Computer Science II — CSci 1200
Test 3 — Overview and Practice

Overview

- Test 3 will be held **Friday, April 20, 2007 2:00-3:30pm, Darrin 308**. No make-ups will be given except for emergency situations, and even then a written excuse from the Dean of Students office will be required.
- Coverage: Lectures 15-20, Labs 9-11, HW 7-8. Material from earlier in the semester, especially on pointers may also be covered on the test.
- Closed-book and closed-notes. **BUT,** you may bring a one-page (8.5x11) double-sided crib sheet without anything written or printed on it that you want. One potential use of this crib sheet is to outline the member functions of the list, vector and map container classes.
- Below are relevant sample questions from previous tests. Solutions will be posted on-line.
- **Important note:** This review does not cover problems on hashing and hash tables. This is new material to CSci II. You can expect to see some coverage of this on the test, however.
- *How to study?*
  - Review lecture notes
  - Review and re-do lecture exercises, lab and homework problems.
  - Do as many of the practice problems below as you can. Focus on the ones that cause you trouble. Practice writing solutions using pencil (or pen) and paper.

Practice Problems

1. Consider the `mergesort` function discussed in detail in lecture. Suppose the vector passed to `mergesort` has 7 items in it. Specify the EXACT set of function calls that are made and the exact order that they are made. In specifying this, you do not need to show the contents of the vector, just the values of `low` and `high` (and `mid`, for calls to `merge`). This is not a question that will appear on the test, but it is instructive about the behavior of merge sort.

   Solution:

```
mergesort:  low = 0, high = 6
mergesort:  low = 0, high = 3
mergesort:  low = 0, high = 1
mergesort:  low = 0, high = 0
mergesort:  low = 1, high = 1
merge:  low = 0, mid = 0, high = 1
mergesort:  low = 2, high = 3
mergesort:  low = 2, high = 2
mergesort:  low = 3, high = 3
merge:  low = 2, mid = 2, high = 3
merge:  low = 0, mid = 1, high = 3
mergesort:  low = 4, high = 6
```

```
mergesort:  low = 4, high = 5
mergesort:  low = 4, high = 4
mergesort:  low = 5, high = 5
merge:  low = 4, mid = 4, high = 5
mergesort:  low = 6, high = 6
merge:  low = 4, mid = 5, high = 6
merge:  low = 0, mid = 3, high = 6
```

2. Consider the public stack interface.

```
template <class T>
class stack {
public:
  stack();
  stack( stack<T> const& other );
  ~stack();
  void push( T const& value );
  void pop( );
  T const& top( ) const;
  int size();
  bool empty();
};
```

In Lab 10 you implemented the stack using a `std::vector`. In this question you are not allowed to use either a `std::vector` or a `std::list`. Instead you are to implement the stack using a dynamically-allocated array. To answer the question, show what private member variables are needed and provide the implementation of the default constructor, the destructor, `stack<T>::push`, and `stack<T>::pop`. All operations should be as efficient as possible.

Solution:

```
template <class T>
class stack {
public:
  stack() : arr(0), size(0), alloc_size(0)
  { }

  stack( stack<T> const& other );

  ~stack()
  {
    delete [] arr;
  }

  void push( T const& value )
  {
    if ( size == alloc_size )
      {
```

```
        int alloc_size *= 2;
        if ( alloc_size < 2 ) alloc_size = 2;  // could be 1 or 3 or ...
        T * new_arr = new T[alloc_size];
        for ( int i=0; i<size; ++i )  new_arr[i] = arr[i];
        delete [] arr;
        arr = new_arr;
      }
    arr[size] = value;
    ++ size;
  }

  void pop( )
  {
     -- size;
  }

  T const& top( ) const;
  int size();
  bool empty();
private:
  T * arr;
  int size;
  int alloc_size;
};
```

3. Suppose that a monster is holding you captive on a computational desert island, and has a large file containing double precision numbers that he needs to have sorted. If you write correct code to sort his numbers he will release you and when you return home will be allowed to move on to DSA. If you don't write correct code, he will eventually release you, but only under the condition that you retake CS 1. The stakes indeed are high, but you are quietly confident — you know about the standard library sort function. (Remember, you are supposed to have forgotten all about bubble sort.) The monster startles you by reminding you that this is a computational desert island and because of this the only data structure you have to work with is a queue.

