Test 2: CPS 100

Owen Astrachan        Susan Rodger

November 14, 1995

Name: ________________________________

Honor code acknowledgement (signature) ________________________________

<table>
<thead>
<tr>
<th></th>
<th>value</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1</td>
<td>6 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 2</td>
<td>32 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 3</td>
<td>8 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 4</td>
<td>6 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 5</td>
<td>20 pts.</td>
<td></td>
</tr>
<tr>
<td>Problem 6</td>
<td>6 pts.</td>
<td></td>
</tr>
<tr>
<td>TOTAL:</td>
<td>72 pts.</td>
<td></td>
</tr>
</tbody>
</table>

This test has 11 pages, be sure your test has them all. Do NOT spend too much time on one question — remember that this class lasts 75 minutes.
Declarations

For all problems on this test you can assume the following declarations have been made (sometimes a type other than int may be used for the info field).

```c
struct Tree  // "standard binary tree declaration"
{
    int info;
    Tree * left;
    Tree * right;
    Tree (int val, Tree * lchild = 0, Tree * rchild = 0)
    {
        info = val;
        left = lchild;
        right = rchild;
    }
};
```
PROBLEM 1:  \((\text{Big-Oh Time: 6 points})\)

Consider the following data structures:

- array in sorted order
- minheap (each node is smaller than its two children)
- binary search tree (not necessarily balanced)

and operations on the data structures:

- Find\((x)\) returns true if \(x\) is present (in the structure), otherwise returns false
- FindMax returns the maximum element

Give the worst case running time (big-Oh) that best describes the running time of each operation.

<table>
<thead>
<tr>
<th></th>
<th>array - sorted order</th>
<th>minheap</th>
<th>binary search tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find((x))</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FindMax()</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
PROBLEM 2: (Lovelier than What?)

Part A (6 points)
Draw the binary search tree that results from inserting the integers 23, 8, 40, 12, 32, 5, 15, 27 (in that order) into an initially empty tree.

Part B (6 points)
Give the preorder and postorder traversals of the tree shown below.

preorder:

postorder:

Part C (2 points)
The function below returns a copy of its tree parameter (See the Tree struct definition on page 2).

```c
Tree * Copy(Tree * t)
// postcondition: returns copy of t
{
    Tree * temp = 0;
    if (t != 0)
    {
        temp = new Tree(t->info,
                        Copy(t->left),
                        Copy(t->right));
    }
    return temp;
}
```

What is the average case complexity of this function (using big-Oh notation)? Briefly justify your answer.
Part D (4 points) Indicate how to modify the function Copy to return a tree that is the mirror-image of the Tree parameter. The trees below are mirror images of each other (all left children in the mirror image are right children in the original tree and vice-versa).

```
Part E (4 points)
Assume that a function NumNodes exists that returns the number of nodes in its Tree parameter.

int NumNodes(Tree * t)
  // postcondition: returns number of nodes in t
The function Copy from Part C is modified so that the info field of each node in the copy returned is replaced by the count of the number of nodes in the tree rooted at t — the function is renamed CopyCount.

Tree * CopyCount(Tree * t)
  // post: returns "counted" copy of t
  {
    Tree * temp = 0;  // holds copy
    if (t != 0)
      {
        temp = new Tree(NumNodes(t),
                        CopyCount(t->left),
                        CopyCount(t->right));
        }
    return temp;
  }
For example, if t is the tree in part B, the call CopyCount(t) returns the tree diagrammed above on the right.
The average case complexity of CopyCount is NOT $O(n)$; what is the complexity and why (briefly justify).
Part F (10 points)
Rewrite CopyCount so that it does have average case complexity \( O(n) \). You may find it useful to use an auxiliary function called by CopyCount. The header for such an auxiliary function is shown below, you do NOT need to use this header but may. You MUST indicate how your auxiliary function is called from CopyCount.

