note

Hack Chyson

November 2, 2019

Contents

1 Preface

Algorithms lies at the heart of computering.

Website: `http://mitpress.mit.edu/algorithms/`

Psudocode: present algorithm clearly and succintly, without idiosyncrasies of a particular programming language obscuring the its essence.

2 Foundations

2.1 The Role of Algorithms in Computing

2.1.1 what is algorithm?

input -> algorithm -> output
An algorithm is a sequence of computational steps that transform the input into the output.
A algorithm describes a specific computational procedure for achieving the input/output relationship.

2.1.2 instance of a problem

the input needed to compute a solution to the problem.

2.1.3 correct algorithms

For every input instance, it halts out the correct output.

2.1.4 problem list

- Internet, finding good routes and finding pages
- Electronic commerce, public-key cryptography and digital singatures
- oil company, where to place its wells in order to maximize its profit, linear programming
- road map, shortest route
- two ordered sequences, find a longest common subsequence
- a mechanical design, each part may include instances of other parts, to list the parts in order so that each part appears before any part that uses it, topological sorting
- convex hull

2.1.5 two characteristics of many algorithms

1. There are many candidate solutions, but finding the one that solve or the one is best is challenge.
2. They have practical applications.

2.1.6 data structure

a way to store and organize data in order to facilitate access and modifications.

No single data structure works well for all purposes, and it is important to know the strengths and limitations of several of them.

2.1.7 the core technique

learn the technique of algorithm design and analysis.

2.1.8 hard problems

Like the NP-complete problem, there are problem that has no efficient solutions.
Before you delve into the real problem, take a overview of it.

2.1.9 NP-complete problem

1. no one knows whether or not efficient algorithms exist.

2. a solution for one NP-complete probelm will works for all NP-complete problems

3. several NP-complete problems are similar, but not identical to problems for which we do know of efficient algorithms.

2.1.10 algorithm efficiency

Computers are not infinitely fast and memory is not free, thus the efficiency of a algorithm matters.
[2019-06-21 Fri]

2.1.11 algorithms as a technology

Algorithms are at the core of most technologies.

[2019-06-22 Sat]

2.2 Getting Started

2.2.1 loop invariant

Loop invariant is used to help us to understand why an algorithm is correct.

The comparison of loop invariant and mathematical induction

Loop Invariant	Mathematical Induction
initialization	base case
maintenance	inductive step
termination	

Initialization: It is true prior to the first iteration of the loop.
Maintanance: If it is true before an iteration of the loop, it remains true before the next iteration.
Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.

2.2.2 psudocode conventions

```
INSERTION-SORT(T)

for j = 2 to A.length
    key = A[j]
    // Insert A[j] into the sorted sequence A[1..j-1].
    i = j - 1
    while i > 0 and A[i] > key
        // in place sort
        A[i + 1] = A[i]
i = i - 1
    // when the loop terminate, i = 0
    A[i + 1] = key
```

1. Indentation indicates block structures.

2. A loop counter retains its value after exiting the loop. (deffer from C++, Java...)

3. Variable are local to the given procedure.

4. The keyword **error** indicates that an error occurred.

2.2.3 analyzing algorithms

analyzing an algorithm: predict the resources.
resources: time and space (memory, communication bandwidth, computer hardware, computational time...)

2.2.4 model

Before analyzing an algorithm, there must be a model to measure the resource cost.

2.2.5 algorithm vs RAM

The focus is algorithm, not the tedious hardware detail.
To yield a clear insight into algorithm design and analysis, RAM model is simplified.

2.2.6 RAM model

instructions	arithmetic	movement	control	each instructions takes a constant amount of time
	add, abstruct, multiply, divide, remainder, floor, ceiling	load, store copy	conditional and unconditional branch, subroutine call, return	
data types	integer, floating			represented by clgn bits
memory hierarchy	do not model caches or virtual memory			

$c \lg n$ explaination:

- lg means $\log_2$
- 2 as root because of the binary system
- $c \geq 1$: each word can hold the value of n
- c to a constant: the word size does not grow arbitrarily

2.2.7 core idea in modeling

show tha important characteristcs of algorithms and suppress the tedious details.