   After panicking a bit (or a lot), you calm down and think about the problem. You realize that if you maintain the values in the queue in increasing order, and insert each value into the queue one at a time, then you can solve the rest of the problem easily. Therefore, you must write a function that takes a new double, stored in `x`, and stores it in the queue. Before the function is called, the values in the queue are in increasing order. After the function ends, the values in the queue must also be in increasing order, but the new value must also be among them.

   Here is the function prototype.

   ```
   void insert_in_order( double x, queue<double>& q )
   ```

   You may only use the public queue interface (member functions) as specified in lab. You may use a second queue as local variable scratch space or you may try to do it in a single queue

(which is a bit harder). Give an "O" estimate of the number of operations required by this function.

Solution: Here's the version with a scratch queue

```
void insert_in_order( double x, queue<double>& q )
{
  if ( q.empty() )
    q.push( x );
  else
    {
      queue<double> temp( q );           // copy q;
      while ( !q.empty() ) q.pop();      // empty q out

      while ( !temp.empty() && x > temp.front() )
        {
          double item = temp.front();
          temp.pop( );
          q.push( item );
        }

      q.push( x );    // insert x in its proper position

      while ( !temp.empty() )
        {
          double item = temp.front()
          temp.pop( );
          q.push( item );
        }
    }
}
```

This function requires $O(n)$ operations. Copying the queue initially and emptying the queue requires $O(n)$ time each. The second and third while loops, combined, touch each entry in the queue and therefore require $O(n)$ operations. Since none of these loops are nested we add the results and get $O(n)$ time overall.

Here's a version without a scratch queue:

```
void insert_in_order( int x, queue<double>& q )
{
  int n = q.size();
  int position = 0;

  //  Find the position for x in the queue, copying all values
  //  less than x to the back of the queue

  while ( position < n && x < q.front() )
    {
```

```
      q.push( q.front() );    //  copy the front to the back
      q.pop();                //  remove the front
      ++ position;
    }

  q.push( x );                //  put x in position

  //  The first n-position entries on the queue haven't been
  //  touched and are greater than or equal to the value
  //  stored in x.  They need to move to the back of the queue

  int i = position;
  while ( i < n )
    {
      q.push( q.front() );    //  copy the front to the back
      q.pop();                //  remove the front
      ++ i;
    }
}
```

The two while loops, combined touch each entry in the queue once and therefore require $O(n)$ operations. Since none of these loops are nested we add the results and get $O(n)$ time overall.

4. For this question and the next few, consider the following tree node class

```
template <class T>
class TreeNode {
public:
  TreeNode() : left(0), right(0) {}
  TreeNode(const T& init) : value(init), left(0), right(0) {}
  T value;
  TreeNode* left;
  TreeNode* right;
};
```

Write a function to find the largest value stored in a binary search tree of integers pointed to by `TreeNode<int>* root`. Write both recursive and non-recursive versions.

Solution: Recursive version:

```
int FindLargest( TreeNode<int>* root )
{
  if ( ! root -> right )
    return root -> value;
  else
    return FindLargest( root -> right )
}
```

Non-recursive version

```
int FindLargest( TreeNode<int>* root )
{
  while ( root -> right )
    root = root -> right;
  return root -> value;
}
```

5. Write a recursive function to count the number of nodes stored in the binary tree pointed to by `TreeNode<T>* root`.

   Solution:

```
int Count( TreeNode<T>* root )
{
  if ( ! root )
    return 0;
  else
    return 1 + Count( root -> left ) + Count( root -> right );
}
```

6. Write a new member function of the `cs2set<T>` class called `to_vector` that copies all values from the binary search tree implementation of the set into a vector. The resulting vector should be increasing order. You may assume the vector is empty at the start. The function prototype should be

```
template <class T>
void cs2set<T>::to_vector(  vector<T>& vec );
```

   Solution:

```
template <class T>
void cs2set<T>::to_vector(  vector<T>& vec )
{
  to_vector( this->root_, vec );
}


template <class T>
void cs2set<T>::to_vector( TreeNode<T>* p, vector<T>& vec )
{
   if ( p )
     {
       to_vector( p->left, vec );
       vec.push_back( p->value );
       to_vector( p->right, vec );
     }
}
```

7. Write a function called `Trim` that removes all leaf nodes from a tree, but otherwise retains the structure of the tree. Hint: look carefully at the way the pointers are passed in the `insert` and `erase` functions.

