CS 251 - Project 4: Sorting and Trees

Out: Feb/25/2019, 8:00 am
Due: Mar/25/2019 8:00 am

Part 1: Words Rhyme Sorting and Suffix Sharing (50 Points)

In this part, you are going to implement the QuickSort algorithm and use it to sort words based on rhyme (i.e., the correspondence of sound between the endings of words). In addition, you will then group the words that share suffixes (e.g., the two words "Sort**ing**" and "Shar**ing**" share the three-character suffix: "-**ing**"). Given a list of words, you are required to provide the following two operations:

(1) Rhyme Operation : Sort all words in a rhyming order (25 points):

To accomplish this task you need to apply the following steps:

1. Read the input list of words into an array of strings.
2. Reverse the letters in each word (e.g., "confound" becomes "dnuofnoc"). Capitalization does not matter.
3. Sort the resulting array of words using your own implementation of QuickSort.
4. Reverse the letters in each word back to their original state.
5. Print the words to the output file.

Example 1: Rhyming Order:

Given the list of words: "compound", "crabbit", "confound", "flashpit", "astound", and "trampit"
The correct rhyming order of these words is:
confound - compound - astound - crabbit - flashpit - trampit

(2) Suffix Sharing Operation (25 points):

For this operation, you need to Identify the suffixes which have at least *k* words ending with it (i.e., where *k* is an input argument). You will group words based on suffix sharing. You need to make use of use of the rhyme order computed previously in this operation. This will enable you to implement this operation efficiently without the need to do a brute force check.

Example 2: Suffix Sharing (k=2)

Given: "rabbit", "crabbit", "flashpit", "trampit" and k=2, will yield the following suffixes:
t - crabbit, flashpit, rabbit, trampit
it - crabbit, flashpit, rabbit, trampit
bit - crabbit, rabbit
pit - flashpit, trampit
bbit - crabbit, rabbit
abbit - crabbit, rabbit
rabbit - crabbit, rabbit

Note that if k=3 for the same list of words, output would be:
t - crabbit, flashpit, rabbit, trampit
it - crabbit, flashpit, rabbit, trampit

Implementation instructions:

For both operations, you will need to apply the sorting on the reversed version of the words (i.e., for the word: “astound” -> “dnuotsa”). You may implement this reverse function or use the string library for C++ or Java. However, you need to provide a full implementation of the QuickSort algorithm, which should work in-place.

In your implementation of the QuickSort, you do not necessarily need to implement string comparing functions, You can use strcmp functions in C++. Also, in Java, just make sure you implements the Comparator interface, so that you can use the (<,>) operators against strings. For the Suffix share implementation, you may want to use a hashmap to store the word lists associated with each suffix. Feel free to use the data structure that comes with the language library (C++ or Java). The test cases will measure the correctness and performance of your implementation using different input sequences.

Sample Input/output format:

- The first integer is a 1 or a 2 indicating this is the input file for part 1 or 2, respectively;
- If it is part 1, second line integer implies required operation 1: rhyme or 2: suffix
- If it is rhyme operation: Third line integer implies that there are n lines following (i.e., n words).
- If it is suffix operation: Third line integer implies *k* and then the fourth line integer implies that there are n lines following (i.e., n words).
- After that, each line will have one word.

Test Cases:

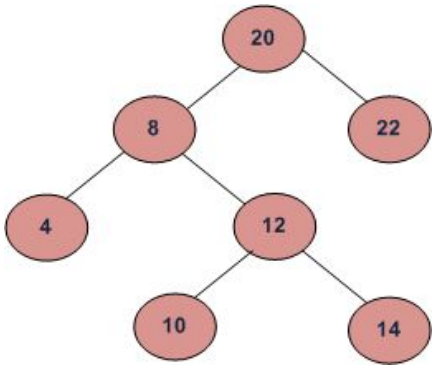
Separate test cases will be given for the two operations and you will be graded independently. For example, if you only complete the Rhyme order, you will get 25pts.

