CSC-17C Final

Emiliano Panday Manalo IV

June 11, 2025

1 Problem

1. (15 points) Hashing We would like to use initials to locate an individual. For instance, MEL- 635 should locate the person Mark E. Lehr. Note: this is all upper case. Generate a hash function for the above using the numbers on your telephone. You know, each letter has a number associated with it, so examine your telephone keypad. Generate 512 random 3 letter initials and take statistics on a linked list array size 512 to hold this information. Report how many have no elements, 1 element, 2 elements, etc... Also, what is the maximum number of collisions. Do both results agree with the hashing statistics distribution? Show the comparison and analysis

PROGRAM:

```
#include <ctime>
#include <iostream>
#include <list>
#include <map>
#include <random>
#include <sstream>
#include <string>
#include <unordered_map>
using namespace std;
int teleHash512(string);
string genI();
int maxCol(int, int);
map<char, int> teleKey = {
    {'A', 2}, {'B', 2}, {'C', 2}, {'D', 3}, {'E', 3}, {'F', 3}, {'G', 4},
    {'H', 4}, {'I', 4}, {'J', 5}, {'K', 5}, {'L', 5}, {'M', 6}, {'N', 6},
    {'O', 6}, {'P', 7}, {'Q', 7}, {'R', 7}, {'S', 7}, {'T', 8}, {'U', 8},
    {'V', 8}, {'W', 9}, {'X', 9}, {'Y', 9}, {'Z', 9}};

int main() {
  unordered_map<int, int> bucketStat;
  list<string> hashTable[512];
  for (int i = 0; i < 511; i++) {
    string I = genI();
    hashTable[teleHash512(I)].push_back(I);
  }
  for (int i = 0; i < 511; i++) {
    bucketStat[hashTable[i].size()]++;
  }
  for (const auto &pair : bucketStat) {
    cout << "Bucket " << pair.first << " has " << pair.second
         << " elements...P = " << static_cast<float>(pair.second) / 512 << endl;
  }
  return 0;
}
```

```
int teleHash512(string toHash) {
  ostringstream comb;
  for (char c : toHash) {
    c = toupper(c);
    comb << teleKey[c];
  }
  return stoi(comb.str()) % 512;
}

string genI() {
  static std::mt19937 rng(static_cast<unsigned int>(std::time(nullptr)));
  static std::uniform_int_distribution<int> dist('A', 'Z');
  string I;
  for (int i = 0; i < 3; ++i) {
    I += static_cast<char>(dist(rng));
  }
  return I;
}
```

SAMPLE OUTPUT:

```
Bucket 6 has 3 elements...P = 0.00585938
Bucket 4 has 14 elements...P = 0.0273438
Bucket 5 has 9 elements...P = 0.0175781
Bucket 3 has 37 elements...P = 0.0722656
Bucket 2 has 81 elements...P = 0.158203
Bucket 1 has 119 elements...P = 0.232422
Bucket 0 has 248 elements...P = 0.484375
```

Let's calculate the max amount of collisions given we know how many available elements are in the hash table using: Let N = number of elements in hash table and let M = number of initials. Knowing:

$$k! \geq 2N$$

We can solve for k

$$k \geq (!^{-1})2N$$

Thus:

$$k \geq (!^{-1})2(512)$$
$$k \geq (!^{-1})1024$$
$$7! \geq (!^{-1})1024$$
$$k = 7$$

There can only be a max of 7 collisions, and since there is no bucket 8, this checks. However, we now need to check if the distributions match, using Poission approximation: Let $\lambda = \frac{M}{N}$ be the average number of items per bucket. For large M and N (with $M \approx N$), we approximate the collision distribution with the Poisson distribution:

$$P(X = j) = \frac{\lambda^j e^{-\lambda}}{j!}$$

$M = 512$, $N = 512$, $\lambda = 1$ Thus we have the theoretical distribution:

$$P(X = 0) = \frac{1}{e} \approx 0.3679$$
$$P(X = 1) = \frac{1}{e} \approx 0.3679$$
$$P(X = 2) = \frac{1}{2e} \approx 0.1839$$
$$P(X = 3) = \frac{1}{6e} \approx 0.0613$$
$$P(X = 4) = \frac{1}{24e} \approx 0.0153$$
$$P(X = 5) = \frac{1}{120e} \approx 0.00307$$
$$P(X = 6) = \frac{1}{720e} \approx 0.000511$$
$$P(X = 7) = \frac{1}{5040e} \approx 0.000073$$

So no, they do not match! I assume this is because three letters are getting the same value and thus can't really truly be hashed properly. If all letters had a unique value, I'm sure it would have had a huge impact on how the approximation and simulation would be compared.

