Score: / 80 CIS 263

Name: last, first 2011:Dec:14(Wed)/15(Thu)

Final Examination

- This examination is open-book and open-notes etc. — except you are not allowed to use any devices (**no** calculators, telephones, etc.). You can consult the instructor.
- You generally need to show intermediate work / explanations appropriate for obtaining answers.

A. For each of the following statements, note whether it is false or true off to the left.
Rem. "$\Theta()$" means the same as "$O()$", except that "$\Theta()$" is just more precise than "$O()$". For example $3*N^2 + 5$ is $O(N^2)$ but it's also $O(N^3)$ and $O(N^4)$ and so on, whereas $3*N^2 + 5$ is only $\Theta(N^2)$ and **not** $\Theta(N^3)$ or $\Theta(N^4)$ or so on. You actually don't need to provide any intermediate work / explanations for your answers for these questions.

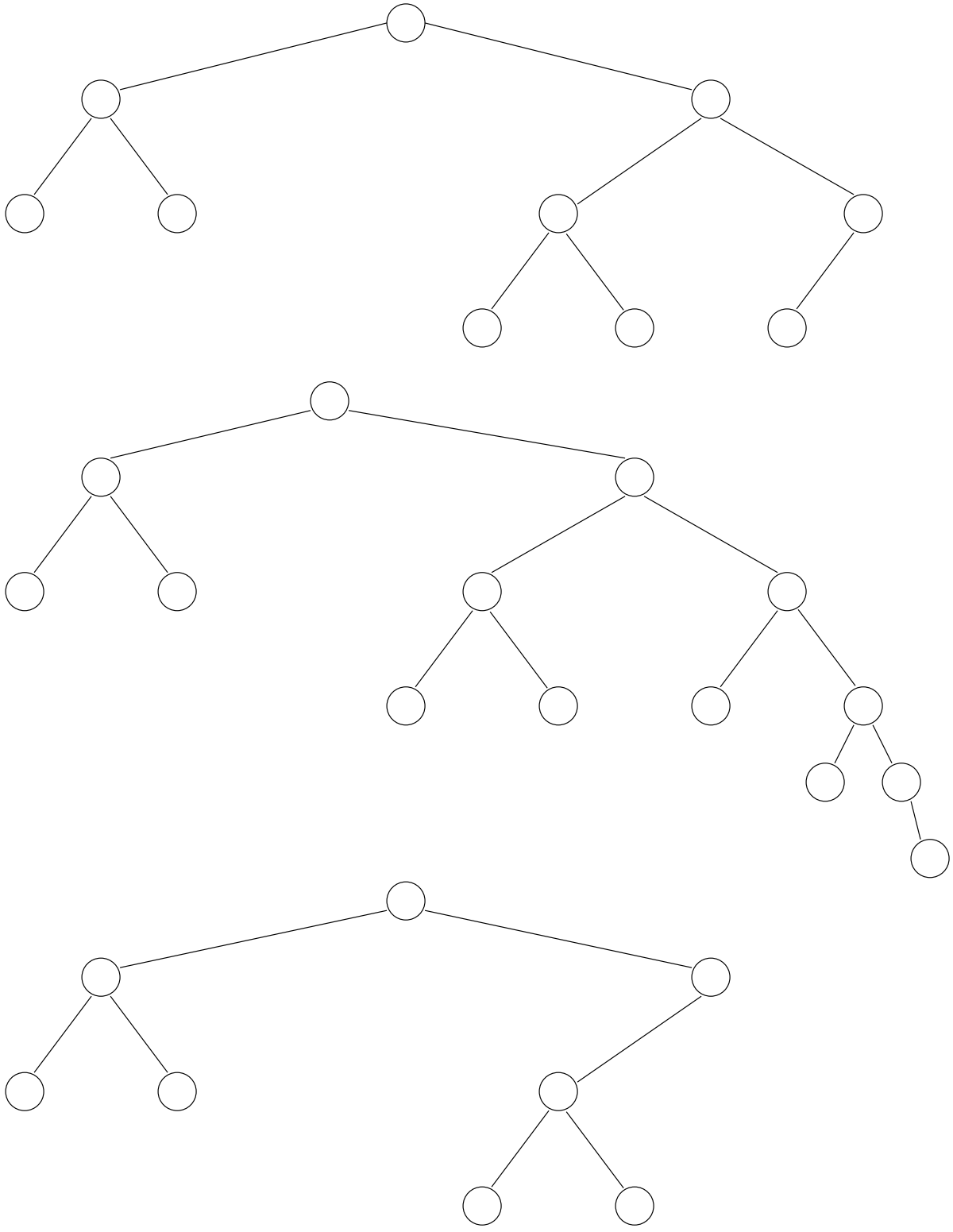
1. [1 point] A $\Theta(n)$ algorithm could require 7.3 seconds for 10 items of data, 14.6 seconds for 20 items, 29.2 seconds for 40 items, 58.4 seconds for 80 items, ...
2. [1 point] Suppose a sort of combination-lock has `n` 'wheels' which are simply binary — 0 or 1 — e.g. the combination to unlock the lock might be 1110100011001. Then the time required to try all the combinations to solve the lock is $\Theta(\lg(n))$.
3. [1 point] $3*2^{(N-1)}$ is $\Theta(2^N)$.
4. [1 point] $\lg(N^2)$ is $\Theta(\lg(N))$.
5. [1 point] The postorder algorithm for processing/traversing a binary tree is to recursively process the element at the given node, then its left subtree, and then its right subtree.
6. [1 point] The time required for `BinarySearchTree::remove()` is **not** $\Theta(N^2)$; the time is significantly smaller than that (a smaller function of `N`).
7. [1 point] The time required for `BinarySearchTree::insert()` is at least $\Theta(N*\lg(N))$.
8. [1 point] Obtaining a record in a B^+ tree always requires navigating down to a leaf node, by contrast with the way one can obtain a record stopping part-way down in a binary-search tree.
9. [1 point] A red-black tree qualifies also as a binary-search tree.
10. [1 point] A heap qualifies as a binary search tree.

