16.3 Huffman codes

- Huffman codes compress data very effectively
  - savings of 20% to 90% are typical, depending on the characteristics of the data being compressed
- We consider the data to be a sequence of characters
- Huffman's greedy algorithm uses a table giving how often each character occurs (i.e., its frequency) to build up an optimal way of representing each character as a binary string

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- We have a **100,000**-character data file that we wish to store compactly
- We observe that the characters in the file occur with the frequencies given in the table above
- That is, only 6 different characters appear, and the character a occurs **45,000** times
- Here, we consider the problem of designing a ***binary character code*** (or ***code*** for short) in which each character is represented by a unique binary string, which we call a ***codeword***

- Using a ***fixed-length code***, requires 3 bits to represent 6 characters:

  $a = \mathbf{000}, b = \mathbf{001}, \ldots, f = \mathbf{101}$

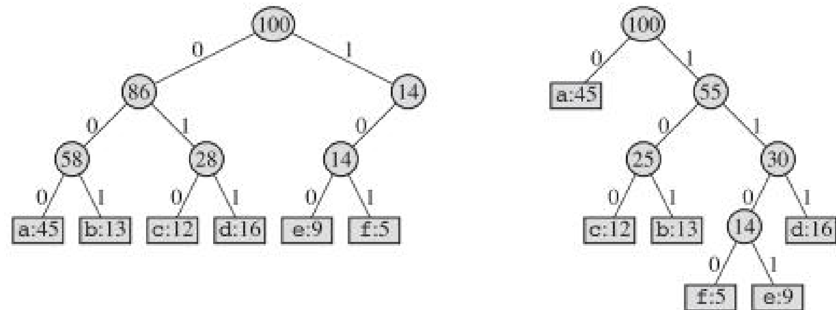
- We now need **300,000** bits to code the entire file
- A ***variable-length code*** gives frequent characters short codewords and infrequent characters long codewords
- Here the 1-bit string **0** represents a, and the 4-bit string **1100** represents f
- This code requires $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000 = 224{,}000$ bits (savings **25%**)

Prefix codes

- We consider only codes in which no codeword is also a prefix of some other codeword
- A prefix code can always achieve the optimal data compression among any character code, and so we can restrict our attention to prefix codes
- Encoding is always simple for any binary character code; we just concatenate the codewords representing each character of the file
- E.g., with the variable-length prefix code, we code the 3-character file abc as $\mathbf{0} \cdot \mathbf{101} \cdot \mathbf{100} = \mathbf{0101100}$

- Prefix codes simplify decoding
  - No codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous
- We can simply identify the initial codeword, translate it back to the original character, and repeat the decoding process on the remainder of the encoded file
- In our example, the string **001011101** parses uniquely as **0 · 0 · 101 · 1101**, which decodes to $aabe$

- The decoding process needs a convenient representation for the prefix code so that we can easily pick off the initial codeword
- A binary tree whose leaves are the given characters provides one such representation
- We interpret the binary codeword for a character as the simple path from the root to that character, where 0 means “go to the left child” and 1 means “go to the right child”
- Note that the trees are not BSTs — the leaves need not appear in sorted order and internal nodes do not contain character keys



The trees corresponding to the fixed-length code a **= 000, … ,** f **= 101** and the optimal prefix code a **= 0,** b **= 101, … ,** f **= 1100**

- An optimal code for a file is always represented by a full binary tree, in which every nonleaf node has two children
- The fixed-length code in our example is not optimal since its tree is not a full binary tree: it contains codewords beginning **10 …**, but none beginning **11 …**
- Since we can now restrict our attention to full binary trees, we can say that if C is the alphabet from which the characters are drawn and
  - all character frequencies are positive, then
  - the tree for an optimal prefix code has exactly $|C|$ leaves, one for each letter of the alphabet, and
  - exactly $|C| - 1$ internal nodes

- Given a tree T corresponding to a prefix code, we can easily compute the number of bits required to encode a file
- For each character c in the alphabet C, let the attribute $c.freq$ denote the frequency of c and let $d_T(c)$ denote the depth of c's leaf
- $d_T(c)$ is also the length of the codeword for c
- Number of bits required to encode a file is thus