   Solution:

```
template <class T>
void Trim( TreeNode<T> *& root )   //  Passing by reference is crucial here
{
  if ( root ) //  Only do something for non-empty trees
    {
      if ( !root->left && !root->right )  // Leaf
        {
          delete root;
          root = 0;     // This sets the appropriate pointer in the parent node to 0.
        }
      else
        {
          Trim( root->left );
          Trim( root->right );
        }
    }
}
```

8. Write a function that takes a set of strings and returns a pointer to an array containing the strings in the set that have exactly 4 characters. The function must also return the number of such strings so that the size of the newly constructed array is known. You may assume there is at least one such string. Here is the prototype, with `arr` being the reference to the array pointer and `n` being the number of strings with 4 characters.

```
void four_character_strings(const set<string>& values,
                            string* & arr, int& n);
```

   Solution:

```
void four_character_strings(const set<string>& values,
                            string* & arr, int& n)
{
   set<string>::const_iterator p;

   // count the number of four letter strings
   n = 0;
   for (p = values.begin(); p != values.end(); p++)
     if (p->length() == 4) n++;

   // allocate space for n strings
   arr = new string[n];
```

```
  // go through a 2nd time to store the strings
  int i = 0;
  for (p = values.begin(); p != values.end(); p++)
    {
      if (p->length() == 4)
        {
          arr[i] = *p;
          i++;
        }
    }
}
```

9. A *word ladder* is a sequence of words that connect a source and target word such that each neighboring pair of words in the sequence differs by exactly one character. For example, here is a word ladder between the words *"sea"* and *"oil"*:

   ```
   sea tea tee tie til oil
   ```

   The characters can only be replaced one character at a time — they cannot be rearranged. Furthermore, each word must appear in the provided dictionary and words cannot be repeated in the ladder.

   (a) First, write a function `next_word` that enumerates all possible words that can be adjacent to an input word, `current`. The function takes a second parameter, `dictionary`, which is simply the set of all valid words that may appear in a word ladder. The function returns a vector of the possible next words. Here is the prototype for your function:

   ```
   vector<string> next_word(const string &current,
                            const set<string> &dictionary);
   ```

   Solution:

   ```
   vector<string> next_word(const string &current,
                            const set<string> &dictionary)
   {
     vector<string> answer;
     for (int i = 0; i < current.size(); i++)
       {
         for (int j = 0; j < 26; j++)
           {
             string next = current;
             next[i] = char('a'+j);
             if (next != current && dictionary.find(next) != dictionary.end())
               answer.push_back(next);
           }
       }
     return answer;
   }
   ```

(b) (Challenging!) Now write a *recursive function* `word_ladder` that uses `next_word` and performs a brute force search to find the shortest ladder between the source and target words. For example, here is code to print the ladder between *"sea"* and *"oil"*:

```
set<string> dictionary;
// dictionary initialization omitted
vector<string> current_ladder, shortest_ladder;
current_ladder.push_back("sea");
word_ladder(current_ladder, "oil", dictionary, shortest_ladder);
for (int i = 0; i < shortest_ladder.size(); i++)
  cout << shortest_ladder[i] << " ";
cout << endl;
```

Solution:

```
bool in_vector(const vector<string> &v, const string &word) {
  for (int i = 0; i < v.size(); i++)
    if (v[i] == word) return true;
  return false;
}

void word_ladder(vector<string> &path, const string &target,
                 const set<string>&dictionary,
                 vector<string> &shortest)
{
  string word = path.back();

  // stop checking this path if it isn't shorter
  if (!shortest.empty() && path.size() >= shortest.size())
    return;

  // found a new shortest path
  if (word == target) {
    shortest = path;
    return;
  }

  // find all possible next words
  vector<string> next_choices = next_word(word,dictionary);
  for (int i = 0; i < next_choices.size(); i++)
    {
      if (in_vector(path,next_choices[i])) continue;

      // if it's not already in the path, add it and recurse
      path.push_back(next_choices[i]);
      word_ladder(path,target,dictionary,shortest);
      path.pop_back();
    }
}
```