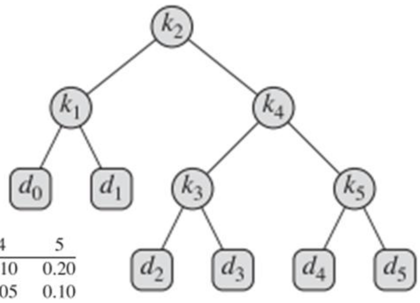
15.5 Optimal binary search trees

- We are designing a program to translate text
- Perform lookup operations by building a BST with n words as keys and their equivalents as satellite data
- We can ensure an $O(\lg n)$ search time per occurrence by using a RBT or any other balanced BST
- A frequently used word may appear far from the root while a rarely used word appears near the root
- We want frequent words to be placed nearer the root
- How do we organize a BST so as to minimize the number of nodes visited in all searches, given that we know how often each word occurs?

- What we need is an ***optimal binary search tree***
- Formally, given a sequence $K = \langle k_1, k_2, \dots, k_n \rangle$ of n distinct sorted keys ($k_1 < k_2 < \cdots < k_n$), we wish to build a BST from these keys
- For each key k_i, we have a probability p_i that a search will be for k_i
- Some searches may be for values not in K, so we also have $n + 1$ "dummy keys" $d_0, d_1, \dots, d_n$ representing values not in K
- In particular, d_0 represents all values less than k_1, d_n represents all values greater than k_n



i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

- For $i = 1, 2, \dots, n-1$, the dummy key d_i represents all values between k_i and k_{i+1}
- For each dummy key d_i, we have a probability q_i that a search will correspond to d_i

- Each key k_i is an internal node, and each dummy key d_i is a leaf
- Every search is either successful (finds a key k_i) or unsuccessful (finds a dummy key d_i), and so we have

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

- Because we have probabilities of searches for each key and each dummy key, we can determine the expected cost of a search in a given BST T

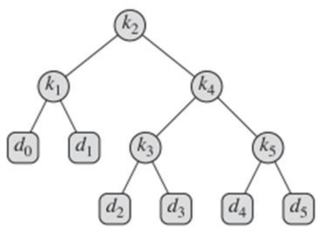
- Let us assume that the actual cost of a search equals the number of nodes examined, i.e., the depth of the node found by the search in $T + 1$
- Then the expected cost of a search in T, $\mathrm{E}[\text{search cost in } T] =$

$$\sum_{i=1}^{n} (\mathrm{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\mathrm{depth}_T(d_i) + 1) \cdot q_i$$

$$= 1 + \sum_{i=1}^{n} \mathrm{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \mathrm{depth}_T(d_i) \cdot q_i \,,$$

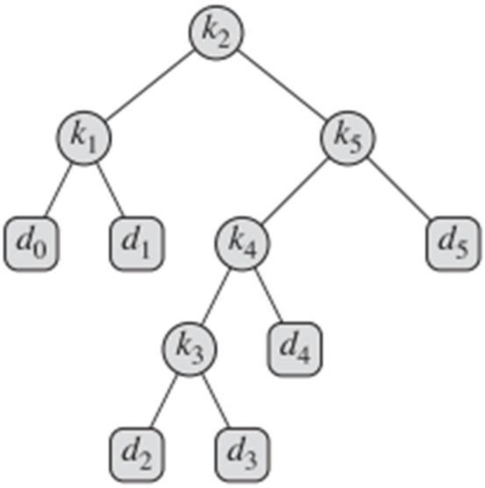
- where depth_T denotes a node's depth in tree T

Node	Depth	Probability	Contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80



i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

- For a given set of probabilities, we wish to construct a BST whose expected search cost is smallest
- We call such a tree an ***optimal binary search tree***
- An optimal BST for the probabilities given has expected cost 2.75
- An optimal BST is not necessarily a tree whose overall height is smallest
- Nor can we necessarily construct an optimal BST by always putting the key with the greatest probability at the root
- The lowest expected cost of any BST with k_5 at the root is 2.85



Step 1: The structure of an optimal BST

- Consider any subtree of a BST
- It must contain keys in a contiguous range $k_i, \dots, k_j$, for some $1 \le i \le j \le n$
- In addition, a subtree that contains keys $k_i, \dots, k_j$ must also have as its leaves the dummy keys $d_{i-1}, \dots, d_j$
- If an optimal BST T has a subtree T' containing keys $k_i, \dots, k_j$, then this subtree T' must be optimal as well for the subproblem with keys $k_i, \dots, k_j$ and dummy keys $d_{i-1}, \dots, d_j$

