr.

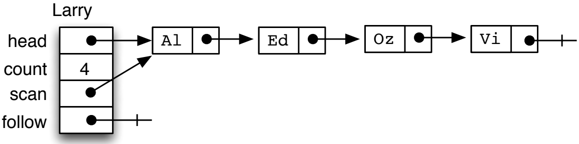


Given these initial assignments, we are ready to search the list.

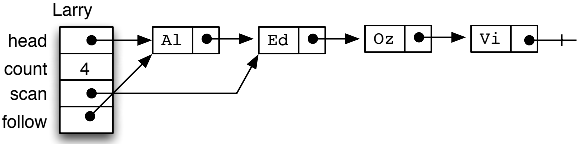
Add Data to a Sorted List.

Suppose we want to insert Li into the list.

Step 2: Compare scan's data to the new data. If the new data is larger, walk one step down the list.

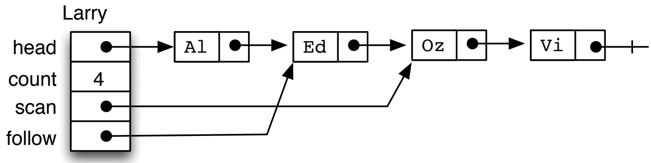


Step 3: To go one step, set **follow = scan,** then set **scan = scan->next.**



Finding the Insertion Location.

Repeat steps 2 and 3 until scan's data is larger than the new data, then stop.



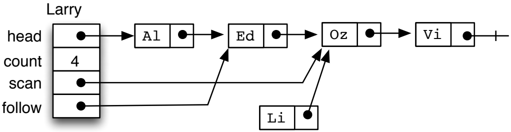
If `scan` reaches the end of the list, it must stop.

If `follow` is still `null` when `scan` stops, the insertion slot is at the head of the list.

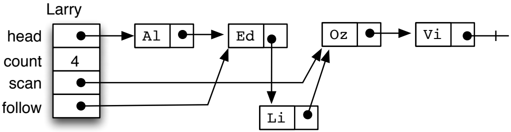
Otherwise, insertion is in the middle of the list.

Do the Actual Insertion.

Step 4. Construct a new cell with the new data and a copy of scan in it.

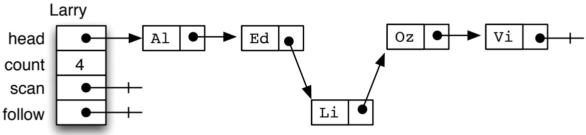


Step 5. Attach follow->next to the new cell.



Do the Actual Insertion.

The insertion is now done. Set scan and follow back to nullptr.



Homework: How would you do a deletion? Explain it. Supply diagrams. Due in 1 week.

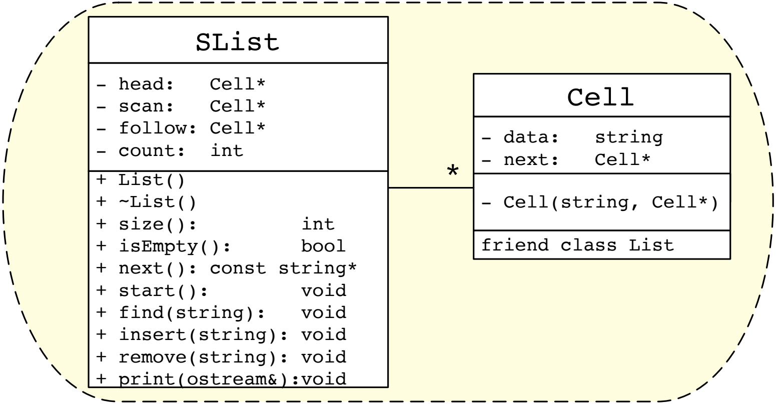
Design the Capabilities, Interface, and Semantics.

First, a programmer must decide what he wants his data structure to do. The functions for a sorted linked-list are different from a sorted array because accessing and allocating patterns are different. Without careful planning the "fixes" could go on for a long time.

- Semantics: The list is either empty or head points at a cell.
- Functions needed: size, isEmpty, insert, remove, find, print. Iterate through the list.
- You can start an iteration at the beginning of the list and get to the end. The current state of the iteration is stored internally in the list.
- You can find a value stored anywhere in the list. The position of that value's cell and the prior cell are stored internally in the list.

Implementing a List Class.

