Chris Gregg

Section Handout #7 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. Binary Search Tree Insertion

a.

```
            Jaques
          /          \
  Beatrice            Sunny
          \        /       \
      Bertrand   Klaus    Violet
                /    \
             Kit    Lemony
```

b.

```
            Leslie
          /        \
      Jerry         Ron
     /   \             \
  Garry   Larry         Tom
   /
 April
  /
Andy
```

c.

```
Aaron
    \
     Anu
       \
        Chris
           \
           Colin
              \
              Jason
                 \
                 Leslie
                     \
                     Wesley
```

2. Is It a BST?

```
bool isBST(TreeNode *node) {
  TreeNode *prev = nullptr;
  return isBSTHelper(node, prev);
}
```

```
bool isBSTHelper(TreeNode *node, TreeNode *&prev) {
  if (node == nullptr) {
    return true;
  } else if (!isBSTHelper(node->left, prev) || (prev && node->data <= prev->data)) {
    return false;
  } else {
    prev = node;
    return isBSTHelper(node->right, prev);
  }
}
```

3. Tree Rotations

```
void rotateLeft(TreeNode *&parent) {
  TreeNode *rightChild = parent->right;
  parent->right = rightChild->left;
  rightChild->left = parent;
  parent = rightChild;
}

void pullToRoot(TreeNode *&root, int value) {
  if ((root == nullptr) || (root->value == value)) return; // no reason to rotate
  if (root->value < value) { // value would be in right subtree
    pullToRoot(root->right, value);
    rotateLeft(root);
  } else {
    pullToRoot(root->left, value);
    rotateRight(root);
  }
}
```

4. Level-Order Heaps

```
void levelOrderTraversal(int *heap, int size) {
  for (int i = 0; i < size; i++) {
    cout << heap[i];
  }
  cout << endl;
}
```

5. On the Level

```
void printLevel(int *heap, int level, int size) {
  int count = pow(2, level);
  for (int i = count; (i < count + count) && (i < size); i++) {
    cout << heap[i] << " ";
  }
  cout << endl;
}
```

6. Hash Functions

hash1 is valid (a value *k* will always map to the same hash value, no matter how many times it's called), but it's not good because everything will get hashed to the same bucket.

hash2 is not valid, because "A" and "a" are equal, but will have different hash values.

hash3 is not valid, because equal strings might not give the same hash value.

hash4 is valid and good.

7. Hash It Out

```
   +---+
0  |   |--> 100:14
   +---+
1  | / |
   +---+
2  | / |
   +---+
3  | / |
   +---+
4  | / |
   +---+
5  | / |
   +---+
6  |   |--> 26:5 --> 6:999
   +---+
7  | / |
   +---+
8  |   |--> 8:33
   +---+
9  | / |
   +---+
10 | / |
   +---+
11 |   |--> 31:19
   +---+
12 | / |
   +---+
13 | / |
   +---+
14 | / |
   +---+
15 | / |
   +---+
16 |   |--> 276:55
   +---+
17 | / |
   +---+
18 |   |--> -18:4 --> 18:22
   +---+
19 | / |
   +---+
```

```
size        =  8
capacity    =  20
load factor =  0.4
```

8. Hashing and Rehashing

Before Rehashing

```
0: baggage (30)
1: badcab (13)
2: feed (20) -> deadbeef (32)
3: cafe (15) -> cabbage (21)
4: null
5: null
```

After Rehashing

```
0: null
1: badcab (13)
2: null
3: cafe (15)
4: null
5: null
6: baggage (30)
7: null
8: deadbeef (32) -> feed (20)
9: cabbage (21)
10: null
11: null
```

9. Open Addressing

```
class OpenHashTable {
public:
  OpenHashTable();
  ~OpenHashTable();

  void put(string key, string value);
  string get(string key);
  void remove(string key);
private:
  OpenEntry *buckets;
  int size;

  int findIndex(string key);
  bool shouldMove(int src,
    int dest);
  int hash(string key);

  struct OpenEntry {
    string key;
    string value;
    bool inUse;
  };
};
```

```
const int CAPACITY = 10;

OpenHashTable::OpenHashTable() {
  buckets = new OpenEntry[CAPACITY];
  size = CAPACITY;

  for (int i = 0; i < size; i++) {
      buckets[i].inUse = false;
  }
}

OpenHashTable::~OpenHashTable() {
  delete[] buckets;
}

void OpenHashTable::put(string key, string value) {
  int hashValue = hash(key) % size;
  int idx = hashValue;

  while (buckets[idx].inUse) {
    if (buckets[idx].key == key) {
      break;
    }

    idx = (idx + 1) % size;
    if (idx == hashValue) {
      throw "The hash map is full!";
    }
  }

  buckets[idx].inUse = true;
  buckets[idx].key = key;
  buckets[idx].value = value;
}
```

```cpp
int OpenHashTable::search(string key) {
  int hashValue = hash(key) % size;
  int idx = hashValue;

  while (buckets[idx].inUse) {
    if (buckets[idx].key == key) {
      return idx;
    }

    idx = (idx + 1) % size;
    if (idx == hashValue) {
      return -1;
    }
  }

  return -1;
}

string OpenHashTable::get(string key) {
  int idx = search(key);
  if (idx == -1) {
    return "";
  }

  return buckets[idx].value;
}

bool OpenHashTable::shouldMove(int src, int dst) {
  int srcHash = hash(buckets[src].key) % size;
  if (src > dst) {
    // moving an element earlier in the array
    return srcHash <= dst || srcHash > src;
  } else if (src < dst) {
    return srcHash <= dst && srcHash > src;
  }
}
```

```
void OpenHashTable::remove(string key) {
  int idx = search(key);
  if (idx == -1) {
    return;
  }

  buckets[idx].inUse = false;
  int dest = idx++;

  while (idx != dest) {
    if (!buckets[idx].inUse) {
      // found a gap; done
      break;
    }

    int hashValue = hash(buckets[idx].key) % size;
    if (shouldMove(idx, dest)) {
      // found an element to move up
      buckets[dest] = buckets[idx];
      buckets[idx].inUse = false;
      dest = idx;
    }

    idx = (idx + 1) % size;
  }
}
```