CS106B Midterm Review Session Notes

Dawson Zhou

Type Definitions

For all of the problems involving linked lists, we use the following struct:

```
struct Cell {
    int value;
    Cell* next;
};
```

For all of the problems involving binary search trees, we use the following struct:

```
struct Node {
    int value;
    Node* left;
    Node* right;
};
```

Problem 1

You are given pointers to the heads of two linked lists, both of which have their elements in sorted increasing order. Write a function which merges the two lists and returns a pointer to the resulting list, which should contain all of the elements of the two lists in sorted increasing order.

You should not allocate any memory for this problem.

```
Cell* merge(Cell* listA, Cell* listB) {
    // If either list is NULL, the problem is a lot simpler; just return
    // the other list as the merged result.
    if (listA == NULL) {
        return listB;
    }
    if (listB == NULL) {
        return listA;
    }

    // result will point to the head of the merged list, but we need to
    // store the tail as well so that we can add elements to the end of
    // the list.
    Cell* result = NULL;
    Cell* tail = NULL;
    // Since we checked for listA and listB being NULL first, we know that
    // the following loop will execute at least once. That means result and
    // tail will both be non-NULL.

    // We want to stop as soon as one list or the other is NULL. When that
    // happens, we can take the other list and just attach it at the end of
    // the merged list.
    while (listA != NULL && listB != NULL) {
        if (listA->value < listB->value) {
            // Save the cell we're moving to the merged list, and then
            // update listA:
            Cell* nextCell = listA;
            listA = listA->next;
            nextCell->next = NULL;

            // We need to do something different depending on whether this
            // is the first item being added to the merged list.
            if (result == NULL) {
                result = nextCell;
                tail = nextCell; // can be moved outside the if-block
            } else {
                tail->next = nextCell;
                tail = nextCell; // can be moved outside the if-block
            }
        // Do the same thing as above, but with listB (see comment below).
        } else {
            Cell* nextCell = listB;
            listB = listB->next;
            nextCell->next = NULL;

            if (result == NULL) {
                result = nextCell;
                tail = nextCell;
            } else {
                tail->next = nextCell;
                tail = nextCell;
            }
        }
    }

    // Take whichever list isn't NULL and add it to the end of the result
    // list.
    if (listA == NULL) {
        tail->next = listB;
    } else if (listB == NULL) {
        tail->next = listA;
    }

    return result;
}
```

Notice that there's a huge amount of duplicated code here. It might actually be beneficial to practice decomposition on your exam if it means you don't have to spend an extra fifteen minutes doing a manual copy and paste. Here is one way to do it:

```
// We pass in the pointers by reference, since they might need to be updated.
void updateList(Cell*& result, Cell*& tail, Cell*& list) {
    Cell* nextCell = list;
    list = list->next;
    if (result == NULL) {
        result = nextCell;
    } else {
        tail->next = nextCell;
    }

    // In either case, update tail.
    tail = nextCell;
}
```

Then, the `while` loop looks as follows:

```
    while (listA != NULL && listB != NULL) {
        if (listA->value < listB->value) {
            updateList(result, tail, listA);
        } else {
            updateList(result, tail, listB);
        }
    }
```

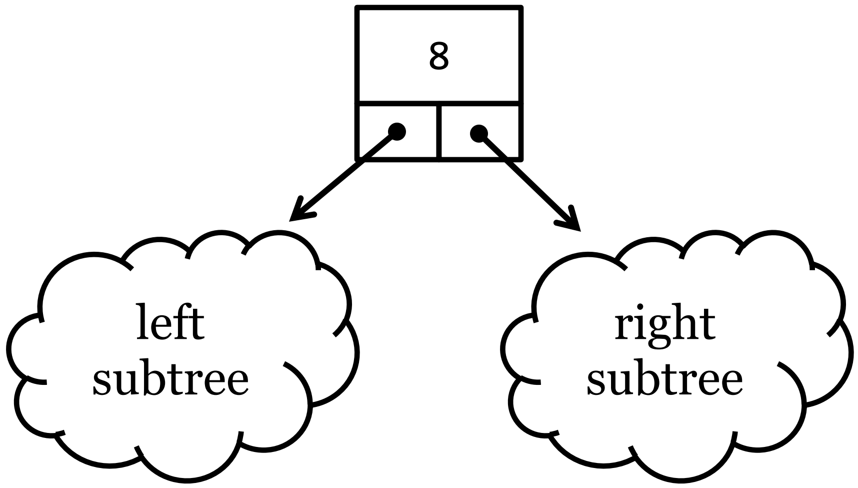
Problem 2

The height of a binary tree is defined as the number of nodes in the longest path from the root node to any leaf node (i.e. a node without children). Write a function which calculates the height of a binary tree, returning it as an integer.

```
int height(Node* root) {
    if (root == NULL) {
        return 0;
    }

    int leftHeight = height(root->left);
    int rightHeight = height(root->right);
    return max(leftHeight, rightHeight) + 1;
}
```

The reason this solution can be so short is because of how it takes advantage of the recursive nature of binary trees. Suppose we have the following tree structure:



The tree can be thought of as a node containing the value 8, with pointers to two children. Each of those children are themselves smaller tree structures.

If we want to find the height of the tree with 8 as the root, we should first find the heights of the left subtree and the right subtree. The height of the overall tree is defined by the longest path to any leaf node; that path will go through whichever subtree has the larger height. Therefore, the height of the overall tree is simply the larger number between the left tree's height and the right tree's height, incremented by one to account for the 8 node.

