NAME: **Tyler Hu**

UH ID: **0276538**

COSC3320 – ASSIGNMENT 2

Theory 1.

(a) Derive a lower bound:

Let *A* = linked list #1 and *B* = linked list #2. Searching if an element in *A* is an element in *B* requires an iteration though every single element in *A* (*n* times with a worst complexity of O(*n*)) and then a check of each element in *B* (*n* times with a worst complexity of O(*n*)). Therefore, the lower bound time complexity would be an order of O(2*n*) -> O(*n*).

(b) Design an algorithm in pseudocode:

Step 1: Copy elements from *A* to a new linked list *C* with a complexity of O*(n).*

```
struct node *copy(struct node *cuA)
   if cuA == NULL
      return
   struct node *cuC = new node
   cuC->info = cuA->info
   cuC->link = copy(cuA->link)
   return cuC
```

Step 2: Modify *B* by incrementing all elements in *B* by 1 with a complexity of O*(n).*

```
while currentOfB != NULL
   currentOfB->info += 1
   currentOfB = currentOfB->link
```

Step 3: Check if *C* has been modified by comparing it to *A*. If there is a difference between *A* and *C*, then there is at least one element in *A* that is an element of *B*. Time complexity is O*(n).*

```
bool checkIfModified(struct node *cuA, struct node *cuC)
   while cuA != NULL && cuC != NULL
      if cuA->info != cuC->info
         return false
      cuA = cuA->link
      cuC = cuC->link
   return true
```

Step 4: Return B back to its original state by decrementing each element by 1 with a complexity of O*(n).*

```
while currentOfB != NULL
   currentOfB->info -= 1
   currentOfB = currentOfB->link
```

Therefore the total time complexity is four operations of O(*n*) time complexity, which is 4*(O(*n*)). This simplifies to a lower bound time complexity of O(*n*).

Theory 2.

We first assume that all values in an AVL tree are integers. As the root of the AVL tree will need to be the median element, the insert and delete functions will need to be modified according to the value of [n/3].

The Insert(x) and Delete(x) algorithms are implemented similarly to the one discussed in class but will be modified by adding in the following pseudocode at the end of each algorithm:

```
int k = n/3

if k < value of element
   recursively shift left child node by four-step process:
      1) shift node to right subtree
      2) shift the left child to the root
      3) rebalance tree except the root
      4) adjust node count counter

if k > value of element
   recursively shift right child node by four-step process:
      1) shift node to left subtree
      2) shift the right child to the root
      3) rebalance tree except the root
      4) adjust node count counter
```

These modifications will change the Find function to a time complexity of O(1) because the median is always at the root, and the function will only be able to directly access the value at the root. Thus, the Find function will have an improvement of its time complexity from $O(\log_2 n)$ to O(1). However, the insertion and deletion algorithms will still have a slower time complexity due to the recursion. The space complexities will remain unchanged with Insert and Delete at $O(\log_2 n)$ and Find at O(1).

Theory 3.

Assume the *n* x *n* matrix from the textbook is implemented. According to the textbook, the average work of multiplying *n* matrices would be equal to the total sum of the scalar multiplications of the k-values from the textbook divided by the total number of k-values used in the operation.

The k-value can therefore be calculated as:

$S[i,j] = S[i,k] + S[k+1,j] + n_i + n_{k+1} + n_{j+1}$

The algorithm in pseudocode is as follows:

```
int k = 0
for i=1 to n-1 (n = number of columns)
   for j=i to n
      for k=i to j
         S[i,j] = S[i,k] + S[k+1,j] + ni + nk+1 + nj+1
         k++
average = sum of (S[i,j] / k)
```

Since there are three nested loops, the time complexity would be $O(n^3)$. The space complexity is $O(n^2)$ due to the matrix construct.

Programming 1.

Background:

The workstation executing this program contains an AMD Ryzen 7 2700x CPU, 16GB of DDR4 3200 SDRAM, and Microsoft Windows 10 Pro 64-bit operating system (supports VMM). The program build was executed via the Microsoft Visual Studio IDE.