```c
void CCAux(Tree * t, Tree * & copy, int & count)
// postcondition: copy is a "copy count" copy of the tree t
// e.g., all info fields replaced by node count
// count is the number of nodes in tree t (rooted at t)
{
```
PROBLEM 3:  (Double Vision: 8 points)

Write the function `DoubleUp` whose header is given below. `DoubleUp` adds one new node to the tree \( t \) for every node except the root. For each non-root node, a new node is created with the same info field value and the same orientation (e.g., a left child or a right child). The newly created node will have only one child, that child will be the node that generates the newly created copy.

For example, if \( t \) is the tree on the left, `DoubleUp(t)` should modify \( t \) to look like the tree on the right.

![Diagram showing before and after of `DoubleUp` function](image)

Complete the function `DoubleUp` below. Use the `Tree` struct from page 2.

```c
void DoubleUp(Tree * t) // postcondition: modifies t so that it is "doubled up"
{
```

```c
```
PROBLEM 4:  \((Recurring \ Recurrence: \ 6 \ points)\)

Solve the following recurrence relation and give the \(O()\) that best describes the running time. Justify your answer.

\[
\begin{align*}
T(1) &= 1 \\
T(n) &= 2T(n/2) + n^2
\end{align*}
\]
PROBLEM 5: (ginorst stuff)

Part A (10 points)
Suppose a struct Student for students in a class is (partially) declared as below. Each student has a name, a list of grades, the number of grades stored in the list, and a class identification number. In a class of $N$ students, each student is given an ID number in the range $1 \ldots N$, e.g., in a class of 20 students the numbers $1 \ldots 20$ are assigned one per student.

```cpp
struct Student
{
    int classID; // ID number
    string name; // name of student "Jane Doe"
    Vector<int> grades; // list of grades
    int numGrades; // # of items stored in grades
};
```

Write the function SortStudents whose header is given below. SortStudents sorts the information in list so that it is in order from smallest (1) to largest ($N$) student ID number.

**Your solution MUST sort in O(N) time!!**
(hint: in what slot does the student with ID number 16 belong?)

```cpp
void SortStudents(Vector<Student> & list, int numElts)
// precondition: numElts = # of students in list
//               ID numbers are unique and in range 1...numElts
// postcondition: list is sorted into increasing order by
//                student ID number
// performance: O(n)
```
Part B  (10 points)

You must solve the problem of determining if two numbers in an array of $n$ numbers sum to a given number $k$. For example, if the array contains 8, 4, 1, 6, there are numbers that sum to 10 (4 and 6); to 9 (8 and 1); to 16 (use 8 twice); but NOT to 8 (exactly two numbers must be used).

Trying all pairs of numbers yields an $O(n^2)$ algorithm. Instead, you must describe how to:

- implement an $O(n \log n)$ algorithm to solve this problem. You can use the array as storage, but only $O(1)$ other storage for this (e.g., int variables are fine, no vectors). Hint: given $k$ (the number to sum to), if one of the two numbers is determined, the other one is known — you need to find if it’s in the array by searching efficiently.

- implement an $O(n)$ algorithm (average case) by using auxiliary storage to facilitate an efficient search.

In both cases, describe how your algorithm will work (briefly) and justify the running time.
PROBLEM 6:  (*Heapless: 8 points*)
Complete the function *NumLessThan* to determine the number of elements in a minheap that are less than a *key*. A minheap is a heap such that the value in each node is less than the value in the node’s subtrees. A minheap is implemented via a vector with the root at index 1.

In the example below, *NumLessThan(heap, 9, 13)* should return 3 (4, 10, and 6 are less than 13), and *NumLessThan(heap, 9, 8)* should return 1 (only 6 is less than 8).

The worst case running time of your function should be O(k) where k is the number of elements less than the key. (Note: If you want to use recursion, you might want to write an auxiliary function.)

```c++
int NumLessThan(const Vector<int> & heap, int size, int key)
// precondition: size is the number of elements in the heap,
// heap is a minheap
// postcondition: returns the number of elements in the heap
// less than key
```