Hash Tables and Huffman Coding

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 2012 ; distribution outside this group of students is NOT allowed.*

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

Introduction

Some applications require the implementation of a **dictionary data structure** with operations **Insert, Search and Delete**. If one implements a dictionary using a linked list or an unordered array, Search would require $\Theta(n)$ time. The introduction of **hash tables** allows faster dictionary operations ON THE AVERAGE. Though the worst case time for such operations are as bad as that of linked lists and arrays, under some reasonable assumptions hash tables implement implement operations like Search in **expected constant time** $O(1)$. Hash tables, because of their simplicity have many practical applications. For example, a hash table is used in compilers to implement a keyword look-up table and constitutes an integral part of languages that deal with manipulation of string structures (e.g. Perl).

Case 1. In **direct addressing of an array** we allocate an array of size equal **to the number of possible keys** and store a key in the corresponding table slot in $O(1)$ time. Let $U = \{0, \ldots u-1\}$ be a universe of key values. We allocate a table $T[0..u-1]$ and allocate key k to the k-th entry of the table. x below is a record and $key(x)$ contains the key of record x, and $other(x)$ contains other relevant information.

```
DAddress-Search (T, k)  DAddress-Insert(T,x)  DAdress-Delete(T,x)
 return(T(k))            T[key(x)]=x           T[key(x)]=NIL;
```

Case 2. If allocation of a table of size equal to $m = u$ is not possible, direct addressing is not an option as it wastes space. An alternative to direct addressing is **hashing**.

What is a hash table? A hash table is a generalization of an array.

A hash table as opposed to a direct addressing scheme would require space **proportional to the (maximum) number of keys that we plan to store in the hash table** NOT TO THE NUMBER OF POSSIBLE KEY VALUES. For example the number of alphabetic variables (26 letters) of length 10 is $2^{4 \cdot 10} < 26^{10} < 2^{5 \cdot 10}$, a huge number bigger than 2^{40} (bigger than a trillion itself) and smaller than 2^{50} (a quadrillion), compared to the maximum number of keywords in a very big program (say, one million keywords/variables). Allocation of space proportional to 26^{10} (i.e. direct addressing) is prohibitively expensive as opposed to allocating space for a hash table of size one million.

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

Instead of using the key value as index to the table, we first compute from the key value a hash index and then use this hash index as table index. In direct addressing, key k is stored in slot k of the table, i.e. in $T[k]$. In hashing, starting with key k, we first compute the hash index of k using a hash function h. The hash index of k is then $h(k)$. Then we access slot $h(k)$ of the (hash) table T to store k (or a more extensive record x whose key is k).

Let $U = \{0, \ldots u-1\}$ be a universe of key values. Instead of allocating a hash table $T[0..u-1]$ of size u we allocate a hash table of size m, where m is an estimate of the (maximum) number of keys that will be stored in the hash table. Since in general $m \ll u$, there may be two keys k_1 and k_2 that hash into the same slot, i.e. it may be possible (and likely) that $h(k_1) = h(k_2)$. This occurrence is called a **key collision**. As $u > m$, it is unrealistic not to expect collisions.

A minimization of collisions is a reasonable objective in designing a hash table and choosing a hash function.

Key collision resolution is, however, essential in dealing with hashing.

```
In hashing starting from  string "ABD" we first get its ordinal
number 732, and then use the hash function to hash 732 into
a slot of the hash table

  x    mapping   k=key(x)       h           h(k)
"ABD" ---------->  732   ---------------->   5   assuming h(732)=5
```

For the remainder we shall assume that all keys are not in the x form but in the k form (i.e. they are represented as decimal numbers).

n *will denote the number of keys to be hashed*

m *will denote the size of the hash table*

Collision resolution through chaining

One scheme in resolving collisions is by **chaining** In **chaining**, slot j stores a pointer to a linked list containing all keys k with hashed value $h(k) = j$.