Hypothesis:

Assume the matrix additions are of integers. Since the matrix addition of Version 1 is a row loop with a nested column loop and Version 2 is a column loop with a nested row loop, there should be minimal timing differences of the matrix addition from Version 1 and Version 2 for the larger matrices. Furthermore, the doubling of the the matrix size should result in a quadrupling of time spent on the matrix addition.

Methodology:

1. The program was written in C++ using the Microsoft Visual Studio IDE.

2. Though the program worked initially for small matrices with $n \leq 4096$, the program would crash. To solve this, the program build had to be changed to x64, as x86 limited the program usage to 2GB of RAM.

3. Even with the change to x64, the workstation still did have enough RAM to calculate the larger matrix additions. Thus I had to increase the swap file size in Windows to 55GB in order to try and simulate the program in core.

Conclusion:

Below is a table containing the average run-times from ten program executions:

Matrix size	Version 1 time (ms)	Version 2 time (ms)
128	0	0
256	0	1
512	2	3
1024	7	15
2048	19	64
4096	53	292
8192	194	1215
16384	893	11275
32768	3853	73276
65536	39160	341645

Based on the results, my hypothesis was clearly wrong. Starting from a matrix size of n = 1024, Version 1 ran much faster on average when compared with Version 2. I surmise that this is due to other factors, as both algorithms execute the same number of operations. One factor may be the lower probability of encountering page faults with Version 1, which means that the OS spends less time in paging memory to disk for Version 1. Another factor is the fact that C++ I/Os memory in row-major order, which is why Version 1 runs faster than Version 2. For example, if the program was written in Java, I believe that the run-times would be more similar as the JVM would I/O the matrices as 1D arrays containing references to other arrays instead of 2D arrays like in C++. Java's method is similar to row-major order but not exactly the same, as Java would not store the rows contiguously.

In conclusion, the doubling of the matrix size does generally result in a quadrupling of the matrix addition processing time, on paper. Realistically however, other limitations such as the OS VMM efficiency, absence of enough physical memory, disk transfer speeds and differing compiler and language usage all influence the real run-times significantly.

Programming #1 in C++

```
#include <iostream>
#include <iomanip>
#include <time.h>
using namespace std;

int main()
{
  //list of matrix sizes
  int size[10] = {128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536};
  //time vars
  double startTime1, stopTime1, startTime2, stopTime2, calcTime1, calcTime2;

  for (int num = 0; num < 10; num++) {
    //matrix initialization
    int** A = new int*[size[num]];
    int** B = new int*[size[num]];
    int** C = new int*[size[num]];

    for (int i = 0; i < size[num]; i++) {
      A[i] = new int[size[num]];
      B[i] = new int[size[num]];
      C[i] = new int[size[num]];
    }

    //fills the matrices with 0
    for (int i = 0; i < size[num]; i++) {
      for (int j = 0; j < size[num]; j++) {
        A[i][j] = 0;
        B[i][j] = 0;
        C[i][j] = 0;
      }
    }

    //matrix addition for version 1
    startTime1 = clock();
    for (int i = 0; i < size[num]; i++) {
      for (int j = 0; j < size[num]; j++) {
        C[i][j] = A[i][j] + B[i][j];
      }
    }
    stopTime1 = clock();

    //matrix addition for version 2
    startTime2 = clock();
    for (int j = 0; j < size[num]; j++) {
      for (int i = 0; i < size[num]; i++) {
        C[i][j] = A[i][j] + B[i][j];
      }
    }
    stopTime2 = clock();

    calcTime1 = (stopTime1 - startTime1) / double(CLOCKS_PER_SEC);
    calcTime2 = (stopTime2 - startTime2) / double(CLOCKS_PER_SEC);
```

```
    //output
    cout << "\nLet n = "<< size[num] << endl;
    cout << "Version 1 time: " << fixed << setprecision(0) << calcTime1*1000 <<
            "ms" << endl;

    cout << "Version 2 time: " << fixed << setprecision(0) << calcTime2*1000 <<
            "ms" << endl;

    cout << "----------------------------------------" << endl;
  }

  return 0;
}
```

Programming 2.

