Test 2: CPS 100

Owen Astrachan

April 4, 2002

Name: ___________________________________________ (1 pt)
Login: ___________________________

Honor code acknowledgment (signature) ___________________________

<table>
<thead>
<tr>
<th>Problem</th>
<th>Value</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1</td>
<td>10 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 2</td>
<td>12 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 3</td>
<td>8 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 4</td>
<td>10 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 5</td>
<td>29 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 6</td>
<td>10 pts.</td>
<td></td>
</tr>
<tr>
<td>TOTAL:</td>
<td>80 pts.</td>
<td></td>
</tr>
</tbody>
</table>

This test has 10 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:

\[
\begin{align*}
T(n) &= T(n/2) + O(1) \quad O(\log n) \\
T(n) &= T(n/2) + O(n) \quad O(n) \\
T(n) &= 2T(n/2) + O(1) \quad O(n) \\
T(n) &= 2T(n/2) + O(n) \quad O(n \log n) \\
T(n) &= T(n-1) + O(1) \quad O(n) \\
T(n) &= T(n-1) + O(n) \quad O(n^2)
\end{align*}
\]
PROBLEM 1: (I’ll huff and I’ll puff and I’ll ... (10 pts))

The Huffman tree below is used for this problem.

Part A (4 points)
Using this tree, decode the following Huffman-encoded bitstream
1111001100110110111

Part B (3 points)
Briefly, what was the reason for using the PSEUD_EOF character in the Huffman coding program?

Part C (3 points)
The `tpqueue` class uses a heap for implementing a priority queue. In a heap, insert is \(O(\log n)\) and deletion is \(O(\log n)\). Suppose you replace this heap-implementation in your Huffman compression program with an implementation that uses an unsorted vector — where the complexity of insert is \(O(1)\) and the complexity of deletion is \(O(n)\). You will **not** notice any substantial change in the time it takes your program to run on nearly all files. Why?
PROBLEM 2: (AVL (12 points))

Part A: 4 points
The strings below are inserted, in the order shown, into an initially empty AVL tree. Draw the resulting tree. Note: there should be no rotations necessary when inserting these strings.
"Monkey", "Giraffe", "Tiger" "Cheetah", "Kangaroo",

Part B: 4 points
Using the tree you drew in Part A, insert the string "Donkey" into the tree so that the tree remains an AVL tree. Draw the resulting tree.

Part C: 4 points
Using the tree you drew in Part A, insert the string "Lion" into the tree so that the tree remains an AVL tree. Draw the resulting tree.
PROBLEM 3:  

(Like finding a needle in a Haystack (8 points))

The code below finds the smallest value in a stack, but empties the stack in finding the minimal value.

```c++
int min(tstack<int>& s) 
    // pre: s.size() >= 1
    // post: returns smallest element in stack
    // stack is empty after function executes
    { 
        int min = s.top();
        s.pop();
        while (s.size() > 0) {
            int value;
            s.pop(value);
            if (value < min) {
                min = value;
            }
        }
        return min;
    }
```

Write a version of `min` so that the stack passed in contains the same elements in the same order after the function executes. You may only define int variables in writing `min`, you may not define stacks, queues, vectors, etc. You’ll need to make the function recursive to solve this problem. Note: the stack will be changed during the execution of `min`, but it should be restored to its original contents before `min` finishes.

```c++
int min(tstack<int>& s) 
    // pre: s.size() >= 1, stack = (s1, s2, ..., sn)
    // post: returns smallest element in stack, stack = (s1, s2, ..., sn)
    { 
        
    }
```
PROBLEM 4:  (Heap o’ Trouble (10 points))

Assume min-heaps are implemented using a vector as we discussed in class so that the root of the heap has index 1 and the children of the value stored at index $k$ have indexes $2 \times k$ (left-child) and $2 \times k + 1$ (right-child).

Here is part of the class declaration for a heap that stores strings.

```cpp
class StringHeap
{
    // public declarations not shown
    private:

        std::vector<std::string> myList;
        int mySize; // number of elements in heap
};
```

The public member functions `StringHeap::getMin()` and `StringHeap::getMax()` follow (note: `getMax` is not a standard priority queue/heap function.)

```cpp
string StringHeap::getMin() const {
    // post: returns minimal element in heap
    return myList[1];
}

string StringHeap::getMax() const {
    // post: returns maximal element in heap
    int maxIndex = 1;
    for (int k=maxIndex+1; k <= mySize; k++) {
        if (myList[k] > myList[maxIndex]) maxIndex = k;
    }
    return myList[maxIndex];
}
```

Part A (3 points)
In a heap with $n$ elements, what is the big-Oh complexity of `StringHeap::getMin()` and why?

Part B (3 points)
In a heap with $n$ elements, what is the big-Oh complexity of `StringHeap::getMax()` and why?

Part C (4 points)
Describe a simple change to the implementation of `StringHeap::getMax()` that will make the function roughly twice as fast although the big-Oh complexity will remain the same.
PROBLEM 5: (Multiset (29 points))

A multiset is similar to a set but stores duplicates. The code below on the left generates the output shown on the right and illustrates some of the multiset functions. `MultiSet::unique` returns the number of different elements in a MultiSet (ignoring duplicates). `MultiSet::size` returns the total number of elements in a MultiSet (including duplicates). `MultiSet::count` returns the number of occurrences of a specific string.

code

```cpp
MultiSet multi;
multi.insert("apple");
multi.insert("cherry");
multi.insert("apple");
multi.insert("cherry");
multi.insert("apple");

cout << multi.count("apple") << endl;
cout << multi.count("cherry") << endl;
cout << multi.count("orange") << endl;
cout << "total = " << multi.size() << endl;
cout << "unique = " << multi.unique() << endl;
```

output

```
 3
 2
 0
total = 5
unique = 2
```

Suppose member functions `MultiSet::insert` and `MultiSet::size` are implemented as follows where `myList` is a vector of strings.

```cpp
void MultiSet::insert(const string& s) {
    myList.push_back(s);
}
int MultiSet::size() const {
    return myList.size();
}
```

Part A (4 pts)

Given these implementations, what is the big-Oh complexity of the function `MultiSet::count()`. Justify your answer.