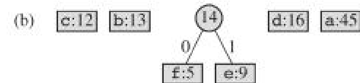
$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$$

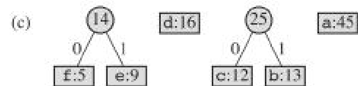
which we define as the cost of the tree T

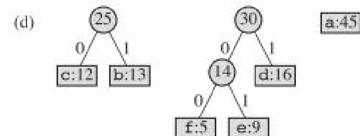
Constructing a Huffman code

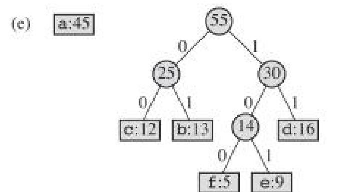
- Let C be a set of n characters and each character $c \in C$ be an object with an attribute $c.freq$
- The algorithm builds the tree T corresponding to the optimal code bottom-up
- It begins with $|C|$ leaves and performs $|C| - 1$ "merging" operations to create the final tree
- We use a min-priority queue Q, keyed on $freq$, to identify the two least-frequent objects to merge
- The result is a new object whose frequency is the sum of the frequencies of the two objects

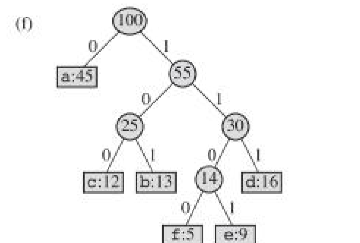
HUFFMAN(C)
1. $n \leftarrow |C|$
2. $Q \leftarrow C$
3. **for** $i \leftarrow 1$ **to** $n - 1$
4. allocate a new node z
5. $z.left \leftarrow x \leftarrow$ EXTRACT-MIN(Q)
6. $z.right \leftarrow y \leftarrow$ EXTRACT-MIN(Q)
7. $z.freq \leftarrow x.freq + y.freq$
8. INSERT(Q, z)
9. **return** EXTRACT-MIN(Q) // return the root of the tree











- To analyze the running time of HUFFMAN, let Q be implemented as a binary min-heap
- For a set C of n characters, we can initialize Q (line 2) in $O(n)$ time using the BUILD-MIN-HEAP
- The **for** loop executes exactly $n - 1$ times, and since each heap operation requires time $O(\lg n)$, the loop contributes $O(n \lg n)$, to the running time
- Thus, the total running time of HUFFMAN on a set of n characters is $O(n \lg n)$
- We can reduce the running time to $O(n \lg \lg n)$ by replacing the binary min-heap with a van Emde Boas tree

Correctness of Huffman's algorithm

- We show that the problem of determining an optimal prefix code exhibits the greedy-choice and optimal-substructure properties

Lemma 16.2 *Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.*

Lemma 16.3

Let C, $c.freq$, x, and y be as in Lemma 16.2.

Let $C' = C - \{x, y\} \cup \{z\}$. Define $freq$ for C' as for C, except that $z.freq = x.freq + y.freq$.

Let T' be any tree representing an optimal prefix code for the alphabet C'. Then the tree T, obtained from T' by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for C.

Theorem 16.4 *Procedure* HUFFMAN *produces an optimal prefix code.*

17 Amortized Analysis

- We average the time required to perform a sequence of data-structure operations over all the operations performed
- Thus, we can show that the average cost of an operation is small, even if a single operation within the sequence might be expensive
- Amortized analysis differs from average-case analysis in that probability is not involved
  - Amortized analysis guarantees the average performance of each operation in the worst case

17.1 Aggregate analysis

- We show that for all n, a sequence of n operations takes worst-case time $T(n)$ in total
- In the worst case, average cost, or ***amortized*** cost, per operation is therefore $T(n)/n$
- This amortized cost applies to each operation, even when there are several types of operations in the sequence
- The other two methods we shall study, may assign different amortized costs to different types of operations

