```
//=============================================================
// heapPQ.cpp
// This file contains excerpts from the MinHeap class,
// Priority Queue class, and Huffman Coding.
//=============================================================

//=============================================================
// MINHEAP
//=============================================================

template <class KeyType>
class MinHeap
{
  public:
    MinHeap(int n = DEFAULT_SIZE);
    MinHeap(KeyType initA[], int n);
    MinHeap(const MinHeap<KeyType>& heap);
    ~MinHeap();

    void heapSort(KeyType sorted[]);

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

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

        void heapify(int index);
    void buildHeap();
        int leftChild(int index) { return 2 * index + 1; }
        int rightChild(int index) { return 2 * index + 2; }
        int parent(int index) { return (index - 1) / 2; }
    void swap(int index1, int index2);
    void copy(const MinHeap<KeyType>& heap);
    void destroy();
};


//=============================================================
// buildHeap
// Builds a MinHeap
// Pre-Conditions:
//              none
// Post-Conditions:
//              the heap is definitely a Min-Heap
//=============================================================
template<class KeyType>
void  MinHeap<KeyType>::buildHeap()
{
        heapSize = capacity;
        for(int i = heapSize / 2 - 1; i >= 0; i--)
        {
                heapify(i);
        }
}

//=============================================================
// heapify
// Makes a heap into a min heap
// Pre-Conditions:
//              Both children must be roots of a Min-Heap
```

```
// Post-Conditions:
//              The heap is a Min-Heap (if the
//              Pre-Condition is satisfied)
//=============================================================
template<class KeyType>
void  MinHeap<KeyType>::heapify(int index)
{
        int l = leftChild(index);
        int r = rightChild(index);
        int min;
        if(l < heapSize && *(A[index]) > *(A[l]))
                min = l;
        else
                min = index;
        if(r < heapSize && *(A[min]) > *(A[r]))
                min = r;
        if(min != index)
        {
                swap(index, min);
                heapify(min);
        }
}


//=============================================================
// swap
// Swaps two items
// Pre-Conditions:
//              The indices are valid
// Post-Conditions:
//              The values at the indices
//              have been swapped
//=============================================================
template<class KeyType>
void  MinHeap<KeyType>::swap(int index1, int index2)
{
        KeyType* temp = A[index1];
        A[index1] = A[index2];
        A[index2] = temp;
}



//=============================================================
// heapSort
// Pre-Conditions:
//              The heap must be a MinHeap
// Post-Conditions:
//              sorted is now sorted in ascending order
//=============================================================
template<class KeyType>
void MinHeap<KeyType>::heapSort(KeyType* sorted[])
{
        sorted = new KeyType*[capacity];
        //buildHeap();
        for(int i = capacity - 1; i >= 0; i--)
        {
                sorted[i] = A[0];
                swap(0,i);
                heapSize--;
                heapify(0);
        }
        heapSize = capacity;
```

```
}

//============================================================
// PRIORITY QUEUE
//============================================================

//============================================================
// extractMin()
// removes and returns minimum value
// Pre-Conditions:
//              Calling object must be a minimum
//              priority queue or empty
// Post-Conditions:
//              heapSize will be one less (if not originally empty)
//              and will be a minimum priority
//              queue
//============================================================

template <class KeyType>
KeyType* MinPriorityQueue<KeyType>::extractMin ( void )
{
        if(!empty())
        {
                swap(0,heapSize-1);     //swaps smallest value to end of pq
                heapSize--;
                heapify(0);
                return A[heapSize];                             //returns smallest value
        }
        else
        {
                return NULL;                            //returns NULL if pq is empty
        }
}

//============================================================
// decreaseKey(int index, KeyType* key)
// decreases the key of the given index to the given key
// Parameters:
//              int index               - the index to be decreased
//              KeyType* key    - the new key to be assigned
// Return Value: N/A
// Pre-Conditions:
//              The calling object must be a priority queue
// Post-Conditions:
//              The key of the given index (in the original pq)
//              will be decreased and the calling
//              object will be a minimum priority queue
//============================================================

template <class KeyType>
void MinPriorityQueue<KeyType>::decreaseKey     (int index, KeyType* key)
{
                if(*key > *A[index])
                {
                        throw KeyError();                       //index is out of bounds

                }else{
                        A[index] = key;
                        while(index > 0 && *A[index] < *A[parent(index)])
                        {
                                swap(index, parent(index));     //swaps index with
                                index = parent(index);          //its parent
                        }
```

```
                }
}

//=============================================================
// insert(KeyType* key)
// inserts the given key into the minimum priority queue
// Parameters:
//              KeyType* key - the key to be inserted into the min
//              priority queue
// Return Value: N/A
// Pre-Conditions:
//              The calling object must be a minimum priority queue
// Post-Conditions:
//              The minimum priority queue will now contain the
//              given key and be a minimum priority queue
//=============================================================

template <class KeyType>
void MinPriorityQueue<KeyType>::insert (KeyType* key)
{
        if(heapSize == capacity)
        {
                throw FullError();                                      //if heap is full
        }else{
                heapSize++;
                A[heapSize - 1] = key;          //inserts key at end of array
                decreaseKey(heapSize - 1, key);//sorts key into correct
        }                                                                               //plac
e

        //sorted[heapSize] = key;
        //heapSize++;
        //this->heapSort(sorted);
}


//=============================================================
// HUFFMAN CODING
//=============================================================

//=============================================================
// creates a min priority queue from a text file
//=============================================================
MinPriorityQueue<Node> fileToMPQ ( string fileName, map<char,
        int> &frequencies )
{
        ifstream file(fileName.c_str());

        char c;
        int counter;
        if(file.is_open())
        {
                while(file.get(c))
                {
                        counter++;
                        frequencies[c]++;
                }
        }

        frequencies.erase(frequencies.begin());
        MinPriorityQueue<Node> nodes(frequencies.size());

        for(map<char, int>::iterator it = frequencies.begin();
```

```
        it != frequencies.end(); it++)
        {
                cout << it->first << " " << it->second << endl;
                Node c;
                c.character = it->first;
                c.freq = it->second;
                nodes.insert(&c);
                //string temp = nodes.toString();
                //cout << temp << endl;
        }

        return nodes;
}
//============================================================
// creates a huffman tree from a priority queue
//============================================================
Node buildHuffmanTree ( map<char, int> frequencies,
        MinPriorityQueue<Node> &nodes )
{
        int n = frequencies.size();
        for(int i = 0; i < n - 1; i++)
        {
                Node* z = new Node;
                z->left = nodes.extractMin();
                z->right = nodes.extractMin();
                if(z->left != NULL)
                        z->freq += z->left->freq;
                if(z->right != NULL)
                        z->freq += z->right->freq;
                z->character = '\0';
                nodes.insert(z);
        }
        Node *root = nodes.extractMin();
        Node x = *root;
        return x;
}
//============================================================
// creates huffman codes from huffman tree
//============================================================
void searchHuffmanTree ( map<char, int> frequencies, map<char,
        string> &huffCodes, Node* z, Node* root, string &s )
{
        char c;
        if(z->isLeaf())
        {
                c = z->character;
                huffCodes[c] = s;
        }
        else{
                if(z->left != NULL)
                {
                        s = s + "0";
                        searchHuffmanTree(frequencies, huffCodes, z->left, root, s);
                }
                if(z->right != NULL)
                {
                        s = s + "1";
                        searchHuffmanTree(frequencies, huffCodes, z->right, root, s);
                }
        }
        s = s.substr(0,s.size() - 1);
}
```