Part B (4 pts)

Given these implementations, describe a method for implementing `MultiSet::unique` that runs in \(O(n \log n)\) time. Justify why your solution is \(O(n \log n)\). None of the other methods should change, but `MultiSet::unique` should run in \(O(n \log n)\) time.
The code below is a partial implementation of a MultiSet class implemented using an HMap object. This implementation is used for the rest of this question (this code is duplicated on the separate code handout). This is a different implementation than used in parts A and B.

class MultiSet
{
public:
    MultiSet();
    void insert(const string& s);       // add element to MultiSet
    int count(const string& s) const;  // returns # occurrences of s
    int size() const;                  // returns total # elements
    int unique() const;                // returns # unique elements

private:
    tmap<string,int> * myMap;
};

MultiSet::MultiSet()
    : myMap(new HMap<int>(10001))
// post: this MultiSet is empty
{
}

int MultiSet::unique() const
// post: returns # unique elements (repeats not counted) in this MultiSet
{
    return myMap->size();
}

void MultiSet::insert(const string& s)
// post: s is added to this MultiSet
{
    if (myMap->contains(s)) myMap->get(s)++;
    else                    myMap->insert(s,1);
}

Part C (4 pts)
Given this implementation write MultiSet::count so that it runs in O(1) time in the average case.

int MultiSet::count(const string& s) const
// post: returns # times s occurs in this MultiSet
//       (returns 0 if s not in this MultiSet)
{
}
Part D (6 pts)
The member function MultiSet::size is implemented as follows:

```cpp
int MultiSet::size() const
// post: returns total # elements (including repeated) in MultiSet
{
    int count = 0;
    Iterator<pair<string,int> > * it = myMap->makeIterator();
    for(it->Init(); it->HasMore(); it->Next()) {
        count += it->Current().second;
    }
    return count;
}
```

Describe modifications to the given MultiSet implementation that permit MultiSet::size to execute in \(O(1)\) time. You must describe and write code for every change needed so that MultiSet::size runs in \(O(1)\) time. None of the other methods should have their running time change. You can write the code on the previous page, or describe the changes in detail below (hint: add something to the private section that will engender changes in the class).

---

Part E (3 points)
If the MultiSet constructor creates a BSTMap instead of an HMap, will the implementations of the MultiSet member functions change? Why?
Part F (8 pts)
The code below creates a map for processing web-logs: files that store information about who visits web pages. The input file is in the format shown where the first string on a line is an Internet Protocol (IP) address and the second string is a URL for a web page visited by the corresponding IP address.

32.101.160.49 http://www.cs.duke.edu/~ola
32.101.160.49 http://www.cs.duke.edu/~ola/ethan
172.157.127.206 http://www.cs.duke.edu/education/courses/cps100/spring02
216.35.116.42 http://www.cs.duke.edu/~stefann
32.101.160.49 http://www.cs.duke.edu/~magda
32.101.160.49 http://www.cs.duke.edu/~ola

Here's the code for reading/storing log information from a file.

    tmap<
tmp<
ifstream input("logfile.txt");
while (input >> ip >> url) {
    if (map->contains(ip)) map->get(ip)->insert(url);
    else {
        MultiSet * ss = new MultiSet();
        ss->insert(url);
        map->insert(ip,ss);
    }
}

Write the function maxUniqueVisits that prints the IP address that visits the most different web pages. If there is more than one such IP address your code can print any one of them. In the sample web log above the IP 32.101.160.49 visits 3 different web pages which is more than the number of pages visited by other IP addresses, so 32.101.160.49 should be printed.

    void maxUniqueVisits(tmap<
    // pre: map stores information about IP -> web-page visits
    // post: IP address that visits the most different web pages printed
    {

}
PROBLEM 6:  *(Encore une fois (10 points))*

A map of strings to ints associates an 8-bit/chunk (the value) with its Huffman encoding (the key). For example, "10" could be mapped to 's', and "11" mapped to 't' as shown in three below.

```
    d
   / \\
  s   t
```

Complete the body of the function `map2tree` below that returns a pointer to the root of the Huffman tree representing the encodings in the map parameter. You will most likely find it useful to write a helper function called by `map2tree`, there’s a blank page at the end of this test. You don’t have to use the call of the helper function shown, but it’s a good start. However, you’re welcome to change completely the body of `map2tree`. For example, given the mappings below, the tree above should be returned.

```
"0"  'd'
"10" 's'
"11" 't'
```

```c
struct TreeNode
{
    int info;
    TreeNode * left;
    TreeNode * right;
    TreeNode(int value, TreeNode * lptr, TreeNode * rptr)
    : info(value), left(lptr), right(rptr)
    {
    }
};

void mapTreeHelper(TreeNode * t, const string& path, int value);
// helper function prototype

TreeNode * map2tree(tmap<string, int> * map)
// pre: map represents string->int encodings for a Huffman tree
//      map contains at least 2 (string,int) pairs
// post: returns pointer to root of Huffman tree representing the encodings
{

    Iterator<pair<string, int>> * it = map->makeIterator();

    for(it->Init(); it->HasMore(); it->Next()) {
        mapTreeHelper( ,
                       ,
                       );
    }
}
```