```
#ifndef NODE_H
#define NODE_H
#include <iostream>
#include <string>
#include <sstream>

using namespace std;

class Node{
        public:
                Node();
                Node(int n);
                string toString();
                Node* left;
                Node* right;
                int* freq;
                char key;
};
#include "node.cpp"
#endif
```

```
#include <iostream>
#include <string>
#include <sstream>

using namespace std;

Node::Node(){
        key = '\0';
        left = nullptr;
        right = nullptr;
        freq = new int(0);
}

Node::Node(int n){
        key = '\0';
        left = nullptr;
        right = nullptr;
        freq = &n;
}

string Node::toString(){
        stringstream ss;
        ss << *freq;
        return ss.str();
}

bool operator > (Node left, Node right){
        return left.freq > right.freq;
}

bool operator < (Node left, Node right){
        return left.freq < right.freq;
}

std::ostream& operator<<(std::ostream& stream, Node node)
{
        stream << node.toString();
        return stream;
}

std::ostream& operator<<(std::ostream& stream, Node* node)
{
        if (node != nullptr){
                stream << node->toString();
                return stream;
        }else {
                return stream;
        }
}
```

```
#ifndef DICT_H
#define DICT_H

#include <iostream>
#include <string>
#include <vector>

using namespace std;

class InvalidKey { }; // Class when get invalid key
class InvalidValue { }; // Class when invalid value

template <class T>
class Dict{
        private:
                vector< pair<char, T> > v;
        public:
                void set(char key, T val);

                T getValue(char key);

                char getKeyByIndex(int index);

                char getKeyByValue(T val);

                bool haveValue(T val);

                bool haveKey(char key);

                void modifyValue(char key, T val);

                void remove(char key);

                T operator[] (int i);

                int size();

                vector<char> keys();

                string toString();

                string toHeader();

                void printKeys();
};

#include "dict.cpp"
#endif
```

```
#include <iostream>
#include <string>
#include <sstream>

using namespace std;

template <class T>
void Dict<T>::set(char key, T val){
        pair<char, T> temp (key, val);
        v.push_back(temp);
}

template <class T>
T Dict<T>::getValue(char key){
        for (int i = 0; i < v.size(); i++){
                if (v[i].first == key){
                        return v[i].second;
                }
        }
        throw InvalidKey();
}

template <class T>
char Dict<T>::getKeyByIndex(int index){
        return v[index].first;
}

template <class T>
char Dict<T>::getKeyByValue(T val){
        for (int i = 0; i < v.size(); i++){
                if (v[i].second == val){
                        return v[i].first;
                }
        }
        throw InvalidValue();
}

template <class T>
bool Dict<T>::haveValue(T val){
        for (int i = 0; i < v.size(); i++){
                if (v[i].second == val){
                        return true;
                }
        }
        return false;
}

template <class T>
bool Dict<T>::haveKey(char key){
        for (int i = 0; i < v.size(); i++){
                if (v[i].first == key){
                        return true;
                }
        }
        return false;
}

template <class T>
void Dict<T>::modifyValue(char key, T val){
        for (int i = 0; i < v.size(); i++){
                if (v[i].first == key){
                        v[i].second = val;
                }
        }
}

template <class T>
void Dict<T>::remove(char key){
        for (int i = 0; i < v.size(); i++){
                if (v[i].first == key){
```

```
                        v.erase(v.begin() + i);
                }
        }
}

template <class T>
T Dict<T>::operator[] (int i){
        return v[i].second;
}

template <class T>
int Dict<T>::size(){
        return keys().size();
}

template <class T>
vector<char> Dict<T>::keys(){
        vector<char> out;
        for (int i = 0; i < v.size(); i++){
                out.push_back(v[i].first);
        }
        return out;
}

template <class T>
string Dict<T>::toString(){
        stringstream ss;
        for (int i = 0; i < v.size(); i++){
                string character = string(1, v[i].first);
                if (character == " "){
                        ss << "\\" << "s" << ": " << v[i].second << "\n";
                }else if(character == "\n"){
                        ss << "\\" << "n" << ": " << v[i].second << "\n";
                }else{
                        ss << character << ": " << v[i].second << "\n";
                }
        }
        return ss.str();
}

template <class T>
string Dict<T>::toHeader(){
        std::stringstream ss;
        for (int i = 0; i < v.size(); i++){
                string character = string(1, v[i].first);
                ss << character << v[i].second<<",";
        }
        ss << ",";
        return ss.str();
}

template <class T>
void Dict<T>::printKeys(){
        for (int i = 0; i < v.size(); i++){
                cout<<(v[i].first) <<","<<v[i].second<<endl;
        }
}
```

```
// pq.h
// This MinPriorityQueue template class assumes that the class KeyType has
// overloaded the < operator and the << stream operator.