Stack operations

- The fundamental stack operations PUSH(S, x) and POP(S) each takes $O(1)$ time
- Let's consider the cost of each operation to be 1
- The total cost of a sequence of n PUSH and POP operations is therefore n, and the actual running time for n operations is therefore $\Theta(n)$
- Now we add the stack operation MULTIPOP(S, k), which removes the k top objects of stack S, popping the entire stack if the stack contains fewer than k objects

- The operation STACK-EMPTY returns TRUE if there are no objects currently on the stack, and FALSE otherwise

MULTIPOP(S, k)
1. **while** not STACK-EMPTY(S) and $k > 0$
2. POP(S)
3. $k \leftarrow k - 1$

- The total cost of MULTIPOP is $\min(s, k)$, and the actual running time is a linear function of this

- Let us analyze a sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack
- The worst-case cost of a MULTIPOP operation is $O(n)$, since the stack size is at most n
- The worst-case time of any stack operation is $O(n)$, and hence a sequence of n operations costs $O(n^2)$
- This analysis is correct, but the $O(n^2)$ result is not tight
- Using aggregate analysis, we can obtain a better upper bound that considers the entire sequence of n operations

- We can pop each object from the stack at most once for each time we have pushed it onto the stack
- The number of times that POP can be called on a nonempty stack, including calls within MULTIPOP, is at most the number of PUSH operations, which is at most n
- Any sequence of n PUSH, POP, and MULTIPOP operations takes a total of $O(n)$ time
- The average cost of an operation $O(n)/n = O(1)$

Incrementing a binary counter

- Consider the problem of implementing a k-bit binary counter that counts upward from 0
- We use an array $A[0..k-1]$ of bits, where $A.length = k$, as the counter
- A binary number x that is stored in the counter has its lowest-order bit in $A[0]$ and its highest-order bit in $A[k-1]$, so that

$$x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$$

- Initially, $x = 0$: $A[i] = 0$ **for** $i = 0,1,\ldots,k-1$
- To add 1 (modulo 2^k) to the value in the counter, we use the following procedure

INCREMENT(A)

1. $i \leftarrow 0$
2. **while** $i < A.length$ and $A[i] = 1$
3. $\quad A[i] \leftarrow 0$
4. $\quad i \leftarrow i + 1$
5. **if** $i < A.length$
6. $\quad A[i] \leftarrow 1$

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

- At the start of each iteration of the **while** loop (lines 2–4), we wish to add a 1 into position i
- If $A[i] = 1$, then adding 1 flips the bit to 0 in position i and yields a carry of 1, to be added into position $i + 1$ on the next iteration of the loop
- Otherwise, the loop ends, and then, if $i < k$, we know that $A[i] = 0$, so that line 6 adds a 1 into position i, flipping the 0 to a 1
- The cost of each INCREMENT operation is linear in the number of bits flipped

- A cursory analysis yields a bound that is correct but not tight
- Single execution of INCREMENT takes time $\Theta(k)$ in the worst case, when array A contains all 1s
- Thus, a sequence of n INCREMENT operations on an initially zero counter takes time $O(nk)$
- Tighten the analysis to yield a worst-case cost of $O(n)$ by observing that not all bits flip each time INCREMENT is called
- $A[0]$ does flip each time INCREMENT is called

- The next bit up, $A[1]$, flips only every other time
  - a sequence of n INCREMENT operations on zero counter causes $A[1]$ to flip $\lfloor n/2 \rfloor$ times
- Similarly, bit $A[2]$ flips only every fourth time, or $\lfloor n/4 \rfloor$ times in a sequence of n INCREMENT operations
- In general, bit $A[i]$ flips $\lfloor n/2^i \rfloor$ times in a sequence of n INCREMENT operations on an initially zero counter
- For $i \geq k$, bit $A[i]$ does not exist, and so it cannot flip

- The total number of flips in the sequence is thus

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

  by infinite decreasing geometric series $\sum_{k=0}^{\infty} x^k = 1/(1-x)$, where $|x| < 1$
- Worst-case time for a sequence of n INCREMENT operations on an initially zero counter is therefore $O(n)$
- The average cost of each operation, and therefore the amortized cost per operation, is $O(n)/n = O(1)$