Sample Input for operation 1: RhymeOrder	**Sample Output**
1 1 6 compound crabbit confound flashpit astound trampit	confound compound astound crabbit flashpit trampit

Sample Input for operation 2: SuffixShare	**Sample Output**
1 2 3 4 rabbit crabbit flashpit trampit	**t** -> [crabbit, flashpit, rabbit, trampit] **it** -> [crabbit, flashpit, rabbit, trampit]

Part 2: Binary Search Tree (BST) (50 Points)

A Binary Search Tree (BST) is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child and the topmost node in the tree is called the root. It additionally satisfies the binary search property, which states that the key in each node must be greater than or equal to any key stored in the left sub-tree, and less than to any key stored in the right subtree.



For this project, you can assume **keys and values** are the **same** (so maintaining the key is enough). The keys will be **integers and unique.** You are required to complete the following tasks.

Basic Operation: (10 pts)

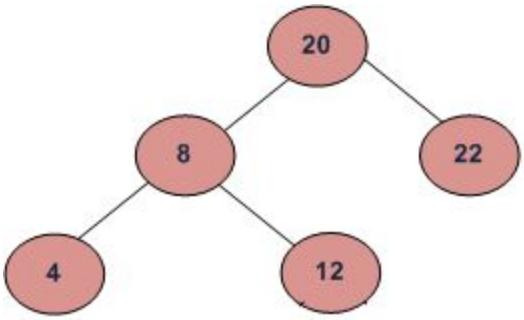
1. **Insert a key into BST:** `insertKey(int key)` – A new key is always inserted at a leaf. We start searching for a key from the root until we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.

2. **Delete a key:** `deleteKey (int key)` – Remove the key node from the tree. If the key is not in the tree, do nothing.

3. **Search a key:** `searchKey(int key)` **-** Search for the key in the BST; return true if found, return false otherwise. This function will be used by you elsewhere.

4. **Range Sum:** `rangeSum(int left, int right)` – Return the summation of all the keys in the BST falls in the range `[left, right]` inclusive. For example, in the above case, the `rangeSum(11,17)` will return `26 [12 14]`. Print “none” to file if no elements found.
   `55`
5. **Height of a node:** `int height(int key)` **-** find and print the height of a node in BST. If a node with the key is not found, print “none” to file. For example, the 14 node in the tree above has height 3.

Traversal: (10pts)

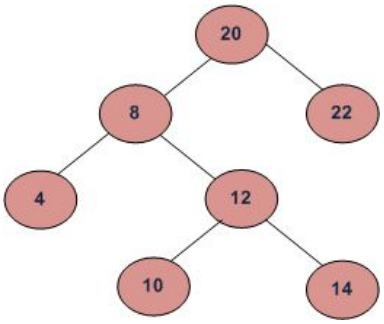


6. **Postorder Traversal Print :** - `postorder()` - get the Postorder traversal of the BST. For the above figure, Postorder (Left, Right, Root) should print “4 12 8 22 20”.

7. **Level Order Traversal Print :** - `levelorder()` - get the Breadth First or Level order traversal of the BST. For the above figure, Breadth First or Level Order Traversal should print “20 8 22 4 12”. One way to implement level order is to use a queue.

For all of the above traversal, print “none” in file if no elements present in tree.

Problem: (15pts)



8. **Finding the lowest common ancestor (LCA) between two nodes in BST:**

```
int LCA(int key1, int key2)
```

Let T be a rooted tree. The lowest common ancestor between two nodes n1 and n2 is defined as the lowest node in T that has both n1 and n2 as descendants (where we allow a node to be a descendant of itself).

The LCA of n1 and n2 in T is the shared ancestor of n1 and n2 that is located farthest from the root.

For example in the above graph,

LCA of 10 and 14 is 12
LCA of 14 and 8 is 8
LCA of 10 and 22 is 20

If any key is invalid or not present, print “none” in file.

9. **Find Floor and Ceil of a key in BST :** `Floor(int key)` Floor of key means, given a key you need to find the node equal or the largest key smaller than given key. For example, in the above example,

   ```
   Floor(21) prints “20”
   Floor(4) prints “4”
   Floor(3) prints “none”
   ```

`ceil(int key)` Ceil of key means, given a key you need to find the node equal or the smallest key larger than given key. For example, in the above example,

```
Ceil(21) prints “22”
Ceil(4) prints “4”
Ceil(22) prints “none”
5
```

10. **Determining the distance between pairs of nodes in a tree**: `dist(int key1, int key2)`

    Given two keys of BST, the distance between two nodes is the minimum number of edges to be traversed to reach one node from other. For example in the above example,

    ```
    dist(8, 22) is 2
    dist(10,22) is 4
    ```

    If any key is not valid, print “none” in file.