2.2.8 analysis of a algorithm

In general, the time grows with the size of the input,
so it is traditional to describe the running time as the function of the size of its input.

		Examples
input size	depends on the problem being studied	number of items, total number of bits ...
running time	the number of primitive operations	

Assumption for simpler analysis:
A constant amount of time is requried to execute each line of the pseudocode.

2.2.9 worst-case analysis

Becuase the behavior of an algorithm may be different for each possible input,
we need a means for summarizing that behavior in simple, easily understood formulas.

the reason to analyze worst-case running time:

1. give an upper bound on the running time
2. worst case ocurrs fairly often
3. the "average case" is often roughly as bad as the worst case

2.2.10 abstraction

Use some simplifying abstraction to ease the analysis.

1. ignore the actual cost of each statement, using the constants c_i to represent these costs.
2. ignore the abstract costs c_i ($an^2 + bn + c$)
3. rate of growth or order of growth of the running time ($\Theta(n^2)$) (pronounced "theta of n-squared")

2.2.11 designing algorithms

1. incremental approch example: insertion-sort

2. devide-and-conquer approch example: like merge-sort

   (a) divide the problem into a number of subproblems

   (b) conquer the subproblems

   (c) combine the solution

2.3 Growth of Functions

Althoght we can sometimes determine the exact running time of an algorithm, the extra procision is not usually worth the effort of computing it.

When we look at input sizes large enought to make only the order of growth of the running time relevant, we are studying the `asymptotic efficiency of algorithms`.

2.3.1 Asymptotic notation

1. Θ-notation

   $\Theta(g(n)) = \{f(n) : \text{there exist positive constant } c_1, c_2 \text{and } n_0 \text{such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}(1)$

   Because $\Theta(g(n))$ is a set, we could write "$f(n) \in \Theta(g(n))$ " to indicate that $f(n)$ is a member of $\Theta(g(n))$. Instead, we will usually write "$f(n) = \Theta(g(n))$ " to express the same notion.

   Since any constant is a degree-0 polynomial, we can express any constant function as $\Theta(n^0)$, or $\Theta(1)$.

2. O-notation

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{and } n_0 \text{ such that} 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \tag{2}$$

O-notation to the worst case ==> to every input
Θ-notation to the worst case =/=> to every input

3. Ω-notation

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \tag{3}$$

4. Theorem For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$
if and only if
$f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$

5. o-notation an upper bound that is not asymptotically tight.

$$o(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{there exist a constant } n_0 > 0 \text{ such that } 0 \leq f(n) \tag{4}$$

or

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \tag{5}$$

6. ω-notation a lower bound that is not asymptotically tight.

$$\omega(g(n)) = \{f(n) : \text{for any positive constant } c > 0, \text{there exist a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) \tag{6}$$

or

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \tag{7}$$

7. comparing functions

(a) Transitivity

$$f(n) = \Theta(g(n)) \quad \text{and} \quad g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$$

The same to O, Ω, o, ω .

(b) Reflexivity

$$f(n) = \Theta(f(n))$$

$$f(n) = O(f(n))$$

$$f(n) = \Omega(f(n))$$

(c) Symmetry

$$f(n) = \Theta(g(n)) \quad \text{if and only if} \quad g(n) = \Theta(f(n))$$

(d) Transpose symmetry

$$f(n) = O(g(n)) \quad \text{if and only if} \quad g(n) = \Omega(f(n))$$

$$f(n) = o(g(n)) \quad \text{if and only if} \quad g(n) = \omega(f(n))$$

2.3.2 Standard notations and common functions

2.4 Divide-and-Conquer

A recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs.

For example:

$$T(n) = \begin{cases} \Theta(1) & \text{ƆƱ} \quad n = 1 \\ 2T(n/2) + \Theta(n) & \text{ƆƱ} \quad n > 1 \end{cases} \tag{8}$$

2.4.1 The master method for solving recurrences

$$T(n) = aT(n/b) + f(n) \tag{9}$$

where $a \geq 1$ and $b > 1$ are constants and f(n) is an asymptotically positive function.