Background:

The workstation executing this program contains an AMD Ryzen 7 2700x CPU, 16GB of DDR4 3200 SDRAM, and Microsoft Windows 10 Pro 64-bit operating system (supports VMM). The program build was executed via the Microsoft Visual Studio IDE.

Hypothesis:

Assume the matrix calculations are of integers with a *m* number of operations. The time complexities of this algorithm execution should all be O(*m*), which would be O(1,677,721,600) and O(13,421,772,800) due to each instance completing *m* number of for loops.

Methodology:

1. The program was written in C++ using the Microsoft Visual Studio IDE.

2. Though the program worked initially for small matrices with $n \leq 4096$, the program would crash. To solve this, the program build had to be changed to x64, as x86 limited the program usage to 2GB of RAM.

3. Even with the change to x64, the workstation still did have enough RAM to calculate the larger matrix additions. Thus I had to increase the swap file size in Windows to 55GB in order to try and simulate the program in core.

Conclusion:

Below is a table containing the run-times from one program execution:

Matrix size	m = 1,677,721,600	m = 13,421,772,800
16	102.45 seconds	131.24 seconds
64	103.95 seconds	129.48 seconds
256	104.52 seconds	131.75 seconds
1024	131.68 seconds	146.89 seconds
4096	156.50 seconds	164.32 seconds
16384	199.38 seconds	244.41 seconds

Based on the results, my hypothesis should be right on paper but is wrong in a practical setting. The results show that for smaller matrix sizes, the execution time does not vary significantly due to the size of the matrix (*n*) or the number of calculations (*m*). However, when the matrix size increases to 1024 and above, the execution time increases significantly for each value of *n* and *m*. I believe that this occurs because as the size of the matrix increases beyond the physical size of the memory, the OS uses VMM to start paging memory to disk, which adds overhead. Thus, matrices of size 1024 and higher started to take longer to calculate.

In conclusion, the computational complexity should be O(*m*), but in reality, limitations in OS VMM efficiency, the amount of physical memory, disk transfer speed, and compiler settings all influence the real run-times significantly.

Programming #2 in C++

```
#include <iostream>
#include <time.h>
using namespace std;

int main()
{
  //seed rng
  srand(time(NULL));
  //list of matrix sizes
  int size[6] = {16, 64, 256, 1024, 4096, 16384};

  for (int num = 0; num < 6; num++) {
    //initializes matrix
    int** matrix = new int*[size[num]];

    for (int i = 0; i < size[num]; i++) {
      matrix[i] = new int[size[num]];
    }

    //fills the matrix with 0
    for (int i = 0; i < size[num]; i++) {
      for (int j = 0; j < size[num]; j++) {
        matrix[i][j] = 0;
      }
    }

    long long int m = 1677721600;
    //long long int m = 13421772800;

    time_t time = clock();
    //adds random numbers from 1-100
    for (int i = 0; i < m; i++) {
      int a = rand() % size[num];
      int b = rand() % size[num];
      int x = rand() % 100 + 1;
      matrix[a][b] = matrix[a][b] + x;
    }
    time = clock() - time;

    printf("Time for matrix size of %d: %.2f seconds", size[num],
          ((float)time) / CLOCKS_PER_SEC);
    cout << endl;
  }

  return 0;
}
```

Programming 3.

Background:

The workstation executing this program contains an AMD Ryzen 7 2700x CPU, 16GB of DDR4 3200 SDRAM, and Microsoft Windows 10 Pro 64-bit operating system (supports VMM). The program build was executed via the Microsoft Visual Studio IDE.

Hypothesis:

Assume the data in the AVL tree nodes consists of an integer and a matrix with random integers. When repeatedly inserting and deleting said from an AVL tree, the OS should be able to handle garbage collection and memory compaction at the cost of extra computational time.