17.2 The accounting method

- Assign differing charges to different operations, some charged more/less than they actually cost
- When an operation's ***amortized cost*** exceeds its actual cost, we assign the difference to specific objects in the data structure as credit
- Credit can help pay for later operations
- We can view the amortized cost of an operation as being split between its actual cost and credit that is either deposited or used up
- Amortized cost of operations may differ

- We want to show that in the worst case the average cost per operation is small by analyzing with amortized costs
  - We must ensure that the total amortized cost of a sequence of operations provides an upper bound on the total actual cost of the sequence
  - Moreover, this relationship must hold for all sequences of operations
- Let c_i be the actual cost of the ith operation and $\hat{c}_i$ its amortized cost, require for all n-sequences

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

- Total credit stored in the data structure is the difference between the total amortized cost and the total actual cost, $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i$
- Total credit associated with the data structure must be nonnegative at all times
- If we ever were to allow the total credit to become negative, then the total amortized cost would not be an upper bound on the total actual cost
- We must take care that the total credit in the data structure never becomes negative

Stack operations

- Recall the actual costs of operations; PUSH: **1**, POP: **1**, and MULTIPOP $\mathbf{min}(k, s)$, where k is the argument of MULTIPOP and s is the stack size
- Let us assign the following amortized costs: PUSH: **2**, POP: **0**, and MULTIPOP **0**
- The amortized cost of MULTIPOP is a constant, whereas the actual cost is variable
- In general, the amortized costs of the operations under consideration may differ from each other, and they may even differ asymptotically

- We can pay for any sequence of stack operations by charging the amortized costs
- We start with an empty stack
- When we push on the stack, we use 1 unit of cost to pay the actual cost of the push and are left with a credit of 1
- The unit stored serves as prepayment for the cost of popping it from the stack
- When we execute a POP operation, we charge the operation nothing and pay its actual cost using the credit stored in the stack

- We can also charge MULTIPOPS nothing
- To pop the 1st element, we take 1 unit of cost from the credit and use it to pay the actual cost of a POP operation
- To pop a 2nd element, we again have an unit of cost in the credit to pay for the POP operation,…
- Thus, we have always charged enough up front to pay for MULTIPOP operations
- For any sequence of n operations, the total amortized cost is an upper bound on actual cost
- Since the total amortized cost is $O(n)$, so is the total actual cost

Incrementing a binary counter

- The running time is proportional to the number of bits flipped, which we use as our cost
- Let us charge an amortized cost of 2 units to set a bit to 1
- We use 1 unit to pay for the setting of the bit, and we place the other unit on the bit as credit to be used later when we flip the bit back to 0
- At any point in time, every 1 in the counter has a unit of credit on it, and thus we can charge nothing to reset a bit to 0

The amortized cost of INCREMENT:

- The cost of resetting the bits within the **while** loop is paid for by the units on the bits that are reset
- The INCREMENT procedure sets at most one bit (line 6) and therefore the amortized cost of an INCREMENT operation is at most 2 units
- The number of 1s in the counter never becomes negative, and thus the amount of credit stays nonnegative at all times
- For n INCREMENT operations, the total amortized cost is $O(n)$, which bounds the total actual cost

17.3 The potential method

- We represent the prepaid work as "potential energy," or just "potential," which can be released to pay for future operations
- Associate the potential with the data structure as a whole rather than with specific objects
- We will perform n operations, starting with an initial data structure D_0
- Let c_i be the actual cost of the ith operation and D_i the data structure that results after applying the ith operation to data structure D_{i-1}

- ***Potential function*** Φ maps data structure D_i to a real number $\Phi(D_i)$, the potential of D_i
- The amortized cost $\hat{c}_i$ of the ith operation with respect to potential function Φ is defined by

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

- The actual cost plus the change in potential
- The total amortized cost of the n operations is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

