Heaps, Priority Queues and Heapsort; Huffman Coding

Chapter 6 (all)

Chapter 16 introduction and section 16.3

Disclaimer: *These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Spring 2019; distribution outside this group of students is NOT allowed.*

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

In the beginning of this class we introduced a couple of sorting algorithms

- Insertion-Sort, an $O(n^2)$ running time algorithm.
- Selection-Sort, an $O(n^2)$ running time algorithm.
- Bubble-Sort, an $O(n^2)$ running time algorithm.
- Merge-Sort an $O(n \lg n)$ running time algorithm.

In the following lectures we are going to introduce a couple of new sorting algorithms as well as data structures.
The algorithms we are going to introduce are

- An algorithm called **heapsort** with $O(n \lg n)$ worst-case running time that uses a new data structure called a **(binary) heap**.
- An algorithm called **quicksort** whose average case running time is $O(n \lg n)$ and although its worst-case running time is $O(n^2)$ in practice it performs better than any other sorting algorithm on the average.
- Linear time sorting algorithms that work when the data type of the input is specific (distribution-based algorithms).

The ADT priority queue with operations as follows will be used.

- A priority queue (PQ) S that implements efficiently the following operations: FindMax(S), Insert(S,x), ExtractMax(S). (We can also have a PQ T with the following operations FindMin(T), Insert(T,x), ExtractMin(T) implemented similarly.)
- The data structure called **(binary) heap** that allows a PQ to be implemented efficiently.

We note that the notion of a **heap** as defined here has nothing to do with the same term used in compilers to mean an alternative memory area to a **stack**, where memory is collected when not in use and returned to the running program (using operations such as malloc, free, new or delete).

Moreover note that some textbooks implement not a (binary) heap but a heap tree. The latter is a binary tree that has some properties (to be defined as a MaxHeap property).

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

A **(binary) heap** is a data structure that has the following properties. From now on we shall call a binary heap, a heap, dropping the binary.

1. Each element of a heap has some priority attached to it; other information may be associated with each element.

2. A (binary) heap is represented by an array $A[1..n]$ of length $length(A)$. The contents of $A[i]$ contain the priority of the i-th element of the heap. A pointer $A[i].pointer$ may point to a memory area storing other useful information associated with this element.

3. The size of a heap is denoted by $heapsize \leq length(A)$.

4a. For every node indexed i of the heap we have that $A[i] \geq A[2i]$ and $A[i] \geq A[2i+1]$, if the two elements belong to the heap. This is equivalent to $A[floor(i/2)] \geq A[i]$. We call this property the **MaxHeap property**.

If on the other hand

4a. For every node indexed i of the heap we have that $A[i] \leq A[2i]$ and $A[i] \leq A[2i+1]$, if the two elements belong to the heap. This is equivalent to $A[floor(i/2)] \leq A[i]$. We call this property the **MinHeap-property**.

In the remainder we will be dealing with Max-Heaps (i.e. heaps that maintain the MaxHeap-property), although the results we will obtain can be easily translated into equivalent results for Min-Heaps.

Another equivalent representation (view) of a (binary) heap is by a complete binary tree whose all levels but the lowest (with highest depth) are filled with elements and in the last level nodes are filling the tree left to right. Nodes of the tree are numbered top-to-bottom, left-to-right starting from the root (first element with label 1). A parent labeled i has at most two children labeled $2i$ and $2i+1$ if they both exist (this is true for an array whose first element is $A[1]$). We call this view heap-tree. The equivalence between the array and its heap-tree view is immediate.

Given the tree view of the array-based (binary) heap,

a. **parent** (i) return floor$(i/2)$;

b. **left** (i) return $(2 * i)$;

c. **right** (i) return $(2 * i + 1)$;

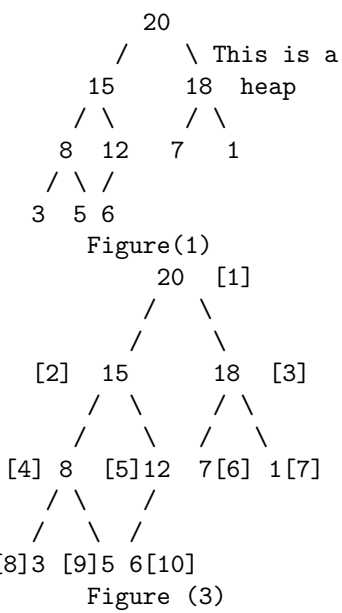
Such operations in a language like C can be implemented with shift operations efficiently.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