Problem 3

We want to write a function which takes in a pointer to the root of a binary tree and makes a deep copy of the entire structure. It should return a pointer to the root of the cloned tree.

```
Node* copyTree(Node* root) {
    if (root == NULL) {
        return NULL;
    }

    Node* rootCopy = new Node;
    rootCopy->value = root->value;
    rootCopy->left = copyTree(root->left);
    rootCopy->right = copyTree(root->right);

    return rootCopy;
}
```

Again, this solution is very concise due to the recursive structure of binary trees. If we want to make a deep copy of the original tree, we first need to have a new node which has the same value as the root node of the original tree. The left child of the copy's root should be a deep copy of the left child of the original root (and likewise for the right).

Problem 4

You are given a pointer to the root of a binary search tree. Write a function which “flattens” the tree. It should return a pointer to the head of a linked list, where the linked list contains the same elements that were in the tree, in sorted order.

```
Cell* flatten(Node* root) {
    if (root == NULL) {
        return NULL;
    }

    Cell* curr = new Cell;
    curr->value = root->value;

    Cell* left = flatten(root->left);
    Cell* right = flatten(root->right);

    // Even if right is NULL, this is safe (and sensible) to do.
    curr->next = right;

    // We have to special case what happens when the left list is NULL.
    // In that case, there are no elements smaller than the one at curr,
    // so we can just return curr.
    if (left == NULL) {
        return curr;
    }

    // curr should be hooked onto the end of the last cell in the linked
    // list pointed at by left.
    Cell* tail = left;
    while (tail->next != NULL) {
        tail = tail->next;
    }
    tail->next = curr;

    return left;
}
```

Once again, this solution works by considering the recursive formulation of the question. Consider the tree diagram at the top of page 4. If we flatten that tree, we know that the cell which contains 8 must appear somewhere in the resulting list. Furthermore, because the original tree is a binary search tree, we know that the left subtree, once flattened, becomes a linked list of values that are smaller than 8. Finally, the right subtree, once flattened, becomes a linked list of values that are greater than 8. The cell containing 8 should simply go in between these two lists.

The above analysis tells us that we can flatten the left and right subtrees recursively, concatenate the cell corresponding to the root node to the end of the left flattened list, and then concatenate the right flattened list to the end of that.

An alternate approach performs an inorder traversal of the binary search tree, building up the linked list along the way. The inorder traversal guarantees that we visit nodes in sorted order, so that the linked list is built in the proper order.

```
Cell* flatten(Node* root) {
    Cell* head = NULL;
    Cell* tail = NULL;
    flattenHelper(root, head, tail);
    return head;
}

// We need to pass these pointers in by reference, since they're being
// modified along the way.
void flattenHelper(Node* root, Cell*& head, Cell*& tail) {
    if (root == NULL) {
        return;
    }

    // First visit all of the nodes that precede root.
    flattenHelper(root->left, head, tail);

    // Then, process root.
    Cell* curr = new Cell;
    curr->value = root->value;

    // If head is NULL, this is the first cell being added to the list.
    if (head == NULL) {
        head = curr;
        tail = curr;
    // Otherwise, we'll just append the cell to the end and update tail.
    } else {
        tail->next = curr;
        tail = tail->next;
    }

    // Finally, visit all of the nodes that follow root.
    flattenHelper(root->right, head, tail);
}
```

Problem 5

Using the `OurMap` class defined in lecture, implement the `rehash` function, which is called whenever there are too many elements. `rehash` should double the number of buckets and reassign all of the existing elements in the hash table to their new locations.

You should only allocate memory for a new buckets array. You should not allocate any memory for individual linked list cells.

```
// In the .h file:
class OurMap {
public:
    // ...

private:
    // Each bucket is a linked list. Each linked list consists of Cells.
    struct Cell {
        string key;
        int value;
        Cell* next;
    };

    // Instance variables:
    Cell** buckets;
    int numBuckets;
    int numElements;

    // What we're writing:
    void rehash();
};
```

```
// In the .cpp file:
void OurMap::rehash() {
    // Our new hash table will have double the number of buckets. We still
    // want to remember the old number so we can perform the transfer.
    int newNumBuckets = numBuckets * 2;
    Cell** newBuckets = new Cell*[newNumBuckets];
    for (int i = 0; i < newNumBuckets; i++) {
        newBuckets[i] = NULL;
    }

    // Here, we’re iterating over the old buckets array. We want to look at
    // every single element that’s being stored there right now; to do so,
    // we’ll use this outer for loop to check out all of the buckets, and for
    // each bucket (which is a linked list), we’ll use a while loop to
    // inspect every cell.
    for (int i = 0; i < numBuckets; i++) {
        Cell* curr = buckets[i];
        while (curr != NULL) {
            // Before we mess around with curr’s pointers, we want to save
            // the address of the Cell that follows curr (i.e. the element
            // we’ll process next).
            Cell* nextSaved = curr->next;

            // We use hashCode to compute the index into the new buckets
            // array where this element belongs. The modulus operator (%)
            // calculates the remainder when the result of hashCode is
            // divided by the increased number of buckets, which gives us
            // a valid index into that array.
            int newBucketIndex = hashCode(curr->key) % newNumBuckets;

            // There might already be elements in that bucket, so we’ll
            // prepend curr to the front of that list. To do so, we’ll say
            // that everything in that list will follow curr, and then set
            // curr to be the new head of that list.
            curr->next = newBuckets[newBucketIndex];
            newBuckets[newBucketIndex] = curr;

            // Now, we can move on to the next Cell we need to move.
            curr = nextSaved;
        }
    }

    // Clean up the old buckets array, and move to the new one:
    delete[] buckets;
    buckets = newBuckets;
    numBuckets = newNumBuckets;
}
```