Methodology:

1. The program was written in C++ using the Microsoft Visual Studio IDE.

2. The program first inserts the data (int & matrix) into an AVL tree with 50 nodes.

3. Then I chose a large number of repeated insertions and deletions to be executed (2,000,000), with the program outputting the average the amount of time needed for the operations to be completed.

Conclusion:

Below are the averaged results from the run-times of ten program executions:

```
Average Initial Insertion time: 8.04 seconds.
Average Insertion time: 2.0343 seconds.
Average Deletion time: 1.8121 seconds.
```

Based on the results, the average initial insertion time is significantly longer than subsequent insertions and deletions. This is surprising, as one would think that the repeated AVL tree rotations after the initial tree build would cause the OS to engage garbage collection and memory compaction. I believe this can be explained by the program sometimes selecting duplicate insertion values, which end up being skipped, thus saving time.

Additionally, insertion is also slower than deletion because deallocating memory is much faster than allocating memory, as insertions will need to require extra time for tree node rotations, garbage collection and memory compaction. Thus, my hypothesis is correct.

In conclusion, Windows has quite an efficient system for garbage collection based on this experiment. Future research can be done in trying to gauge the OS limit before the garbage collector is overloaded.

Programming #3 in C++

source: geeks4geeks - https://www.geeksforgeeks.org/avl-tree-set-2-deletion/

```
#include <iostream>
#include <iomanip>
#include <time.h>
#include <math.h>

using namespace std;

class BST {
  private:
    int size_0 = pow(2, 20);
    int size_1 = pow(2, 19) + pow(2, 20);
    int size_2 = pow(2, 18) + pow(2, 17);
    int node_count=0;
    int insert_count = 0;
    int delete_count = 0;

    struct node
    {
      int data;
      int* matrix;
      node* left;
      node* right;
      int height;
    };

    node* root;

    node* insert(int x, node* t)
    {
      int remainder = x % 3;
      if (t == NULL)
      {
        insert_count++;
        t = new node;
        t->data = x;
        node_count++;
        t->height = 0;
        t->left = t->right = NULL;

        if (remainder == 0)
        {
          t->matrix = new int[size_0];
        }
        else if (remainder == 1)
        {
          t->matrix = new int[size_1];
        }
        else
        {
          t->matrix = new int[size_2];
        }
      }
```

```
  else if (x < t->data)
  {
    t->left = insert(x, t->left);
    if (height(t->left) - height(t->right) == 2)
    {
      if (x < t->left->data)
        t = singleRightRotate(t);
      else
        t = doubleRightRotate(t);
    }
  }
  else if (x > t->data)
  {
    t->right = insert(x, t->right);
    if (height(t->right) - height(t->left) == 2)
    {
      if (x > t->right->data)
        t = singleLeftRotate(t);
      else
        t = doubleLeftRotate(t);
    }
  }

  t->height = max(height(t->left), height(t->right)) + 1;
  return t;
}

node* singleRightRotate(node* &t)
{
  if (t== NULL || t->left == NULL)
  {
    return t;
  }
  else
  {
    node* u = t->left;
    t->left = u->right;
    u->right = t;
    t->height = max(height(t->left), height(t->right)) + 1;
    u->height = max(height(u->left), t->height) + 1;
    return u;
  }
  return t;
}

node* singleLeftRotate(node* &t)
{
  if (t == NULL || t->right == NULL)
  {
    return t;
  }
  else
  {
    node* u = t->right;
    t->right = u->left;
    u->left = t;
    t->height = max(height(t->left), height(t->right)) + 1;
```

```
    u->height = max(height(t->right), t->height) + 1;
    return u;
  }
  return t;
}

node* doubleLeftRotate(node* &t) {
  t->right = singleRightRotate(t->right);
  return singleLeftRotate(t);
}

node* doubleRightRotate(node* &t)
{