The operations Insert, Search, Delete looks as follows. **Note that many time we ignore the distinction between a record x whose key is $k = key(X)$ and thus use as argument k the key directly, instead of x.**

```
ChainHashInsert(T,x)     ChainHashSearch(T,k)          ChainHashDelete(T,x)
   Insert record x at      Searchfor k in T(h(k))        Delete record x from
   head of T(h(key(x)))    if x found with key(x)=k             T(h(key(x)))
                             return(record x)
                           else
                             return(FAILED)
```

A **Hash-Insert** operation inserts an element in the head of the list $T[h(k)]$ provided that the key is not already in the list. A search operation can be successful or unsuccessful and is reduced into a linked-list search. Whether the linked list is a singly or a doubly linked list affects performance of ChainHashDelete. We shall assume that the linked list is a DOUBLY LINKED LIST. The worst-case running time bound for an insertion operation is $O(1)$ as keys are inserted at the head of the list. Deletion takes $O(1)$ worst case time for a doubly linked list.

As far as the search operation is concerned we shall prove that if n elements are stored in a hash table of size m and the collision resolution protocol is chaining, then a successful/unsuccessful search takes on the average time $\Theta(1 + a)$, where $a = n/m$ is the load factor under the **SIMPLE UNIFORM HASHING ASSUMPTION** which states that any key k is equally likely to hash into any of the m slots.

Let us assume that the hash table T contains m entries and it currently stores n keys. This means that the average number of keys per linked list (also called CHAIN) is $a = n/m$. a can be less than, equal to or greater than 1. The cost of computing $h(k)$ from k will be assumed to be $O(1)$. The worst case behavior of chaining is $\Theta(n)$ as all keys may get hashed on the same value. We are however interested in examining the average performance of hashing. The best we can hope for is $\Omega(1 + a)$.

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

Theorem. With collision resolution by chaining under simple uniform hashing, an **unsuccessful** search takes time $\Theta(1+a)$ on the average.

Proof. Any key k NOT IN THE HASH TABLE is equally likely to hash into any one of m slots. There are n elements and m slots i.e on the average n/m elements per slot chain. The result follows.

In a successful search the probability a chain is searched is proportional to its size. On the average half the chain will be searched before the key is found.

Theorem. With collision resolution by chaining under simple uniform hashing, a **successful** search takes time $\Theta(1+a)$ on the average.

Proof. When key k is inserted, its record x is inserted at the head of the list. Subsequent (in time) insertions of keys in the same list occur before x. Prior insertions of keys occurred after x, i.e. traversing the chain from head to just before x, we access keys that were inserted into the chain after key k (whose inserted record was x). Suppose k was the ith key and thus x was the i-th record inserted in the hash table. After k, $n-i$ keys were inserted in the table. As each one of these keys was equally likely to go into any of the m slots, on the average $(n-i)/m$ records will be inserted before x and all these records will be searched before x is located during a successful search, for a total number of records search of $((n-i)/m)+1$. Therefore for the i-th record x with key k of the hash table $((n-i)/m)+1$ records will be searched before it is found. We sum the number of records searched for every $i=1,\ldots,n$ and dividing the sum by n we get the average successful search time.

$$\frac{1}{n}\sum_{i=1}^{n}(1+(n-i)/m) = 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) = 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) = 1+\frac{(n-1)}{2m} = \Theta(1+a).$$

Question Why $\Theta(1+a)$ and not $\Theta(a)$?

In **open addressing** we address the case that $n \leq m$ and we store the keys in the hash table itself. A slot contains either a key or a flag that indicates that the slot is available. The Load factor a is thus at most 1 for open addressing.

```
  0  1  2  3  4  5  6  7  8  9 10
---------------------------------   h(k)=k%11
|  |  |  |  |  |  |  |  |  |  |  |   h(k,i)= (h(k) + i) % 11
|  |  |  |  |  | 5| 6|16|  |  |  |
---------------------------------   Figure 1.
```

In order to insert k we **repeatedly probe** the hash table until we can find an empty slot in which to put the key. Instead of probing slots in a predetermined order $0, 1, \ldots, m-1$, the **probing sequence** depends on k. We thus extend hash function $h(k)$, to include a probe index/number starting from 0

$$h : U \times \{0, \ldots m-1\} \rightarrow \{0, \ldots m-1\}$$

thus defining for every key k a probe sequence,

$$\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$$

that is required to be a permutation of $0, \ldots, m-1$, or there may be some slots that may be missed during the probing.

Thus for key k, we first probe $h(k,0)$, then if necessary $h(k,1)$, and so on. If $h(k,m-1)$ is also full, then we determine that the hash table is full and this causes an overflow.