- Given keys $k_i, \dots, k_j$, one of them, say k_r, is the root of an optimal subtree containing these keys
- The left subtree of the root k_r contains the keys $k_i, \dots, k_{r-1}$ (and dummy keys $d_{i-1}, \dots, d_{r-1}$)
- The right subtree contains the keys $k_{r+1}, \dots, k_j$ (and dummy keys $d_r, \dots, d_j$)
- As long as we
  - examine all candidate roots k_r, where $i \le r \le j$,
  - and determine all optimal BSTs containing $k_i, \dots, k_{r-1}$ and those containing $k_{r+1}, \dots, k_j$,

  we are guaranteed to find an optimal BST

- Suppose that in a subtree with keys $k_i, \dots, k_j$, we select k_i as the root
- k_i's left subtree contains the keys $k_i, \dots, k_{i-1}$
- Interpret this sequence as containing no keys
- Subtrees, however, also contain dummy keys
- Adopt the convention that a subtree containing keys $k_i, \dots, k_{i-1}$ has no actual keys but does contain the single dummy key d_{i-1}
- Symmetrically, if we select k_j as the root, then k_j's right subtree contains no actual keys, but it does contain the dummy key d_j

Step 2: A recursive solution

- We pick our subproblem domain as finding an optimal BST containing the keys $k_i, \dots, k_j$, where $i \geq 1$, $j \leq n$, and $j \geq i - 1$
- Let us define $e[i, j]$ as the expected cost of searching an optimal BST containing the keys $k_i, \dots, k_j$
- Ultimately, we wish to compute $e[1, n]$
- The easy case occurs when $j = i - 1$
- Then we have just the dummy key d_{i-1}
- The expected search cost is $e[i, i-1] = q_{i-1}$

- When $j > i$, we need to select a root k_r from among $k_i, \dots, k_j$ and make an optimal BST with keys $k_i, \dots, k_{r-1}$ as its left subtree and an optimal BST with keys $k_{r+1}, \dots, k_j$ as its right subtree
- What happens to the expected search cost of a subtree when it becomes a subtree of a node?
  - Depth of each node increases by 1
  - Expected search cost of this subtree increases by the sum of all the probabilities in it
- For a subtree with keys $k_i, \dots, k_j$, let us denote this sum of probabilities as

$$w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$$

- Thus, if k_r is the root of an optimal subtree containing keys $k_i, \dots, k_j$, we have

$$e[i,j] = p_r + \big(e[i,r-1] + w(i,r-1)\big) + \big(e[r+1,j] + w(r+1,j)\big)$$

- Noting that

$$w(i,j) = w(i,r-1) + p_r + w(r+1,j)$$

we rewrite

$$e[i,j] = e[i,r-1] + e[r+1,j] + w(i,j)$$

- We choose the root k_r that gives the lowest expected search cost: $e[i,j] =$

$$\begin{cases} q_{i-1} & \text{if } j = i-1 \\ \min_{i \le r \le j} e[i,j] = e[i,r-1] + e[r+1,j] + w(i,j) & \text{if } i \le j \end{cases}$$

- The $e[i,j]$ values give the expected search costs in optimal BSTs
- To help us keep track of the structure of optimal BSTs, we define $\mathit{root}[i,j]$, for $1 \leq i \leq j \leq n$, to be the index r for which k_r is the root of an optimal BST containing keys $k_i, \ldots, k_j$
- Although we will see how to compute the values of $\mathit{root}[i,j]$, we leave the construction of an optimal binary search tree from these values as en exercise

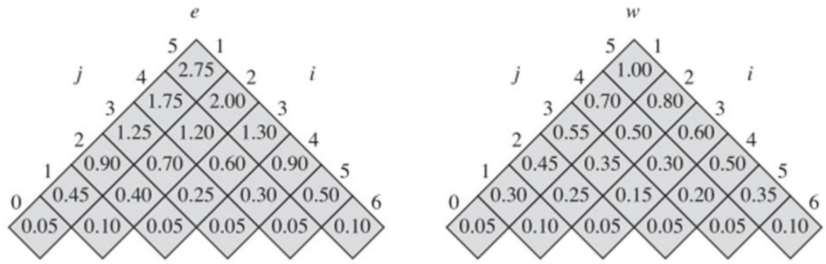
Step 3: Computing the expected search cost of an optimal BST