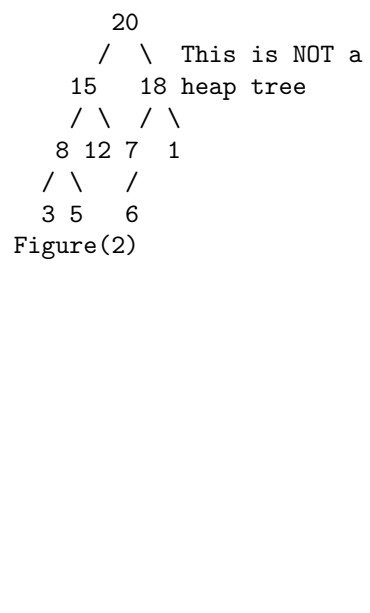
Representation

Array A below is a Max-Heap. Figure (1) shows the array as a heap-tree that has the MaxHeap-property. Figure (2) is not a heap as nodes indexed 10 and 11 are not filled with elements (the node indexed 9 holds a 5, and the node index 12 holds a 6: there is a gap, i.e. nodes in last level not filled left to right). In Figure (3) in brackets we indicate the index of a node that is the same as the index of that element within array A.

```
      1   2   3  4   5   6  7  8  9 10
A=[ 20  15  18  8  12   7  1  3  5  6]
```



Figure(1)



Figure(2)

Figure (3)

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Sometimes the tree view of the heap that is filled with nodes top to bottom and left to right is referred to as a complete binary tree view.

Sometimes, the tree view of a heap (i.e. a complete binary tree with the MinHeap or MaxHeap property) is referred to as the MinTree or MaxTree (compare to MinHeap and MaxHeap).

The **height** of a tree node, is the size (in edges) of the longest **downward** path from the node to a descendant leaf of the tree. (or equivalently, it is the height of the tree minus the depth of the node). The height of a tree is the height of the root.

A heap with n elements has a tree view of height $\lfloor \lg n \rfloor$. A heap with n elements has a tree view with $\lceil \lg (n+1) \rceil$ levels (the root is at level 1 and so on).

Some operations defined on heaps are the following ones.

Heapify (A,i) preserves the heap property at the node indexed i if it is not valid. The precondition is that the left and right subtree views of that node HAVE the heap property; the tree view rooted at i may not. Depending on the heap we use we can call it Max-Heapify or Min-Heapify. In some other textbooks the Heapify operation is generically called **Sift or Shift Fix, etc**. Heapify is implied in the operation of Program 9.4 in section 9.3.

Build-Heap(A). Given an unordered array A a (binary) heap is built in place. Such an operation is sometimes referred to as a **Heap Initialization** operation (section 9.3).

Extract-Max (A). Extract the element with the maximum value of the heap;after the extraction the heap-property is maintained for the remainder. For a min-heap we can equivalently define **Extrac-Min** (A). ExtractMax is described in Program 9.4 of section 9.3.

Insert(A,**x**). An element x is inserted into the heap. Insert is described in Program 9.3 of section 9.3.

HeapSort (A). Given an (unordered) array A this procedure sorts A in place.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Max-Heapify

When Max-Heapify is called on the element indexed i the implicit assumption is that the subtree views rooted at left(i) and right(i) are heaps; $A[i]$ however may violate the heap property. The purpose of Heapify is to slide down this key so that the subtree view rooted at i becomes a heap.

```
                                Heap property may not be true v for i
 Max-Heapify (A,i)                                                i
// Subtrees rooted at 2i and 2i+1 are heaps       Heap-----v  /   \  v---Heap
// and thus have max-heap property                          -----  ------
1. largest= index of the maximum key among                | 2i  | |2i+1 |
           A[i], A[left(i)], A[right(i)];                 |     | |     |
2. if (largest != i) {                                    ------  -------
3.   exchange(A[i],A[largest]);
4.   Max-Heapify(A,largest); //recursively continue
   }
```