- 2nd equality follows because the terms telescope

- If we can define a potential function Φ so that $\Phi(D_n) \geq \Phi(D_0)$ then the total amortized cost $\sum_{i=1}^{n} \hat{c}_i$ gives an upper bound on the total actual cost $\sum_{i=1}^{n} c_i$
- In practice, we do not always know how many operations might be performed
- Therefore, if we require that $\Phi(D_i) \geq \Phi(D_0)$ for all i, then we guarantee, as in the accounting method, that we pay in advance
- We usually just define $\Phi(D_0)$ to be 0 and then show that $\Phi(D_i) \geq \mathbf{0}$ for all i

- If the potential difference $\Phi(D_i) - \Phi(D_{i-1})$ of the ith operation is …
  - *positive*, then the amortized cost $\hat{c}_i$ represents an overcharge to the ith operation, and the potential of the data structure increases
  - *negative*, then the amortized cost represents an undercharge to the ith operation, and the decrease in the potential pays for the actual cost of the operation
- Different potential functions may yield different amortized costs yet still be upper bounds on the actual costs

Stack operations

- We define the potential function Φ on a stack to be the number of objects in the stack
- For the empty stack D_0, we have $\Phi(D_0) = 0$
- Since the number of objects in the stack is never negative, the stack D_i that results after the ith operation has nonnegative potential, and thus
$$\Phi(D_i) \geq 0 = \Phi(D_0)$$
- The total amortized cost of n operations with respect to Φ therefore represents an upper bound on the actual cost

- If the ith operation on a stack containing s objects is a PUSH operation, then the potential difference is
$$\Phi(D_i) - \Phi(D_{i-1}) = (s+1) - s = 1$$
- The amortized cost of this PUSH operation is
$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2$$
- Suppose that the ith operation on the stack is MULTIPOP(S, k), which causes $k' = \min(k, s)$ objects to be popped off the stack
- The actual cost of the operation is k', and the potential difference is
$$\Phi(D_i) - \Phi(D_{i-1}) = -k'$$

- Thus, the amortized cost of the MULTIPOP operation is
  $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0$
- Similarly, the amortized cost of an ordinary POP operation is 0
- The amortized cost of each of the three operations is $O(1)$, and thus the total amortized cost of a sequence of n operations is $O(n)$
- Since $\Phi(D_i) \geq \Phi(D_0)$, the total amortized cost of n operations is an upper bound on the total actual cost
- The worst-case cost of n operations is $O(n)$

Incrementing a binary counter

- We define the potential of the counter after the ith INCREMENT to be b_i, the number of 1s in the counter after the ith operation
- Suppose that the ith INCREMENT operation resets t_i bits
- The actual cost of the operation is therefore at most $t_i + 1$, since in addition to resetting t_i bits, it sets at most one bit to 1
- If $b_i = 0$, then the ith operation resets all k bits

- In this situation $b_{i-1} = t_i = k$
- If $b_i > 0$, then $b_i = b_{i-1} - t_i + 1$
- In either case, $b_i \leq b_{i-1} - t_i + 1$, and the potential difference is
  $\Phi(D_i) - \Phi(D_{i-1}) \leq (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i$
- The amortized cost is therefore
  $$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$
  $$\leq (t_i + 1) + (1 - t_i) = 2$$
- If the counter starts at zero, then $\Phi(D_0) = 0$
- $\Phi(D_i) \geq 0$ for all i, the total amortized cost of n INCREMENTs is an upper bound on the total actual cost, and so the worst-case cost is $O(n)$

- The potential method gives us a way to analyze the counter even when it does not start at zero
- The counter starts with b_0 1s, and after n INCREMENTs it has b_n 1s, where $0 \leq b_0, b_n \leq k$
- Rewrite total actual cost in terms of amortized cost as
  $$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \hat{c}_i - \Phi(D_n) + \Phi(D_0)$$
- We have $\hat{c}_i \leq 2$ for all $1 \leq i \leq n$

- Since $\Phi(D_0) = b_0$ and $\Phi(D_n) = b_n$, the total actual cost of n INCREMENTs is $\leq 2n - b_n + b_0$
- Note that since $b_0 \leq k$, as long as $k = O(n)$, the total actual cost is $O(n)$
- I.e., if we execute at least $n = \Omega(k)$ INCREMENT operations, the total actual cost is $O(n)$, no matter what initial value the counter contains