- We store $e[i,j]$ values in a table $e[1..n+1, 0..n]$
- The first index needs to run to $n+1$ because to have a subtree containing only the dummy key d_n, we need to compute and store $e[n+1, n]$
- The second index needs to start from 0 because to have a subtree containing only the dummy key d_0, we need to compute and store $e[1,0]$

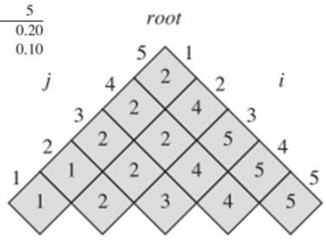
- We use only the entries $e[i,j]$ for which $j \geq i-1$
- We also use a table $root[i,j]$, for recording the root of the subtree containing keys $k_i, \dots, k_j$
- This table uses only the entries $1 \leq i \leq j \leq n$
- We also store the $w(i,j)$ values in a table $w[1..n+1, 0..n]$
- For the base case, we compute $w[i, i-1] = q_i$
- For $j \geq i$, we compute
$$w[i,j] = w[i,j-1] + p_j + q_j$$
- Thus, we can compute the $\Theta(n^2)$ values of $w[i,j]$ in $\Theta(1)$ time each

OPTIMAL-BST(p, q, n)

let $e[1..n+1, 0..n]$, $w[1..n+1, 0..n]$, $root[1..n, 1..n]$ be new tables
for $i = 1$ **to** $n+1$
  $e[i, i-1] = q_{i-1}$
  $w[i, i-1] = q_{i-1}$
for $l = 1$ **to** n
  for $i = 1$ **to** $n - l + 1$
    $j = i + l - 1$
    $e[i,j] = \infty$
    $w[i,j] = w[i,j-1] + p_j + q_j$
    for $r = i$ **to** j
      $t = e[i, r-1] + e[r+1, j] + w[i,j]$
      if $t < e[i,j]$
        $e[i,j] = t$
        $root[i,j] = r$
return e and $root$



i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10



- The OPTIMAL-BST procedure takes $\Theta(n^3)$ time, just like MATRIX-CHAIN-ORDER
- Its running time is $O(n^3)$, since its **for** loops are nested three deep and each loop index takes on at most n values
- The loop indices in OPTIMAL-BST do not have exactly the same bounds as those in MATRIX-CHAIN-ORDER, but they are within ≤ 1 in all directions
- Thus, like MATRIX-CHAIN-ORDER, the OPTIMAL-BST procedure takes $\Omega(n^3)$ time

16 Greedy Algorithms

- Optimization algorithms typically go through a sequence of steps, with a set of choices at each
- For many optimization problems, using dynamic programming to determine the best choices is overkill; simpler, more efficient algorithms will do
- A ***greedy algorithm*** always makes the choice that looks best at the moment
- That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution

16.1 An activity-selection problem

- Suppose we have a set $S = \{a_1, a_2, \dots, a_n\}$ of n proposed activities that wish to use a resource (e.g., a lecture hall), which can serve only one activity at a time
- Each activity a_i has a start time s_i and a finish time f_i, where $0 \leq s_i < f_i < \infty$
- If selected, activity a_i takes place during the half-open time interval $[s_i, f_i)$

- Activities a_i and a_j are ***compatible*** if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap
- I.e., a_i and a_j are compatible if $s_i \geq f_j$ or $s_j \geq f_i$
- We wish to select a maximum-size subset of mutually compatible activities
- We assume that the activities are sorted in monotonically increasing order of finish time:

$$f_1 \leq f_2 \leq f_3 \leq \cdots \leq f_{n-1} \leq f_n$$

- Consider, e.g., the following set S of activities:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

- For this example, the subset $\{a_3, a_9, a_{11}\}$ consists of mutually compatible activities
- It is not a maximum subset, however, since the subset $\{a_1, a_4, a_8, a_{11}\}$ is larger
- In fact, it is a largest subset of mutually compatible activities; another largest subset is $\{a_2, a_4, a_9, a_{11}\}$

The optimal substructure of the activity-selection problem

- Let S_{ij} be the set of activities that start after a_i finishes and that finish before a_j starts
- We wish to find a maximum set of mutually compatible activities in S_{ij}
- Suppose that such a maximum set is A_{ij}, which includes some activity a_k
- By including a_k in an optimal solution, we are left with two subproblems: finding mutually compatible activities in the set S_{ik} and finding mutually compatible activities in the set S_{kj}