#ifndef PQ_H
#define PQ_H

#include <iostream>
#include "heap.h"

template <class KeyType>
class MinPriorityQueue : public MinHeap<KeyType>
{
  public:
    MinPriorityQueue();             // default constructor
    MinPriorityQueue(int n);        // construct an empty MPQ with capacity n
    MinPriorityQueue(const MinPriorityQueue<KeyType>& pq);  // copy constructor

    KeyType* minimum() const;                   // return the minimum element
    KeyType* extractMin();                      // delete the minimum element and return
it
    void decreaseKey(int index, KeyType* key);  // decrease the value of an element
    void insert(KeyType* key);                  // insert a new element
    bool empty() const;                         // return whether the MPQ is empty
    int length() const;                         // return the number of keys
    std::string toString() const;               // return a string representation of the
MPQ

    // Specify that MPQ will be referring to the following members of MinHeap<KeyType>.

    using MinHeap<KeyType>::A;
    using MinHeap<KeyType>::heapSize;
    using MinHeap<KeyType>::capacity;
    using MinHeap<KeyType>::parent;
    using MinHeap<KeyType>::swap;
    using MinHeap<KeyType>::heapify;

    /* The using statements are necessary to resolve ambiguity because
       these members do not refer to KeyType.  Alternatively, you could
       use this->heapify(0) or MinHeap<KeyType>::heapify(0).
    */
};

template <class KeyType>
std::ostream& operator<<(std::ostream& stream, const MinPriorityQueue<KeyType>& pq);

class FullError { };    // MinPriorityQueue full exception
class EmptyError { };   // MinPriorityQueue empty exception
class KeyError { };     // MinPriorityQueue key exception

#include "pq.cpp"

#endif
```

```
// pq.cpp
using namespace std;
// These 3 constructors just call the corresponding MinHeap constructors.  That's all.
template <class KeyType>
MinPriorityQueue<KeyType>::MinPriorityQueue() : MinHeap<KeyType>()
{ }

/*============================================================
MinPriorityQueue(int n)      //default constructor
Precondition: Must be given a capacity size (n)
Postcondition: An empty queue with capacity of n
============================================================*/
template <class KeyType>
MinPriorityQueue<KeyType>::MinPriorityQueue(int n) : MinHeap<KeyType>(n)
{ }

/*============================================================
MinPriorityQueue(const MinPriorityQueue<KeyType>& pq)      //construct queue from another
 queue
Precondition: Must be given a priority queue
Postcondition: A queue deep copied from the given queue
============================================================*/
template <class KeyType>
MinPriorityQueue<KeyType>::MinPriorityQueue(const MinPriorityQueue<KeyType>& pq) : MinHea
p<KeyType>(pq)
{ }

/*============================================================
minimum()      //Return the pointer to minimium value of the heap
Precondition: Must be given a non-empty priority queue
Postcondition: The pointer to the minimum element in the queue
============================================================*/
template <class KeyType>
KeyType* MinPriorityQueue<KeyType>::minimum() const
{
        if (this->empty())
        {
                throw EmptyError();
        }
        return this->A[0];
}

/*============================================================
extractMin()      //Return the minimium value of the heap
Precondition: Must be given a non-empty priority queue
Postcondition: The minimum element in the queue
============================================================*/
template <class KeyType>
KeyType* MinPriorityQueue<KeyType>::extractMin()
{
        if (this->heapSize < 1)
        {
                throw EmptyError();
        }
        KeyType* minElement = this->A[0];
        this->A[0] = this->A[heapSize-1];
        this->heapSize -= 1;
        this->heapify(0);
        return minElement;
}

/*============================================================
decreaseKey()      //Decrease value of the given index and maintain the heap
Precondition: The private array A must be a heap. The value at the given index of the hea
p must be larger or equal to the given key