2 Problem

2. (15 points) Stacks Take my hyperbolic sin/cos recursive function (at the end of the final) place the angle on a sine or cosine stack that represents a call to the sine or cosine. When the program returns, examine the stack for how many times the hyp sine was called and how many times hyp sine/cosine was called vs. the value you were trying to find. Put the results in a table. Range of values from -1 to 1 in .1 radian increments. Does the number of function calls agree with what you predict it should be?

PROGRAM

```
#include <cmath>
#include <cstdlib>
#include <ctime>
#include <iomanip>
#include <iostream>
#include <stack>
using namespace std;
float h(float);
float g(float);

stack<float> sineStack;
stack<float> cosStack;
int main(int argc, char **argv) {
  cout << "Hyperbolic Sine/Cosine Function Call Analysis" << endl;
  cout << "=============================================" << endl;
  cout << left << setw(10) << "Angle" << setw(15) << "sinh(x)" << setw(15)
       << "Our h(x)" << setw(12) << "Sine Calls" << setw(18) << "Cosine Calls"
       << setw(12) << "Difference" << endl;
  cout << string(82, '-') << endl;

  for (float i = -1.0; i <= 1.0; i += 0.1) {
    stack<float> clear;
    sineStack.swap(clear);
    cosStack.swap(clear);
```

```
    cout << fixed << setprecision(1);
    cout << left << setw(10) << i;
    cout << setprecision(6);
    cout << setw(15) << sinh(i) << setw(15) << h(i) << setw(12)
         << sineStack.size() << setw(18) << cosStack.size() << setw(12)
         << abs(sinh(i) - h(i)) << endl;
  }
  return 0;
}
float h(float angR) {
  float tol = 1e-6;
  sineStack.push(angR); // Track hyperbolic sine calls only
  if (angR > -tol && angR < tol)
    return angR + angR * angR * angR / 6;
  return 2 * h(angR / 2) * g(angR / 2);
}
float g(float angR) {
  cosStack.push(angR);
  float tol = 1e-6;
  if (angR > -tol && angR < tol)
    return 1 + angR * angR / 2;
  float b = h(angR / 2);
  return 1 + 2 * b * b;
}
```

SAMPLE OUTPUT

```
Hyperbolic Sine/Cosine Function Call Analysis
==============================================
```

Angle	sinh(x)	Our h(x)	Sine Calls	Cosine Calls	Difference
-1.0	-1.175201	-1.175201	28656	17710	0.000000
-0.9	-1.026517	-1.026517	28656	75022	0.000000
-0.8	-0.888106	-0.888106	28656	75022	0.000000
-0.7	-0.758584	-0.758583	28656	75022	0.000000
-0.6	-0.636653	-0.636653	28656	75022	0.000000
-0.5	-0.521095	-0.521095	17710	68257	0.000000
-0.4	-0.410752	-0.410752	17710	46365	0.000000
-0.3	-0.304520	-0.304520	17710	46365	0.000000
-0.2	-0.201336	-0.201336	10945	42184	0.000000
-0.1	-0.100167	-0.100167	6764	26070	0.000000
0.0	0.000000	0.000000	1	13528	0.000000
0.1	0.100167	0.100167	6764	4182	0.000000
0.2	0.201336	0.201336	10945	20292	0.000000
0.3	0.304520	0.304520	17710	32835	0.000000
0.4	0.410752	0.410752	17710	46365	0.000000
0.5	0.521095	0.521095	17710	46365	0.000000
0.6	0.636654	0.636654	28656	53130	0.000000
0.7	0.758584	0.758584	28656	75022	0.000000
0.8	0.888106	0.888106	28656	75022	0.000000
0.9	1.026517	1.026517	28656	75022	0.000000

I have no idea how to prove this mathematically, in which the recursive case is not based on an element but on a floating point number. Visually I can see this being something like 2^N due to one of the recursive functions calling two more functions. Which does look likely, as the behavior of the sine and cosine call shows this.