- Let $A_{ik} = A_{ij} \cap S_{ik}$ and $A_{kj} = A_{ij} \cap S_{kj}$, so that
  - A_{ik} contains the activities in A_{ij} that finish before a_k starts and
  - A_{kj} contains the activities in A_{ij} that start after a_k finishes
- Thus, $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$, and so the maximum-size set A_{ij} in S_{ij} consists of $|A_{ij}| = |A_{ik}| + |A_{kj}| + 1$ activities
- The usual cut-and-paste argument shows that the optimal solution A_{ij} must also include optimal solutions for S_{ik} and S_{kj}

- This suggests that we might solve the activity-selection problem by dynamic programming
- If we denote the size of an optimal solution for the set S_{ij} by $c[i,j]$, then we would have the recurrence

$$c[i,j] = c[i,k] + c[k,j] + 1$$

- Of course, if we did not know that an optimal solution for the set S_{ij} includes activity a_k, we would have to examine all activities in S_{ij} to find which one to choose, so that

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max_{a_k \in S_{ij}} \{c[i,j] = c[i,k] + c[k,j] + 1\} & \text{if} S_{ij} \neq \emptyset \end{cases}$$

Making the greedy choice

- For the activity-selection problem, we need consider only the greedy choice
- We choose an activity that leaves the resource available for as many other activities as possible
- Now, of the activities we end up choosing, one of them must be the first one to finish
- Choose the activity in S with the earliest finish time, since that leaves the resource available for as many of the activities that follow it as possible
- Activities are sorted in monotonically increasing order by finish time; greedy choice is activity a_1

- If we make the greedy choice, we only have to find activities that start after a_1 finishes
- $s_1 < f_1$ and f_1 is the earliest finish time of any activity $\Rightarrow$ no activity can have a finish time $\leq s_1$
- Thus, all activities that are compatible with activity a_1 must start after a_1 finishes
- Let $S_k = \{a_i \in S: s_i \geq f_k\}$ be the set of activities that start after activity a_k finishes
- Optimal substructure: if a_1 is in the optimal solution, then an optimal solution to the original problem consists of a_1 and all the activities in an optimal solution to the subproblem S_1

Theorem 16.1 *Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.*

Proof Let A_k be a max-size subset of mutually compatible activities in S_k, and let a_j be the activity in A_k with the earliest finish time. If $a_j = a_m$, we are done, since a_m is in a max-size subset of mutually compatible activities of S_k.

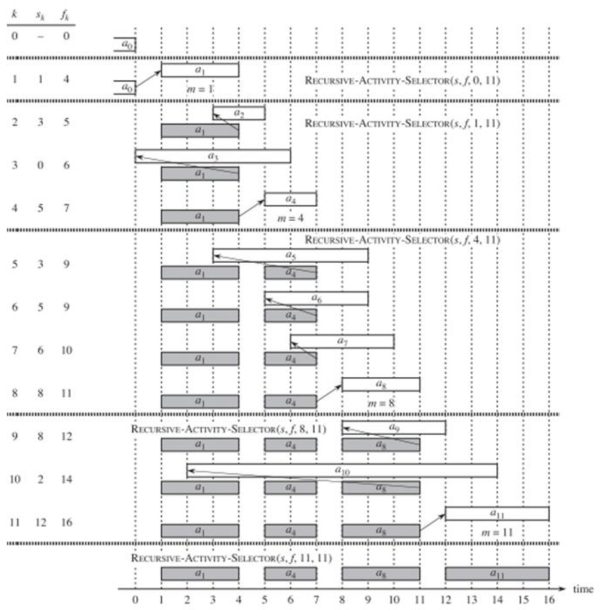
If $a_j \neq a_m$, let the set $A'_k = A_k - \{a_j\} \cup \{a_m\}$. The activities in A'_k are disjoint because the activities in A_k are disjoint, a_j is the first activity in A_k to finish, and $f_m \leq f_j$. Since $|A'_k| = |A_k|$, we conclude that A'_k is a maximum-size subset of mutually compatible activities of S_k and includes a_m. ■