Example. Let us assume that we have the very simple case depicted in Figure 1. The first key that needs to be hashed has value $k = 5$. Since $h(5) = 5$, for $i = 0$ we have $h(k,0) = 5$ and we probe slot 5. It is empty (no key) present. Thus we insert 5 in slot 5. Next comes key 6. Similarly it is inserted in slot 6. Next come key 16. As $h(16) = 5$, we first try $h(k,0) = 5$. Slot 5 is occupied by 5 and thus a **collision** occurs; the collision is due to the fact that 5,6 are such that $h(5) = h(16)$. We then try $h(k,1) = 6$ which is also occupied and a collision also occurs. In this case the cause of the collision is not that $h(16)$ is equal to $h(6)$: they are not the same. The reason is that $h(6,0) = h(16,1)$. The next possible slot is $h(16,2) = 7$, which is available and 16 is inserted there.

```
  Insert(T,k)
1. i=0;
2. do  j=h(k,i);
3.   if EMPTY(T[j]){   //  EMPTY(T(j)) :  (T(j)==NULL OR T(j)==DELETED)
4.       T[j]=k
5.       return j
     }
6.   else i=i+1;
7. while (i<m);
8. error("Hash overflow");

  Search(T,k)
1. i=0;
2. do j=h(k,i);
3.    if T[j]=k
4.        return j
5.    i=i+1;
6. while ((i<m) && (T[j] != NULL));
7. return (-1)
```

In a deletion a deleted slot for key x cannot be marked NULL. If it is marked NULL then, a subsequent search for key y that was inserted after x and with $key(x) = key(y)$ (i.e. the slot for x was searched and found full during the Insert operation for y) will not be found, because the NULL in x's slot will disrupt the search. To avoid such a problem a freed slot is marked DELETED.

In the analysis of open addressing we shall make the assumption of **uniform hashing that is, each key is equally likely to have any of the $m!$ permutations as its probing sequence**.

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

Linear Probing

Given an ordinary hash function h, linear probing uses h such that $h(k,i) = (h(k) + i) \bmod m$. The first position is $h(k)$. The problem with this method is **primary clustering**. Long runs of occupied slots are being built.

Say we have $m/2$ keys in T. In case 1, these keys occupy the first $m/2$ position of T (clustering). In case 2, they occupy the odd positions of the array. In case 1 average probing is $m/8$, in case 2 it is 1.5.

Quadratic Probing Given an ordinary hash function h', quadratic probing uses $h(k,i) = (h'(k) + c_0 + c_1 i + c_2 i^2) \bmod m$. The first position is $h(k,0) = h_1(k)$. If no c_0, c_1 and c_3 are specified (e.g. in homeworks or exams), then $c_1 = c_2 = 0$ and $c_3 = 1$. A milder form of clustering appears under quadratic probing which is called **secondary clustering**.

Double Hashing Given two hash function h_1, h_2, double hashing uses h such that $h(k,i) = (h_1(k) + ih_2(k)) \bmod m$. The probe sequence depends on two hash functions i.e. depends twice on k. In addition $h_2(k)$ must be relatively prime to m for the entire hash table to be searched. If their gcd is d instead only a $1/d$ fraction of the hash table is searched. This can be enforced by making m a power of 2 and force $h_2(k)$ to produce an odd number.

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

Analysis of unsuccessful Search: Theorem Given an open addressing hash table with load factor a the expected number of probes in an unsuccessful search is $1/(1-a)$ assuming uniform hashing.

Proof. The time for an unsuccessful search in a hash table of size m with n keys in it is the same as the time for inserting an $n+1$-st key in the hash table. Insert(n,m) gives the AVERAGE number of probes required for inserting a key in a table of m slots ALREADY containing n keys. Therefore

$$UnSearch(n,m) = Insert(n,m)$$

It is obvious that $Insert(0,m) = 1$, for any m. $Insert(n,m) = 1$ if the $n+1$-st key hashes into an empty slot. The probability of this happening is $(m-n)/m$. Otherwise, with probability n/m the key hashes to an occupied slot, so the number of probes is 1 plus the number of probes required to be inserted in the rest of the table. Therefore

$$Insert(n,m) = 1 \cdot \frac{m-n}{m} + \frac{n}{m}\left(1 + Insert(n-1,m-1)\right) = 1 + \frac{n}{m} Insert(n-1,m-1).$$

By induction on n and for $n < m$ we can prove that

$$Insert(n,m) = \frac{m+1}{m+1-n} \approx \frac{1}{1-a}$$

Analysis of Insert: Theorem Given an open addressing hash table with load factor a the expected number of probes for an insert is $1/(1-a)$ assuming uniform hashing.