B. [4 points] Order the following functions by growth rates: N^2, $\lg(N)$, 2^N, $N*\lg(N)$, N

C. [5 points] Draw a red-black tree with *greatest* height for a collection of data records whose key values are $1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11$. To show 'red' and 'black', draw doubled parentheses `(())` around red nodes and rectangular brackets `[]` around black nodes.
Draw **just one tree** — you do **not** need to show how the tree was built with any insertions.

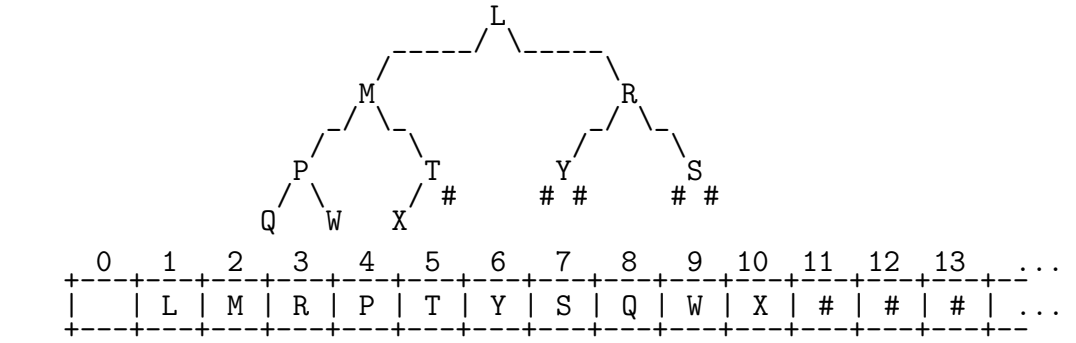
D. [8 points] For each of the following trees, try to color it as a red-black tree — obeying the specifications for such trees: write "r" on a node to 'color' it red, and write "b" on a node to 'color' it black.

If it is not possible to color a tree red-black while obeying the specifications for such trees, then explain clearly why it is not possible.

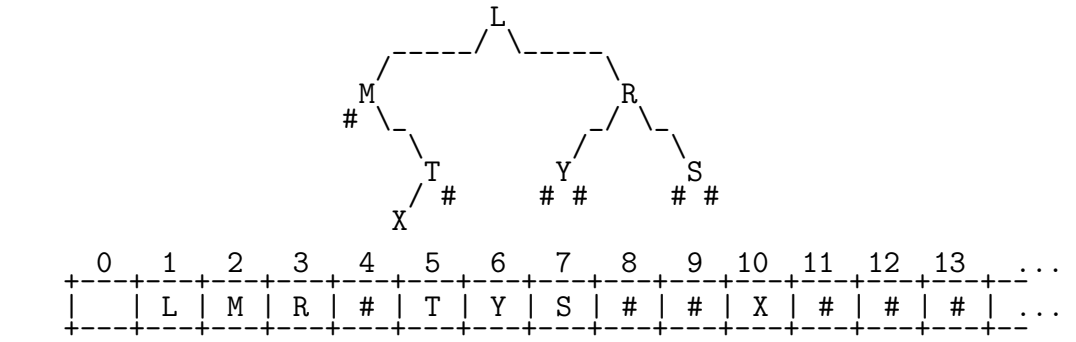


E. Suppose some program needs to use the same binary tree each time it gets executed. Then, it needs to be able to store the tree in a file, and later retrieve it. Your task for this exercise is to write code for these purposes.