3 Problem

. (20 points) Queues Let us say you are in a line at the grocery store or bank like I was last weekend. One line, yet there are 3 clerks/tellers which service the same line. Simulate the following, Clerk 1 - Services customers on the average 1/min Clerk 2 - Services customers on the average 0.5/min Clerk 3 - Services customers on the average 0.75/min Customers - Arrive at 4/minute intervals. When the line gets to 5 customers add one more Clerk with the same service rate as Clerk 1. Add one more clerk similarly for each 5 customers. Take tellers away when they have serviced the line according to how they were added. For instance, if a 4th clerk was added to the line because there were 5 customers waiting then remove the clerk when the customer count in line goes to zero. What is my average customer wait time? What is the max number of customers in the line? If you randomize servicing and arrival times by +-50results?

PROGRAM:

```
#include <iomanip>
#include <iostream>
#include <queue>
#include <random>
#include <vector>
using namespace std;
//---RNG SETUP---
random_device rd;
mt19937 gen(rd());                          // Mersenne Twister RNG
uniform_int_distribution<> arr(30, 90);  // Arrival times
uniform_int_distribution<> tel(30, 180); // Teller one processes
//---GLOBAL VALUES---
class Customer {
public:
  int initialTime;
  Customer(int t) { initialTime = t; }
};
class Teller {
public:
  int secondsPerCustomer;
  bool isRandom = false;
  Teller(int s) { secondsPerCustomer = s; }
  Teller(int s, bool b) {
    secondsPerCustomer = s;
    isRandom = b;
  }
  void startServing(queue<Customer> &line, const int time, int &totalTime,
                    int &customersServed) {
    if (isRandom == true)
      secondsPerCustomer = tel(gen);
    if (!line.empty() && time % secondsPerCustomer == 0) {
      totalTime += abs(time - line.front().initialTime);
      line.pop();
      customersServed++;
    }
  }
};
void regSim() {
  queue<Customer> line;
  vector<Teller> tellers = {Teller(60), Teller(120), Teller(80)};
  int maxQueueSize = 0;
  int totalTime = 0;
  int customersServed = 0;
```

```
  for (int i = 0; i < 28800; i++) {
    if (i % 60 == 0) {
      line.push(Customer(i));
      line.push(Customer(i));
      line.push(Customer(i));
      line.push(Customer(i));
    }
    for (auto &teller : tellers)
      teller.startServing(line, i, totalTime, customersServed);

    if (line.size() > 4) {
      tellers.push_back(Teller(60));
    }
    if (line.empty() && tellers.size() > 3) {
      tellers.pop_back();
    }
    if (line.size() > maxQueueSize)
      maxQueueSize = line.size();
  }
  cout << fixed << setprecision(2);
  cout << "\n===== Simulation Results =====\n";
  cout << left << setw(25) << "Total Wait Time:" << totalTime << " seconds\n";
  if (customersServed > 0)
    cout << left << setw(25)
         << "Average Wait Time:" << (float)totalTime / customersServed
         << " seconds\n";
  cout << left << setw(25) << "Max Queue Size:" << maxQueueSize
       << " customers\n";
  cout << "==============================\n";
}
void ranSim() {
  queue<Customer> line;
  vector<Teller> tellers = {Teller(60, true), Teller(120, true),
                            Teller(80, true)};
  int maxQueueSize = 0;
  int totalTime = 0;
  int customersServed = 0;
  int initialArrival = 60;

  for (int i = 0; i < 28800; i++) {
    if (i % initialArrival == 0) {
      line.push(Customer(i));
      line.push(Customer(i));
      line.push(Customer(i));
      line.push(Customer(i));
      initialArrival = arr(gen);
    }
    for (auto &teller : tellers)
      teller.startServing(line, i, totalTime, customersServed);

    if (line.size() > 4) {
      tellers.push_back(Teller(60, true));
    }
    if (line.empty() && tellers.size() > 3) {
      tellers.pop_back();
```

```
    }
    if (line.size() > maxQueueSize)
      maxQueueSize = line.size();
  }
  cout << fixed << setprecision(2);
  cout << "\n===== Simulation Results =====\n";
  cout << left << setw(25) << "Total Wait Time:" << totalTime << " seconds\n";
  if (customersServed > 0)
    cout << left << setw(25)
         << "Average Wait Time:" << (float)totalTime / customersServed
         << " seconds\n";
  cout << left << setw(25) << "Max Queue Size:" << maxQueueSize
       << " customers\n";
  cout << "================================\n";
}
int main() {
  regSim();
  ranSim();
  return 0;
}
```