Proof An element is inserted if there is space and it is preceded by an unsuccessful search and placement of key in the first empty slot found, and the time of an unsuccessful search is thus $1/(1-a)$.

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

Performance Analysis continued

Analysis of successful Search: Theorem Given an open addressing hash table with load factor a, the expected number of probes in a successful search is $1/a \ln(1/(1-a))$, assuming uniform hashing.

Proof A search for k has the same probing sequence as when k was inserted. When k was inserted, let it have been the $i+1$-st inserted key. Then insertion required on the average $1/(1-i/m) = m/(m-i)$ probes. Averaging over all keys we get (note $H_n = 1 + 1/2 + 1/3 + \ldots + 1/n \approx \ln(n)$).

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} = \frac{m}{n}(H_m - H_{m-n}) = \frac{1}{a}(H_m - H_{m-n}) = \frac{1}{a}(\ln(m) - \ln(m-n)) = \frac{1}{a}\ln(1/(1-a))$$

and the result follows. If $n/m = 0.5$ expected number is 1.387. If $n/m = 0.9$ then 2.55.

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

A **(binary) heap** is a data structure that has the following properties.

1. Each element of a heap has some priority attached to it.

2. A 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 about this element.

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

4a. For any element of a heap in position i 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 any element of a heap in position i 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**.

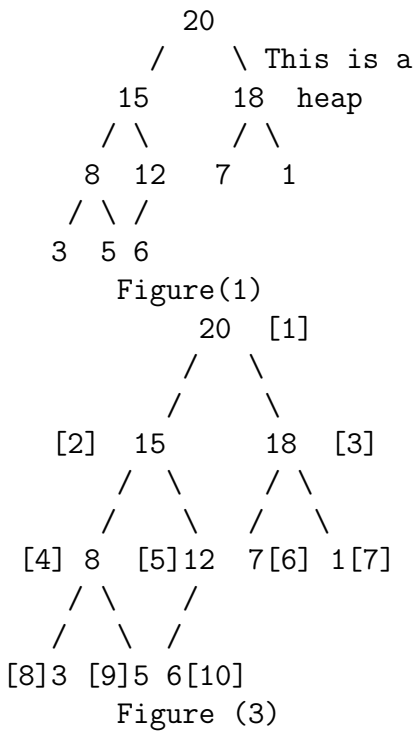
- A priority queue (PQ) S that implements efficiently the following operations: FindMax(S), Insert(S,x), ExtractMax(S). We can also have a PQ T with the following operations FindMin(T), Insert(T,x), ExtractMin(T) implemented similarly.
- The data structure called **heap** that allows a PQ to be implemented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

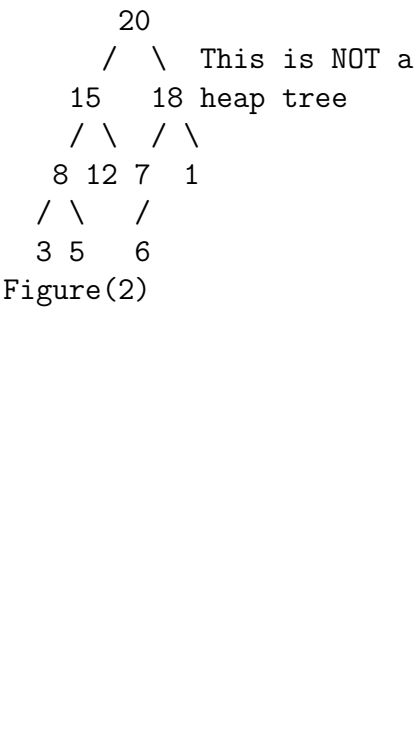
(c) Copyright A. Gerbessiotis. All rights reserved.

