CS106X
Winter 2017

Chris Gregg

Section Handout #5 Solutions

If you have any questions about the solutions to the problems in this handout, feel free to reach out to your section leader, Aaron, or Chris for more information.

1. No, *You're* Out of Order

a. pre-order: 3 5 1 2 4 6
in-order: 1 5 3 4 2 6
post-order: 1 5 4 6 2 3

b. pre-order: 19 47 23 -2 55 63 94 28
in-order: 23 47 55 -2 19 63 94 28
post-order: 23 55 -2 47 28 94 63 19

c. pre-order: 2 1 7 4 3 5 6 9 8
in-order: 2 3 4 5 7 1 6 8 9
post-order: 3 5 4 7 8 6 1 2

2. Height

```
int height(TreeNode *node) {
  if (node == nullptr) {
    return 0;
  } else {
    return 1 + max(height(node->left), height(node->right));
  }
}
```

3. Count Left Nodes

```
int countLeftNodes(TreeNode *node) {
  if (node == nullptr) {
    return 0;
  } else if (node->left == nullptr) {
    return countLeftNodes(node->right);
  } else {
    return 1 + countLeftNodes(node->left) + countLeftNodes(node->right);
  }
}
```

4. Balanced

```
bool isBalanced(TreeNode *node) {
  if (node == nullptr) {
    return true;
  } else if (!isBalanced(node->left) || !isBalanced(node->right)) {
      return false;
  } else {
    int leftHeight  = height(node->left);   // using code from a previous problem
    int rightHeight = height(node->right);
    return abs(leftHeight - rightHeight) <= 1;
  }
}
```

5. Prune a Tree

```
void pruneTree(TreeNode *&node) {
  if (node != nullptr) {
    if (node->left == nullptr && node->right == nullptr) {
        delete node;
        node = nullptr;
    } else {
        pruneTree(node->left);
        pruneTree(node->right);
    }
  }
}
```

6. Complete To Level

```
void completeToLevel(StringNode *&node, int k) {
  if (k < 1) {
    throw k;
  } else {
    completeToLevelHelper(node, k, 1);
  }
}

void completeToLevelHelper(StringNode *&node, int k, int level) {
  if (level <= k) {
    if (node == nullptr) {
      node = new StringNode;
      node->str = "??";
    }
    completeToLevelHelper(node->left,  k, level + 1);
    completeToLevelHelper(node->right, k, level + 1);
  }
}
```

7. Word Trees

```
bool wordExists(CharNode *node, string str) {
  if (str.empty()) {
    return true;   // the empty tree contains the empty string
  } else if (node == nullptr) {
    return false;
  }  else if (suffixExists(node, str)) {
    return true;
  } else {
    return wordExists(node->left, str) || wordExists(node->right, str);
  }
}
```

```
// helper to search the given subtree for the rest of the string
bool suffixExists(CharNode *node, string suffix) {
  if (suffix.empty()) {
    return true;
  } else if (node == nullptr) {
    return false;
  } else if (suffix[0] != node->ch) {
    return false;
  } else {
    return suffixExists(node->left, suffix.substr(1))
        || suffixExists(node->right, suffix.substr(1));
  }
}
```

8. List to Tree
Provided are two solutions to this problem. The first is recursive, and is a bit shorter.

```
TreeNode *listToBinaryTree(ListNode *front) {
  if (front == NULL) return NULL;
  TreeNode *root = new TreeNode;
  root->data = head->data;
  root->left = listToBinaryTree(front->next);
  root->right = listToBinaryTree(front->next);
  return root;
}
```

The second solution is iterative, and uses a Queue of double pointers to line up the locations of the `TreeNode *`s that need to be considered during the next iteration. It's trickier to understand, but it's good practice for the types of operations you can do with double pointers, an advanced programming technique.

```
TreeNode *listToBinaryTree(ListNode *front) {
  TreeNode *root;
  Queue<TreeNode **> children;
  children.enqueue(&root);
  for (ListNode* curr = head; curr != nullptr; curr = curr->next) {
    int numChildren = children.size(); // take a snapshot of the size
    for (int i = 0; i < numChildren; i++) {
      TreeNode **nodePtr = children.dequeue();
      *nodePtr = new TreeNode;
      (*nodePtr)->data = curr->data;
      children.enqueue(&(*nodePtr)->left);
      children.enqueue(&(*nodePtr)->right);
    }
  }

  // everything in Queue points to what needs to be NULLed out
  while (!children.isEmpty()) {
    TreeNode **nodep = children.dequeue();
    *nodep = nullptr;
  }
  return root;
}
```

9. Cartesian Trees

```
int findIndexOfMinimum(Vector<int> &inorder, int low, int high) {
  int index = low;
  for (int i = low + 1; i <= high; i++) {
    if (inorder[i] < inorder[index]) {
      index = i;
    }
  }
  return index;
}

TreeNode *arrayToCartesianTree(Vector<int> &inorder, int low, int high) {
  if (low > high) return nullptr;
  int index = findIndexOfMinimum(inorder, low, high);
  TreeNode *root = new TreeNode;
  root->value = inorder[index];
  root->left = arrayToCartesianTree(inorder, low, index - 1);
  root->right = arrayToCartesianTree(inorder, index + 1, high);
  return root;
}

TreeNode *arrayToCartesianTree(Vector<int> &inorder) {
  return arrayToCartesianTree(inorder, 0, inorder.size() - 1);
}
```

10. Quad Trees

```
QuadTreeNode *gridToQuadtree(Grid<bool> &image, int minX, int maxX, int minY, int maxY) {
  QuadTreeNode *qt = new QuadTreeNode;
  qt->minX = minX;
  qt->maxX = maxX - 1;
  qt->minY = minY;
  qt->maxY = maxY - 1;

  if (allPixelsAreTheSameColor(image, minX, maxX, minY, maxY)) {
    qt->isBlack = image[minX][minY];
    for (int i = 0; i < 4; i++) {
      qt->children[i] = nullptr;
    }
  } else {
    int midX = (maxX - minX) / 2;
    int midY = (maxY - minY) / 2;

    qt->children[0] = gridToQuadtree(image, minX, midX, midY, maxY); // NW
    qt->children[1] = gridToQuadtree(image, midX, maxX, midY, maxY); // NE
    qt->children[2] = gridToQuadtree(image, midX, maxX, minY, midY); // SE
    qt->children[3] = gridToQuadtree(image, minX, midX, minY, midY); // SW
  }

  return qt;
}

QuadTreeNode *gridToQuadtree(Grid<bool> &image) {
  return gridToQuadtree(image, 0, image.numCols(), 0, image.numRows());
}
```