

# CSSE 230 – Fundamentals of Computing

## Exam 2

Spring term, 2009-2010

Your name: \_\_\_\_\_

Instructions: This exam is open book, notes, computer. In addition:

- All the work you turn in must be your own.
- You must not use any forms of communication or cooperation.
- Disable all chat tools (IM, ICQ, etc) before the exam starts!
- For the first **four** problems, your computer has to be closed. You may look at all problems of this exam and work on them. However, once you open up your computer, you need to turn in the first three problems.
- The code you provide should follow good style and should be efficient. Due to time constraints, you do not have to comment your code.
- Write all answers on these pages. Use the back as necessary.

<b>Problem</b>	<b>Points available</b>	<b>Your score</b>
<b>1</b>	<b>12</b>	
<b>2</b>	<b>15</b>	
<b>3</b>	<b>10</b>	
<b>4</b>	<b>8</b>	
<b>5</b>	<b>55</b>	
<b>Total</b>	<b>100</b>	

1) [12 points] Consider the searching algorithm shown below. Analyze the **worst**, **best**, and **average** case run times of this algorithm. Count the number of times that you access array.

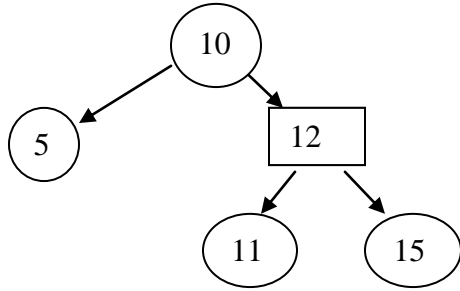
a) **Best case.** Describe a case in which the best case occurs or give an example. Either give formulae for the run times of the operations you count or provide a good argument for the run times. Express the run times using the big-oh notation.

b) **Worst case.** Describe a case in which the worst case occurs or give an example. Either give formulae for the run times of the operations you count or provide a good argument for the run times. Express the run times using the big-oh notation.

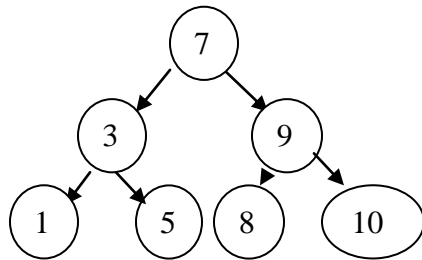
c) **Average case.** Provide and justify the average case big-Oh running time.

```
public static int search(int[] sortedArray, int toFind){
    // Returns index of toFind in sortedArray, or -1 if not found
    int low = 0;
    int high = sortedArray.length - 1;
    int mid;
    while (sortedArray[low] <= toFind && sortedArray[high] >= toFind) {
        mid = low + ((toFind - sortedArray[low]) * (high - low)) /
            (sortedArray[high] - sortedArray[low]);
        if (sortedArray[mid] < toFind)
            low = mid + 1;
        else if (sortedArray[mid] > toFind)
            high = mid - 1;
        else
            return mid;
    }
    if (sortedArray[low] == toFind)
        return low;
    else
        return -1; // Not found
}
```

2) [15 points] a) [8 pts] Show the steps required to extend the Red Black tree below. Red nodes are depicted by squares. Use top-down insertion and include rotations for the following insertion sequence: 2-3-4-6



b) [7 pts] Show the steps required to remove from the Red Black tree below. Use top-down removal and include rotations for the following removal sequence: 10-3-1



3) [10 pts] *Restaurant Finder*. Consider the following problem and its proposed solution. In order to assist diners in locating restaurants that serve their preferred dishes, a software system is developed. In it, restaurants are stored in a hash-table by name. You may suppose that there are  $R$  restaurants in the system and no two restaurants have the same name. For each restaurant, we store its name, address and the menu items they serve. The menu items of a particular restaurant are stored in an additional hash-table, again by name. Suppose that each restaurant serves the same number of dishes and that this number is  $M$ .

a) [4 pts] Consider the problem of searching for a menu item. Describe an algorithm for searching for a menu item by name. This search should list all restaurants that carry that menu item. Count the number of times a hash-table is accessed. Give an informal big-Oh analysis. Show your work.

b) [4 pts] Consider the problem of inserting a new menu item for a given restaurant. Count the number of times a hash-table is accessed. Give an informal big-Oh analysis. Show your work.

b) [2 pts] Evaluate the choice of this data structure when it comes to the operations described above. If you think it is a good choice, justify it. If you feel that a more efficient way of storing the data is in order, describe and justify it.

4) [8 pts] Consider the following min-heap. Insert the following numbers into it: 7-2-9 Show your work.

/	3	5	4	6					
---	---	---	---	---	--	--	--	--	--

5) [50 points] **On-the-Computer part**

- Download the BinarySearchTree project from the communal folder located at: <http://rose-hulman.edu/class/csse/csse230/exams/RBT.zip>
- Study the code and modify it as follows. You may extend or implement any methods you would like.
- All code you write should be **correct, efficient, and use good style**. However, no documentation is required, because of time constraints.
- Implement the `equalBlackNodeHeight()` [15pts], `noSuccessiveRedNodes` [20 pts] and `isMaxRedBlackTree` [20 pts] methods as specified in the JavaDocs of the method stubs. Here are the specifications from the JavaDocs:

```
/** This method determines whether all paths from the root to
 * each of the null nodes have the same number of black nodes.
 * This method runs in O(n) time. An empty tree trivially satisfies
 * this property.
 *
 * @return true, if all paths from the root to the null nodes
 * have the same number of black nodes.
 */
public boolean equalBlackNodeHeight() {

/** This method determines whether the tree ever has two red
 * nodes in succession. This method runs in O(n) time. An empty
 * tree trivially does not satisfy this property.
 *
 * @return true, if there are never two successive red nodes.
 * False otherwise.
 */
public boolean noSuccessiveRedNodes() {

/** This method determines whether the tree is a red black tree
 * and whether it contains a maximum number of red nodes for a given
 * height. This method runs in O(n) time. An empty tree trivially
 * satisfies this property. Please study the test cases for examples.
 *
 * @return true, if the tree contains a maximum number of red nodes.
 * False otherwise.
 */
public boolean isMaxRedBlackTree() {
```

- Please submit your **RedBlackTree.java** file to the **Exam2** drop box on our Angel site.