  t->left = singleLeftRotate(t->left);
  return singleRightRotate(t);
}

node* findMin(node* t)
{
  if (t == NULL)
    return NULL;
  else if (t->left == NULL)
    return t;
  else
    return findMin(t->left);
}

node* findMax(node* t)
{
  if (t == NULL)
    return NULL;
  else if (t->right == NULL)
    return t;
  else
    return findMax(t->right);
}

node* remove(int x, node* t)
{
  node* temp;

  // Element not found
  if (t == NULL)
    return NULL;
  // Searching for element
  else if (x < t->data)
    t->left = remove(x, t->left);
  else if (x > t->data)
    t->right = remove(x, t->right);

  // Element found with 2 children
  else if (t->left && t->right)
  {
    temp = findMin(t->right);
    t->data = temp->data;
    t->right = remove(t->data, t->right);
  }
```

```
  // With one or zero child
  else
  {
    temp = t;
    if (t->left == NULL)
      t = t->right;
    else if (t->right == NULL)
      t = t->left;

    delete[] temp->matrix;
    delete temp;
    delete_count++;
    node_count--;
  }
  if (t == NULL)
    return t;

  t->height = max(height(t->left), height(t->right)) + 1;

  // If node is unbalanced
  // If left node is deleted, right case
  if (height(t->left) - height(t->right) == 2)
  {
    // right right case
    if (height(t->left->left) - height(t->left->right) == 1)
      return singleLeftRotate(t);
    // right left case
    else
      return doubleLeftRotate(t);
  }
  // If right node is deleted, left case
  else if (height(t->right) - height(t->left) == 2)
  {
    // left left case
    if (height(t->right->right) - height(t->right->left) == 1)
      return singleRightRotate(t);
    // left right case
    else
      return doubleRightRotate(t);
  }
  return t;
}

int height(node* t)
{
  return (t == NULL ? -1 : t->height);
}

int getBalance(node* t)
{
  if (t == NULL)
    return 0;
  else
    return height(t->left) - height(t->right);
}

void inorder(node* t)
```

```
    {
      if (t == NULL)
        return;
      inorder(t->left);
      cout << t->data << " ";
      inorder(t->right);
    }

  public:
    BST()
    {
      root = NULL;
    }

    void insert(int x)
    {
      root = insert(x, root);
    }

    void remove(int x)
    {
      root = remove(x, root);
    }

    void display()
    {
      inorder(root);
      cout << endl;
    }

    int getcount()
    {
      return node_count;
    }

    int insertcount()
    {
      return insert_count;
    }

    int deletecount()
    {
      return delete_count;
    }

    void setcount()
    {
      insert_count = 0;
    }
  };

int main()
{
  srand((unsigned int)time(0));
  double initial_insertion = 0;
  double insertion = 0;
  double deletion = 0;
```

```
  double start, stop, time;
  int *num_array = new int[2000000];
  int counter = 0;

  for (int i = 0; i < 2000000; i++)
  {
    num_array[i] = rand() % 299 + 1;
  }

  BST node;
  int j = 0;
  start = clock();

  while (node.getcount() < 50) {
    j++;
    node.insert(num_array[j]);
  }

  stop = clock();
  time = (stop - start);
  initial_insertion += time;

  node.setcount(); //resets insert count to 0
  for (int i = 0; i < 1000000; i++)
  {
    while (node.getcount() < 50)
    {
      j++;
      start = clock();
      node.insert(num_array[j]);
      stop = clock();
    }

    time = (stop - start);
    insertion += time;

    while (node.getcount()>= 50)
    {
      start = clock();
      node.remove(num_array[j]);
      stop = clock();
    }

    time = (stop - start);
    deletion += time;

  }
  cout << "Average Initial Insertion time: " << setprecision(5) <<
          initial_insertion / 50.00 << " seconds." << endl;
  cout << "Average Insertion time: " << setprecision(5) <<
          insertion / node.insertcount() << " seconds." << endl;
  cout << "Average Deletion time: " << setprecision(5) <<
          deletion / node.deletecount() << " seconds." << endl;

  return 0;
}
```

Programming 4.

