CS106B
Spring 2016

Instructor: Cynthia Lee
Final Exam

FINAL EXAM - SOLUTIONS

---

1. Pointers and Memory

```
// Sample solution 1
ListNode *curr = front->next;
front->next = front->next->next;
delete curr;
curr = front->next->next->next;
front->next->next->next = NULL;
delete curr->next;
delete curr;

// Sample solution 2
delete front->next->next->next->next->next;
delete front->next->next->next->next;
ListNode *temp = front->next;
front->next = front->next->next;
delete temp;
front->next->next->next = NULL;
```

---

2. Big-O

(a) O(N)
(b) O(N^2)
(c) O(N)

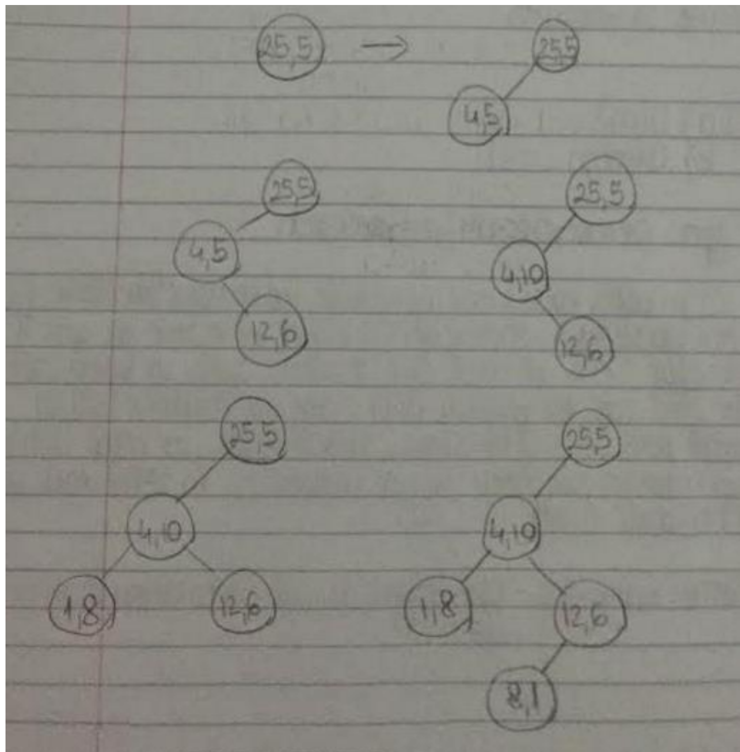
---

3. Heaps

(a) 15, 44, 25, 55, 60, 75, 50, [next 3 array boxes blank]; size=7, capacity=10
(b) 25, 44, 50, 55, 60, 75, [next boxes blank]; size=6, capacity=10

---

4. BST



---

5. **Graph**

Solutions need to use the provided function signature as a wrapper function for a recursive helper that performs a DFS on the graph (it's technically possible to solve this without a helper, but only making really creative use of the auxiliary storage inside the BasicGraph). One thing to notice is that the wrapper does NOT need to iterate over all vertices and launch a DFS from each one (as is a common solution template for many of our practice exam and section problems). A Hamiltonian Cycle creates a continuous chain that touches every vertex, so you once you have a Hamiltonian Cycle ordering, you could "rotate" it so that it starts/ends at any other vertex, and you would have another Hamiltonian Cycle. Therefore, it doesn't matter which vertex you use as the start of the DFS, and you only need to use one arbitrarily chosen one.

```
// Sample Solution 1
Vector<Vertex*> findHamCycle(BasicGraph& graph) {
  Vector<Vertex*> currentItinerary;
  Set<Vertex*> vertices = graph.getVertexSet();
  if (vertices.size() < 2) {
      return currentItinerary;
  }
  Vertex *start = vertices.first();
  start->visited = true;
  currentItinerary.add(start);
  for (Vertex *neighbor : graph.getNeighbors(start)) {
      if (goDFS(graph, neighbor, start, currentItinerary)) {
          return currentItinerary;
      }
  }
  currentItinerary.clear();
```

```
  return currentItinerary;
}

bool goDFS(BasicGraph &graph, Vertex *current, Vertex *start,
         Vector<Vertex*> &currentItinerary) {
  if (current == start && currentItinerary.size() != graph.size()) {
      return false;
  }
  if (current->visited && current != start) {
      return false;
  }
  current->visited = true;
  currentItinerary.add(current);
  if (currentItinerary.size() == graph.size() + 1 && current == start) {
      return true;
  }
  for (Vertex *neighbor : graph.getNeighbors(current)) {
      if (goDFS(graph, neighbor, start, currentItinerary)) {
          return true;
      }
  }
  current->visited = false;
  currentItinerary.remove(currentItinerary.size()-1);
  return false;
}

// Sample Solutions 2 & 3
// This wrapper only checks one possible starting vertex
Vector<Vertex*> findHamCycle(BasicGraph& graph) {
   Vector<Vertex*> path;
   Set<Vertex*> toVisit = graph.getVertexSet();
   path.add(toVisit.first());
   if (helper(graph, path, toVisit))
       return path;
   path.clear();
   return path;
}

// This wrapper checks all possible starting vertices
Vector<Vertex*> findHamCycle(BasicGraph& graph) {
   Vector<Vertex*> path;
   for (Vertex* v : graph.getVertexSet()) {
       path.add(v);
       Set<Vertex*> toVisit = graph.getVertexSet(); // Need a copy to pass by ref.
       if (helper(graph, path, toVisit))
           return path;
       path.clear();
   }
   return path;
}


// This helper goes with both wrappers above
bool helper(BasicGraph& graph, Vector<Vertex*>& path, Set<Vertex*>& toVisit) {
```

```
    if (toVisit.isEmpty())
        return true;

    for (Vertex* next : graph.getNeighbors(path[path.size() - 1])) {
        if (toVisit.contains(next) && (toVisit.size() == 1 || next != path[0])) {
            path.add(next);
            toVisit.remove(next);
            if (helper(graph, path, toVisit))
                return true;
            path.remove(path.size() - 1);
            toVisit.add(next);
        }
    }
    return false;
}
```

