CS 106B Spring 2022 Final Exam Key

Problem 1: Unique Grid Distance Puzzle

```
bool isUniqueSolution(Vector<GridLocation> circles) {
    Set<double> distances;
    for (int i = 0; i < circles.size(); i++) {
        for (int j = i + 1; j < circles.size(); j++) {
            double distance = euclidianDistance(circles[i], circles[j]);
            if (distances.contains(distance)) {
                return false;
            }
            distances.add(distance);
        }
    }
    return true;
}

void findUniqueDistancesHelper(const int n, Vector<GridLocation> &circles,
Set<Vector<GridLocation>> &solutions,
                               int row, int col) {
    // n is the number of circles we need to place on a grid of n x n locations
    // base cases
    if (!isUniqueSolution(circles)) {
        // not a solution
        return;
    }
    if (circles.size() == n) {
        solutions.add(circles);
        return;
    }
    if (row == n) {
        return;
    }

    int newRow = row;
    int newCol = col + 1;
    if (newCol == n) {
        newRow++;
        newCol = 0;
    }
    // recursive cases
    // don't add a new circle
    findUniqueDistancesHelper(n, circles, solutions, newRow, newCol);

    // add a new circle (choose)
    circles.add({row, col});
    findUniqueDistancesHelper(n, circles, solutions, newRow, newCol);
    // unchoose
    circles.remove(circles.size() - 1);
}

Set<Vector<GridLocation>> findUniqueDistances(int n) {
    // n is the number of circles we need to place on a grid of n x n locations
    Vector<GridLocation> circles;
    Set<Vector<GridLocation>> solutions;
    findUniqueDistancesHelper(n, circles, solutions, 0, 0);
    return solutions;
}
```

Problem 2: Problem 2 – Classes and Dynamic Memory

Example header file:

```
#pragma once
#include "vector.h"

using namespace std;

struct Student {
  string name;
  int suid;
  Vector<string> preferences;
};

class Section {

public:
  Section(int capacity, string time);
  ~Section();

  bool tryAddStudent(Student s);

  Student getStudent(int index);

  void swap(int index1, int index2);

private:
  // Students given empty private section
  Student *_students;
  int _capacity;
  int _numEnrolled;
  string _time;

  bool checkStudent(Student s);
  int thisSectionPreference(int index);

};
```

Example class implementation:

```
#include "Section.h"

// #2. Write full class implementation

Section::Section(int capacity, string time) {
    _students = new Student[capacity];
    _capacity = capacity;
    _numEnrolled = 0;
    _time = time;
}

Section::~Section() {
    delete[] _students;
}

bool Section::checkStudent(Student s) {
    /* Condition 1: section is full */
    if (_numEnrolled == _capacity) return false;

    /*
   * Condition 2: this section's time is not in the
   * top 3 of student preferences.
   */
    for (int i = 0; i < s.preferences.size() && i < 3; i++) {
        if (s.preferences[i] == _time) {
            return true;
        }
    }

    return false;
}

int Section::thisSectionPreference(int index) {
    Vector<string> studentPref = _students[index].preferences;
    for (int i = 0; i < studentPref.size(); i++) {
        if (studentPref[i] == _time) {
            return i;
        }
    }
    return -1;
}
```

```
bool Section::tryAddStudent(Student s) {
    if (!checkStudent(s)) {
        return false;
    }

    _students[_numEnrolled] = s;

    /* Student can be added. Now let's insert in sorted order by preference.
*/
    for (int i = _numEnrolled; i > 0; i--) {

        /*
     * Check if student at index i's preference for this section
     * is less than student at index i - 1's preference for this
     * section.
     */
        if (thisSectionPreference(i) < thisSectionPreference(i - 1)) {
            swap(i, i - 1);
        }
    }

    _numEnrolled++;
    return true;

}

Student Section::getStudent(int index) {
    if (index < 0 || index >= _numEnrolled) {
        error("Out of bounds!");
    }

    return _students[index];
}

void Section::swap(int index1, int index2) {
    Student temp = _students[index1];
    _students[index1] = _students[index2];
    _students[index2] = temp;
}
```

Problem 3: Linked Lists

```
ListNode* addToEndOfList(ListNode* node, ListNode*& front,
ListNode*& end) {
    if (front == nullptr) {
        front = node;
        end = node;
    } else {
        end->next = node;
        end = node;
    }
    ListNode* next = node->next;
    // make sure that the new node is in fact the end of
our list node->next = nullptr;
    node->next = nullptr;
    return next;
}

ListNode* listDivisibleBy(ListNode*& front, int divisor) {
    ListNode* divisibleFront = nullptr;
    ListNode* divisibleEnd = nullptr;
    ListNode* indivisibleFront = nullptr;
    ListNode* indivisibleEnd = nullptr;
    ListNode* current = front;
    while (current != nullptr) {
        if (current->data % divisor == 0) {
            current = addToEndOfList(current,
divisibleFront, divisibleEnd);
        } else {
            current = addToEndOfList(current,
indivisibleFront, indivisibleEnd);
        }
    }
    front = indivisibleFront;
    return divisibleFront;
```