This code you need to write here is related to the way heaps are stored in arrays, e.g. as follows:



As I mentioned in a lecture, that scheme works even if the binary tree isn't a heap; there just may be more '`NULL`s', e.g. as follows:



With this scheme, there's a unique index corresponding to each node in a binary tree. The key specification for tree nodes' positions in the array from our textbook is as follows: "For any element in array position `i`, the left child is in position `2*i`, the right child is in the cell after the left child (`2*i + 1`), and the parent is in position $\lfloor$`i/1`$\rfloor$."

Here's some demonstration code for binary trees containing `string`s:

```
struct BinaryNode {     // everything is public
    string element;             // The data in the node
    BinaryNode * left;          // Left child
    BinaryNode * right;         // Right child

    // Constructor:
    BinaryNode(string theElement,
               BinaryNode * lt = NULL, BinaryNode * rt = NULL
               )
        : element(theElement), left(lt), right(rt)
        { }
};

class BinaryTree {
    BinaryNode * root;
  public:
```

```
    BinaryTree(BinaryNode * root_given = NULL) : root(root_given) { }
    int height() const { return height(root); }
  private:
    int height(BinaryNode * n) const {
        return  n == NULL  ?  -1  :  1 + max(height(n->left), height(n->right));
        }
  /*
  friend ostream & operator<<(ostream & os, const BinaryTree & bt);
  friend istream & operator>>(istream & is, BinaryTree & bt);
  */
};
int
main()
{
    BinaryTree family(
        new BinaryNode("Margaret",
                       new BinaryNode("John",
                                      new BinaryNode("Hugh"),
                                      new BinaryNode("Amy")
                                      ),
                       new BinaryNode("Kay",
                                      new BinaryNode("Peter"),
                                      new BinaryNode("Craig")
                                      )
                       )
        );
    cout << "family.height(): " << family.height() << endl;
    /*
    cout << family;
    ofstream ofs("family_bst.txt");
    ofs << family;
    ofs.close();
    ifstream ifs("family_bst.txt");
    BinaryTree family_restored;
    ifs >> family_restored;
    ifs.close();
    */
}
```

1. [15 points]

   Here is `operator<<()`:

   ```
   ostream &
   operator<<(ostream & os, const BinaryTree & bt)
   {
       os << bt.height();
       store(os, bt.root, 1);
       return  os;
   }
   ```

   In some sense, this is like the first `height()` method provided in `class BinaryTree` above.

   Then, like the second method named `height()` provided in `class BinaryTree` above, you need to write a function named `store()` doing a preorder traversal of the tree's nodes, and using the `ostream` argument to output for each node a line comprising (i) the unique index corresponding to the node as indicated above, (ii) a space, and then (iii) the

node's element. For example, the output for the sample tree `family` above should be as follows:

```
7
1 Margaret
2 John
4 Hugh
5 Amy
3 Kay
6 Peter
7 Craig
```

2. [18 points] Next, for recreating the binary tree after its contents have been stored in the manner indicated above, you need to write `operator>>()` as follows (and as declared above):
   a. Read the first integer `h` from the `istream` argument. This integer is the height of the binary tree. Allocate an array of `string`s named say `ta[]` for holding a binary tree with height `h`, like the ones indicated above. Again from our textbook, the maximum possible quantity of items for a height `h` is $2^{h+1} - 1$ (but then rem. the array's element #0 isn't used).
   b. Then, process each of the remaining `h` lines from the `istream`: read the index at the head of the line and the node element in the rest of the line (use `getline(is, string)`), and store the node element at that index in the array `ta[]` you allocated as specified above. E.g.:

```
      ta[]:
    +-------------+
0   |             |
    +-------------+
1   | "Margaret"  |
    +-------------+
2   | "John"      |
    +-------------+
3   | "Kay"       |
    +-------------+
4   | "Hugh"      |
    +-------------+
5   | "Amy"       |
    +-------------+
6   | "Peter"     |
    +-------------+
7   | "Craig"     |
    +-------------+
```

   c. Then, copy the following instruction:

```
bt.root = treeify(ta, 1, h);
```

   d. Then, conclude writing the function `operator>>()`.

3. [20 points] Next, write the function `treeify()`, taking as arguments the array `ta[]`, an index `i`, and the value `h` from above. This function needs to return a `BinaryNode` constructed from the element stored at the given index `i` in the given array `ta[]` — plus the fields `left` and `right` for the `BinaryNode` being constructed need to be obtained via recursive subinvocations of `treeify()` with appropriate arguments.
   Remember to return an appropriate value when the height `i` is `-1` or a relevant array element is empty (actually the empty string).