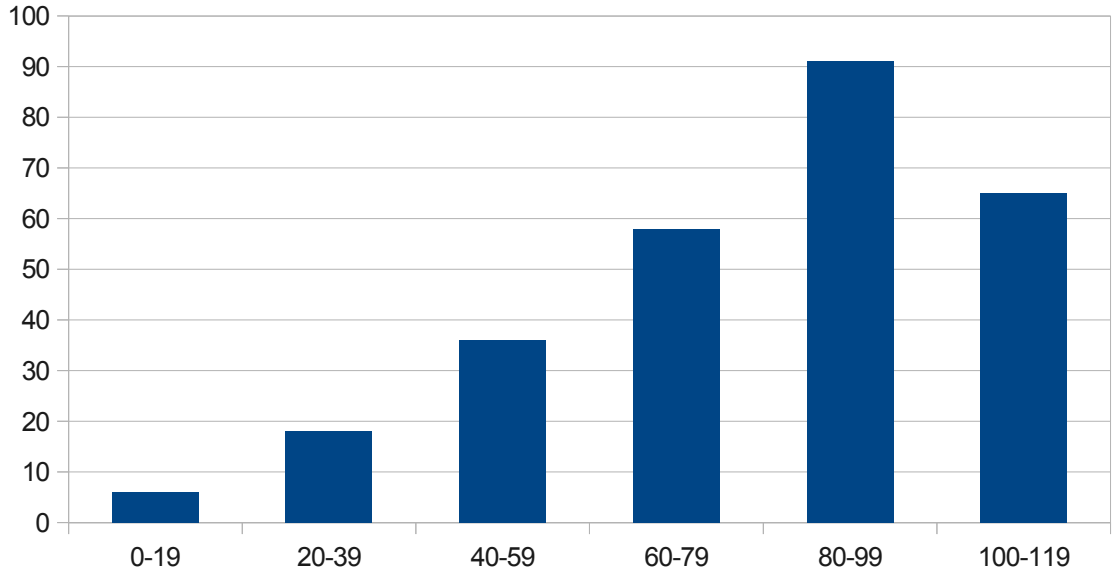
CS106B Midterm Exam #2 Solutions

Below is the score distribution for the second midterm exam:



Overall, the final statistics were as follows:

Mean: 79 / 120

Median: 85 / 120

Standard Deviation: 25

We are **not** grading this course using raw point totals and will instead be grading on a (fairly generous) curve. Roughly speaking, the median score corresponds to roughly a B/B+. As always, if you have any comments or questions about the midterm or your grade on the exam, please don't hesitate to drop by office hours with questions! You can also email Zach or Keith with any questions.

If you think that we made any mistakes in our grading, please feel free to submit a regrade request to us. Just write a short (one-paragraph or so) description of what you think we graded incorrectly, staple it to the front of your exam, and hand your exam to either Zach or Keith by Monday, June 11 at 2:15PM. We reserve the right to regrade your entire exam if you submit it for a regrade.

Problem One: Perfect Hash Tables (40 Points)

Here is one possible implementation:

```
PerfectHashTable::PerfectHashTable(Set<string>& values) {
    /* Set up the buckets array and number of elements. */
    numBuckets = values.size() * values.size();
    elems = new Bucket[numBuckets];

    /* Keep picking hash functions until we find one that works. */
    while (true) {
        whichFunction = randomInteger(INT_MIN, INT_MAX);

        /* Mark all buckets as empty. */
        for (int i = 0; i < numBuckets; i++) {
            elems[i].isUsed = false;
        }

        /* Hash each element into a bucket and see if we can do so
         * without collisions.
         */
        bool success = true;
        foreach (string value in values) {
            int bucket = hashCode(value, whichFunction) % numBuckets;
            if (elems[bucket].isUsed) {
                success = false;
                break;
            }
            elems[bucket].value = value;
            elems[bucket].isUsed = true;
        }

        if (success) break;
    }
}

PerfectHashTable::~PerfectHashTable() {
    delete[] elems;
}

bool PerfectHashTable::containsKey(string value) {
    int bucket = hashCode(value, whichFunction) % numBuckets;
    return elems[bucket].isUsed && elems[bucket].value == value;
}
```