Postcondition: The maintained heap with the new replaced key
============================================================*/
template <class KeyType>
void MinPriorityQueue<KeyType>::decreaseKey(int index, KeyType* key)
{
```

```
        if (*(key) > *(this->A[index]))
        {
                cout << "key: " << *(key) << " " << "index: "<< *(this->A[index]) << endl
;
                throw KeyError();
        }
        this->A[index] = key;
        while (index > 0 && *(this->A[parent(index)]) > *(this->A[index]))
        {
                this->swap(index, parent(index));
                index = parent(index);
        }
        return;
}

/*============================================================
insert()      //insert a new key into the heap
Precondition: The private array A must be a heap
Postcondition: The maintained heap with the new inserted key
============================================================*/
template <class KeyType>
void MinPriorityQueue<KeyType>::insert(KeyType* key)
{
        this->heapSize += 1;
        int inf = ~(1 << 31);
        KeyType* temp = new KeyType(inf);
        this->A[heapSize-1] = temp;
        this->decreaseKey(heapSize-1, key);
}

/*============================================================
empty()      //Return true if the heap is empty
Precondition:
Postcondition: Return true if the heap is empty and false if the heap is not empty
============================================================*/
template <class KeyType>
bool MinPriorityQueue<KeyType>::empty() const
{
        return (this->heapSize == 0);
}

/*============================================================
length()      //Return size of the heap
Precondition:
Postcondition: Return size of the heap
============================================================*/
template <class KeyType>
int MinPriorityQueue<KeyType>::length() const
{
        return this->heapSize;
}


template <class KeyType>
std::string MinPriorityQueue<KeyType>::toString() const
{
        std::stringstream ss;

        if (heapSize == 0)
        {
                ss << "[ ]";
        }
        else
        {
                ss << "[";
                for (int index = 0; index < heapSize - 1; index++)
                        ss << *(A[index]) << ", ";
                ss << *(A[heapSize - 1]) << "]";
        }
        return ss.str();
```

```
}

template <class KeyType>
std::ostream& operator<<(std::ostream& stream, const MinPriorityQueue<KeyType>& pq)
{
        stream << pq.toString();

        return stream;
}
```

```
// heap.h
// a binary min heap

#ifndef HEAP_H
#define HEAP_H

#include <iostream>

const int DEFAULT_SIZE = 100;

template <class KeyType>
class MinHeap
{
  public:
    MinHeap(int n = DEFAULT_SIZE);          // default constructor
    MinHeap(KeyType* initA[], int n);        // construct heap from array
    MinHeap(const MinHeap<KeyType>& heap);  // copy constructor
    ~MinHeap();                             // destructor

    void heapSort(KeyType* sorted[]);  // heapsort, return result in sorted

    MinHeap<KeyType>& operator=(const MinHeap<KeyType>& heap);  // assignment operator
    std::string toString() const;     // return string representation

  protected:
    KeyType **A;     // array containing the heap
    int heapSize;   // size of the heap
    int capacity;   // size of A

    void heapify(int index);          // heapify subheap rooted at index
    void buildHeap();                 // build heap
    int leftChild(int index) { return 2 * index + 1; }  // return index of left child
    int rightChild(int index) { return 2 * index + 2; } // return index of right child
    int parent(int index) { return (index - 1) / 2; }   // return index of parent
    void heapifyR(int index);                  // recursive heapify
    void heapifyI(int index);                  // iterative heapify
    void swap(int index1, int index2);         // swap elements in A
    void copy(const MinHeap<KeyType>& heap);  // copy heap to this heap
    void destroy();                            // deallocate heap
};

template <class KeyType>
std::ostream& operator<<(std::ostream& stream, const MinHeap<KeyType>& heap);

#include "heap.cpp"

#endif
```

```
// heap.cpp

#include <sstream>
#include "heap.h"
// Implement heap methods here.

