Test 2: CPS 100

Owen Astrachan Jeffrey Forbes

March 31, 2004

Name: _____

Login: _____

Honor code acknowledgment (signature) _____

	value	grade
Problem 1	20 pts.	
Problem 2	22 pts.	
Problem 3	22 pts.	
TOTAL:	64 pts.	

This test has 13 pages, be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts 75 minutes.

In writing code you do not need to worry about specifying the proper `#include` header files. Assume that all the header files we've discussed are included in any code you write.

Some common recurrences and their solutions.

A	`T(n) = T(n/2) + O(1)`	$O(\log n)$
B	`T(n) = T(n/2) + O(n)`	$O(n)$
C	`T(n) = 2T(n/2) + O(1)`	$O(n)$
D	`T(n) = 2T(n/2) + O(n)`	$O(n \log n)$
E	`T(n) = T(n-1) + O(1)`	$O(n)$
F	`T(n) = T(n-1) + O(n)`	$O(n^2)$
G	`T(n) = 2T(n-1) + O(1)`	$O(2^n)$

PROBLEM 1 : (***Miscellaneous Stuff (20 points)***)

A. The postfix expression `2 3 + 4 *` has the value 20. What is an equivalent infix expression and why is postfix possibly a better representation than infix for arithmetic expressions? Be brief.

B. Chaining as a collision-resolution strategy is more widely-used in implementating hash tables than linear probing. Briefly, why is chaining more used in practice?

C. The code below sorts a vector. What is its big-Oh complexity and why?

```
void sort(tvector<string>& v)
{
    tpqueue<string> pq;
    for(int k=0; k < v.size(); k++){
        pq.insert(v[k]);
    }
    for(int k=0; k < v.size(); k++){
        pq.deletemin(v[k]);
    }
}
```

D. In C++ the standard library call `sort` is based on a different algorithm than the standard library call `stable_sort`. Provide a reason to prefer a stable sort to the faster, but unstable `sort` algorithm.

E. Describe the purpose/use of the struct `Greater` in the code below.

```
struct Greater
{
    bool operator()(const string& lhs, const string& rhs) {
        return lhs > rhs;
    }
};

int main()
{
    Greater gr;
    tvector<string> vec;
    // fill vec with values
    sort(vec.begin(), vec.end(), gr);
}
```

PROBLEM 2 : (***Mind your P's and Q's (22 points)***)

Part A (4 points)

The code below is partially complete, it is intended to merge two sorted queues into one sorted queue. Complete the function so that works as specified by writing code in the loop body.

```
tqueue<string> mergeq(tqueue<string>& a, tqueue<string>& b)
// pre: a and b are sorted (first item in each queue is the smallest)
// post: a and b are empty, queue returned is sorted and contains
//       elements from original a and b
// requirements: runs in O(n) time for n-element queues.
{
    tqueue<string> result;
    string item;
    while (a.size() != 0 && b.size() != 0){

      // write code to set item to the minimum of
      // of the elements at the front of queues a and b









        result.enqueue(item);
    }
    while (a.size() != 0){     // elements left in a?
        a.dequeue(item);
        result.enqueue(item);
    }
    while (b.size() != 0){     // elements in left in b?
        b.dequeue(item);
        result.enqueue(item);
    }
    return result;
}
```

Part B (4 points)

Code to sort the elements of a queue is shown below. Explain in general terms how the sort works and provide the asymptotic/big-Oh complexity of the code. Justify your answer.

```
void qsort(tqueue<string>& q)
{
    if (q.size() <= 1) return;

    tqueue<string> a, b;
    string item;
    int size = q.size();
    for(int k=0; k < size; k++){
        q.dequeue(item);
        if (k % 2 == 0){
            a.enqueue(item);
        }
        else {
            b.enqueue(item);
        }
    }
    qsort(a);
    qsort(b);
    q = mergeq(a,b);
}
```

Part C (4 points)