Analysis. A Max-Heapify operation takes constant time for steps 1, 2, and 3. Line 4 is run at most H times where H is the height of the heap. A heap with n nodes has height $H = \Theta(\lg n)$. In general if the height of node labeled i is $h \leq H$, then the time for heapify is $T(h)$

$$T(h) = T(h-1) + \Theta(1) \text{ and thus, } T(h) = O(h) = O(H) = O(\lg n).$$

(c) Copyright A. Gerbessiotis.

```
 Build-MaxHeap(A)  // Heap Initialization procedure.
1. heapsize(A)=length(A);
2. for(i=floor(length(A)/2);i>=1;i--) {
3.   Max-Heapify(A,i);
   }
```

The input to Build-MaxHeap is an unordered array A. The outcome of running Build-MaxHeap is to create a MaxHeap out of an unordered array A. For a array of size n half its elements already maintain the heap property (elements indexed length(A/2)+1 through n). This is so because these elements do not have any children and thus have the heap property by default. Therefore the elements that may not maintain the heap property are the remaining elements located in positions 1 through $\lfloor length(A)/2 \rfloor$. The heap construction process moves from the lowest level towards the top of the tree view. Therefore subtree views rooted at successive levels maintain the heap property, and finally the whole tree view does.

Analysis. A naive analysis shows that each of $O(n/2)$ elements spends time $O(h) = O(\lg n)$ in line 3 for a total time $O(n \lg n)$. A tighter bound follows from the simple observation that only the root might require $\lg n$ iterations for Heapify; more than half of the nodes are leaves and thus the cost of Heapify is zero, for one quarter or more the cost of Heapify is one iteration per node and so on. The number of nodes at height h is ceiling$(n/2^{h+1})$. Each of them will perform at most h iterations (calls/operations) of Max-Heapify, each one of them computing the maximum of 3 elements i.e. 2 comparisons. Thus in total the number of iterations of Max-Heapify is at most $hn/2^{h+1}$ and twice as much is the number of comparisons. The total running time is then the order of the total number of Max-Heapify iterations which is at most

$$\sum_{h=0}^{\lg n} hn/2^h = n \cdot \sum_{h=0}^{\lg n} h(1/2)^h = n \cdot \sum_{i=0}^{\lg n} i(1/2)^i \leq n/2 \cdot 2 = n$$

where we used the fact (Appendix) that for $x < 1$, $\sum_{i=0}^{\infty} ix^i \leq x/(1-x)^2$ and for $i = h$, $x = 1/2$ we get $\sum_{i=0}^{\infty} i(1/2)^i \leq 2$. Each Line 1 of Heapify performs at most 2 comparisons. Thus the number of Max-Heapify function-calls is at most n, the number of comparisons performend is at most $2n$ and the running of Build-MaxHeap is thus $O(n)$.

(c) Copyright A. Gerbessiotis.

Introduction

The **heapsort** algorithm utilizes Build-MaxHeap to build a heap from the input keys into an array A of size n. The key with the maximum value is stored in the top of the heap (root of heap tree or key in index position 1 of array A), and thus it can be extracted from the heap and placed in its correct position in the output sequence (the last position). This is achieved by exchanging $A[1]$ and $A[n]$. This exchange, may violate the heap property for elements $A[1 \ldots n-1]$ (note that the extraction of max makes the heap of size one less). It is obvious however that the "children" of $A[1]$ in the tree view have the heap-property since the rest of the heap was untouched by the exchange. A call to Max-Heapify will maintain the heap property for $A[1]$ and the resulting $A[1..n-1]$ becomes a heap. We continue similarly with the extraction of the current maximum key from the heap $A[1..n-1]$.

```
 HeapSort(A,n)              // A is an unordered array of n keys
1. Build-Max-Heap(A);       // A  has become a heap
2. for(i=n;i>=2;i--){       // A[1..i] is a heap A[i+1..n] is a sorted subarray
3.   exchange(A[1], A[i]); //Extract Max and place it into A[i],move A[i] to A[1]
     // A[1..i-1] may not be a heap, A[i..n] is a sorted subarray
4.   heapsize(A) = heapsize(A)-1;   //one key was extracted from heap;adjust size
5.   Max-Heapify(A,1);     //Root may not have Heap Property; Run Heapify
                           // A[1..i-1] is guaranteed to be a heap
   }
```

Step 1 requires time $O(n)$. The loop of lines 2-5 is executed $n-1$ times and each call to Max-Heapify takes $O(\lg n)$ steps. The total running time is thus $O(n \lg n)$ in the worst case.