/*============================================================
MinHeap(int n = DEFAULT_SIZE)      //default constructor
Precondition: Must be given a capacity size (n)
Postcondition: An empty heap with capacity of n (1000 (default))
============================================================*/
template <class KeyType>
MinHeap<KeyType>::MinHeap(int n)
{
        A = new KeyType*[n];
        this->heapSize = 0;
        this->capacity = n;
}


/*============================================================
MinHeap(KeyType initA[], int n)      //construct heap from array
Precondition: Must be given an array and a capacity
Postcondition: A min heap constructed from the array
============================================================*/
template <class KeyType>
MinHeap<KeyType>::MinHeap(KeyType* initA[], int n)
{
  A = new KeyType*[n];
  this->capacity = n;
  this->heapSize = n;
  for (int i=0; i<n; i++)
  {
    this->A[i] = initA[i]; //traverse through initA and copy each element to current heap

  }
  buildHeap();
}


/*============================================================
MinHeap(const MinHeap<KeyType>& heap);  // copy constructor
Precondition: Must be given a heap
Postcondition: A min heap copied from the given heap
============================================================*/
template <class KeyType>
MinHeap<KeyType>::MinHeap(const MinHeap<KeyType>& heap)
{
  copy(heap); //call copy method which copies each element of heap parameter to current h
eap
}


/*============================================================
~MinHeap();                              // destructor
Precondition: Given a heap
Postcondition: The heap is deallocated
============================================================*/
template <class KeyType>
MinHeap<KeyType>::~MinHeap()
{
        this->destroy(); //call destroy method that deallocates current object
}


/*============================================================
heapSort(KeyType sorted[]);  // heapsort, return result in sorted
Precondition: Must be given a heap to be sorted
Postcondition: The heap is sorted in ascending order and the result is stored in sorted
============================================================*/
```

```
template <class KeyType>
void MinHeap<KeyType>::heapSort(KeyType* sorted[])
{
  // One by one extract an element from heap
        int n = this->heapSize;
  for (int i = n-1; i > 0; i--)
  {
    // Swap current root and end
    swap(0, i);
          this->heapSize -= 1;
    // call max heapify on the reduced heap
    heapify(0);
  }
        this->heapSize = n;
        for (int i = 0; i < this->heapSize; i++){
                sorted[n-i-1] = this->A[i]; //reverse max heap to get min heap
        }
}


/*============================================================
operator = (const MinHeap<KeyType>& heap);  // assignment operator
Precondition: Must be given a heap to be copied
Postcondition: Assign the heap to a new heap
============================================================*/
template <class KeyType>
MinHeap<KeyType>& MinHeap<KeyType>::operator=(const MinHeap<KeyType>& heap)
{
  this->copy(); //call copy method that copies each element of the heap parameter to curr
ent heap
  return *this;
}


/*============================================================
heapify(int index);          // heapify subheap rooted at index
Precondition: Must be given an index. Used on an array
Postcondition: The min heap property is maintained by calling the heapifyR
============================================================*/
template <class KeyType>
void MinHeap<KeyType>::heapify(int index)
{
  heapifyR(index); //calls recursive heapify
}


/*============================================================
heapifyR(int index);          // heapify subheap rooted at index
Precondition: Must be given an index. Used on an array
Postcondition: The min heap property is maintained by recursively calling heapifyR
============================================================*/
template <class KeyType>
void MinHeap<KeyType>::heapifyR(int index)
{
  int smallest = index;
  int left = leftChild(index);
  int right = rightChild(index);

  if (left < heapSize  && *(A[left]) < *(A[smallest]))
  {
    smallest = left;
  }
  if (right < heapSize && *(A[right]) < *(A[smallest]))
  {
    smallest = right; //switch smallest to right if A[right] is the smaller of the two ch
ildren
  }

  if (smallest != index)
  {
```

```
    swap(smallest, index); //swap current index with the index of the smaller child
    heapify(smallest); //recursively call heapify on lower children
  }
}


/*============================================================
heapify(int index);          // heapify subheap rooted at index
Precondition: Must be given an index. Used on an array
Postcondition: The min heap property is maintained by iteratively heapifying
============================================================*/
template <class KeyType>
void MinHeap<KeyType>::heapifyI(int index)