The code below prints every element in a queue. The storage/memory used (in addition to the queue) is $O(1)$. Explain how this code works but why it's not possible to print the contents of a stack using this approach, and why printing a stack without destroying its contents requires $O(n)$ storage. Be brief.

```
void print(tqueue<string>& q){

    int size = q.size();
    string item;
    for(int k=0; k < size; k++){
        q.dequeue(item);
        cout << item << " ";
        q.enqueue(item);
    }
    cout << endl;
}
```

Part D (6 points)

Write function `filterQ` that removes some elements from a queue of strings. The elements that remain in the queue should be in the same relative order they were before the removal process executes.

To check whether an element is removed, your code will call the Predicate object `pred`'s method *satisfies*. If the *satisfies* method returns true, the element is removed.

For example, if the queue `qu` is represented by

```
"juice", "mom", "racecar", "peace", "radar", "police"
```

then the call `filterQ(qu,palindrome)` would change the queue `qu` to be the following:

```
"juice", "peace", "police"
```

assuming `palindrome` returns true precisely when its string parameter is a palindrome (a word reading the same backward as forward).

The `Predicate` class is defined as:

```
struct Predicate
{
    virtual bool satisfies(const string& arg) = 0;
};
```

Complete the function `filterQ` below. You can allocate O(1) additional storage in writing `filterQ`, you cannot create another queue or vector or stack or any structure in which more than O(1) elements from the queue being filtered are stored.

```
void filterQ(tqueue<string>& q, Predicate& pred)
{









}
```

Part E (4 points)

Write the definition for a struct/class named `VowelStarter` that can be used to filter all strings from a queue that start with a vowel. A vowel is 'a', 'e', 'i', 'o', 'u'. Assume strings are all lowercase letters. For example, the code below should remove all strings that start with a vowel from the queue `sq`.

```
VowelStarter vs;
tqueue<string> sq;
// fill sq with values

filterQ(sq,vs);
// now sq contains no strings that begin with vowels
```

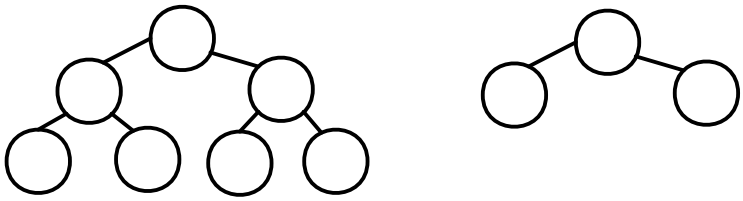
In writing your class/struct you may find this code snippet useful.

```
void snippet(const string& s)
{
  string vowels = "aeiou";
  if (vowels.find(s[0]) != string::npos) {
     cout << "begins with vowel" << endl
  }
}
```

Write the struct/class definition for `VowelStarter` below.

PROBLEM 3 : (*Trees*)

For the purposes of this problem, a *full, complete* binary tree with n levels has 2^{n-1} leaf nodes and, more generally, 2^{k-1} nodes at level k where the root is at level 1, the root's two children are at level 2, and so on. The diagram below shows two such trees, the tree on the left is a level-3 full, complete tree and the tree on the right is a level-2 full, complete tree.



In this problem tree nodes have parent pointers. The declaration for such tree nodes follows.

```
struct TreeNode
{
    string info;
    TreeNode * left, * right, * parent;
    TreeNode(const string& s,  TreeNode * lptr, TreeNode * rptr, TreeNode * pptr)
        : info(s),left(lptr),right(rptr),parent(pptr)
    { }
};
```

Part A (6 points)

Write the function *makeComplete* that returns a full-complete binary tree with the specified number of levels. The call `makeComplete(3,0)` should return a tree such as the one above on the left; `makeComplete(1,0)` should return a single-node tree. The root of the tree has a NULL/0 parent; all other tree nodes should have correct parent pointers. Use the empty string `""` for the `info` value when creating nodes.

