CS 104 (Spring 2014) — Final Exam
05/09/2014

Good Luck

Your Name, USC username, and Student ID:

This exam has 8 pages and 8 questions. If yours does not, please contact us immediately. Please read and reread each question carefully before trying to solve it.

In your solutions, you should try hard to write mostly correct C++ unless explicitly told otherwise. Minor syntax errors will (at most) lose small numbers of points, so don't fret too much about those; but you should also demonstrate a mastery of most C++ syntax. Unless otherwise noted, do not worry about `#include` directives or `using namespace std`.

You don't need to duplicate the code we're giving you — in response to questions, just give the piece of code to add to what we are giving you.

Some of the questions may become easier to solve if you use one or more of the following: Hashtables, Balanced Search Trees, Priority Queues, Sorting Algorithms. Unless otherwise specified, you may assume that you have implementations of these, and refer to them using the STL syntax or the names we used in class. As a refresher, here is the basic STL syntax (a little modified for simplicity in some cases).

Hashtable: Write `unordered_map <keyType, valueType>`. You may assume that all three standard operations take constant time.

Balanced Search Tree: Write `map <keyType, valueType>`. You may assume that all three standard operations take time $\Theta(\log n)$, and that a linear-time iterator is provided.

Priority Queues: Write `priority_queue <T>`. That's not quite the C++ syntax, but we'll pretend as much for now.

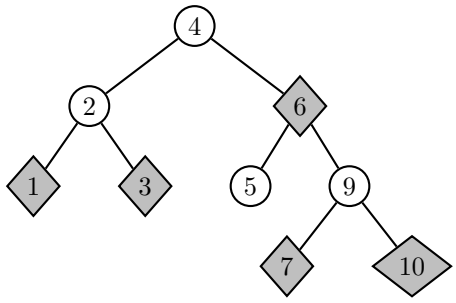
Sorting: Write `sort(a,left,right)` to sort an array `a` between `left` and `right` (inclusive) in ascending order. This is not exactly STL syntax, but cleaner for now. You may assume that it runs in time $O(n \log n)$, where $n = \text{right} - \text{left}$, and is stable. If `a` is an array of items of type `T`, `sort` will use the `<` operator of `T` to sort, unless you explicitly pass as a fourth argument a comparator object or function.

For some questions, we may explicitly not want you to use these functionalities. For those, we will say "No STL" at the beginning of the problem. That means that if you want Hashtables, Search Trees, Priority Queues, or Sorting Algorithms, you have to give us an implementation. You will still be allowed to use sequence containers and other adapters, though (including `vector`), unless explicitly specified.

Question	1	2	3	4	5	6	7	8	Total
Points possible	10	10	12	12	12	16	12	16	100
Points									

(1) [10 points]
In class, we saw that Dijkstra's Algorithm correctly finds a shortest s-t path in weighted graphs so long as all edge costs are non-negative, $c_e \geq 0$ for all e. Draw an example graph with one or more negative edge weights on which Dijkstra's Algorithm will *fail* to find a shortest s-t path. (Hint: there are some pretty easy examples.) Explain what the actual shortest path would be, and which path Dijkstra's Algorithm finds.

(2) [10 points]
Below, you are given a correct Red-Black tree, in which red nodes are shown as shaded diamonds, while black nodes are shown as empty circles. Give the final tree obtained from inserting the key '8' into this tree, after the `insert` function correctly restores the Red-Black Tree properties. It is not necessary to show intermediate steps, but if your final answer is wrong, you will get partial credit if earlier steps are correct.



(3) [12 points]

An education researcher has the hypothesis that higher education only reinforces existing education inequalities, rather than making students catch up and excel. More specifically, he makes the strong hypothesis that the student with the highest SAT score coming in will have the highest GPA upon graduation, the student with the second-highest SAT score will have the second-highest GPA, and so on. To be consistent, he hypothesizes that students coming in with the same SAT score will also have the exact same GPA.