{
        int smallest = index;
  int left = leftChild(index);
  int right = rightChild(index);

  if (left < heapSize  && *(A[left]) < *(A[smallest]))
  {
    smallest = left; //intitialize smallest
  }
  if (right < heapSize && *(A[right]) < *(A[smallest]))
  {
    smallest = right; //switch smallest to right if A[right] is the smaller of the two ch
ildren
  }

  while(smallest != index) //iterative call
  {
    swap(smallest, index);
    index = smallest;
    left = leftChild(index);
    right = rightChild(index);

    if (left < heapSize  && *(A[left]) < *(A[smallest]))
    {
      smallest = left;
    }
    if (right < heapSize && *(A[right]) < *(A[smallest]))
    {
      smallest = right;
    }
  }
}


/*============================================================
buildHeap();                 // build heap
Precondition: Used on an array
Postcondition: Build a new heap from the array
============================================================*/
template <class KeyType>
void MinHeap<KeyType>::buildHeap()
{
  for (int i = heapSize/2; i>=0; i--)
  {
    heapify(i); //call heapify on the first half of the array
  }
}


/*============================================================
swap(int index1, int index2);        // swap elements in A
Precondition: Must be given two indices of the array
Postcondition: The values of the two indices are exchanged
============================================================*/
template <class KeyType>
void MinHeap<KeyType>::swap(int index1, int index2)
{
```

```
        KeyType* temp = this->A[index1]; //temporary variable to store A[index1]
        this->A[index1] = this->A[index2];
        this->A[index2] = temp;
}


/*============================================================
copy(const MinHeap<KeyType>& heap);  // copy heap to this heap
Precondition: Must be given a heap to copy
Postcondition: Copy the heap to this heap
============================================================*/
template <class KeyType>
void MinHeap<KeyType>::copy(const MinHeap<KeyType>& heap)
{
        this->heapSize = heap.heapSize;
        this->capacity = heap.capacity;
        A = new KeyType*[this->capacity];
        for (int i = 0; i < heap.heapSize; i++){
                this->A[i] = heap.A[i]; //traverses through the heap parameter and copies
 each of the element to the current heap
        }
}


/*============================================================
destroy();                              // deallocate heap
Precondition: Given a heap
Postcondition: The heap is deallocated
============================================================*/
template <class KeyType>
void MinHeap<KeyType>::destroy()
{
        delete []A; //deallocate object
}


// Use the following toString() for testing purposes.

template <class KeyType>
std::string MinHeap<KeyType>::toString() const
{
        std::stringstream ss;

        if (capacity == 0)
                ss << "[ ]";
        else
        {
                ss << "[";
                if (heapSize > 0)
                {
                        for (int index = 0; index < heapSize - 1; index++)
                                ss << *(A[index]) << ", ";
                        ss << *(A[heapSize - 1]);
                }
                ss << " | ";
                if (capacity > heapSize)
                {
                        for (int index = heapSize; index < capacity - 1; index++)
                                ss << *(A[index]) << ", ";
                        ss << *(A[capacity - 1]);
                }
                ss << "]";
        }
        return ss.str();
}

template <class KeyType>
std::ostream& operator<<(std::ostream& stream, const MinHeap<KeyType>& heap)
{
        return stream << heap.toString();
```

```
}
```

```
#include <iostream>
#include "pq.h"
#include <assert.h>
using namespace std;

void testDefaultConstructor()
{
        cout << "Test Default Constructor:"<<endl;
        MinPriorityQueue<int>queue(5);
        cout<<"\tTest1: Passed"<<endl;

        return;

}

void testInsert()
{
        cout << "Test Insert function:"<<endl;
        MinPriorityQueue<int>queue(5);
        int* pointer1 = new int(10);
        queue.insert(pointer1);
        assert (queue.toString() == "[10]");
        cout<<"\tTest1: Passed"<<endl;

        int* pointer2 = new int(1);
        queue.insert(pointer2);
        assert (queue.toString() == "[1, 10]");
        cout<<"\tTest2: Passed"<<endl;
        int* pointer3 = new int(3);
        queue.insert(pointer3);

        int* pointer4 = new int(5);
        queue.insert(pointer4);

        int* pointer5 = new int(0);
        queue.insert(pointer5);
        assert(queue.toString() == "[0, 1, 3, 10, 5]");
        cout<<"\tTest3: Passed"<<endl;

}

void testExtractMinimum()
{
        cout<< "Test Extract-Minimum and Minimum:"<<endl;
        MinPriorityQueue<int>queue(5);
        int* pointer1 = new int(10);
        queue.insert(pointer1);

        int* pointer2 = new int(1);