(c) Copyright A. Gerbessiotis.

The heap data-structure finds however many application in operating system design (scheduling) and (event-driven) simulation. It implements a highly efficient priority queue ADT.

A **priority queue** is an ADT for maintaining a set S of elements each with an associated value/priority called a **key**. A priority queue supports the following operations.

- **Insert(S,x)** inserts element x into queue S.
- **FindMax(S)** returns the element of S with the highest key value or priority (the element is not removed from S). This is the Search operation for a priority queue.
- **ExtractMax(S)** extracts the element of S with the highest key value (the element is removed form S). This is the Delete operation for a priority queue.

We can also define a priority queue with FindMin and ExtractMin operations.

0. The priority queue ADT can be realized using a max-heap or a min-heap as appropriate.
1. Operation **FindMax(S)** can be implemented easily by having a heap return its first element $A[1]$.

```
FindMax(A)
 return(A[1]);
```

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

2. The **ExtractMax(S)** operation works similarly with the exception that it removes the MAX element from the heap. Its operation is also similar to heapsort.

```
 Extract-Max(A)
1. if heap-size(A) < 1
     return("Error. No heap elements");
2. max=A[1];
3. A[1]=A[heap-size(A)];
4. heap-size(A) = heap-size(A)-1;
5. Max-Heapify(A,1);
6. return(max);
```

The running time of Extract-Max is $O(\lg n)$.

3. Insert-MaxHeap inserts a node into a heap/priority queue. It first expands a heap by adding a leaf to the tree. Then similarly to the insertion operation of Insertion-Sort it traverses a path from this leaf toward the root to find the proper place for the inserted element. (A 'right' to 'left' operation in the array but with jumps.)

```
 Insert-MaxHeap(A,key)
1. heap-size(A) = heap-size(A)+1;
2. i=heap-size(A);
3. while ((i>1) and A[parent(i)] < key) {
4.   A[i]=A[parent(i)];
5.   i=parent(i);
   }
6. A[i]=key;
```

The running time is $O(\lg n)$ for inserting a single element in a heap or priority queue.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

In the remainder we deal with yet another application of priority queues, that of text compression.

Text S with characters from alphabet X. Convert it into a text C with characters from alphabet Y. In text compression we are given a text S of characters drawn from some alphabet X and we want to efficiently encode the characters of S into a text C that uses another alphabet Y. This leads to compression if the size of C is less than the size of S. Moreover to accomplish this we need to find the code of every character of S. Each such character is drawn from alphabet X and gets mapped into one or more characters of alphabet Y used in C.

Encoding: E from X to Y. In **data or text compression** the first step is to determine an encoding E (aka determining the **code**) of every character of alphabet X using characters from alphabet Y. These codes must be unique and uniquely identify the characters of X. In C the characters of X are stored through their codes that contain characters from alphabet Y.

Decoding: E' from Y to X. If we want to recover S from C we also need to be able to compute E' that maps the codes of characters in X drawn from Y into the original characters of X.

Data compression: from $S \stackrel{E}{\rightarrow} C$. In data compression we first compute $E : X \rightarrow Y$ and establish the codes of every character of S that belongs to X. Those codes use characters from alphabet Y. This is the **encoding** phase. We then use E to encode each character in S from alphabet X using its corresponding code drawn from Y. The result is text C originating from S. This is the **compression phase**.

Data decompression: from $C \stackrel{E'}{\rightarrow} S$. The way **decompression** is performed is by assigning alphabet letters from X to every code that uses letters from Y i.e. we need to invert function E i.e. to decode the characters of Y to those of X. We then recover S from C. This is **data decompression.** For this to be able to be done in a self-sustained function, C must include information about E, if E is invertible, or how to recover E', in addition to the converted/encoded text of S. Thus C contains the encoded text of S plus more; the more is to retrieve information about X and E or E'.

The process that converts S into C can be colloquially referred to as **encoding or compression** and the inverse process that retrieves S from C can be referred to as **decoding or decompression**. Thus colloquially one of the two distinct steps can be used to describe both!