Representation

```
      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. All rights reserved.

Heapify (A,i) maintains the heap property at the node indexed i. The left and right subtree of that node MUST maintain the heap property; the tree rooted at i may not maintain the tree property. Depending on the heap we use we can define MaxHeapify or MinHeapify.

BuildMaxHeap(A) **/BuldMinHeap**(A). Given an unordered array A a heap is built in place. Such an operation is sometimes referred to as a **Heap Initialization** operation.

ExtractMaxHeap (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 **ExtracMinHeap** (A).

InsertMinHeap(A,**x**). An element x is inserted into the key.

(c) Copyright A. Gerbessiotis. CS 667-101 : Spring 2012 . All rights reserved.

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.

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

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 frequency 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 constituent 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.

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

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.   BuildMinHeap(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(ExtractMinHeap(Q));
```

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

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 (BuildHeap 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.

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

Theorem Huffman. Let C be an alphabet in which each character c has frequency $f[c]$. Let x and y be two characters of C with the lowest frequencies. Then there exists an optimal prefix code for C 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.

Theorem Huffman2 Let T be a full binary tree representing an optimal prefix code for C 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 $C' = C - \{x, y\} \cup \{z\}$.

Proof. For every c in C 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 C'. 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 C'.

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.

In real-life the open-addressing scheme is used more often than hashing by chaining. Instead of using the simple hash functions related to linear probing or quadratic probing, more elaborate hash functions are used. The hashing scheme (e.g. table organization) is also more involved. In any such scheme rehashing and reorganization might be necessary if there is an overflow condition. In such a case the size of the hash table gets increased (say by a factor of two or more), and the keys are deleted one by one from the previous (overflowed) hash table to be inserted into the new. Such a reorganization is slow and time consuming and should thus be avoided or done as infrequently as possible. Sometimes this reorganization can be avoided by the following method: when an overflow condition occurs in hashing by open-addressing in which the size m of the table was carefully chosen, any new keys are inserted into a balanced binary search tree data structure. This way one avoids reorganization by paying a penalty of using more space per key (pointers) and more time for search/insertion/deletion requests for the newly inserted keys.

In addition, keys are not plain numbers of fixed size (e.g. 32-bit integers) but arbitrarily long strings. Thus it makes then sense to store in the table key values that are pointers to memory in which the string itself resides. However pointers might be a very crude representation of a key value itself (and not very arbitrarily chosen). In such cases associated with a string s we might have not only a pointer p in memory that points to s itself, but also a key value k that can be some kind of a hash of s itself. (Thus p and k are different from each other.) A checksum function (such as a 32-bit CRC, or 128-bit MD5, or variable length SHA functions can serve as such) can then be used to generate k from s. A `docID` identifies a document better than a variable and usually long sting that a URL is. A `docID` can be obtained through a hash function such as MD5 but usually is user assigned (eg popular documents have short `docID`s.)

Google's DocIndex. For example, with respect to the Google original system described in the paper by Brin and Page (1997), one of the tables in DocIndex stores URLs and docIDs. One table related to DocIndex uses docID as the key and stores for each document with a given docID a variety of data associated with it such as the status of the document, a pointer to the Google repository holding the document (the "cached" copy of the Web document), the document checksum established by Google's internal checksum function (which can be one of the previously listed functions), statistics about the document, the URL itself and its title (if any).

In order to access this table, one needs to previously access a more compact table that contains for each document only a URL checksum and the docID for the document. This latter table is sorted based on checksum. Thus binary search on the checksum can retrieve the corresponding docID. Thus depending on the problem requirements, hashing can be avoided by using a binary-search method. This of course is possible if the table is already sorted; the application in question maintains that secondary table sorted by merging a (sorted) new instance into the previous (also sorted) instance of the table.

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

Case study: Three organizations for `word` or `wordIDs`

Case T1. Suppose we would like to maintain a table of n words using a hash table. Collision resolution by open addressing will be used with a hash-function that might look like the one used for quadratic probing earlier in the introduction of this method. For open addressing to work well, the load factor need to be kept low, that is $n/m < 1/2$, i.e. the table should be at least twice as large as the number of words that will be maintained in it. In previous examples the keys had numeric values that could fit into a slot of the table. However if words are to be stored inside a slot, it should be wide enough to accommodate the longest expected word. One can see where this leads in the case of chemical compounds, where words can exceed 32 characters and thus ones might need $> 64n$ bytes for the table (do not forget $m > 2n$ and each slot must thus be at least 32 bytes)!