        queue.insert(pointer2);

        int* pointer3 = new int(3);
        queue.insert(pointer3);

        int* pointer4 = new int(5);
        queue.insert(pointer4);

        int* pointer5 = new int(0);
        queue.insert(pointer5);

        assert(*(queue.minimum()) == 0);
        assert(*(queue.extractMin()) == 0);
        assert(queue.toString() == "[1, 5, 3, 10]");
        cout<<"\tTest1: Passed"<<endl;
        assert(*(queue.minimum()) == 1);
        assert(*(queue.extractMin()) == 1);
        cout<<"\tTest2: Passed"<<endl;
        assert(*(queue.minimum()) == 3);
        assert(*(queue.extractMin()) == 3);
        cout<<"\tTest3: Passed"<<endl;
```

```
        assert(*(queue.minimum()) == 5);
        assert(*(queue.extractMin()) == 5);
        assert(*(queue.minimum()) == 10);
        assert(*(queue.extractMin()) == 10);
        cout<<"\tTest4: Passed"<<endl;
}

void testEmptyandLength()
{
        cout<< "Test Empty and Length:"<<endl;
        MinPriorityQueue<int>queue(5);
        assert(queue.empty());
        assert(queue.length() == 0);
        cout<< "\tTest1: Passed"<<endl;

        int* pointer1 = new int(10);
        queue.insert(pointer1);
        assert(!(queue.empty()));
        assert(queue.length() == 1);
        cout<< "\tTest2: Passed"<<endl;

        queue.extractMin();
        assert(queue.empty());
        assert(queue.length() == 0);
        cout<< "\tTest3: Passed"<<endl;

}

void testCopyConstructor()
{
        cout<< "Test Copy Constructor:"<<endl;

        MinPriorityQueue<int>queue(5);
        MinPriorityQueue<int>test_queue1(queue);
        assert(test_queue1.toString() == "[ ]");
        cout<< "\tTest1: Passed"<< endl;
        int* pointer1 = new int(10);
        queue.insert(pointer1);

        int* pointer2 = new int(1);
        queue.insert(pointer2);

        int* pointer3 = new int(3);
        queue.insert(pointer3);

        int* pointer4 = new int(5);
        queue.insert(pointer4);

        int* pointer5 = new int(0);
        queue.insert(pointer5);

        MinPriorityQueue<int>test_queue2(queue);
        assert(test_queue2.toString() == "[0, 1, 3, 10, 5]");
        cout<< "\tTest2: Passed"<< endl;
}

void testDecreaseKey()
{
        cout<< "Test Decrease Key:"<<endl;
        MinPriorityQueue<int>queue(5);
        int* pointer1 = new int(10);
        queue.insert(pointer1);

        int* pointer2 = new int(1);
        queue.insert(pointer2);

        int* pointer3 = new int(3);
        queue.insert(pointer3);

        int* pointer4 = new int(5);
```

```
        queue.insert(pointer4);

        int* pointer5 = new int(0);
        queue.insert(pointer5);

        assert(queue.toString() == "[0, 1, 3, 10, 5]");
        queue.decreaseKey(3, new int(-1));
        assert(queue.toString() == "[-1, 0, 3, 1, 5]");
        cout<<"\tTest1: Passed"<<endl;
}
int main()
{
        testDefaultConstructor();
        testInsert();
        testExtractMinimum();
        testEmptyandLength();
        testCopyConstructor();
        testDecreaseKey();
}
```

```
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <vector>

#include "pq.h"
#include "node.h"
#include "dict.h"

using namespace std;

Dict<int> countFrequency(string input){
        Dict<int> dictionary;
        stringstream s;
        ifstream inputFile (input);
        s << inputFile.rdbuf();
        string fileString = s.str();
        for (int i = 0; i < fileString.size(); i++){
                char character = fileString[i];
                if (dictionary.haveKey(character)){
                        int currentFreq = dictionary.getValue(character);
                        dictionary.modifyValue(character, currentFreq + 1);
                }else {
                        dictionary.set(character, 1);
                }
        }
        return dictionary;
}

Node* huffman(Dict<int> dict){
        int n = dict.size();
        MinPriorityQueue<Node> q;
        for (int i = 0; i < n; i++){
                Node* tempNode = new Node;
                tempNode->key = dict.getKeyByIndex(i);
                int* tempInt = new int(dict[i]);
                tempNode->freq = tempInt;
                q.insert(tempNode);
        }
        for (int i = 0; i < dict.size() - 1; i++){
                Node* tempNode = new Node;
                Node* x = q.extractMin();
                Node* y = q.extractMin();
                tempNode->left = x;
                tempNode->right = y;
                *(tempNode->freq) = *(x->freq) + *(y->freq);
                q.insert(tempNode);
        }
        return q.extractMin();
}

Dict<string> treeToDict(Node* node, Dict<string> dictionary, string current){
        if (node!= NULL){
                if (node->left == nullptr && node->right == nullptr){
                        dictionary.set(node->key, current);
                        return dictionary;
                }
                current.push_back('0');
                dictionary = treeToDict(node->left, dictionary, current);