Compression ratio. The compression ratio is defined as the ratio between the size of the original text to the size of the compressed text i.e. $|C|/|S|$; (the compressed text usually contains overhead related to the encoding of the characters of X.)

(c) Copyright A. Gerbessiotis.

4.0 Example.

```
Text S: ACTGAACA , Text size: 8B or 64 bit    X={ A, C, G, T } We will use  Y= { 0, 1 }
One possible encoding E:X-->Y                 Another possible encoding E: X-->Y
  E  A: 0                                     E A: 0
     C: 10   A bit aligned code                 C: 1      A bit aligned code
     G: 110  that is a prefix code              G: 10     that IS NOT a prefix code
     T: 111                                     T: 01      (0 is a prefix of 01, 1 of 10)
Compress S to C with E, C: 01011111000100    Compress S to C with E, C: 0101100010
Compressed text size:  14bit or 2B           Compressed text size:  10bit or 2B
Compr. ratio: (bit:64/14, byte: 8/2)         Compr. ratio: (bit:64/10 byte: 8/2)
Note: C does not store overhead about E, E' or how to retrieve A from C.
      GIVEN C only and E/ E' how can we retrieve S from C?
 S from C: straighforward                  S from C: is 0101100010 a TTGAAG? ACTGAACA?
```

4.1 Character Encoding: ASCII, UNICODE, UTF-8. The bit sequence assigned to a given character of an alphaber is called the **code** of that character. In ASCII English characters, digits and punctuation marks are assigned binary codes (from an alphabet of 0,1). Thus the code for character, i.e. digit, `0` in ASCII is 48 in decimal, or 30 (0x30) in hexadecimal, or **0110000** in binary: we usually store 7-bit ASCII codes in a byte that uses 8 bits. Thus digit 0 is denoted **00110000**. Likewise digit 1 is **00110001**. In UNICODE this character would be `0x0030` in hexadecimal i.e. two bytes are being used, one of which is zero and the other containing the ASCII value of character `0`. (For information only, a third encoding UTF-8 stores all possible 1,000,000+ UNICODE characters and thus also its ASCII subset by using 1-4 bytes. Thus ASCII characters are encoded using 1 byte in UTF-8.) Since n bits allow for 2^n distinct values to be represented, an alphabet such as alphabet A of the previous paragraphs with $\mid A \mid$ characters requires $\approx \lg |A|$ bits if a fixed number of bytes or bits are to be used to encode the characters of the alphabet.

4.1.1 Byte-aligned codes are code that respect byte boundaries. Thus ASCII, UNICODE, UTF-8 character codes are byte-aligned codes. In ASCII a character is assigned one byte, but in UNICODE at least two bytes, and UTF-8 1-4 bytes.

4.1.2 Bit-aligned codes are code that respect bit boundaries and not necessarily byte boundaries. If each character is equally likely to appear in a document (or string) one can use a byte-aligned code for those characters. In real-life however this is not the case. In English, certain characters (i.e. letters) are more likely to appear than others (e.g. e is more frequent than w or x). Thus it makes sense to assign fewer bits to more frequent characters and more bits to infrequent characters. In a **bit-aligned** code a variable number of bits is used to encode a character. It is thus possible that the same byte can be used to encode one or more or part of one or more characters. The separation between one word (code) and another one can occur at any bit position, not necessarily at multiples of 8 bits (byte). Several of the codes that we will be discussing in the remainder are bit-aligned codes.

(c) Copyright A. Gerbessiotis.

ALARM! ALARM! PAY ATTENTION! ALARM! ALARM! The symbol C is being used to denote a character C of text S i.e. a member of alphabet X. Yet the same symbol C is being used to denote the compressed text C. Beware of context, and do not get confused!

4.2. Codes : Prefix codes. Bit-aligned codes may cause problems if not properly chosen. For bit-aligned codes to be used effectively, it is imperative that no code is the **prefix** of another code. Such desirable codes are called **prefix codes** (and imply the prefix-free property): no code can be a prefix of another code!

In example 4.0 the second code used is not a prefix code. The code for A that is a 0 is a prefix of the code for T that is 01. Moreover, the code for C that is a 1 is a prefix of the code for G that is a 10. Absent of any other information, the compressed text `0101100010` may get decompressed as `TTGAAG` as the first 2-bit sequence `01` is decompressed as T, rather than a A followed by a C.