Problem Two: Encoding Trees (30 Points)

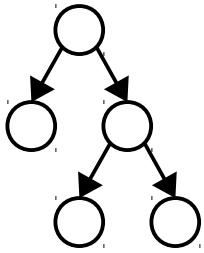
(i) Implementing Tree Encoding (20 Points)

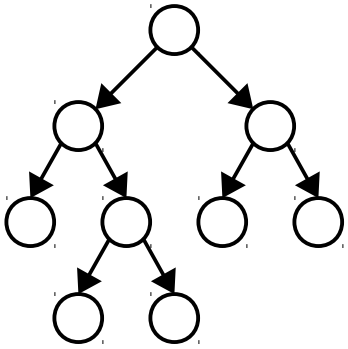
```
void compressTree(Node* root, obstream& out) {
    if (root == NULL) {
        out.writeBit(0);
    } else {
        out.writeBit(1);
        compressTree(root->left, out);
        compressTree(root->right, out);
    }
}

Node* decompressTree(ibstream& in) {
    if (in.readBit() == 0) {
        return NULL;
    } else {
        Node* result = new Node;
        result->left = decompressTree(in);
        result->right = decompressTree(in);
        return result;
    }
}
```

(ii) Encoding Encoding Trees (10 Points)

One encoding system is as follows: A node with no children is written as a `0`, and a node with two children is written as `1`, then the encoding of its left subtree, then the encoding of its right subtree. These trees have the following encodings:





Left tree: `10100`

Right tree: `110100100`

The code for this (which wasn't required on the exam) is actually quite short:

```
void compressPureTree(Node* root, obstream& out) {
    if (root->left == NULL && root->right == NULL) {
        out.writeBit(0);
    } else {
        out.writeBit(1);
        compressPureTree(root->left, out);
        compressPureTree(root->right, out);
    }
}
```

Problem Three: Spaghetti Stacks (40 Points)

(i) Making Spaghetti (20 Points)

There are two main approaches that can be used to solve this problem. The first approach is to recursively convert each tree to a spaghetti stack by constructing each tree with knowledge of its parent. This is shown here:

```
Set<SpaghettiNode*> spaghettify(TreeNode* root) {
   Set<SpaghettiNode*> result;
   spaghettifyRec(root, NULL, result);
   return result;
}

/* Builds a spaghetti stack from root whose parent in the spaghetti stack is the
 * node parent.
 */
void spaghettifyRec(TreeNode* root, SpaghettiNode* parent,
                    Set<SpaghettiNode>& result) {
    /* If there is nothing to build, we're done. */
    if (root == NULL) return;

    /* Construct a new spaghetti node wired into the parent. */
    SpaghettiNode* sNode = new SpaghettiNode;
    sNode->value = root->value;
    sNode->parent = parent;

    /* If this is a leaf node, add it to the result set. */
    if (root->children.isEmpty()) {
        result += sNode;
    }
    /* Otherwise, build up all the children of this node as spaghetti stacks
     * that use the current node as a parent.
     */
    else {
        foreach (TreeNode* child in root->children) {
            spaghettifyRec(child, sNode, result);
        }
    }
}
```