He would like you to write a program to test his theory on data sets. Each data set will contain students sorted alphabetically, with their GPA and SAT score given. Your program should determine whether his hypothesis is true. Add your code to the skeleton on the next page.

Because he expects to run on several large data sets, your algorithm must run in time $O(n \log n)$ for data sets of n students.

```
struct StudentRecord {
  string name;  // name of the student
  int SAT;      // SAT score
  double GPA;   // GPA at graduation

// feel free to add functionality here if you need it.









};

int main (int nargs, char** args) {
  int n;           // number of students in data set
  StudentRecord *students;

  read_data (args[1], &n, students);
    /* this function reads all the data correctly from the file and puts them in the array 'students'.
       When the function returns, the students in the array will be sorted by name.
       n will be the size of the array. No two students will have the same name. */

  // Your code goes here. You should output "Yes" if the hypothesis is true, and "No" otherwise.









  return 0;
}
```

(4) [12 points]
You put k keys into a hash table with m hash buckets. We assume that the hash function is good enough that you can treat the destination of each key as independently uniformly random. What is the probability that there are no collisions at all, i.e., that all keys end up in different positions of the array? Show your work as you derive your answer.

(5) [12 points]
We have a Bloom Filter with an array of 10 elements (the elements of the set are integers), and using three hash functions

$$\begin{aligned} h_1(x) &= (7x+4) \bmod 10, \\ h_2(x) &= (2x+1) \bmod 10, \\ h_3(x) &= (5x+3) \bmod 10. \end{aligned}$$

We execute the sequence of operations given below. What does the program output? Which of the answers are false positives (the Bloom filter says "Yes", even though the correct answer is "No")? Which are false negatives (the Bloom filter says "No", even though the correct answer is "Yes")? If your final answer is incorrect, you may get more partial credit if you show enough work for us to isolate the mistake.

```
BloomFilterSet<int> bf (10);
bf.add (0);
bf.add (1);
bf.add (2);
bf.add (8);
// Show us what the Bloom Filter's Array looks like at this point.
if (bf.contains (2)) std::cout << "2\n";
if (bf.contains (3)) std::cout << "3\n";
if (bf.contains (4)) std::cout << "4\n";
if (bf.contains (9)) std::cout << "9\n";
```

(6) [10+6=16 points]
Implement and analyze lookup of keys in general $(b, 2b-1)$ trees. Remember that these are trees in which each node has at least b and at most $2b-1$ children. Other than that, they are exactly like 2–3 trees, which are the special case $b = 2$. You are already given the class `BTreeNode` below, which only has integers keys, and no values (to keep it simple). The child pointers will be `NULL` when a certain child does not exist. You get to assume that your function will be passed the root of a correct $(b, 2b-1)$-tree and a key to search for. Your analysis in big-$O/\Omega/\Theta$ notation should tell us how long the `lookup` function takes, as a function of n (the number of keys in the tree) and b.

(a) Give us your implementation of `lookup`.

```
#define b 10000   // Your code should work also if someone gives b another value
struct BTreeNode {
  int c;  // number of children this node actually has.
          // Will always be between b and 2b-1.
  int k [2b-2];  // the sorted array of keys in the node.
  BTreeNode *child[2b-1];  // the ordered array of children.
};

bool lookup (BTreeNode *root, int key)
  /* This is the function you implement.
     It should return true if the key exists in the tree, and false otherwise.
     You may define helper functions if you want. */
```

(b) Analyze the running time of your implementation in big-$O/\Omega/\Theta$ notation.

(7) [12 points]
You have a correctly implemented `Max-Heap` class (indices starting at 1), which provides the two following helper functions (in addition to `insert`, `remove`, `peek`):

(a) `trickleUp (int i)` (David's lecture) a.k.a. `insert (int i)` (Mark's lecture) takes the element at position i and keeps swapping it with its parent (and grandparent etc.) until it is no larger than the parent at the time.

(b) `trickleDown (int i)` (David's lecture) a.k.a. `heapify (int i)` (Mark's lecture) takes the element at position i and keeps swapping it with the larger of its children (and grandchildren etc.) until it is no smaller than both children.