SAMPLE OUTPUT

```
===== Simulation Results =====
Total Wait Time:         64800 seconds
Average Wait Time:       33.75 seconds
Max Queue Size:          5 customers
================================

===== Simulation Results =====
Total Wait Time:         81963 seconds
Average Wait Time:       23.24 seconds
Max Queue Size:          22 customers
================================
```

Looking at these results it's not surprising that the Queue will possibly take longer for an ideal day with no benefits of faster or slow tellers or amount of customers. The Max Queue size is also larger due to the fact that it will be incredibly short to have four customers back to back.

4 Problem

4. (15 points) Sorting Create a list of 100000 random short integers. Use the merge and selection sorting routines we discussed to choose the top p elements in the list. Let us say for example p is 8. Then show by timing, recording operations and analysis for the Order of each algorithm like you did on the midterm except of course this is a different kind of selection sort. You get to stop after p elements have been sorted. Does the size of p change the analysis and when does 1 sort outperform the other.

PROGRAM:

```
#include <algorithm>
#include <chrono>
#include <iostream>
#include <list>
#include <random>
#include <vector>
using namespace std;
```

```
//---RNG VALUE---
random_device rd;
mt19937 gen(rd());
uniform_int_distribution<short> shortIntDist(0, 32767);

//---GLOBAL VALUES---
int mergeSortOpp = 0;
int selectionSortOpp = 0;

// Merge function for merge sort
void merge(list<short int> &lst, list<short int> &left,
           list<short int> &right) {
  lst.clear();
  mergeSortOpp++;

  auto leftIt = left.begin();
  auto rightIt = right.begin();

  while (leftIt != left.end() && rightIt != right.end()) {
    mergeSortOpp++;
    if (*leftIt <= *rightIt) {
      lst.push_back(*leftIt);
      ++leftIt;
    } else {
      lst.push_back(*rightIt);
      ++rightIt;
    }
  }

  // Append remaining elements
  while (leftIt != left.end()) {
    lst.push_back(*leftIt);
    ++leftIt;
    mergeSortOpp++;
  }

  while (rightIt != right.end()) {
    lst.push_back(*rightIt);
    ++rightIt;
    mergeSortOpp++;
  }
}

// Complete merge sort
void mergeSort(list<short int> &lst) {
  if (lst.size() <= 1)
    return;

  mergeSortOpp++;

  // Split the list into two halves
  list<short int> left;
  list<short int> right;

  auto slow = lst.begin();
  auto fast = lst.begin();
```

```
  // Use fast/slow iterator technique to find middle
  while (fast != lst.end()) {
    ++fast;
    if (fast != lst.end()) {
      ++fast;
      ++slow;
    }
  }

  // Move elements from lst to left and right
  left.splice(left.begin(), lst, lst.begin(), slow);
  right.splice(right.begin(), lst, lst.begin(), lst.end());

  // Recursively sort the halves
  mergeSort(left);
  mergeSort(right);

  // Merge sorted halves back into lst
  merge(lst, left, right);
}

// Partial selection sort - only sort first p elements
void partialSelectionSort(list<short int> &lst, int p) {
  if (p <= 0 || lst.empty())
    return;

  auto end_pos = lst.begin();
  advance(end_pos, min(p, (int)lst.size()));

  for (auto pos = lst.begin(); pos != end_pos; ++pos) {
    auto minIt = pos;
    for (auto it = next(pos); it != lst.end(); ++it) {
      selectionSortOpp++;
      if (*it < *minIt) {
        minIt = it;
      }
    }
    if (minIt != pos) {
      swap(*pos, *minIt);
      selectionSortOpp++;
    }
  }
}

// Get top p elements using merge sort
list<short int> getTopPMergeSort(list<short int> arr, int p) {
  mergeSortOpp = 0;
  mergeSort(arr);

  list<short int> result;
  auto it = arr.begin();