We need a class and a helper class because a list has manycells.



```
SList
- head:   Cell*
- scan:   Cell*
- follow: Cell*
- count:  int
+ List()
+ ~List()
+ size():          int
+ isEmpty():       bool
+ next(): const string*
+ start():         void
+ find(string):    void
+ insert(string): void
+ remove(string): void
+ print(ostream&):void
```

*

```
Cell
- data:    string
- next:    Cell*
- Cell(string, Cell*)
friend class List
```

The SList has Internal State

The two-pointer scan supports all list operations.

- Scan and follow are class members so that they can be used to communicate from one class function to the next.
- The `find()` function sets scan and follow for use by other functions. This permits all functions to be short and clear.
- `find()` stops on a $<=$ comparison. Thus, `scan->value` is either greater than the value you are looking for, or they are equal.
- Follow always trails one cell behind scan.
- All operations that change the list or the internal state must be careful about the head and the tail of the list.

After calling `find()` . . . .

After calling `find()`, a function must check for and handle relevant special cases:

1. Scan will be nullptr if the list is empty. Caution: A program must check `scan` before trying to use it
2. Scan will be nullptr if `find()` reached the end of the list.
3. Follow will be nullptr if `find()` stopped at the head of the list. In this case, an insertion would go at the head of the list.

`insert()` and `getVal()` have special cases for (1) and (2).
`remove` needs to test (1) and (3).

The Cell Class.

This is a private class, only accessible to SList. Note the ctors. The SList class declaration follows in the same file.

```
#include "tools.hpp"
typedef string BT;
//------------------------------------------------------
class Cell {
friend class SList;
private:
    BT data;
    Cell* next;
    //--------------------------------------------------
    Cell(BT dt, Cell* nx=nullptr) : data(dt), next(nx){}
};
```

The SList class

Data members and inline functions:

```
class SList {
private:
  Cell* head = nullptr;     // Permanent head pointer.
  int count = 0;            // The # of items currently.
  Cell* scan = nullptr;     // for walking down list.
  Cell* follow = nullptr;

public:
  SList() = default;
  int   size(){ return count; }
  bool  isempty(){ return count==0; }
```

The SList class

Functions defined in the .cpp file:

```
~SList();
void  start();              // Position scan at head.
const BT* next();           // Return scan->data, go next.
void  find( BT value );     // Position scan and follow
void  insert( BT value );   // Calls find().
void  remove( BT value );   // Calls find().
void  print ( ostream& out ) const;   };
```

Some of these are given below. Others are left for the student to complete.

Destructor for SLists.

All memory created with new must be freed.

```
// --------------------------------------------------
SList::~SList()
{
    for(;;) {
        scan = head;
        if (scan==NULL) break;
        head = head->next;
        delete scan;
    }
}
```

SList::print

List must provide the interface for itself and for the helper class.

```
// --------------------------------------------------
// Print contents of all Cells in the List.
void SList::print ( ostream& out )
{
    out <<"---------------\n";
    scan = head;
    while ( scan != nullptr ) {
        out << scan->data <<endl;
        scan = scan->next;
    }
    out <<"================\n";
}
```

SList::remove

```
void SList::remove( BT val ) {
    find( val );
    if (scan == nullptr || scan->data != val )
        cout <<endl <<val <<" is not in the list.\n\n";
    else {
        --count;
        if(follow == nullptr) {
            head = head->next;          delete scan;
        }
        else {
            follow->next = scan->next;  delete scan;
        }
    }
}
```

Written Homework, Due Feb. 16

1. Implement insertion for the unsorted list class.
   void insert(string st):

   Implement these functions for the sorted list class:
2. void insert(string st):
   When the insertion is done, make follow point at the new cell. Beware of null pointers.
3. const BT* next( ):
   Return nullptr if scan is nullptr. Otherwise, return the value under scan and move follow and scan to the next cell(s). Beware of null pointers. Use & to return a pointer to the data in a cell.

Program, Due Feb. 16

The little brown dog.

1. Open a text file that contains an indefinite number of lines of text.
2. Read entire lines one at a time.
3. Insert each line in a linked list of strings in alphabetic order by the first letter of each string.
4. When eof happens, print out the contents of the list, one cell per output line. Check your output to be sure that the last input line does not occur twice. Do not modify the data file I give you.
5. Make a main function in one file, and also a class for the data structure with a header file and a code file.

Turn in your source code files and your output, zipped into a single file.