Suppose that you get an instance whose array elements are not in correct heap order, i.e., the heap property does not hold yet. You want to establish the correct arrangement, by using the two helper functions given above. Which of the following two alternatives do this correctly? Explain why they are correct or incorrect. For incorrect ones, give an example of an initial heap which is not correctly processed, and explain what happens when the incorrect variant is executed on it.

```
Heap h;

// Variant 1
for (int i = 1; i <= n; i ++) h.trickleUp (i);

// Variant 2;
for (int i = 1; i <= n; i ++) h.trickleDown (i);
```

(8) [16 points]

In class, we saw Tries and how to insert and look up words in them. Here, you will implement deletion from a trie. We will assume that our strings only consist of lower-case letters.

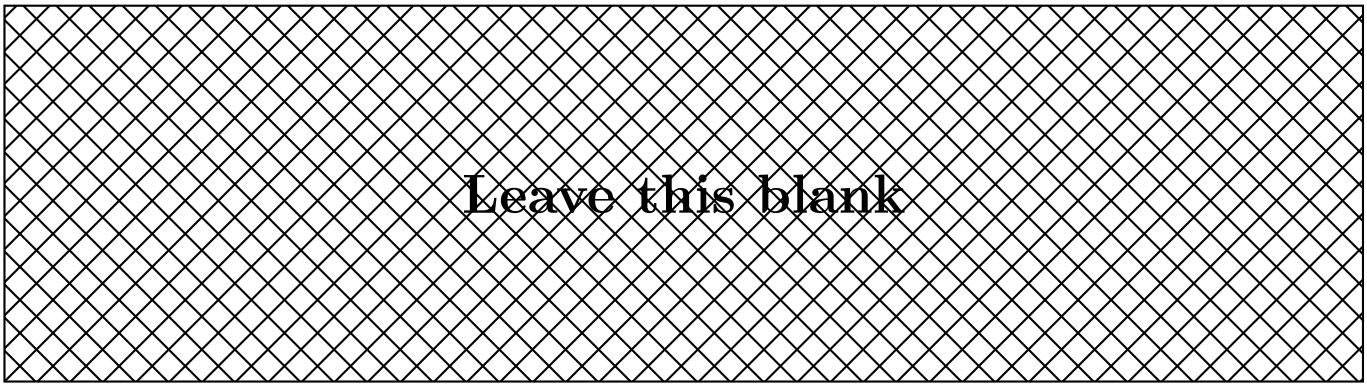
A friend has already written the following piece of code. She is only sharing the header file with you, but since she's a Trojan, you're sure that it's all correctly implemented.

```
struct TrieNode {
  bool isInSet;        // true if there is a word ending at this node.
  TrieNode *parent;    // the parent in the trie, NULL if the node is the root.
  TrieNode *child[26]; // the (up to) 26 children for the letters. NULL if there is no such child.

  int convert (char letter) // turns a letter into its index in the child array
    { return (int) (letter - 'a'); }
  bool isLeaf () { // returns whether the node is a leaf
    for (int i = 0; i < 26; i ++)
        if (child[i] != NULL) return false;
    return true;
  }
};

class InsertOnlyTrie {
  protected:
    TrieNode *root;

  public:
    InsertOnlyTrie () { root = NULL; }
    ~InsertOnlyTrie (); // correctly implemented destructor

    void add (string s);
      // adds the string to the trie correctly. Is already implemented.

    TrieNode *lookup (string s);
      /* returns the pointer to the node at which the search for s terminates in the trie.
         Is already implemented. If s is not in the trie, returns NULL. */
};
```

On the next page, you should use your friend's class to build a "real" `Trie` class that also provides the following public function.

```
void remove (string s);
      /* If s is not in the trie, the function should not do anything.
         Otherwise, it should remove s from the trie as though it had never been added. */
```

Remember that you cannot alter your friend's code, so you must find a suitable way to build the `Trie` class. For full credit, your solution must not leak any memory.



```
// your code goes here
```