4.3. Compression: Lossless and lossy.

Data compression techniques are **lossless** or `noisy/lossy`. For a lossless technique, and compressed text C, we can fully recover the original uncompressed S from C during decompression. There are also **noisy or lossy** techniques where full-recovery of S from C is not possible; an **approximation** S' of S can only be retrieved. Such techniques can be used in image or voice compression. JPEG is a lossy image compression algorithm. GIF is a losless data compression algorithm based on the Lempel-Ziv-Welch (LZW) algorithm. PNG is also lossless (and for sometime was used as a replacement for GIF).

4.4. A lossless method that constructs prefix codes: Huffman's.

The primary method that will be presented in this subject for text compression is Huffman's algorithm (aka `Huffman's encoding aka Huffman's compression method aka Huffman's coding`) that yields what is known as Huffman codes. **Huffman codes are bit-aligned prefix codes**. They lead to lossless compression.

The text S, its alphabet X and the frequencies of the characters of X. For a given set of characters X and their associated frequencies Huffman's compression method determines a set of codes, known as Huffman codes. Huffman encoding thus requires that we know in advance the alphabet X and also the frequency of each character in text S.

(c) Copyright A. Gerbessiotis.

4.5. Potential space savings. Huffman codes result in space savings of 20% to 90% depending on the structure of text S that will be compressed. Nowadays it is not very often that anyone is using Huffman compresison (aka coding) to compress a text or strings of characters. Average savings observed is of 3 bits per character thus using 5bits or so for character encoding. This is not good enough any more. Lempel Ziv (LZ78) uses only 4.2 bits on the average, and LZ77 a bit less. The UNIX `compress` function uses approximately 3.31 bits and `gzip` close to 2.53. And `bzip2` uses 2.23 bits per character which implies close to a 4:1 compression. However instead of doing character compression/encoding using Huffman coding, we can do word compression/encoding, and Huffman can exhibit substantiallhy better performance to the level of 2.95 bits per word. The **important property** of Huffman/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 text uniquely. No 'separator' is needed to separate adjacent codes.

4.6. Algorithm Design Technique 4: The greedy principle. Huffman's algorithm for encoding and decoding a text is a **greedy algorithm** that during its course of execution makes a decision/choice that is the best possible decision at the time (that's the greedy principle in algorithm design).

4.7. Establishing a prefix code: Huffman's construction of a binary tree T. For a given text S and its alphabet X, a Huffman tree T is a binary tree. It is in fact a full tree with $n-1$ internal nodes, where n is equal to $|X|$. (And in practice $n \approx 256$.) The leaves are the characters of alphabet X (appearing in S). Every node has a frequency field associated with it; the frequency of a leaf is the frequency of the corresponding character of X in S. A code is a description of the path from the root to the specified character/leaf. The length of that path is the number of bits in the code for that character/leaf. Since T is a full tree, every internal node of the binary tree has a left child and a right child: the left branch is encoded as a 0 and the right branch as an 1. A "Huffman tree" is not necessarily unique.

4.8. Building the binary tree of prefix codes (aka Huffman tree T). For a text S we first determine X, its alphabet and for every $c \in X$, the frequency $f(c)$ of c in S. Then a binary tree T of prefix codes is being built using Huffman's construction (i.e Huffman's coding) utilizing the character frequencies. The frequency field of every node in T records the sum of the frequencies of its left and right children; this is also the sum of the frequenciess of the leaves in the subtree rooted at that node. The number of bits required to encode S for a given Huffman tree T is $B(T) = \sum_{c \in X} f(c)d(c)$, where $d(c)$ is the depth of character c in the tree (i.e. its distance from the root, which is the number of bits in its code), and $f(c)$ is the frequency of character c in the text. The Huffman tree is being built using the greedy principle bottom to top.

4.9 Initialization and Work. After we identify the distinct characters that form alphabet X, and determine the frequency of each such character c in X, we form individual one-node trees, one per character c and associate the character c and also frequency $f(c)$ with the node representing c. (This is done for non-zero frequency characters.)

The single node representing c is the root of its own tree. The algorithm works bottom-up by combining two available trees into a single one by creating a new root node for a new tree whose left child is the root of one of the two trees to be combined and the right child the other one. The frequency field of the new root is the sum of the frequencies of the roots of the two constituent trees.