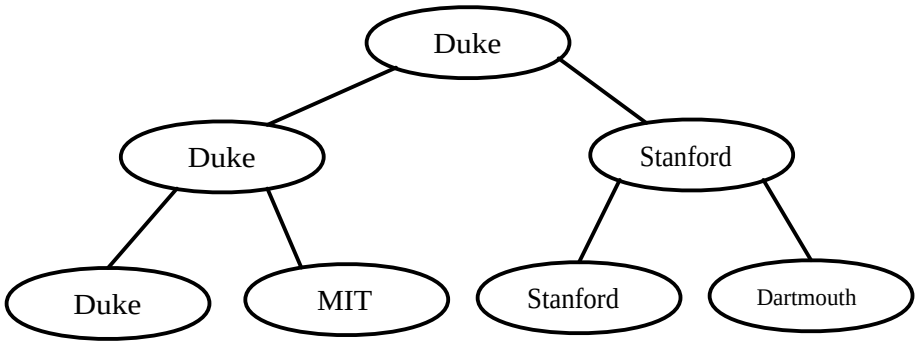
```
TreeNode * makeComplete(int level, TreeNode * parent)
// pre: 1 <= level, parent points to parent of node created at this level
// post: returns a full, complete tree with # levels specifed by level
{









}
```

Part B (4 points) What is the recurrence relation, and big-Oh solution for the code you wrote for part A for an n-level tree? Justify your answer

Part C (6 points)

For this problem you'll treat the full complete tree like a *tournament tree*. In a tournament tree, leaf-value store names, or more generically items. Each internal node stores the winner of the values stored in its two children (since the tree is complete, all non-leaf/internal nodes have two children).

For example, the tree below shows a hypothetical tournament tree with the leaf value storing the names of schools competing in a computer programming contest tournament.



Assume you have a full, complete binary tree, e.g., as would be returned by the function `makeComplete` from Part A. Write the function `assign2leaves` that assigns values in a stack passed to the function to the leaves. For example, suppose the stack is created by the code below:

```
tstack<string> names;
names.push("Dartmouth");
names.push("Stanford");
names.push("MIT");
names.push("Duke");
```

then the call `assign2leaves(root, names)` where root is the root of a level-3 full, complete tree should assign values as shown above. Note that `"Duke"` is the value at the top of the stack and is stored as the left-most leaf of the leaves. Your code should do this – this means also that the right-most leaf gets the first value pushed onto the stack.

(continued)

Complete the function below.

```
void assign2leaves(TreeNode* root, tstack<string>& names)
// pre: names has at least as many values as their are leaves
//      in the full, complete tree pointed to by root.
// post: values in names have been assigned to leaf-nodes.
//       names will decrease in size each time a value is added to a leaf
{









}
```

Part D (6 points)

In this problem you'll assign winners to the internal nodes of a tree. Assume all leaf nodes have been assigned values, e.g., as in the tournament tree diagrammed previously. You can also assume that a map makes it possible to look up the winners of any pair of teams. For example, here's code to determine the winner of a match between `"Duke"` and `"MIT"`. This code shows syntactically and semantically how to use the map that stores the winner of a contest between any two teams.

```
string DukeMITwinner(tmap<pair<string,string>, string> * winnerMap)
{
    pair<string,string> pp = make_pair("Duke", "MIT");
    return winnerMap->get(pp);
}
```

Write the function `assignwinners` whose header is given below. The function is passed the root of a tournament tree like the one diagrammed above. Assume all the leaf values have been filled in. The function should assign values to internal nodes so that each internal node stores the winner of the match played between the internal node's children. The winner is determined by using the map parameter.

```
void assignwinners(TreeNode * root,
                   tmap<pair<string,string>, string> * winnerMap)
// pre: leaf values of tournament tree with root have values assigned
// post: internal nodes of tournament tree have values assigned
//       consistent with winner information represented in winnerMap
{









}
```