  for (int i = 0; i < p && it != arr.end(); ++i, ++it) {
    result.push_back(*it);
  }
  return result;
```

```
}

// Get top p elements using partial selection sort
list<short int> getTopPSelectionSort(list<short int> arr, int p) {
  selectionSortOpp = 0;
  partialSelectionSort(arr, p);

  list<short int> result;
  auto it = arr.begin();
  for (int i = 0; i < p && it != arr.end(); ++i, ++it) {
    result.push_back(*it);
  }
  return result;
}

int main() {
  const int ARRAY_SIZE = 100000;
  vector<int> p_values;
  for (int i = 1; i <= 8; i++)
    p_values.push_back(i);

  cout << "Sorting Algorithm Performance Analysis" << endl;
  cout << "Array Size: " << ARRAY_SIZE << " elements" << endl;
  cout << "=======================================" << endl;

  for (int p : p_values) {
    cout << "\nAnalyzing for p = " << p << " elements:" << endl;
    cout << "---------------------------------------" << endl;

    // Run multiple trials for better accuracy
    const int TRIALS = 5;
    long long totalMergeTime = 0, totalSelectionTime = 0;
    long long totalMergeOps = 0, totalSelectionOps = 0;

    for (int trial = 0; trial < TRIALS; trial++) {
      // Generate random data
      list<short int> arr;
      for (int i = 0; i < ARRAY_SIZE; i++) {
        arr.push_back(shortIntDist(gen));
      }

      // Test Merge Sort
      auto start = chrono::high_resolution_clock::now();
      list<short int> mergeResult = getTopPMergeSort(arr, p);
      auto end = chrono::high_resolution_clock::now();

      auto mergeTime =
          chrono::duration_cast<chrono::nanoseconds>(end - start).count();
      totalMergeTime += mergeTime;
      totalMergeOps += mergeSortOpp;

      // Test Selection Sort
      start = chrono::high_resolution_clock::now();
      list<short int> selectionResult = getTopPSelectionSort(arr, p);
      end = chrono::high_resolution_clock::now();
```

```
      auto selectionTime =
          chrono::duration_cast<chrono::nanoseconds>(end - start).count();
      totalSelectionTime += selectionTime;
      totalSelectionOps += selectionSortOpp;
    }

    // Calculate averages
    long long avgMergeTime = totalMergeTime / TRIALS;
    long long avgSelectionTime = totalSelectionTime / TRIALS;
    long long avgMergeOps = totalMergeOps / TRIALS;
    long long avgSelectionOps = totalSelectionOps / TRIALS;

    // Display results
    cout << "Merge Sort    - Time: " << avgMergeTime
         << " ns, Operations: " << avgMergeOps << endl;
    cout << "Selection Sort - Time: " << avgSelectionTime
         << " ns, Operations: " << avgSelectionOps << endl;

    // Analysis
    if (avgMergeTime < avgSelectionTime) {
      cout << "Winner: Merge Sort (faster by "
           << ((double)avgSelectionTime / avgMergeTime) << "x)" << endl;
    } else {
      cout << "Winner: Selection Sort (faster by "
           << ((double)avgMergeTime / avgSelectionTime) << "x)" << endl;
    }

    // Time complexity analysis
    cout << "Time Complexity Analysis:" << endl;
    cout << "  Merge Sort: O(n log n) = O(" << ARRAY_SIZE << " * log("
         << ARRAY_SIZE << "))  O("
         << (long long)(ARRAY_SIZE * log2(ARRAY_SIZE)) << ")" << endl;
    cout << "  Selection Sort (partial): O(p * n) = O(" << p << " * "
         << ARRAY_SIZE << ") = O(" << (long long)p * ARRAY_SIZE << ")" << endl;
  }

  cout << "\n========================================" << endl;
  cout << "ANALYSIS SUMMARY:" << endl;
  cout << "- Selection Sort is better for small p values" << endl;
  cout << "- Merge Sort becomes better as p increases" << endl;
  cout << "- Crossover point depends on implementation and data" << endl;
  cout << "- Selection Sort: O(p*n), Merge Sort: O(n log n)" << endl;

  return 0;
}
```