There is no character representation associated with the subsequently formed root nodes. If we start with an alphabet X that has $n =\mid X \mid$ characters (aka trees) with non-zero frequencies, after $n - 1 =\mid X \mid -1$ such combining operations there is only one tree left whose frequency field is the text size (i.e the sum of frequencies of all characters in S). This is the Huffman tree!

4.10. Applying the greedy principle into combining 2 trees into one. The criterion for choosing the two trees to combine into one is the "greedy" principle: the two trees with the lowest frequency fields at their respective roots are chosen.

4.11 Use a priority queue to maintain tree frequency information. We use a priority queue to maintain the frequency field of the trees (in fact, their roots) that are being maintained. The priority queue is implemented using a binary heap, a MINHEAP. (The priority of a tree is the frequency of its root.)

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Encoding

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

```
   Encode (file S, compressedfile C)                      // Text is AACCAGTA
1. Read S, form X, and for each char c compute X[c].freq  // A:4 C:2 G:1 T:1
2. Tr= Huffman(X[]); // Build Huffman tree Tr ;
3. From Tr generate for every c of X,   X[c].code, the Huffman code of  c
4. EncodeFile(S,X,Tr,C) // Encoded file is C
5. // Somehow a representation of Tr is to be stored in C as well !
```

The following pseudocode shows the steps of building Huffman's tree.

```
    Huffman (X[ ])//Every character in X has  X[.].freq  X[.].char
0.   PriorityQueue Q ;  // a definition/declaration  of Q
1.   BuildMinHeap(Q,X);
2.   for(c=0;c<|X|;c++){
       // create single node trees
3.    Tr[c].char=X[c].char; Tr[c].freq=X[c].freq; Tr[c].left= Tr[c].right= NULL;
4.    // connect elements of Q with the tree nodes of Tr ; i.e. Q points to Tr elements
5.   }
6.   for(i=0;i<n-1;i++) {
7.     z=ALLOCATENODE();
8.     x=z.left  = Extract-Min(Q); // x would point to an element of Tr by step 4 above
9.     y=z.right = Extract-Min(Q); // y would point to an element of Tr by step 4 above
10.    z.freq = x.freq + y.freq;
11.    z.char = -1;
12.    InsertMinHeap(Q,z); // we Extracted twice from Q in 8-9; we insert once here
13.  }                     // net loss of one character/element of Q.
14.  return(ExtractMinHeap(Q)); // There is only one left in the Heap; the root of T!
```

The running time of Huffman is $O(n \lg n)$, where $n =\mid X \mid$. Note that n is the number of distinct characters of S not the size of S. Regular English texts might be Megabytes in size. Yet n is no more than 100 as English texts utilize a subset of the ASCII character set. Lines 0-4 require $O(\mid X \mid) = O(n)$ time. Line 5 (BuildHeap operation) requires $O(n)$ time and lines 6-14 require $O(\lg n)$ time per iteration for a total of $O(n \lg n)$.

(c) Copyright A. Gerbessiotis.

Decoding

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

```
    DecodeFile (compressedfile C, file S)
0.  Obtain Tr from C; B[]=codedtext(C); pos=0; // codedtext(C) is C minus the portion describing Tr
1.  pointer = root(Tr);
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,S);
9.     pointer=Tr;
10.   }
11. }
12. return(S);
```

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

In the remainder when we refer to a Huffman tree T we will not be using the **Tr** notation.
We shall call it just **T**.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

4.12 Theorem Huffman 1. Let X be an alphabet in which each character c has frequency $f[c]$. Let x and y be two characters of X with the lowest frequencies. Then there exists an optimal prefix code for X in which x and y have same length codewords that only differ in the last bit.

Proof. We start with an optimal but arbitrary prefix code and modify it to obtain another optimal prefix code where x and y are maximum depth sibling in the new tree (i.e. codewords have same length and differ in last/rightmost bit).

Let in the arbitrary optimal prefix code b and c are the siblings of maximum depth. Without loss of generality let $f[b] \leq f[c]$ and also, $f[x] \leq f[y]$. Since $f[x], f[y]$ are lowest leaf frequencies, then $f[b], f[c]$ must be such that $f[x] \leq f[b]$ and $f[y] \leq f[c]$.

