

# Test 2: CPS 100

Owen Astrachan

Robert Duvall

Jeff Forbes

April 11, 2001

Name: \_\_\_\_\_

Login: \_\_\_\_\_

Honor code acknowledgment (signature) \_\_\_\_\_

|                   | value   | grade |
|-------------------|---------|-------|
| Problem 1         | 18 pts. |       |
| Problem 2         | 20 pts. |       |
| Problem 3         | 8 pts.  |       |
| Problem 4         | 30 pts. |       |
| Problem 5 (extra) | 6 pts.  |       |
| TOTAL:            | 76 pts. |       |

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 50 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.

## Recurrences and Solutions

$$T(n) = T(n/2) + O(1) \quad O(\log n)$$

$$T(n) = T(n/2) + O(n) \quad O(n)$$

$$T(n) = 2T(n/2) + O(1) \quad O(n)$$

$$T(n) = 2T(n/2) + O(n) \quad O(n \log n)$$

$$T(n) = T(n-1) + O(1) \quad O(n)$$

$$T(n) = T(n-1) + O(n) \quad O(n^2)$$

The declaration for binary search tree nodes on this test is:

```
struct TreeNode
{
    string info;
    TreeNode * left;
    TreeNode * right;

    TreeNode(const string& s, TreeNode * lt, TreeNode * rt)
        : info(s), left(lt), right(rt)
    { }
};
```

**PROBLEM 1 :** (*A Picture is Worth a Thousand Words (18 points)*)

**Part A (4 points)**

Build the Huffman tree for the following set of frequencies

|                  |  |    |    |    |    |   |   |
|------------------|--|----|----|----|----|---|---|
| <b>letter</b>    |  | a  | b  | c  | d  | e | f |
| <b>frequency</b> |  | 45 | 13 | 12 | 16 | 9 | 5 |

**Part B (4 points)**

What are the codes, e.g., 100 or 11, associated with the letters below

a \_\_\_\_\_

e \_\_\_\_\_

**Part C (4 points)**

Draw each **AVL tree** that results from inserting the following numbers in order into an initially empty tree. You should draw as many trees as there are numbers, i.e., draw the tree after inserting each number (rotate as necessary).

8, 3, 1, 7

**Part D (6 points)**

Draw each **AVL tree** that results from inserting the following numbers in order into an initially empty tree. You should draw as many trees as there are numbers, i.e., draw the tree after inserting each number (rotate as necessary).

5, 1, 10, 11, 8, 7

**PROBLEM 2 : (Completely Unbalanced (20 points))**

As discussed in class, a tree is **height-balanced** if, for every node  $N$  in the tree, the heights of the left and right subtrees of  $N$  differ by at most one.

We discussed the function `isBalanced` below that returns true if a tree is height-balanced, and false otherwise. The functions called by `isBalanced` are also shown.

```
int max(int a, int b)
{
    if (a > b) return a;
    else      return b;
}

int height(TreeNode * t)
{
    if (t == 0) return 0;
    return 1 + max(height(t->left), height(t->right));
}

bool isBalanced(TreeNode * t)
// post: returns true iff t is balanced
{
    if (t == 0) return true;

    return isBalanced(t->left) && isBalanced(t->right) &&
           abs(height(t->left) - height(t->right)) <= 1;
}
```

**Part A (4 points)**

The complexity of `isBalanced` is  $O(n \log n)$  for a tree of  $n$  nodes. Justify this claim (a recurrence is the best justification).

**Part B (4 points)**

Suppose `isBalanced` is rewritten as follows, draw a tree for which this version of `isBalanced` returns true, but which is **not** height-balanced.

```
bool isBalanced(TreeNode * t)
// post: returns true iff t is balanced
{
    if (t == 0) return true;
    else      return abs(height(t->left) - height(t->right)) <= 1;
}
```

**Part C (12 points)**

Complete the function `auxBalanced` below so that it satisfies its postcondition and so that when used with `isBalanced2` (also below) determines if a tree is height-balanced in  $O(n)$  time.

```
void auxBalanced(TreeNode * t, int & height, bool & balanced)
// post: height is height of tree rooted at t
//       balanced = true if t is height-balanced, else balanced = false
//
{
    if (t == 0) {

    }
    else {

    }
}

bool isBalanced2(TreeNode * t)
// post: returns true iff t is balanced
{
    int height;
    bool balanced;
    auxBalanced(t, height, balanced);
    return balanced;
}
```

**PROBLEM 3 :** (*Deep Stuff (8 points)*)

**Part A (4 points)**

The function `deepLeaf` below returns a pointer to the deepest leaf in a tree (furthest from the root). What is its big-Oh complexity (in the average case where trees are roughly balanced). Justify your answer (a recurrence is the best justification).

```
bool isLeaf (TreeNode * t)
// post: returns true iff both children of t are null
{
    if (t == 0) return false;
    else      return (t->left == 0) && (t->right == 0);
}

TreeNode * deepLeaf(TreeNode * t)
// post: returns pointer to deepest leaf in t
{
    if (t == 0)      return 0;
    else if (isLeaf(t)) return t;

    else if (height(t->left) >= height(t->right))
        return deepLeaf(t->left);
    else
        return deepLeaf(t->right);
}
```

**Part B (4 points)**

Does your answer change for the complexity of `deepLeaf` if the tree isn't roughly balanced? Consider, for example, a tree of  $n$  nodes in which every non-leaf node has no left child and one right child. Justify your answer briefly (a recurrence is the best justification).