Red Black Tree Insertion (15pts):

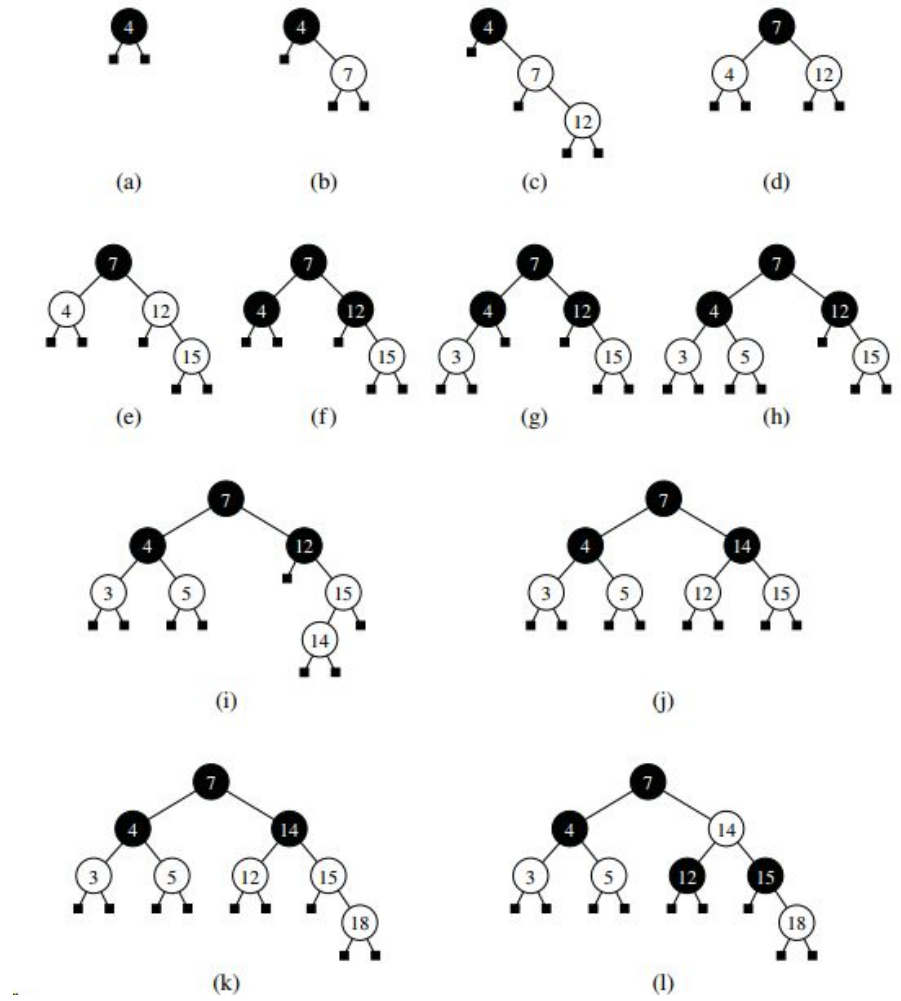
In this part, modify your code to handle Red Black Tree insertion portion only. There will be no deletion, therefore your tree will remain balanced. The insert method used may be slightly different for the C++ and the Java sections. Please follow the instructions accordingly.