SAMPLE OUTPUT:

```
Sorting Algorithm Performance Analysis
Array Size: 100000 elements
========================================

Analyzing for p = 1 elements:
Merge Sort    - Time: 65207664 ns, Operations: 1868926
Selection Sort - Time: 3806535 ns, Operations: 100000
Winner: Selection Sort (faster by 17.1305x)
Time Complexity Analysis:
  Merge Sort: O(n log n) = O(100000 * log(100000))  O(1660964)
```

```
  Selection Sort (partial): O(p * n) = O(1 * 100000) = O(100000)

Analyzing for p = 2 elements:
----------------------------------------
Merge Sort    - Time: 97068945 ns, Operations: 1868926
Selection Sort - Time: 7376793 ns, Operations: 199999
Winner: Selection Sort (faster by 13.1587x)
Time Complexity Analysis:
  Merge Sort: O(n log n) = O(100000 * log(100000))  O(1660964)
  Selection Sort (partial): O(p * n) = O(2 * 100000) = O(200000)

Analyzing for p = 3 elements:
----------------------------------------
Merge Sort    - Time: 95865090 ns, Operations: 1868926
Selection Sort - Time: 9795122 ns, Operations: 299997
Winner: Selection Sort (faster by 9.78702x)
Time Complexity Analysis:
  Merge Sort: O(n log n) = O(100000 * log(100000))  O(1660964)
  Selection Sort (partial): O(p * n) = O(3 * 100000) = O(300000)

Analyzing for p = 4 elements:
----------------------------------------
Merge Sort    - Time: 109401168 ns, Operations: 1868926
Selection Sort - Time: 11107194 ns, Operations: 399994
Winner: Selection Sort (faster by 9.84958x)
Time Complexity Analysis:
  Merge Sort: O(n log n) = O(100000 * log(100000))  O(1660964)
  Selection Sort (partial): O(p * n) = O(4 * 100000) = O(400000)

Analyzing for p = 5 elements:
----------------------------------------
Merge Sort    - Time: 115320511 ns, Operations: 1868926
Selection Sort - Time: 13974683 ns, Operations: 499990
Winner: Selection Sort (faster by 8.2521x)
Time Complexity Analysis:
  Merge Sort: O(n log n) = O(100000 * log(100000))  O(1660964)
  Selection Sort (partial): O(p * n) = O(5 * 100000) = O(500000)

Analyzing for p = 6 elements:
----------------------------------------
Merge Sort    - Time: 90341341 ns, Operations: 1868926
Selection Sort - Time: 14344234 ns, Operations: 599985
Winner: Selection Sort (faster by 6.29809x)
Time Complexity Analysis:
  Merge Sort: O(n log n) = O(100000 * log(100000))  O(1660964)
  Selection Sort (partial): O(p * n) = O(6 * 100000) = O(600000)

Analyzing for p = 7 elements:
----------------------------------------
Merge Sort    - Time: 94903802 ns, Operations: 1868926
Selection Sort - Time: 16646525 ns, Operations: 699979
Winner: Selection Sort (faster by 5.70112x)
Time Complexity Analysis:
  Merge Sort: O(n log n) = O(100000 * log(100000))  O(1660964)
  Selection Sort (partial): O(p * n) = O(7 * 100000) = O(700000)
```

```
Analyzing for p = 8 elements:
----------------------------------------
Merge Sort    - Time: 90946180 ns, Operations: 1868926
Selection Sort - Time: 18240013 ns, Operations: 799972
Winner: Selection Sort (faster by 4.98608x)
Time Complexity Analysis:
  Merge Sort: O(n log n) = O(100000 * log(100000))  O(1660964)
  Selection Sort (partial): O(p * n) = O(8 * 100000) = O(800000)

========================================
ANALYSIS SUMMARY:
- Selection Sort is better for small p values
- Merge Sort becomes better as p increases
- Crossover point depends on implementation and data
- Selection Sort: O(p*n), Merge Sort: O(n log n)
```

