

# Computer Science II — CSci 1200

## Test 3 — Overview and Practice

### Overview

- Test 3 will be held **Friday April 18, 2008 2:00-3:45pm, 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 14-20, Labs 8-11, HW 7-8. Material from earlier in the semester, especially on maps may also be covered on the test.
- The test will be closed-book and closed-notes. We will provide a single sheet, attached to the test, summarizing some of the syntax from `std::map`, `std::list` and `std::set`.
- The test questions will be drawn from these questions, often with minor modifications, **and** from the lecture, homework and lab problems. Solutions to many of these questions will be provided a day or two before the exam.
- *How to study?*
  - Work through the sample questions, writing out solutions! Redo the questions on maps from the Test 2 practice. You are welcome (and encouraged) to do this with other students.
  - Review and re-do lecture exercises, lab and homework problems.
  - Identify the problems that cause you difficulty and review lecture notes and background reading on these topic areas.

### Questions

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.
2. Write an efficient function that merges two **unordered** vectors of integers to create a single vector that has only the values that appear in both vectors. This single vector should have no duplicates. As an example, if the vectors contain the values (one vector per line)

```
8, 11, -1, 14, 89, 27, 8, 11, 34, 17, 8, 17
-1, 12, 27, 8, 34, 17, 55, 45, 8, 17
```

Then the resulting vector should contain the values

```
-1, 8, 17, 27, 34
```

Much less credit will be given for a  $O(N^2)$  solution, where  $N$  is the length of the vectors, but give such a solution if it is the only one you can think of. You may specify the function prototype however you wish.

3. 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.

4. 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.

5. 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.

- 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.
- Suppose you are given a vector that is known to contain many duplicate values. To illustrate, there may be 1,500 values, but only 75 of them are distinct. You could use `std::sort` to sort this vector, but a more efficient means is to use a `std::map<T,int>`. Implement a sort function based on this idea. The prototype is

```

template <class T>
void map_sort( vector<T> & v )

```

For example, if the vector contains the following integers before the call to `map_sort`,

```
6, 10, 3, 7, 6, 11, 2, 3, 6, 10, 11, 6, 3
```

then after the call it should contain

```
2, 3, 3, 3, 6, 6, 6, 6, 7, 10, 10, 11, 11
```

- Write a function that takes a pointer to the root of a binary search tree and two values, `x0` and `x1`. The function should return the number of values stored in the binary search tree that are greater than or equal to `x0` and less than or equal to `x1`. The function should be as efficient as possible, which means it should not visit all of the nodes and test their values unless it is absolutely necessary to do so (which would happen if all values in the tree were in the range `[x0..x1]`). Here is the function prototype:

```

template <class T>
int count( TreeNode<T>* root, T const& x0, T const& x1 )

```

You may assume type `T` has any necessary comparison operators defined on it.

- 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);
```

- (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;
```

10. Draw all valid three-node binary search trees containing the values 2, 5, 9.
11. Is the following a good hash function? Why or why not?

```
unsigned int test3_hash( const string& key, int N )
{
    unsigned int value = 1;
    for ( int i=0; i<key.size(); i += 2 )
        value = value + (key[i]*13);
    return value % N;
}
```

12. Write a function that determines if the values stored on a stack form a palindrome. Recall that a palindrome is a sequence that is the same when scanned forward and scanned backward. As examples, the following three sequences of characters are all palindromes

```
otto
hannah
able was I ere I saw elba
```

If any one of them was stored on a stack of chars your function should return true. The problem with using a stack is that it can not be “scanned” forward and backward (e.g. using iterators) in the sense that a vector or a list may be scanned. The only container classes your function may use are **other stacks**, and only the public interface of the stack class may be used:

```

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();
};

```

You may assume that an `operator==` is defined for objects of type `T`. To receive full credit your solution must run in time  $O(N)$ , where  $N$  is the number of items on the stack. Your function prototype must be

```

template <class T>
bool is_palindrome( const stack<T>& s )

```

13. Suppose we implemented a hash table as a vector of vectors of keys instead of a vector of lists of keys. In other words the declaration of `hash_set` includes the member variables

```

template <class KeyType, class HashFunc>
class hash_set {
private:
    std::vector< std::list<KeyType> > m_table; // actual table
    HashFunc m_hash; // hash function
    unsigned int m_size; // number of keys stored in the table
public:

```

The iterator class inside the `hash_set` class starts with

```

class iterator {
public:
    friend class hash_set; // allows access to private variables
private:
    hash_set* m_hs; // pointer to the hash_set object
    int m_index; // which entry in m_table
    int m_loc; // which location in m_table[m_index];

```

- (a) The iterator class includes the following operator:

```

    iterator & operator--( )
    {
        this->prev();
        return *this;
    }

```

Write the iterator class member function `prev` which this operator depends on. You may write it as though it is implemented inside the class declaration (and therefore you need not specify class scope in your function declaration).

- (b) Write the `insert` member function of the `hash_set` class, again assuming that it is implemented inside the class declaration.
- (c) Rewrite the `resize_table` member function for this case. Recall that its prototype is

```
// resize the table with the same values but a
void resize_table( unsigned int new_size )
```

and that `operator()` of the `m_hash` takes an object of type `KeyType` and returns a non-negative integer.

14. In hashing using open addressing, when a collision occurs during an insert operation, other locations in the table are checked to see if either the key is already there or to find a location to insert the key. The simplest method is *linear probing* where, if  $i$  is the hashed table location for the key, then locations  $i$ ,  $(i+1)\%N$ ,  $(i+2)\%N$ , ...,  $(i+N-1)\%N$  are checked in order. When an empty location is found, the key is inserted. When the key is already there, checking stops. Checking also stops (with an unsuccessful insert) when all  $N$  possible locations have been checked. This process gets a bit more complicated when keys may be erased. In this case, the insert operation must skip over “erased” locations when determining if a key is already there, but must use the first “erased” or “empty” location found when looking for a location to insert. Erase must mark locations as “erased”. Find must ignore erased locations.

- (a) Consider a hash table of size 6, and a hash function where

```
f("ant") = 3,  f("cat") = 5, f("elf") = 4,  f("bat") = 5,
f("dog") = 4,  f("fish") = 3
```

Show the contents of the table after inserting "dog", "ant", "elf", "cat" and "fish".

- (b) Now show the contents of the table after subsequently erasing "elf" and "dog" followed by inserting "elf" and "bat".

15. This question considers a simplified `hashset` class that uses open addressing based on linear probing. It has no iterators. Here is the class prototype

```
template < typename KeyType, typename HashFunc >
class hash_set {
private:
    typedef enum { EMPTY, ERASED, FULL } status_type;
    class entry {
public:
        entry() : status(EMPTY) {}
        status_type status;           // EMPTY, ERASED or FULL
        KeyType key;                 // The actual value stored
        unsigned int hash_value;     // reuse when resizing the table
    };

private:
    std::vector< entry > m_table;    // actual table
    HashFunc m_hash;                // hash function
    unsigned int m_size;             // number of keys stored in the table
};
```

```
public:
    // Return true if the value was not already there and was inserted
    bool insert( KeyType const& value );

    // Return true if the value was there and was removed
    // Set status of value's location to ERASED
    bool erase( KeyType const& value );

    // Return true if the value is in the hash set.
    bool find( KeyType const& value ) const;
};
```

Implement the `insert`, `erase` and `find` functions. Do not worry about the possibility of resizing the table, even though that would be important in practice.