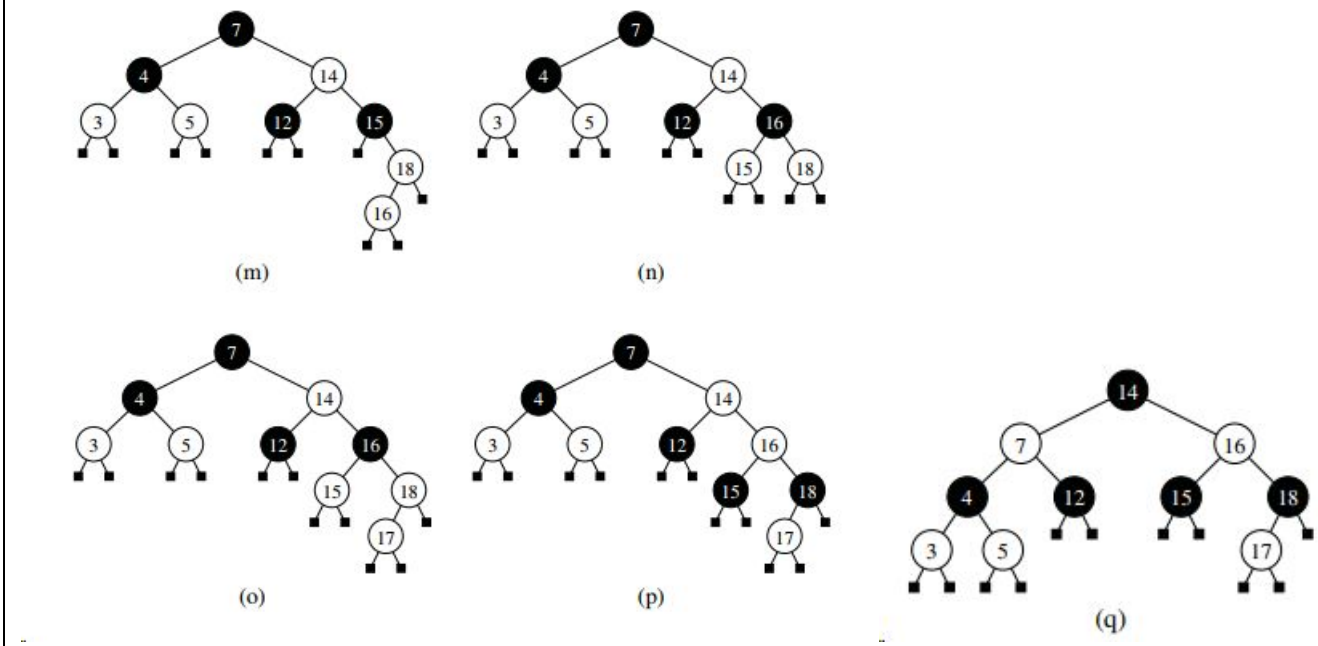
C++ Section:

11. **Insert to maintain balance in tree** : `insertRB(int key)`: Follow the insertion procedure described in **Data Structures and Algorithms in C++, 2nd Edition, M. T. Goodrich, R. Tamassia, D. M. Mount, February 2011**[link]

    Sample example from book,

```
insertRB 4
insertRB 7
insertRB 12
insertRB 15
insertRB 3
insertRB 5
insertRB 14
insertRB 18
insertRB 16
insertRB 17
```

