Data Structures and Algorithms — CSCI 230
Lists, Stacks and Queues

The Basic Data Structures

The material on linear structures such as lists, stacks and queues should be review and will be treated as such.

- The list abstract data type (ADT) consists of a linear structure and a set of operations on this structure.
- One basic implementation of a linear structure is an array. In modern programming terms, this should be thought of as a dynamically sized structure through use of a `vector` from the standard library.
- The second basic implementation is a linked-list.
- Stacks and queues are specialized linear structures.
- Stacks allow insertion and deletion from only one end (the "top"), making them "LIFO" structures.
  - The allowed insert and delete operations take place in $O(1)$ ("constant") time.
  - Stacks are easily implemented in arrays or linked-lists.
  - Applications include implementations of function calls during program execution and evaluation of postfix expression.
- Queues allow insertion at one end (the "rear") and deletion from the other, making them "FIFO" structures.
  - The allowed insert and delete operations take place in $O(1)$ ("constant") time.
  - Queues are most easily implemented in a linked-list, although array implementations are possible (recall lab 1). STL does not allow implementation in a `vector`.
  - Applications include any operation where jobs are handled on a "first-come-first-served" basis.

Exercise — Linear Structure Efficiency

Different linear structures differ in the efficiency of the operations applied to them. One must be aware of these efficiencies when choosing between the different linear structures for a particular problem (or when determining whether any of them are appropriate).

There are three main operations on a linear structure: search, insert and delete. Please fill in the following table to give the worst-case efficiencies of these operations for the different linear data structures. In the table, the "Insert" operation is broken up into "Finding" the location to insert and actually performing the insert. Similarly, the "Delete" operation is broken up into "Finding" the location to delete and actually performing the deletion. The operations on sorted lists must preserve the ordering!

Be careful of the questions on stacks and queues: some of the operations are not appropriate!

		Insert		Delete	
Linear Structure	Search	Find	Insert	Find	Delete
Lists: Arrays					
Lists: Linked					
Sorted Lists: Arrays					
Sorted Lists: Linked					
Stacks					
Queues					

The Polynomial Class

Central and powerful features of C++ are the mechanisms for creating class objects that mimic real life objects, defining natural operations on them, and then using them as building blocks in much larger programs. The extended example here is of polynomials.

- Many operations should be defined on a polynomial object, including: addition, subtraction, multiplication, division, and evaluation. These operations determine the public interface (the member functions) of the polynomial class.
- The private (internal) representation of the polynomial must be designed to allow these operations to be implemented as easily and as efficiently as possible. We will consider several alternative implementations in class.
- After the public interface and private representation are determined, the next step is to write the algorithms and implementations of the member functions. The two main algorithms (operators) we will focus on are addition and multiplication.
- Here is an outline of an algorithm to add two polynomials. It assumes the terms of each input polynomial are sorted by non-decreasing order of exponent and that only terms with non-zero coefficients are represented. The resulting polynomial will have the same properties

```
let f and g be the two polynomials to be added
let h be the resulting polynomial

while unprocessed terms remain in both f and g {
  let t1 be the remaining term in f with greatest exponent
  let t2 be the remaining term in g with greatest exponent

  if (t1.exponent == t2.exponent)
     append a new term to h with
        coefficient = t1.coefficent + t2.coefficient and
        exponent = t1.exponent
  else if (t1.exponent > t2.exponent)
     append a copy of t1 to the end of h
  else
     append a copy of t2 to the end of h
}

copy the remaining terms of f to the end of h
copy the remaining terms of g to the end of h
```

- Attached to the end of the notes is the class declaration and some of the implementation.

Exercises exploring the Polynomial class

1. Review the class design and implementation. Do you have any questions?

2. What is the worst-case time required for the addition operation? Assume there are m terms with non-zero coefficients in f and n terms with non-zero coefficients in g. What would have to change in the algorithm and what would be the resulting time complexity if the vector of terms was not sorted by exponent?

3. Design an algorithm to multiply two polynomials. The algorithm must preserve the ordering of the exponents and it must merge terms having the same exponent! What is the worst-case time required by your algorithm? Can you come up with an algorithm requiring $O(mn \log(mn))$ time? Write the details of the algorithm in C++ code.

Amortized analysis and dynamic arrays

This material will only be included if we have time. It shows that the cost of using dynamically resized arrays, such as the `vector` container class from the standard library, is not significantly greater than the cost of using ordinary arrays.