Problem 4: Trees

```
// 1. Difference of max and min, BST
int diffMaxMinBst(TreeNode *root) {
    // Solution should be O(log n)
    // assumes at least one node in tree
    int min, max;
    // find min
    TreeNode *temp = root;
    while (temp->left != nullptr) {
        temp = temp->left;
    }
    min = temp->data;

    // find max
    temp = root;
    while (temp->right != nullptr) {
        temp = temp->right;
    }
    max = temp->data;

    return max - min;
}

// 2. Difference of max and min, non-BST

void diffMaxMinNonBstHelper(TreeNode *root, int &min, int &max) {
    // base case
    if (root != nullptr) {
        // pre-order traversal
        if (root->data < min) {
            min = root->data;
        }
        if (root->data > max) {
            max = root->data;
        }
        diffMaxMinNonBstHelper(root->left, min, max);
        diffMaxMinNonBstHelper(root->right, min, max);
    }
}

int diffMaxMinNonBst(TreeNode *root) {
    // Solution should be O(n)
    // assumes at least one node in tree
    int min = root->data;
    int max = root->data;
    diffMaxMinNonBstHelper(root, min, max);
    return max - min;
}
```

```
// 3. Given a sorted vector, insert the values into a BST to create the most
balanced
//    tree possible

TreeNode *balancedBstFromSortedVec(Vector<int> sortedVec) {
    int vecSize = sortedVec.size();
    // base cases
    if (vecSize == 0) {
        return nullptr;
    }
    return new TreeNode(sortedVec[vecSize / 2],
        balancedBstFromSortedVec(sortedVec.subList(0, vecSize / 2)),
        balancedBstFromSortedVec(sortedVec.subList(vecSize / 2 + 1)));

}
```

Problem 5: Hashing

Question A:

A good hash function suitable for a hash table should:

A. Be deterministic
~~B. Be reversible~~
C. Create a good distribution
D. Be fast, e.g., O(1)
~~E. Be fast, e.g., O(log n)~~

B is not correct because given a hash, you cannot determine the original input
E is not correct because hash functions should not depend on n at all

Question B:

What does it mean for a hash table to use chaining?

A. Each hash bucket holds a linked list for elements that collide when hashed.
B. ~~The next hash is based on the previous valued added to the hash table, so collisions are avoided.~~
C. ~~All values in the table are linked together with hash nodes, avoiding collisions.~~
D. ~~The hash function for each bucket depends on the function for the previous bucket, limiting collisions.~~
E. ~~The hash values form a increasing or decreasing sequence based on the hash table direction, meaning that collisions can be differentiated from each other.~~

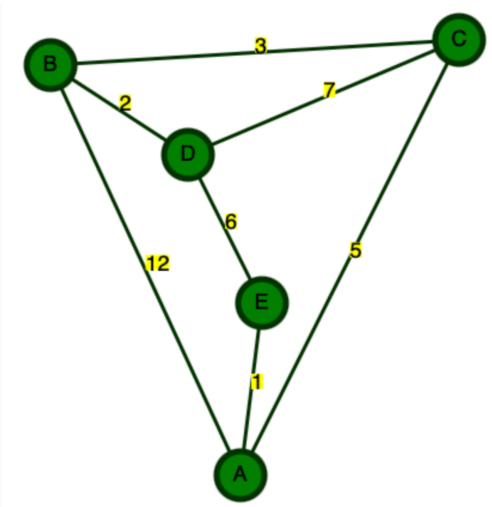
Question C:

Which of the following would be true of an open addressing hash table with linear probing?

A. Unlike a hash table that uses chaining, an open addressing table can fill up, and must be expanded when the table becomes full.
B. A disadvantage of the linear probing is that hashed values can end up in “clusters,” near each other in the table. I.e., many values will end up next to each other in the table.
C. ~~Open addressing guarantees O(1) time, unlike chaining, which can have O(n) behavior.~~
D. ~~Open addressing would keep the keys in sorted order, unlike chaining.~~

C is not correct because open addressing is also O(n) in the worst case (so is chaining)
D is not correct because hashing does not have anything to do with ordering the keys

Problem 6: Shortest Path Algorithms



Question A:

If a breadth first search (BFS) was run to determine the shortest path from node A to node B in the graph, what would the algorithm return?

A.A->B
B.A->E->D->B
C.A->C->B
D.A->C->D->B

(BFS does not take into account weights)

Question B: Dijkstra's Algorithm – multiple choice, choose the best answer.

If Dijkstra's Algorothm was run to determine the shortest path from node A to node B in the graph, what would the algorithm return?

A. A->B
B. A->E->D->B
C. A->C->B
D. A->C->D->B

(Dijkstra's always finds shortest path with weights)

Question C: A* Algorithm – multiple choice, choose the best answer.
If an A* Algorithm was run to determine the shortest path from node A to node B in the graph, what would the algorithm return?
A. A->B
B. A->E->D->B
C. A->C->B
D. A->C->D->B

(A* always finds shortest path with weights)

Note: Because there was confusion over the heuristic for A*, we gave everyone full credit (3/3) for Question C.