We obtain tree T' by exchanging in T x with b and T'' by exchanging y with c in T'.

Then we have

$$\begin{aligned} B(T) - B(T') &= \sum_c f(c)d_T(c) - \sum_c f(c)d_{T'}(c) \\ &= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T'}(x) - f(b)d_{T'}(b) \\ &= f(x)d_T(x) + f(b)d_T(b) - f(x)d_T(b) - f(b)d_T(x) \\ &= (f(b) - f(x))(d_T(b) - d_T(x)) \\ &\geq 0. \end{aligned}$$

Note that since $f(x)$ is minimum frequency leaf $f(b) - f(x) \geq 0$. Since b is maximum depth leaf in T $d_T(b) \geq d_T(x)$.

We can similarly prove that $B(T') - B(T'') \geq 0$. Therefore $B(T'') \leq B(T)$. Since T is optimal $B(T) \leq B(T'')$. Therefore $B(T) = B(T'')$.

(c) Copyright A. Gerbessiotis. All rights reserved.

4.13 Theorem Huffman2. Let T be a full binary tree representing an optimal prefix code for X with frequencies $f[c]$. Consider two sibling leaves x and y in T and let z be their parent. If z is considered a character with $f[z] = f[x] + f[y]$, the tree $T' = T - \{x, y\}$ represents an optimal prefix code for $X' = X - \{x, y\} \cup \{z\}$.

Proof. For every c in A other than x, y we have $d_T(c) = d_{T'}(c)$. For $d_T(x) = d_T(y) = d_{T'}(z) + 1$, we have

$$f(x)d_T(x) + f(y)d_T(y) = (f(x) + f(y))(d_{T'}(z) + 1) = f(z)d_{T'}(z) + f(x) + f(y).$$

Then, $B(T) = B(T') + f(x) + f(y)$. If $B(T')$ is not optimal then there is an optimal tree O such that $B(O) < B(T')$, and its leaves are characters in X'. Then z appears as a leaf in O. If we add x, y in O we get a prefix code whose total cost is $B(O) + f(x) + f(y) < B(T) = B(T') + f(x) + f(y)$. This, however, contradicts the minimality of $B(T)$. Therefore such an O can not exist because T' is the optimal prefix code for X'.

4.14 Theorem Huffman-Algorithm Huffman produces an optimal prefix code.

Proof. A direct result of Theorems Huffman and Huffman2.

(c) Copyright A. Gerbessiotis. All rights reserved.

Appendix
An equation derivation

We already know the first two derivations: the first one has n terms, the second one has infinite term. In the latter one the right-hand side is to the limit $n \to \infty$. The third one is to be derived from the second one.

$$\begin{aligned}
\sum_{i=0}^{n-1} x^i &= 1 + x + \ldots + x^i + \ldots + x^{n-1} = \frac{x^n - 1}{x - 1} \qquad x \neq 1 \\
\sum_{i=0}^{\infty} x^i &= 1 + x + \ldots + x^i + \ldots = \frac{1}{1 - x} \qquad 0 < x < 1 \\
\sum_{i=0}^{\infty} i \cdot x^i &= 1 \cdot x^1 + 2 \cdot x^2 + \ldots + i \cdot x^i + \ldots = ? \qquad 0 < x < 1
\end{aligned}$$

Starting, from the second equation we work as follows; (a) take the first derivative of both sides, (b) multiply the result (both sides) by x. We get:

For any $x < 1$, we have.

$$\begin{aligned}
\sum_{i=0}^{\infty} x^i &= \frac{1}{1 - x} \\
1 + x + \ldots + x^i + \ldots &= \frac{1}{1 - x} \\
\left(1 + x + \ldots + x^i + \ldots\right)' &= \left(\frac{1}{1 - x}\right)' \\
0 + 1 \cdot x^0 + \ldots + i \cdot x^{i-1} + \ldots &= \frac{1}{(1 - x)^2} \\
0 \cdot x + 1 \cdot x^1 + \ldots + i \cdot x^i + \ldots &= \frac{x}{(1 - x)^2} \\
\sum_{i=0}^{\infty} i \cdot x^i &= \frac{x}{(1 - x)^2}
\end{aligned}$$

If we substitute $x = 1/2$ we obtain

$$\sum_{i=0}^{\infty} i/2^i = 2$$