Background:

The workstation executing this program contains an AMD Ryzen 7 2700x CPU, 16GB of DDR4 3200 SDRAM, and Microsoft Windows 10 Pro 64-bit operating system (supports VMM). The program build was executed via the Microsoft Visual Studio IDE.

Hypothesis:

The Windows VMM should be able to handle paging with linear time complexity at the very worst, or else excessive thrashing would render the OS unusable.

Methodology:

1. The program was written in C++ using the Microsoft Visual Studio IDE.

2. The program utilizes the Windows library API to get VMM status readings in real-time.

3. The program fills a very large dynamic array with arithmetic operations to simulate calculations.

Conclusion:

Below is a table containing the available memory in bytes from one program execution:

Cache size	Avail. Physical Memory	Available Page File	Avail. Virtual Memory	Run-time (seconds)
0.5M	9294967295	9292525142	9301945391	0.045 s
0.6M	9312426969	9300412423	9300614005	0.052 s
0.7M	9310341782	9300154039	9294826274	0.054 s
0.8M	9305783520	9282934754	9293916453	0.066 s
0.9M	9293725095	9299093311	9303119173	0.069 s
0.95M	9304639413	9291836324	9309416061	0.075 s
0.99M	9275704758	9287501780	9301605745	0.077 s
1.0M	9202832165	9299143762	9288724883	0.087 s
1.01M	9199540332	9173025153	9192990417	0.087 s
1.1M	9192896501	9174691509	9188099160	0.088 s
1.5M	9196274251	9192525142	9101945391	0.094 s
2.0M	9112426969	9100412423	9100614005	0.117 s
5.0M	9010341782	9200154039	9094826274	0.377 s
10.0M	8705783520	9282934754	8703916453	0.797 s
50.0M	6293725095	9299093311	6203119173	3.877 s

Based on the results, my hypothesis was correct in that there is a linear increase for the run-times up until 5.0M where starts showing some irregular increases upwards. Thus I believe that possibly some small thrashing occurs between 2.0M and 50.0M. The thrashing is negligible enough to prove that the Windows VMM is efficient at paging and replacement.

Programming #4 in C++

```
#include<iostream>
#include<tchar.h>
#include<Windows.h>
#include<limits.h>
#include<time.h>

using namespace std;

int main()
{
  long long int numBytesTotal;
  long long int numFreeBytes;
  long long int numFreePage;
  long long int numFreeVirtual;

  //representation of different caches
  double cacheSize[15] = {0.5, 0.6, 0.7, 0.8, 0.9, 0.95, 0.99, 1.0, 1.01, 1.1, 1.5,
                          2, 5, 10, 50};

  //initial view of VMM
  MEMORYSTATUS memInfo;
  GlobalMemoryStatus(&memInfo);

  numFreeBytes = memInfo.dwAvailPhys;
  numFreePage = memInfo.dwAvailPageFile;
  numFreeVirtual = memInfo.dwAvailVirtual;

  cout << "Available Physical Memory: " << memInfo.dwAvailPhys << endl;
  cout << "Available Page File: "       << memInfo.dwAvailPageFile << endl;
  cout << "Available Virtual Memory: "  << memInfo.dwAvailVirtual << endl;

  //outputs VMM status after calculations
  for (int i=0; i < 15; i++) {
    cout<< "======================================" << endl;
    cout<< "Cache Size: " << cacheSize[i] << "*M" << endl;

    clock_t startTime;
    startTime = clock();

    //gets cache size in bytes
    numBytesTotal = abs((int)(cacheSize[i] * (numFreeBytes)));
    int total = numBytesTotal / sizeof(int);

    //initialize array for synthetic calculations
    int *numArray = new int[total];

    GlobalMemoryStatus(&memInfo);
```

```
    cout << "Available Physical Memory: " << memInfo.dwAvailPhys << endl;
    cout << "Available Page File: "       << memInfo.dwAvailPageFile << endl;
    cout << "Available Virtual Memory: "  << memInfo.dwAvailVirtual << endl;

    //synthetic calculations
    for (int i=0; i < total; i++) {
      numArray[i] = i;
    }
    for (int i=0; i < total; i++) {
      numArray[i] += i;
    }

    //output
    cout << "Time elapsed: " <<
            ((double)(clock() - startTime) / (double)CLOCKS_PER_SEC) <<
            " seconds " << endl;

    delete[] numArray;
  }
  return 0;
}
```

Programming 5.