---

6. **Trees**

```
/* APPROACH #1 observes that we really wish to return two values: (1) boolean subtree
* is valid, and (2) int sum of leaves in this subtree. It achieves this using a
* helper with two pass-by-reference parameters. This wrapper calls a helper. The sum
* is not needed by the wrapper (only needed inside the recursion).
*/
bool isValidSumTree_Approach1(TreeNode *tree){
  int sum = 0;
  bool isValid = false;
  isValidHelper(tree, sum, isValid);
  return isValid;
}


/* APPROACH #1 helper:
* This recursive helper has these two parameters. It performs a post-order traversal
* to gather the sum and then check self for validity.
*
* sum and isValid are used as OUTPUT ONLY (essentially return values)
*/
void isValidHelper(TreeNode *tree, int &sum, bool &isValid) {
  // empty tree is trivially valid
  if (tree == NULL) {
      isValid = true;
      sum = 0;
      return;
  }

  // leaf
  if (tree->left == NULL && tree->right == NULL) {
      // leaf cannot have -1 (or any non-negative) key
      if (tree->key < 0) {
          isValid = false;
          sum = 0;
          return;
      }
      // any other key is fine for leaf
```

```
        isValid = true;
        sum = tree->key;
        return;
    }

    // post-order traversal for non-leaves
    bool leftIsValid = false;
    bool rightIsValid = false;
    int leftSum = 0;
    int rightSum = 0;
    isValidHelper(tree->left, leftSum, leftIsValid);
    isValidHelper(tree->right, rightSum, rightIsValid);
    sum = leftSum + rightSum;
    // check for problems
    if (!leftIsValid || !rightIsValid /* subtree invalid */
            || (tree->key != -1 && tree->key != sum) /* sum is wrong */) {
        isValid = false;
        sum = 0;
        return;
    }

    isValid = true;
  }

  /* APPROACH #2 just re-traverses the tree to gather the descendent leaves' sum at
  * every node. Inefficient but easy to write. */
  bool isValidSumTree_Approach2(TreeNode *tree){
    // empty tree is trivially valid
    if (tree == NULL) {
        return true;
    }

    // leaf is valid if key is non-negative
    if (tree->left == NULL && tree->right == NULL) {
        return tree->key >= 0;
    }

    // check our own key for problems
    if ((tree->key != -1
            && tree->key != sumLeaves(tree->left) + sumLeaves(tree->right))) {
        return false;
    }

    // recursively check our subtrees for problems
    if (!isValidSumTree_Approach2(tree->left)
        || !isValidSumTree_Approach2(tree->right)) {
        return false;
    }

    return true;
  }

  /* APPROACH #2 helper calculates the sum of descendant leaves. It assumes
```

```
* the tree is valid, so that must be checked separately. */
int sumLeaves(TreeNode *tree) {
  // null contributes nothing to sum
  if (tree == NULL) return 0;

  // is leaf?
  if (tree->left == NULL && tree->right == NULL) return tree->key;

  // traversal to sum leaves (ignore own key since we are not leaf)
  return sumLeaves(tree->left) + sumLeaves(tree->right);
}

/* APPROACH #3 overwrites -1 keys with the actual sum, making checking children by
* the parent trivial. */
bool isValidSumTree_Approach3(TreeNode *tree) {
  // empty tree is trivially valid
  if (tree == NULL) {
      return true;
  }

  // leaf is valid if key is non-negative
  if (tree->left == NULL && tree->right == NULL) {
      return tree->key >= 0;
  }

  // traversal of children will set both left and right children's keys
  // to the actual sums (overwriting -1 if necessary), and also check
  // left and right subtrees for validity
  if (!isValidSumTree_Approach3(tree->left)
          || !isValidSumTree_Approach3(tree->right)) {
      return false;
  }

  // check our own key for problems
  int leftSum = 0;
  int rightSum = 0;
  if (tree->left != NULL) leftSum = tree->left->key;
  if (tree->right != NULL) rightSum = tree->right->key;
  if (tree->key != -1 && tree->key != leftSum + rightSum) {
      return false;
  }

  // make sure our key is actual sum (not -1), to make parent's job of checking
  // its key easier
  tree->key = leftSum + rightSum;

  return true;
}
```

7. **Inheritance**
    - E 1 / M 1
    - Compiler error
    - A 3
    - E 1 / M 1
    - P 2
    - P 2 / E 3
    - P 2
    - P 2
    - E 3
    - E 1 / M 1
    - P 2
    - E 3
    - Crash
    - E 1 / M 1
    - P 2
    - Compiles: NO