The other major approach is to spaghettify each tree recursively and have the recursive function hand back a pointer to the root node of the new tree. Given this pointer, we can then change its parent to the new node that we constructed.

```
/* Builds a spaghetti stack from root whose parent in the spaghetti stack is the
 * node parent.
 */
SpaghettiNode* spaghettifyRec(TreeNode* root, Set<SpaghettiNode>& result) {
    /* If there is nothing to build, we're done. */
    if (root == NULL) return NULL;

    /* Construct a new spaghetti node with no parent. */
    SpaghettiNode* sNode = new SpaghettiNode;
    sNode->value = root->value;
    sNode->parent = NULL;

    /* If this is a leaf node, add it to the result set. */
    if (root->children.isEmpty()) {
        result += sNode;
    }
    /* Otherwise, build up all the children of this node as spaghetti stacks
     * that use the current node as a parent.
     */
    else {
        foreach (TreeNode* child in root->children) {
            spaghettifyRec(child, result)->parent = sNode;
        }
    }

    /* Return this node as the root of the new spaghetti stack. */
    return sNode;
}
```

(ii) Cleaning up Spaghetti **(20 Points)**

The tricky part of this problem was finding a way to clean up the tree without deleting the same node twice. Simply walking up from each leaf node to the root cleaning as you go will cause problems if any node has two children.

There were two main approaches we saw for solving this problem. The first option was to just build up a set of all the nodes in the spaghetti stack, then to free each of them.

```
void freeSpaghettiStack(Set<SpaghettiNode*> leaves) {
    Set<SpaghettiNode*> nodes;
    foreach (SpaghettiNode* leaf in leaves) {
        for (SpaghettiNode* curr = leaf; curr != NULL; curr = curr->parent) {
            nodes += curr;
        }
    }

    foreach (SpaghettiNode* node in nodes) {
        delete node;
    }
}
```

Another option is to walk the tree deleting nodes and keeping track of what was deleted so that we don't delete anything twice.

```
void freeSpaghettiStack(Set<SpaghettiNode*> leaves) {
    Set<SpaghettiNode*> deletedNodes;
    foreach (SpaghettiNode* leaf in leaves) {
        while (leaf != NULL) {
            if (deletedNodes.contains(leaf)) break;
            deletedNodes += leaf;

            SpaghettiNode* next = leaf->parent;
            delete leaf;
            leaf = next;
        }
    }
}
```

Problem Four: The Good, the Bad, and the Average (10 Points)

Fill in the following table with the big-O runtimes of each of the following operations in the best-case, worst-case, and average-case. The first row has been filled in for you.

	Best-Case	Worst-Case	Average-Case
Searching an unsorted `Vector` of *n* `int`s for a value using linear search.	O(1)	O(*n*)	O(*n*)
Looking up a value in a chained hash table (the version we built in lecture) containing *n* `int`s.	**O(1)**	**O(*n*)**	**O(1)**
Inserting a value into a treap containing *n* `int`s.	**O(1)**	**O(*n*)**	**O(log *n*)**
Determining whether a string of length *L* is a prefix in a trie holding *n* words.	**O(1)**	**O(*L*)**	*n.a.*

Here is a description of each answer:

- **Hash table lookup, best-case:** Looking up a value in an empty bucket or in a bucket with at most a constant number of elements in it will run in O(1).
- **Hash table lookup, worst-case:** If all *n* elements in the hash table hash to the same bucket, then a lookup might require worst-case O(*n*) time to look at all of them.
- **Hash-table lookup, average-case:** The expected number of elements in a bucket, assuming that the load factor is kept at a constant, is O(1).
- **Treap insertion, best-case:** The treap could have the maximum value in the tree stored at the root. Inserting a larger value, in this case, is O(1) because we have to follow one pointer and do one rotation.
- **Treap insertion, worst-case:** The treap could degenerate into a linked list, in which case insertion can take time O(*n*).
- **Treap insertion, average-case:** The expected height of a treap is O(log *n*), so on expectation insertion takes time O(log *n*) to walk to the bottom of the tree, then does at most O(log *n*) rotations to pull the node up.
- **Trie prefix search, best-case:** The first letter of the prefix could not be in the trie, in which case the search terminates immediately in time O(1).
- **Trie prefix search, worst-case:** We might have to look at all letters of the prefix without walking off the trie, which takes time O(*L*).