CS 106B Spring 2022 Practice Final Exam Key

1.

```
ListNode *mergeListsAndReverse(ListNode *listA, ListNode *listB){
    ListNode *reversed = nullptr;
    ListNode *tmp;
    // traverse both lists and compare ListNodes
    // don't stop until both lists are nullptr
    while (listA != nullptr || listB != nullptr) {
        // only check data if listA is not nullptr
        if (listA != nullptr
            && (listB == nullptr || listA->data < listB->data)) {
            // rewire listA
            tmp = listA->next;
            listA->next = reversed;
            reversed = listA;
            listA = tmp;
        } else {
            // rewire listB
            tmp = listB->next;
            listB->next = reversed;
            reversed = listB;
            listB = tmp;
        }
    }
    return reversed;
}
```

Problem 2:

```
// Do not modify BST_Node
struct BST_Node {
    int data;
    BST_Node *left;
    BST_Node *right;
};

class BST {
// Do not modify public class members
public:
    BST();  // constructor
    ~BST(); // destructor

    // insert value into BST, ignoring duplicates
    void insert(int value);

    // removes all nodes less than min and greater than max
    void removeOutOfRangeNodes(int min, int max);

    // prints the in-order representation of the tree
    // in the following form:
    // 1,2,3,4
    // with an endl after the list of values.
    void printTreeInOrder();

private:
    BST_Node *root;

    // helper function for insert
    void insert(BST_Node *&n, int value);

    // cleans up the BST
    void deleteTree(BST_Node *n);

    // add more private functions if needed
    BST_Node *removeOutOfRangeNodes(BST_Node *current, int min, int
max);
    void printTreeInOrder(BST_Node *n, int lefts);
};
```

```
// ----------------------------------------

// bst.cpp

// constructor
BST::BST() {
    root = nullptr;
}

// destructor
BST::~BST() {
    deleteTree(root);
}

void BST::insert(int value) {
    insert(root,value);
}

// helper function for insert
void BST::insert(BST_Node *&n, int value) {
    if (n == nullptr) {
        n = new BST_Node;
        n->data = value;
        n->left = nullptr;
        n->right = nullptr;
    } else {
        if (value < n->data) {
            insert(n->left, value);
        } else if (value > n->data) {
            insert(n->right, value);
        }
    }
}

void BST::removeOutOfRangeNodes(int min, int max) {
    root = removeOutOfRangeNodes(root, min, max);
}


BST_Node *BST::removeOutOfRangeNodes(BST_Node *current,
                                     int min, int max) {
    if (current == nullptr) {
        return nullptr;
    }

    current->left = removeOutOfRangeNodes(current->left, min, max);
    current->right = removeOutOfRangeNodes(current->right, min, max);

    if (current->data < min) {
```

```
        BST_Node *temp = current->right;
        delete current;
        return temp;
    }

    if (current->data > max) {
        BST_Node *temp = current->left;
        delete current;
        return temp;
    }

    return current;
}

void BST::printTreeInOrder() {
    printTreeInOrder(root,0);
}

void BST::printTreeInOrder(BST_Node *n, int lefts) {
    if (n == nullptr) return;
    printTreeInOrder(n->left, lefts+1);
    cout << n->data;
    if ((lefts == 0 && n->right == nullptr)) {
        // we are at the max node
        cout << endl;
    } else {
        cout << ", ";
    }
    printTreeInOrder(n->right, lefts);
}

void BST::deleteTree(BST_Node *n) {
    if (n != nullptr) {
        // post-order traversal
        deleteTree(n->left);
        deleteTree(n->right);
        delete n;
    }
}
```

Problem 3:

```
/*
 * Function: filenameMatches
 * Usage: if (filenameMatches(filename, pattern)) . . .
 * ----------------------------------------------------
 * This function checks to see whether filename matches the pattern,
 * which consists of three types of characters:
 *
 * 1. The character ?, which matches any single character
 * 2. The character *, which matches any string of characters
 * 3. Any other character, which matches only that character
 */

bool filenameMatches(string filename, string pattern) {
   if (pattern == "") return (filename == "");
   int n = filename.length();
   switch (pattern[0]) {
    case '?':
      if (filename == "") return false;
      return filenameMatches(filename.substr(1), pattern.substr(1));
    case '*':
      for (int i = 0; i <= n; i++) {
         if (filenameMatches(filename.substr(i), pattern.substr(1))) {
            return true;
         }
      }
      return false;
    default:
      if (filename == "" || pattern[0] != filename[0]) return false;
      return filenameMatches(filename.substr(1), pattern.substr(1));
   }
}
```

Problem 4:

```
void breakSentence(Lexicon &lex, string str, int n, string result) {
    // look at all potential prefixes
    for (int i=0; i < n; i++) {
        // get the prefix
        string prefix = str.substr(0, i+1);

        // if the lexicon contains the prefix, use it,
        // and check the rest of the string;
        // otherwise, we ignore it
        if (lex.contains(prefix)) {
            // if we're at the end of the string, print
            if (i == n-1) {
                result += prefix;
                cout << result << endl;
            } else {
                breakSentence(lex, str.substr(i+1, n-i-1), n-i-1,
                              result + prefix + " ");
            }
        }
    }
}

void breakSentence(Lexicon &lex, string str) {
    breakSentence(lex, str, str.size(), "");
}
```

Problem 5:

```
Finished table:

0:  20, your -> 40, hash
1:
2:
3:
4:
5:
6: 6, map
7: 87, skills -> 27, are
8: 88, fantastic -> 28, and
9: 29, you
10:
11: 51, will
12:
13: 53, get -> 13, full
14:
15:
16:
17:
18: 78, credit
19:

Final number of elements: 12
Capacity: 20
Load factor: 12/20 = 0.6 (or 3/5)
```

Problem 6:

```
Vector<string> allPaths(Grid<bool> &g) {
    return allPaths(g, 0, 0, "");
}

// helper function:
Vector<string> allPaths(Grid<bool> &g, int r, int c,
                        string path) {
    Vector<string> paths;
    if (!g.inBounds(r,c) or g[r][c]) {
        // already visited, or out of bounds
        return paths;
    }

    if (r == g.numRows() - 1 and c == g.numCols() - 1) {
        // we've reached the end
        paths.add(path);
        return paths;
    }

    // general idea: mark current position as visited,
    // then visit all neighbors
    g[r][c] = true; // visited
    paths += allPaths(g, r-1, c, path + "n");
    paths += allPaths(g, r, c+1, path + "e");
    paths += allPaths(g, r+1, c, path + "s");
    paths += allPaths(g, r, c-1, path + "w");
    // unmark
    g[r][c] = false;
    return paths;
}
```