- The *amortized cost* of k operations is the average of the worst-case costs of the individual operations. It is found by summing the cost of all k operations and dividing by k.
- For example, if exactly 2 of the k operations requires k steps and each of the remaining $k-2$ operations requires 1 step, then the amortized cost of each operation is $O(1)$.
- We will consider as an example a simplified version of the `vector` and its `PushBack` function.
- When inserting values at the end of a vector, `PushBack` usually just puts each new value in the next available location. On the rare occasions when the array is full, a new array is allocated, the contents of the old array are copied into it, and the old array is deleted.
- These operations are shown in the following C++ code.

```
const int Mult=2;
const int Add=100;

template <class T>
class Vector {
public:
  Vector( int init_size = 1 );
  void PushBack( const T& item );
  //  and many more useful member functions
private:
  T * values_;
  int count_;   // number of current values
  int size_;    // size of the current allocation
};

template <class T>
Vector<T>::Vector( int init_size );
{
  values_ = new T[init_size];
  count_ = 0;
  size_ = init_size;
}

template <class T>
void Vector<T>::PushBack( const T & item )
{
  if ( count_ == size_ ) {         // reallocation
    int new_size = size_ + Add;    // #1a
    int new_size = size_ * Mult;   // #1b
    T * temp = new T[new_size];
    for ( int i=0; i<size_; i++ )
      temp[i] = values_[i];        //  #2
    delete [] values_;
    values_ = temp;
    size_ = new_size;
  }
  values_[count_++] = item;        //  #3
}
```

Only one of the statements 1a and 1b will be used. Part of the problem is to figure out which.

Consider the following operation:

```
Vector<int> v;
for ( int i=0; i<n; i++ ) v.PushBack(10*i);
```

Now, do the following exercises, in order:

1. For $i = 0$, $i = 1$, $i = 2$, $i = 3$, $i = 4$, $i = 5$, etc., how many operations are required by the `PushBack` function? Count only statements labeled `#2` and `#3` and consider option `#1a` only. Can you see a pattern?

2. What is the cost of operation `#3` when it is summed over all n iterations of the `for` loop? This should be straightforward.

3. What is the cost of operation `#2` when it is summed over all n iterations of the `for` loop? This should be more involved.

4. Add the answers to the previous two questions and divide by n to calculate the amortized cost of each `PushBack` operation.

5. Repeat the previous four steps for the case where statement `#1b` is used instead of `#1a`.

Review Problems

In addition to the material covered in class, the quiz for chapter 3 will include material from the lab on doubly-linked lists (lab 3) and the lab on the standard library (STL) vectors and lists (lab 4). Students should not worry too much about STL syntax, but should worry about semantics. For example, if you assume there is a `push_front` method on an STL vector you will be penalized because this operation makes no sense in terms of the logical properties of vectors and STL.

1. Why is the standard library `vector` container class not a good implementation for queues?

2. I need a program to store and access a set of objects called *foos*, and I need to figure out what type of data structure to use. Each *foo* has an id number and other information, which is not important to this question. The operations I need on my set of *foos* include inserting a *foo*, searching for the *foo* having a particular id number, and printing the *foos* in order of id number. The special feature of the *foo*s is that the order they are input is *almost sorted*. In particular, at the time *foo* f is input it is *guaranteed* that there will be no more than k *foos* currently stored (already input) with greater id numbers. This special feature does not carry over to the search operation — each id number in the set is equally likely to be requested at any time.

   (a) Suppose I use a dynamic array such as an STL vector to store my *foos*. What is the cost of an insert, a search and a print operation, as a function of n (the number of currently stored *foos*) and k? Explain your answer briefly, including any assumptions you need to make.

   (b) Repeat the question for a linked list.

   (c) Which of these linear data structures would you choose? Why? What effect does the value of k have on your answer?

3. Write a function that reads in a set of floats from an input stream (provided as a parameter) called `instr`. First, the function should store all of the values in a STL `list` container class. Then, it should output (using `std::cout`) every nth value in the list, starting from the end of the list, where n is passed in as an integer parameter. The first value displayed will always be the last item in the list; the second will be the $(n + 1)$st from the end; etc. Repeat using an STL `vector` instead of a list.

4. Write a function to reverse the direction of the pointers in a singly-linked list and another one to reverse the direction of the pointers in a doubly-linked list. The former is much harder than the latter. Assume the structure

```
template <class T>
struct Node {
   T value;
   Node * next;
};
```

for singly-linked lists and

```
template <class T>
struct Node {
   T value;
   Node * next;
   Node * prev;
};
```

for doubly-linked lists. Assume each function call is

```
void Reverse( Node* & head )
```

and that `head` is to be changed to point to the node that previously was the tail.