5 Problem

5. (15 points) Binary Trees Take problem 1 and put each of the 3 letters in a sorted binary tree. Compare number of nodes to identify a match with the hash vs. the tree. Use the AVL technique to balance the tree. **PROGRAM**

```
#include "MyTree.h"
#include <ctime>
#include <iostream>
#include <list>
#include <map>
#include <random>
#include <sstream>
#include <string>
#include <unordered_map>
using namespace std;
int teleHash512(string);
string genI();
int maxCol(int, int);
int stringToIntASCII(const string &input) {
  string combined = "";
  for (char c : input) {
    combined += to_string(static_cast<int>(c));
  }
  return stoi(combined); // Be careful: this fails if the string is too long
}
map<char, int> teleKey = {
    {'A', 2}, {'B', 2}, {'C', 2}, {'D', 3}, {'E', 3}, {'F', 3}, {'G', 4},
    {'H', 4}, {'I', 4}, {'J', 5}, {'K', 5}, {'L', 5}, {'M', 6}, {'N', 6},
    {'O', 6}, {'P', 7}, {'Q', 7}, {'R', 7}, {'S', 7}, {'T', 8}, {'U', 8},
    {'V', 8}, {'W', 9}, {'X', 9}, {'Y', 9}, {'Z', 9}};

int main() {
  unordered_map<int, int> bucketStat;
  list<string> hashTable[512];
  MyTree tree;
  for (int i = 0; i < 511; i++) {
    string I = genI();
    tree.insert(teleHash512(I));
```

```
    hashTable[teleHash512(I)].push_back(I);
  }
  for (int i = 0; i < 511; i++) {
    bucketStat[hashTable[i].size()]++;
  }
  for (const auto &pair : bucketStat) {
    cout << "Bucket " << pair.first << " has " << pair.second
         << " elements...P = " << static_cast<float>(pair.second) / 512 << endl;
  }
  cout << "Amount Of Nodes: " << tree.countNodes() << endl;
  cout << "Depth: " << tree.getHeight() << endl;
  tree.print();
  return 0;
}

int teleHash512(string toHash) {
  ostringstream comb;
  for (char c : toHash) {
    c = toupper(c);
    comb << teleKey[c];
  }
  return stoi(comb.str()) % 512;
}

string genI() {
  static std::mt19937 rng(static_cast<unsigned int>(std::time(nullptr)));
  static std::uniform_int_distribution<int> dist('A', 'Z');
  string I;
  for (int i = 0; i < 3; ++i) {
    I += static_cast<char>(dist(rng));
  }
  return I;
}
```

SAMPLE OUTPUT:

```
Bucket 5 has 8 elements...P = 0.015625
Bucket 6 has 2 elements...P = 0.00390625
Bucket 4 has 18 elements...P = 0.0351562
Bucket 3 has 41 elements...P = 0.0800781
Bucket 1 has 124 elements...P = 0.242188
Bucket 2 has 70 elements...P = 0.136719
Bucket 0 has 248 elements...P = 0.484375
Amount Of Nodes: 511
Depth: 11
```

Comparing the operational depth of each data structure. The hash with chaining is faster. This is because the max amount of operations with the given 512 inputs is 6, compared to the tree's depth of 11. However, the AVL balanced binary tree is always consistent with worst and best case scenarios. A hash with chaning into infinity has the possible issue of becoming a linked list of O(N) compared to a tree of Log(N) due to its divide and conquer nature. Thus, for the sake of this problem that the hash with chaining is faster, but its scalability poses further issues as N elements are added into infinity.

6 Problem

```
6. (20 points) Weighted Graph
In the vertex and edge structure defined below
```

```
Vertex Edge Vertex
SFO 2703 BOS
SFO 1847 ORD
ORD 868 BOS
ORD 743 JFK
JFK 189 BOS
SFO 1465 DFW
DFW 803 ORD
DFW 1124 MIA
MIA 1093 JFK
MIA 1257 BOS
SFO 338 LAX
LAX 1234 DFW
LAX 2341 MIA
Calculate by hand as well as developing a program as a check.
a) Find the shortest distance between ORD and LAX.
b) Find the shortest distance between JFK and SFO.
c) Find the minimum spanning tree.
```