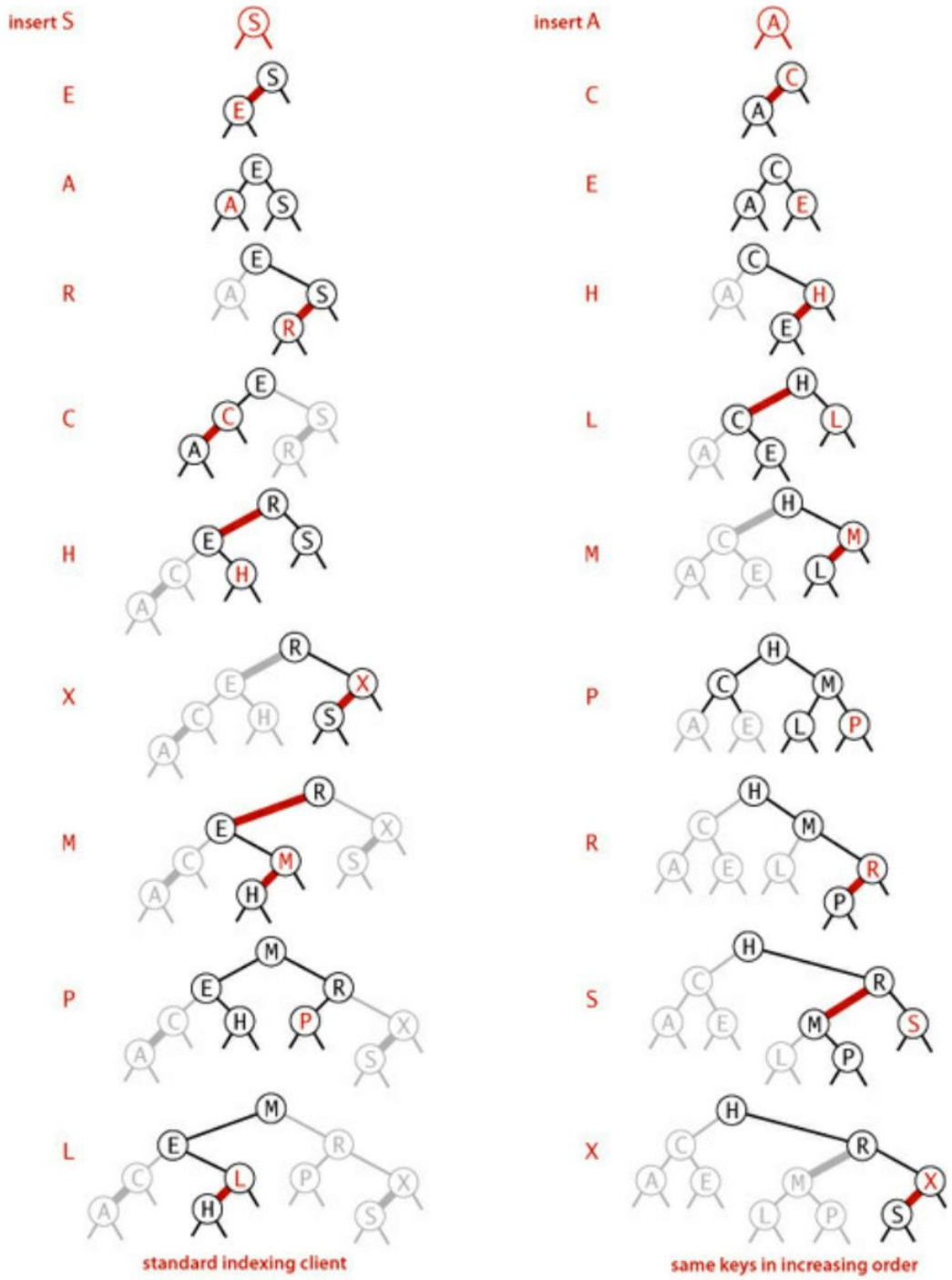




Java Section:

12. **Insert to maintain balance in tree** : `insertRB(int key)` : Follow the insertion procedure described in section 3.3 of Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne

`Sample example from book,`



Red-black BST construction traces

Both Sections:

13. **Black Height of a Red-Black Tree :** Black height is number of black nodes on a path from a node to a leaf (excluding given node). Leaf nodes are also counted black nodes.

```
getBlackHeight(int key)
```

In the above scenario, `getBlackHeight(14)` would print “2” to file. Print “none” if it’s an invalid key.

Sample Input/output format:

- The first integer 1 or 2 indicates this is the input file for part 1 or 2.
- The second integer n implies that there are n lines below.
- You can assume all numbers will be of Integer datatype.
- The command and corresponding input/output can be understandable from the sample test cases given bellow.
- For part2, each line of output should end with \r\n and there will be an additional line at the end. Match your output exactly with provided test cases.

Test Cases:

Test cases will be given for each of the 12 functionalities listed above. Your score will be based on how many complete units you are passing. For example, if you only complete basic operations, you will get 10pts. If you do basic and traversals you would get total of 25 pts.