- We can repeatedly choose the activity that finishes first, keep only the activities compatible with this activity, and repeat until no activities remain
- Moreover, because we always choose the activity with the earliest finish time, the finish times of the activities we choose must strictly increase
- We can consider each activity just once overall, in monotonically increasing order of finish times

A recursive greedy algorithm

RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)

1. $m \leftarrow k + 1$
2. **while** $m \leq n$ **and** $s[m] < f[k]$ // find the first // activity in S_k to finish
3. $\quad m \leftarrow m + 1$
4. **if** $m \leq n$
5. $\quad$ **return** $\{a_m\} \cup$ RECURSIVE-ACTIVITY-SELECTOR(s, f, m, n)
6. **else return** $\emptyset$



An iterative greedy algorithm

GREEDY-ACTIVITY-SELECTOR(s, f)

1. $n \leftarrow s.length$
2. $A \leftarrow \{a_1\}$
3. $k \leftarrow 1$
4. **for** $m \leftarrow 2$ **to** n
5. **if** $s[m] \geq f[k]$
6. $A \leftarrow A \cup \{a_m\}$
7. $k \leftarrow m$
8. **return** A

- The set A returned by the call

  GREEDY-ACTIVITY-SELECTOR(s, f)

  is precisely the set returned by the call

  RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)
- Both the recursive version and the iterative algorithm schedule a set of n activities in $\Theta(n)$ time, assuming that the activities were already sorted initially by their finish times

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:

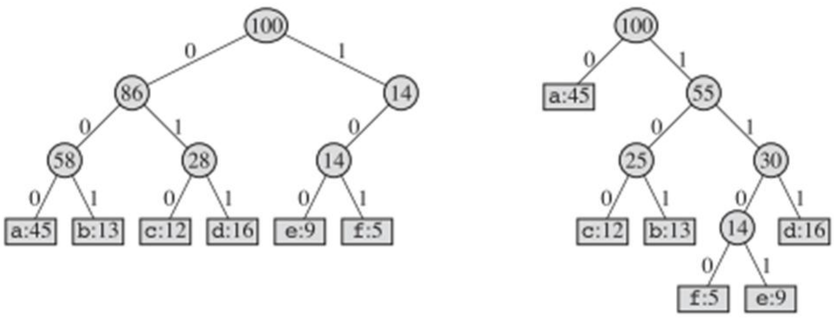
  $a = 000, b = 001, \ldots, f = 101$
- We now need 300,000 bits to code the entire file
- A ***variable-length code*** gives frequent characters short codewords and infrequent characters long codewords
- Here the 1-bit string 0 represents a, and the 4-bit string 1100 represents f
- This code requires $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000 = 224{,}000$ bits (savings $\approx$ 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 $0 \cdot 101 \cdot 100 = 0101100$

- 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 $0 \cdot 0 \cdot 101 \cdot 1101$, 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

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

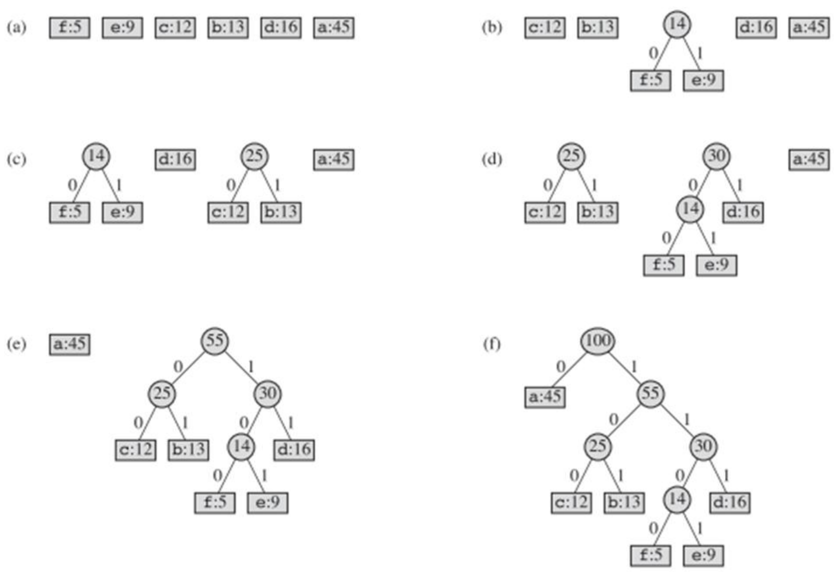
which we define as the cost of the tree T

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.*