Background:

The workstation executing this program contains an AMD Ryzen 7 2700x CPU, 16GB of DDR4 3200 SDRAM, and Microsoft Windows 10 Pro 64-bit operating system (supports VMM). The program build was executed via the Microsoft Visual Studio IDE.

Huffman Algorithm Design:

1. The Input is an array of unique characters along with their frequency of occurrences and the output is Huffman Tree.

2. Create a leaf node for each unique character and build a min heap of all leaf nodes. Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root.

3. Extract two nodes with the minimum frequency from the min heap.

4. Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

5. Repeat steps #3 and #4 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Sources: https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/
https://en.wikipedia.org/wiki/Huffman_coding

Conclusion:

Given input:

```
array of characters = { 'a', 'b', 'c', 'd', 'e', 'f' };
frequency of characters = { 5, 9, 12, 13, 16, 45 };
```

Resulting output of one program execution:

```
f: 0
c: 100
d: 101
a: 1100
b: 1101
e: 111
```

Based on the results, the Huffman algorithm generates an optimal, variable-length prefix code for a character. The algorithm derives these codes from the estimated probability or frequency of occurrence (freq) for each possible value of the source character. Thus, common characters are generally represented using fewer bits than less common

characters, resulting in net loss-less data compression. Note that since the Huffman encoding algorithm extensively uses min heaps, its complexity should be similar to the min heap algorithm, i.e., time complexity of O($n\log_2 n$) and space complexity of O(1) where n is the number of unique characters.