Sample Input	**Sample Output**
2 17 insert 20 insert 22 insert 8 insert 12 insert 4 insert 10 insert 14 height 10 height 22 height 20 height 12 delete 10 delete 15 search 14 search 15 range 1 22 range 11 17	3 1 0 2 deleted deletion failed found not found 80 26

2 9 postorder levelorder insert 20 insert 22 insert 8 insert 12 insert 4 postorder levelorder	none none 4 12 8 22 20 20 8 22 4 12
2 19 insert 20 insert 22 insert 8 insert 12 insert 4 insert 10 insert 14 lca 4 10 lca 20 14 lca 11 20 ceil 21 ceil 22 ceil 23 floor 5 floor 4 floor 3 dist 10 22 dist 4 14 dist 1 0	8 20 none 22 22 none 4 4 none 4 3 none

Section	RB Tree Sample Input	RB Tree Sample Output
C++	2 13 insertRB 4 insertRB 7 insertRB 12 insertRB 15	2 2 14 7 16 4 12 15 18 3 5 17

	insertRB 3 insertRB 5 insertRB 14 insertRB 18 insertRB 16 insertRB 17 Bheight 14 Bheight 16 levelorder	
Java	2 13 insertRB 1 insertRB 2 insertRB 3 insertRB 4 insertRB 5 insertRB 6 insertRB 7 insertRB 8 insertRB 9 insertRB 10 Bheight 1 Bheight 4 levelorder	1 3 4 2 8 1 3 6 10 5 7 9

Part 3: Programming Environment and Grading

Assignments will be tested in a Linux environment. You will be able to work on the assignments using the Linux workstations in HAAS and LAWSON (use your username and password).

3A: Java

- Compiler we use: javac (v10.0.2)
- File to submit: Rhymer.java, BinarySearchTree.java, Node.java, Main.java

Your project must compile using the javac compiler (v10.0.2) on data.cs.purdue.edu. Main.java is provided to you so that you can generate output file on your own to see if your code works correctly, you may change it as your wish.

Compiling process: **following commands assume you are at project root and you have NOT changed project file structure!**

- To compile Main:
  javac src/*
- To run Main:
  java -cp src/ Main InputFilePath OutputFilePath
  Note: You should replace InputFilePath OutputFilePath with actual file path in the above command.

Grading process:

1. Compiling your program with JUnit test files using javac v10.0.2.
2. Running JUnit tests. It read input files and manipulates data structures implemented by you and see if it works as expected. Any forms of difference between your output and expected output will cause points deduction.
3. Inspecting your source code.

3B: C++

The compilation will be done using g++ and makefiles. You must submit all the source code as well as the Makefile that compiles your provided source code into an executable named "program".

Your project must compile using the standard g++ compiler (v 4.9.2) on data.cs.purdue.edu. For convenience, you are provided with a template Makefile and C++ source file. You are allowed to modify such file at your convenience as long as it follows the I/O specification. Note some latest features from C++14 are not available in g++ 4.9.

The grading process consists of:

1. Compiling and building your program using your supplied makefile.
2. The name of the produced executable program must be "program" (must be lowercase)
3. Running your program automatically with several test input files we have pre-made according to the strict input file format of the project and verifying correct output files thus follow the above instruction for output precisely – do not "embellish" the output with additional characters or formatting – if your program produces different output such as extra prompt and space, points will be deducted.
4. Inspecting your source code.

The input to the programming projects will be via the command line:

program input-test1.txt output-test1.txt

The file output-test1.txt will be tested for proper output.

Important:

1. If your program does not compile, your grade will be 0.
2. Plagiarism and any other form of academic dishonesty will be graded with 0 points as definitive score for the project and will be reported to the corresponding office.

Part 4: Submit Instructions

The project must be turned in by the due date and time using the turnin command. Follow the next steps:

1. Login to data.cs.purdue.edu (you can use the labs or an ssh remote connection).
2. Create a directory named with your **username** and copy your solution there.
   C++: makefile, all .cpp, .h files.
   Java: Rhymer.java, BinarySearchTree.java, Node.java, Main.java
3. Go to the upper-level directory and execute the following command:
   turnin -c cs251 -p project4 your_username
   (Important: previous submissions are overwritten with the new ones. Your last submission will be the official and therefore graded).
4. Verify what you have turned in by typing **turnin -v -c cs251 -p project4**
   (Important: Do not forget the -v flag, otherwise your submission would be replaced with an empty one). If you submit the wrong file you will not receive credit.