In the remainder we deal with yet another application of priority queues, that of text compression. In text compression we are given a string T of characters drawn from some alphabet Σ and we want to efficiently encode string T into a smaller (binary) string Y. The way compression is performed is by first assigning a distinct bit sequence to each element of Σ and then storing the characters of the string using this encoding consecutively in computer memory. Since n bits have 2^n distinct values an alphabet with $\mid \Sigma \mid$ characters requires $\lg \mid \Sigma \mid$ bits. If each character is equally likely to appear in the documents stored in a computer not much can be done. In real-life however this is not the case. With English certain letters are more likely to appear than others (e.g. e is more frequent than w or x).

The techniques for compression that we will examine are **lossless** i.e. for a given Y we can fully recover T from it. There are also **noisy or lossy** techniques where full-recovery of Y from T is not possible. Such techniques can be used in image or voice compression. The process that converts T into Y is called **encoding or compression**.

The primary method that will be presented in this subject is Huffman encoding that yields what is known as Huffman codes. Huffman codes result in space savings of 20% to 90% depending on the structure of the input string or file.

Huffman's algorithm/encoding is a **greedy algorithm** that during its course of execution makes that choice that looks best at that instant.

Huffman encoding relies on the following observation/example. Suppose a string consists not of all ASCII characters but only of few of them, say the numerals (0..9) plus few other characters (delimiters such as space, tab, newline, carriage return), then we can represent each character not with 7 or 8 bits (i.e. ASCII) but with only 4 bits (16 different values). Huffman encoding requires that we know in advance the alphabet Σ and also the frequency of each character. Each character is represented by a different encoding string or binary character code (or code in short) in such a way that no encoding string is a prefix of another encoding string. Codes that have this property are called **prefix codes**.

The important property of prefix codes is that decoding is quite simple. Since no codeword is a prefix of another one, that codeword at the start of a string or a file is unambiguous and can be removed from the coded file; repeating this process decodes the rest of the string or file uniquely.

Prefix codes and prefix trees

We are going to represent prefix codes by binary trees. For each string or file the characters used are collected in set C. The frequency $f(c)$ of any character c is also found. Each node of the tree has a left child that represents a zero and a right child that represents an one. Each leaf is a character that appears in the string or file, along with its frequency. A path from the root to a leaf "character" represents the code for that character where every left child traversal corresponds to a zero and very right child one to an one. The binary tree representing a prefix code is full with $\mid C \mid -1$ internal children where $\mid C \mid$ is the number of unique leaves/characters that appear in the string/file. An internal node of the tree also has a frequency field. The grequency field of an internal node is the sum of the frequencies of the leaves in the subtree of this node.

The number of bits required to encode the file with a given prefix code tree is $B(T) = \sum_{c \in C} f(c)d(c)$, where $d(c)$ is the depth of character c in the tree, and $f(c)$ is the frequency of character c in the string/file that was used to obtain the tree.

The algorithm that constructs a Huffman code (i.e. an optimal prefix code) is a greedy algorithm. It builds the tree that corresponds to the code bottom to top.

Initially the $\mid C \mid$ characters form individual one-node trees. The single node is the root of its own tree. The information associated with each node is the character representation (e.g. 'A') of the node and the frequency of the character in the string or file. The algorithm works bottom-up by combining two trees into a single one by creating a root node for the new tree whose left child is the root of one of the two nodes and the right child the other node. The frequency field of the new root is the sum of the frequencies of the roots of the two consituent trees (now, children). There is no character representation associated with such an internal node. Therefore after $\mid C \mid -1$ such operations there is only one tree left whose frequency field is the length of the string or file.

The criterion for choosing the two trees to combine is a "greedy" one: the two trees with the lowest frequency fields at their respective roots are chosen.

We use a priority queue to maintain the trees, and implement the priority queue with a MINHEAP. The priority of a tree is the frequency of its root.

Encoding

The following pseudocode shows the steps for encoding a file and constructing the Huffman code.

```
   Encode (uncompressed file)
1. Read file and obtain character array C with frequency of each character i stored in C[i].freq
2. Huffman(C[]);
3. Traverse Tree to obtain prefix codes for each i in C[].
4. Read file replacing character i with a bitstring symbol C[i].symbol
5. Save prefix code table in file.
```

The following pseudocode shows the steps of Huffman's algorithm.

```
    Huffman (C[ ])//Every element of C has a frequency field freq.
0.   priorityqueue Q;
1.   for(c=0;c<|C|;i++){
2.    create single node bin. tree T: T[c].char=c, T[c].freq=C[c].freq
3.    Q[c] =T[c];
4.   }
5.   Build-MinHeap(Q,|C|);
6.   for(i=0;i<n-1;i++) {
7.     z=ALLOCATENODE();
8.     x=z.left  = EXTRACT-MIN(Q);
9.     y=z.right = EXTRACT-MIN(Q);
10.    z.freq = x.freq + y.freq;
11.    z.char = -1;
12.    InsertMinHeap(Q,z);
13.  }
14.  return(RemoveMin(Q));
```


Decoding

The running time of Huffman is $O(n \lg n)$, where $n =\mid C \mid$. Lines 0-4 require $O(\mid C \mid) = O(n)$ time. Line 5 (Build-MinHeap operation) requires $O(n)$ time and lines 6-12 require $O(\lg n)$ time per iteration for a total of $O(n \lg n)$. time.

The decoding operation traverses the tree root to leaf using the bitstring of the string /file.

```
    HuffmanDecode (compressed file,HuffmanTree T, bitstring B, currentposition pos)
0.  Obtain T from file;
1.  pointer = T;
2.  while (B[pos] != EOFile ) {
3.    if (B[pos] == 0)
4.     pointer = pointer.left;
5.    else
6.     pointer = pointer.right;
7.    if (isleaf(pointer) == TRUE) { // ie. pointer.char != -1
8.     Output(pointer.char);
9.     pointer=T;
10.   }
11. }
}
```

The decoding procedure works in time linear to the size in bits of the encoded/compressed file.