**PROBLEM 4 :** (*Balanced but not Stable (30 points)*)

In this question, you'll consider three different problems in which the elements of a vector are distinct integers. The code below solves the problem of finding two integers in the vector that are farthest apart (difference is maximal) in  $O(n^2)$  time.

```
void findMaxDiff(tvector<int> & a, int& first, int& second)
// post: first and second are set to the elements of a
//       that have the greatest difference (farthest apart)
{
    int j,k;
    int max = 0;
    for(j=0; j < a.size(); j++) {
        for(k=j+1; k < a.size(); k++) {
            if (abs(a[j] - a[k]) > max) {
                first = a[j];
                second = a[k];
                max = abs(first - second);
            }
        }
    }
}
```

**Part A (4 points)**

A colleague suggests an alternative algorithm: sort the vector using merge sort; the two integers farthest apart are the first and last integers in the sorted vector. What is the running time of this algorithm? Justify your answer briefly.

**Part B (4 points)**

Describe a method for solving the first problem in  $O(n)$  time. Don't write code, describe a method and the justification that it's  $O(n)$ .

### Part C (4 points)

Describe a method for solving a related problem of finding the two integers closest together (difference is minimal) that runs in better than  $O(n^2)$  time. Don't write code, describe a method and the justification for its running time.

### Part D (6 points)

Describe a method for solving a third problem: given a target number  $T$  determine if there are two integers in the vector whose sum is  $T$ . Write a description of how to solve this problem in  $O(n)$  time using an auxiliary data structure. Don't write code, describe a method and the justification that it's  $O(n)$ .

Hint: given the target number  $T$  and an element  $a[j]$  the other element must have the value  $T - a[j]$  if the sum is in the vector?

### Part E (12 points)

An OO advocate proposes replacing the function `findMaxDiff` from the beginning of the problem with the function `processAll` below (this function uses the class `Property` and a subclass `MaxDiff` described below the code). **In this problem you will implement the class `Sum`, a subclass of the class `Property`.**

```
void processAll(tvector<int> & a, Property& p)
{
    int j,k;
    for(j=0; j < a.size(); j++) {
        for(k=j+1; k < a.size(); k++) {
            p.process(a[j],a[k]);
            if (! p.keepGoing()) return;
        }
    }
}
int main()
{
    tvector<int> list;          // fill list with values
    MaxDiff md;
    processAll(list, md);
    cout << md.getFirst() << " " << md.getSecond() << endl;
}
```

The main class `Property` used in `processAll`, and the subclass `MaxDiff` for finding the maximal difference (two numbers farthest apart) follow. You'll implement `Sum`, whose use is shown below, on the next page.

```
class Property
{
public:
    virtual ~Property() { }
    virtual bool keepGoing() {return true; }
    virtual void process(int & a, int & b) = 0;

    int getFirst() const { return myFirst; }
    int getSecond() const { return mySecond; }

protected:
    int myFirst;
    int mySecond;
};

class MaxDiff : public Property
{
public:
    MaxDiff() : myMax(0) { }
    virtual void process(int& a, int & b)
    {
        if (abs(a - b) > myMax) {
            myFirst = a;
            mySecond = b;
            myMax = abs(a - b);
        }
    }
private:
    int myMax;
};
```

Complete the implementation of the class `Sum` so that the code below solves the problem from **Part D** of finding if two numbers sum to a target number. This solution isn't as efficient as the solution you were asked to write in **Part D**.

```
int target;
cout << "target ";
cin >> target;
Sum s(target);
processAll(list,s);
if (! s.keepGoing()) {
    cout << s.getFirst() << " " << s.getSecond() << endl;
}
else {
    cout << "no solution found" << endl;
}
```

Complete the implementation of `sum` below.

```
class Sum : public Property
{
public:
    Sum(int target)
        :

    {

    }
    virtual void process (int& a, int& b)
    {

    }
    virtual bool keepGoing()
    {

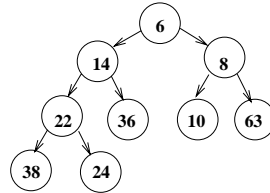
    }

private:
    int myTarget;
    bool myFound;
};
```

**PROBLEM 5 :** (*Heaps of Trouble (7 points)*)

Complete the function `lessCount` to determine the number of elements in a minheap that are less than a target number. A minheap is a heap such that the value in each node is less than the value in the node's subtrees. A min heap is implemented via a vector with the root at index 1 as we discussed in class.

In the example below, `lessCount(heap,13)` should return 3 (6, 8, and 10 are less than 13), and `lessCount(heap,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. We provide an auxiliary function header that can be called from `lessCount`, but you don't have to use it.

```
int lessCount(const tvector<int> & heap, int target)
// pre: heap is a minheap containing heap.size() elements
// post: returns the number of elements in the heap
//       less than target
{

}

int lessAux(const tvector<int> & heap, int index, int target)
// pre: heap is a minheap with heap.size() elements
//       1 <= index < heap.size()
// post: returns # elements in subheap rooted/starting at index
//       that are less than target
{

}

}
```