Case T2a. This scheme looks like **T1** except that the table maintains only some-form-of-a-pointer to strings that represent `word`s. The bitlength of a `pointer` can be 32-bits. However if an `index` to a table is used we can get away with few extra bit and can thus use 24bits, or 16bits or less (application specific). Where do we store the words? In an array of characters where words are separated by NULL characters i.e. \0. For n words of maximum length 32 the previous scheme T1 would need $64n$ bytes (since $m = 2n$ and each slot 32 bytes wide). This scheme would use $w + n$ to store the strings plus the size of table of pointers, where w is the sum of the lengths of the words maintained, and n is the number of NULL characters used as separators (in fact $n - 1$). Since the average length of a word is roughly 7 characters then $w \approx 7n$ and thus this scheme would store the strings in roughly $8n$ bytes. For the table of pointers we have $m = 2n$ and even if the pointer are 4-bytes wide (32-bits) this would lead to a $8n$ bytes for a total of $16n$. Savings of over 75% have been achived ($16n$ vs $64n$).

Case T2b. In this scheme we maintain a `wordID` in addition to the pointer. If we have k word we need roughly $\lg k + 1$ bits to represent a `wordID` in the range 0 to $k - 1$. Thus for $m = 2n$ we need $2n(\lg k + 1)$ bits for `wordID`s. Definitely for $k < 2^{32}$ we would need no more than 4 Bytes per wordID. Thus `wordID`s can consume $8n$ more bytes. If the previous scheme used $16n$ bytes of space this one uses $24n$ still better than $64n$!

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

Size of a lexicon

One application of a hash table is to maintain a vocabulary (or lexicon). The lexicon (vocabulary) may be implemented by a more elaborate version of an open-addressing scheme. The hash table itself (similar to the table used for open-addressing) might be a hash table of pointers. The strings themselves are stored into a separate contiguous memory separated by NULL (`'\0'`) characters. For efficiency purposes vocabularies used in say Web-searching are memory resident.

Problem. Say we have a PC with 256 MB (megabytes) of RAM. We want to build a vocabulary of English words. How many words can it accommodate? How will it be organized?

Answer. The amount of main memory available is a hint that a memory resident lexicon shall be used. We could use a hash table of only pointers and store the strings separately NULL separated. Say the number of words allowed by this organization is n. The average length of an English word (before stopwords and stemming are applied) is around 6 to 7. Including the extra NULL character that will be used as a string/word terminator, we get a rough size of $8n$ bytes for simply storing into consecutive memory locations the strings. This way an array of size $8n$ will be used to store all the strings; we do not need to use in array of variable length strings! (This reminds scheme **T2a**.)

A pointer is 4 bytes, thus a hash table of pointers of length n would need $4n$ bytes. But then the table will be full and hash-table performance will be suboptimal. If the table is made $m = 2n$ long, hash table performance will improve. Such a table of pointers uses $4m = 4 \cdot 2n = 8n$ bytes. In total the amount of memory used is roughly $8n + 8n = 16n$. Thus $16n = 256MB$ and solving for n we get that this scheme can accommodate approximately $16,000,000$ words. Of course in that memory one needs to store the operating system processes, and also the hash table related code for hash table maintenance and thus vocabulary size would be even lower.

One could store along with each pointer, a `wordID` i.e. a hash (or checksum) of the individual word (similarly to a docID). (This become **T2b** then.) Given that the previous estimation can support up to 16,000,000 words, we conclude that we do not need more than 24-bits for a wordID since $\lg 16,000,000 \approx 24$. This is roughly 3 bytes per word. Thus for a hash table of length $m = 2n$ we need 4 bytes for a pointer and 3 bytes for a wordID or a total of $2n \cdot (3 + 4) = 14n$ bytes an increase by $6n$ bytes over the previous $8n$. Adding to that $8n$ for the strings, we get $22n$. Equating this to 256MB we conclude that this scheme can accommodate a vocabulary of slightly smaller size (10-12 million).

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