PROGRAM:

```
#include <algorithm>
#include <climits>
#include <iostream>
#include <queue>
#include <set>
#include <stack>
#include <unordered_map>
#include <vector>
using namespace std;

struct Edge {
  string from, to;
  int weight;
};

unordered_map<string, vector<pair<string, int>>>
makeGraph(const vector<Edge> &edges) {
  unordered_map<string, vector<pair<string, int>>> graph;
  for (auto &e : edges) {
    graph[e.from].push_back({e.to, e.weight});
    graph[e.to].push_back({e.from, e.weight}); // undirected
  }
  return graph;
}

pair<int, vector<string>>
dijkstra(unordered_map<string, vector<pair<string, int>>> &graph, string start,
         string end) {
  priority_queue<pair<int, string>, vector<pair<int, string>>, greater<>> pq;
  unordered_map<string, int> dist;
  unordered_map<string, string> prev;

  for (auto &node : graph)
    dist[node.first] = INT_MAX;
  pq.push({0, start});
```

```
  dist[start] = 0;

  while (!pq.empty()) {
    auto [d, u] = pq.top();
    pq.pop();
    if (u == end)
      break;
    for (auto &[v, w] : graph[u]) {
      if (dist[u] + w < dist[v]) {
        dist[v] = dist[u] + w;
        prev[v] = u;
        pq.push({dist[v], v});
      }
    }
  }

  vector<string> path;
  for (string at = end; at != ""; at = prev.count(at) ? prev[at] : "") {
    path.push_back(at);
    if (at == start)
      break;
  }
  reverse(path.begin(), path.end());
  return {dist[end], path};
}

// Union-Find for Kruskal's
struct DSU {
  unordered_map<string, string> parent;
  string find(string x) {
    if (parent[x] != x)
      parent[x] = find(parent[x]);
    return parent[x];
  }
  void unite(string x, string y) { parent[find(x)] = find(y); }
  void add(string x) {
    if (!parent.count(x))
      parent[x] = x;
  }
};

int kruskal(const vector<Edge> &edges, vector<Edge> &mstEdges) {
  DSU dsu;
  for (const auto &e : edges) {
    dsu.add(e.from);
    dsu.add(e.to);
  }
  vector<Edge> sorted = edges;
  sort(sorted.begin(), sorted.end(),
       [](auto &a, auto &b) { return a.weight < b.weight; });
  int total = 0;
  for (auto &e : sorted) {
    if (dsu.find(e.from) != dsu.find(e.to)) {
      dsu.unite(e.from, e.to);
      total += e.weight;
      mstEdges.push_back(e);
```

```
    }
  }
  return total;
}

void printPath(const string &from, const string &to,
               const pair<int, vector<string>> &result) {
  cout << "Shortest path from " << from << " to " << to << " (" << result.first
       << " miles): ";
  for (size_t i = 0; i < result.second.size(); ++i) {
    cout << result.second[i];
    if (i + 1 < result.second.size())
      cout << " → ";
  }
  cout << "\n";
}

int main() {
  vector<Edge> edges = {
      {"SFO", "BOS", 2703}, {"SFO", "ORD", 1847}, {"ORD", "BOS", 868},
      {"ORD", "JFK", 743},  {"JFK", "BOS", 189},  {"SFO", "DFW", 1465},
      {"DFW", "ORD", 803},  {"DFW", "MIA", 1124}, {"MIA", "JFK", 1093},
      {"MIA", "BOS", 1257}, {"SFO", "LAX", 338},  {"LAX", "DFW", 1234},
      {"LAX", "MIA", 2341}};

  auto graph = makeGraph(edges);

  // (a) Shortest ORD → LAX
  auto pathA = dijkstra(graph, "ORD", "LAX");
  printPath("ORD", "LAX", pathA);

  // (b) Shortest JFK → SFO
  auto pathB = dijkstra(graph, "JFK", "SFO");
  printPath("JFK", "SFO", pathB);

  // (c) Minimum Spanning Tree
  vector<Edge> mstEdges;
  int mstCost = kruskal(edges, mstEdges);
  cout << "\nMinimum Spanning Tree edges:\n";
  for (auto &e : mstEdges) {
    cout << e.from << " | " << e.to << " (" << e.weight << ")\n";
  }
  cout << "Total MST cost: " << mstCost << "\n";

  return 0;
}
```

SAMPLE OUTPUT:

```
Shortest path from ORD to LAX (2037 miles): ORD → DFW → LAX
Shortest path from JFK to SFO (2590 miles): JFK → ORD → SFO

Minimum Spanning Tree edges:
JFK | BOS (189)
SFO | LAX (338)
ORD | JFK (743)
DFW | ORD (803)
MIA | JFK (1093)
```

```
LAX | DFW (1234)
Total MST cost: 4400
```