                current.pop_back();
                current.push_back('1');
                dictionary = treeToDict(node->right, dictionary, current);
        }
        return dictionary;
}

pair<string, int> encode2bit(string encoded_string){
        stringstream bs;
```

```
        char byte = 0;
        int count = 0;
        for (int i=0; i<encoded_string.size(); i++){
                if (encoded_string[i] == '0'){
                        byte = byte << 1;
                }
                else if (encoded_string[i] == '1'){
                        byte = (byte << 1) | 1;
                }
                count += 1;
                if (count == 8){
                        bs << byte;
                        byte = 0;
                        count = 0;
                }
        }

        byte = byte << (8-count);
        bs << byte;
        pair<string, int> result (bs.str(), 8 - count);
        return result;
}

string bit2encode(string bit_string){
        char decode_key = 1;
        int count = 0;
        stringstream es;
        for (int i=0; i < bit_string.size(); i ++){
                for (int count=7; count>=0; count--){
                        if (bit_string[i] & (1<<count)){
                                es << '1';
                        }
                        else{
                                es << '0';
                        }
                }
        }
        return es.str();
}

void encode(string input, string output, Dict<string> dictionary){
        // read input file to inputString
        ifstream inputFile(input);
        ofstream outputFile(output);
        stringstream s;
        s << inputFile.rdbuf();
        string inputString = s.str();

        // encode inputString to encodedString
        stringstream es;
        for (int i = 0; i < inputString.size(); i++){
                char character = inputString[i];
                string encodedChar = dictionary.getValue(character);
                es << encodedChar;
        }
        string encodedString = es.str();
        // encode
        pair<string,int> temp = encode2bit(encodedString);
        string bits = temp.first;
        int pad = temp.second;

        // append dictionary for decode purposes
        outputFile << dictionary.toHeader();

        // append padding to outputFile
        outputFile << pad;

        // convert encodedString to bits and write to outputfile
        outputFile << bits;
}
```

```
void decode(string input, string output){
        ifstream inputFile(input);
        ofstream outputFile(output);

        // read inputFile to text
        stringstream s;
        s << inputFile.rdbuf();
        string inputString = s.str();

        // extract header
        stringstream hs;
        int i = 0;
        while (inputString[i] != ',' || inputString[i+1] != ','){
                const char temp = inputString[i];
                hs << temp;
                i++;
        }
        string headerString = hs.str();
        i = i + 2;

        // extract padding
        int pad = inputString[i] - '0';
        i++;

        // extract data
        stringstream ds;
        while (i < inputString.size()){
                ds << inputString[i];
                i++;
        }
        string dataBitString = ds.str();

        // parse header to dict
        Dict<string> dictionary;
        int j = 0;
        while (j < headerString.size()){
                char key = headerString[j];
                j++;
                stringstream val;
                while (j < headerString.size() && headerString[j] != ','){
                        val << headerString[j];
                        j++;
                }
                j++;
                dictionary.set(key,val.str());
        }
        string dataString = bit2encode(dataBitString);

        // depadding last bit
        dataString = dataString.substr(0, dataString.size() - pad);

        // decode based on dict
        stringstream code;
        for (int i = 0; i < dataString.size(); i++){
                code << dataString[i];
                if(dictionary.haveValue(code.str())){
                        outputFile << dictionary.getKeyByValue(code.str());
                        code.str("");
                }
        }
}

int main(int argc, char* argv[]){
        string action = argv[1];
        string inputFile = argv[2];
        string outputFile = argv[3];

        if (action == "-c"){
                Dict<int> dictFreq = countFrequency(inputFile);
```

```
                Node* root = huffman(dictFreq);
                Dict<string> dictHuffman;
                string current;
                dictHuffman = treeToDict(root, dictHuffman, current);
                encode(inputFile, outputFile, dictHuffman);
        } else if (action == "-d"){
                decode(inputFile, outputFile);
        } else{
                cout << "Invalid argument" << endl;
        }
        return 0;
}
```