Programming #5 in C++

Source: https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

```
// C++ program for Huffman Coding
#include <iostream>
#include <cstdlib>
using namespace std;

#define MAX_TREE_HT 100

// A Huffman tree node
struct MinHeapNode {
    // One of the input characters
    char data;

    // Frequency of the character
    unsigned freq;

    // Left and right child of this node
    struct MinHeapNode *left, *right;
};

// A Min Heap: Collection of min heap (or Huffman tree) nodes
struct MinHeap {
    // Current size of min heap
    unsigned size;

    // capacity of min heap
    unsigned capacity;

    // Attay of minheap node pointers
    struct MinHeapNode** array;
};

// A utility function allocate a new min heap node with given character
// and frequency of the character
struct MinHeapNode* newNode(char data, unsigned freq)
{
    struct MinHeapNode* temp = (struct MinHeapNode*)malloc(sizeof
                               (struct MinHeapNode));

    temp->left = temp->right = NULL;
    temp->data = data;
```

```
    temp->freq = freq;

    return temp;
}

// A utility function to create min heap of given capacity
struct MinHeap* createMinHeap(unsigned capacity) {
    struct MinHeap* minHeap = (struct MinHeap*)malloc(sizeof(struct MinHeap));

    // current size is 0
    minHeap->size = 0;

    minHeap->capacity = capacity;

    minHeap->array = (struct MinHeapNode**)malloc(minHeap->capacity * sizeof
                     (struct MinHeapNode*));
    return minHeap;
}

// A utility function to swap two min heap nodes
void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b) {
    struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}

// The standard minHeapify function.
void minHeapify(struct MinHeap* minHeap, int idx) {
    int smallest = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;

    if (left < minHeap->size && minHeap->array[left]->freq <
        minHeap->array[smallest]->freq)
        smallest = left;

    if (right < minHeap->size && minHeap->array[right]->
        freq < minHeap->array[smallest]->freq)
        smallest = right;

    if (smallest != idx) {
        swapMinHeapNode(&minHeap->array[smallest],
                        &minHeap->array[idx]);
        minHeapify(minHeap, smallest);
    }
}

// A utility function to check size of heap is 1 or not
int isSizeOne(struct MinHeap* minHeap) {
   return (minHeap->size == 1);
}
```

```
// A standard function to extract value node from heap
struct MinHeapNode* extractMin(struct MinHeap* minHeap) {
    struct MinHeapNode* temp = minHeap->array[0];
    minHeap->array[0] = minHeap->array[minHeap->size - 1];

    --minHeap->size;
    minHeapify(minHeap, 0);

    return temp;
}

// A utility function to insert a new node to Min Heap
void insertMinHeap(struct MinHeap* minheap, struct MinHeapNode* minHeapNode) {
    ++minHeap->size;
    int i = minHeap->size - 1;

    while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq) {
        minHeap->array[i] = minHeap->array[(i - 1) / 2];
        i = (i - 1) / 2;
    }
    minHeap->array[i] = minHeapNode;
}

// A standard function to build min heap
void buildMinHeap(struct MinHeap* minHeap) {
    int n = minHeap->size - 1;
    int i;

    for (i = (n - 1) / 2; i >= 0; --i)
        minHeapify(minHeap, i);
}

// A utility function to print an array of size n
void printArr(int arr[], int n)
{
    int i;
    for (i = 0; i < n; ++i)
        cout<< arr[i];

    cout<<"\n";
}

// Utility function to check if this node is leaf
int isLeaf(struct MinHeapNode* root) {
    return !(root->left) && !(root->right);
}

// Creates a min heap of capacity to size and inserts all character of
// data[] in min heap. Initially size of heap is equal to capacity
struct MinHeap* createAndBuildMinHeap(char data[], int freq[], int size) {
```

```
    struct MinHeap* minHeap = createMinHeap(size);

    for (int i = 0; i < size; ++i)
        minHeap->array[i] = newNode(data[i], freq[i]);

    minHeap->size = size;
    buildMinHeap(minHeap);
    return minHeap;
}

// The main function that builds Huffman tree
struct MinHeapNode* buildHuffmanTree(char data[], int freq[], int size) {
    struct MinHeapNode *left, *right, *top;
    struct MinHeap* minHeap = createAndBuildMinHeap(data, freq, size);

    // Iterate while size of heap doesn't become 1
    while (!isSizeOne(minHeap)) {
        left = extractMin(minHeap);
        right = extractMin(minHeap);
        top = newNode('$', left->freq + right->freq);

        top->left = left;
        top->right = right;

        insertMinHeap(minHeap, top);
    }
    return extractMin(minHeap);
}

// Prints huffman codes from the root of Huffman Tree. Uses arr[] to store codes
void printCodes(struct MinHeapNode* root, int arr[], int top) {
    // Assign 0 to left edge and recur
    if (root->left) {
        arr[top] = 0;
        printCodes(root->left, arr, top + 1);
    }

    // Assign 1 to right edge and recur
    if (root->right) {
        arr[top] = 1;
        printCodes(root->right, arr, top + 1);
    }

    // If this is a leaf node, then it contains one of the input
    // characters, print the character its code from arr[]
    if (isLeaf(root)) {
        cout<< root->data <<": ";
        printArr(arr, top);
    }
}
```

```
// The main function that builds a  Huffman Tree and print codes by traversing
// the built Huffman Tree
void HuffmanCodes(char data[], int freq[], int size) {
    // Construct Huffman Tree
    struct MinHeapNode* root
        = buildHuffmanTree(data, freq, size);

    // Print Huffman codes using
    // the Huffman tree built above
    int arr[MAX_TREE_HT], top = 0;

    printCodes(root, arr, top);
}

// Driver program to test above functions
int main()
{
    char arr[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
    int freq[] = { 5, 9, 12, 13, 16, 45 };

    int size = sizeof(arr) / sizeof(arr[0]);

    HuffmanCodes(arr, freq, size);

    return 0;
}
```