Then $T(n)$ has the following asymptotic bounds:

1. If $f(n) = O(n^{\log_b(a-\epsilon)})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$.

3. If $f(n) = O(n^{\log_b(a+\epsilon)})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Intuitively, the larger of the two functions determines the solution to the recurrence.

Note:
Beyond this intuition, you need to be aware of some technicalities. In the first case, not only must $f(n)$ be smaller than $n^{\log_b a}$, it must be polynomially smaller. In the third case, not only must $f(n)$ be larger than $n^{\log_b a}$, it also must be polynomially larger and in addition satisfy the "regularity" condition that $af(n/b) \leq cf(n)$

2.5 Probabilistic Analysis and Randomized Algorithms

Probabilistic analysis is the use of probability in the analysis of problems.

In order to perform a probabilistic analysis, we must use knowledge of, or make assumptions about, the distribution of the inputs. Then we analyze

our algorithm, computing an average-case running time, where we take the average over the distribution of the possible inputs. Thus we are, in effect, averaging the running time over all possible inputs. When reporting such a running time, we will refer to it as the average-case running time.

we call an algorithm randomized if its behavior is determined not only by its input but also by values produced by a random-number generator.

In general, we discuss the average-case running time when the probability distribution is over the inputs to the algorithm, and we discuss the expected running time when the algorithm itself makes random choices.

2.5.1 Indicator random variables

In order to analyze many algorithms, we use indicator random variables. Indicator random variables provide a convenient method for converting between probabilities and expectations.
Suppose we are given a sample space S and an event A. Then the indicator random variable $I\{A\}$ associated with event A is defined as

$$I\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs} \\ 0 & \text{if } A \text{ does not occurs} \end{cases} \tag{10}$$

Lemma:
Given a sample space S and an event A in the sample space S, let $X_A = I\{A\}$. Then $E[X_A] = Pr\{A\}$.

2.5.2 Randomized alogrithms

Most of the time, we do not have knowledge in the input distribution. Instead of assuming a distribution of inputs, we impose a distribution.

1. Randomly permuting arrays

```
PERMUTE-BY-SORTING(A)
let P[1..n] be a new array
for i = 1 to n
    P[i] = RANDOM(1, n^3)  # priority
sort A, using P as sort keys
```

```
RANDOMIZE-IN-PLACE(A)
n = A.length
for i = 1 to n
    swap A[i] with A[RANDOM(i, n)]
```

3 Sorting and Order Statistics

sorting problem:

Input: A sequence of n numbers $a_1, a_2, ..., a_n$.

Output: A permutation (reordering) $a_1', a_2', ..., a_n'$ of the input sequence such that $a_1' \leq a_2' \leq ... \leq a_n'$.

In practice, the numbers to be sorted are rarely isolated values. Each is usually part of a collection of data called a record. Each record contains a key, which is the value to be sorted. The remainder of the record consists of satellite data, which are usually carried around with the key.

3.1 Heapsort

3.1.1 Heaps

The (binary) heap data stucture is an array object that can view as nearly complete tree. Each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.

An array A that represents a heap is an object with two attributes:

1. A.length, which gives the number of elements in the array

2. A.heap-size, which represents how many elements in the heap are stored with array A

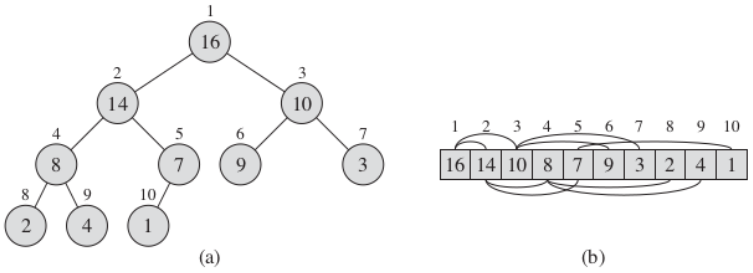


Figure 6.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

PARENT(i)
1 **return** $\lfloor i/2 \rfloor$

LEFT(i)
1 **return** $2i$

RIGHT(i)
1 **return** $2i + 1$

There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a heap property.

In a max-heap, the max-heap property is that for every node i other than root, $A[PARENT(i)] \geq A[i]$.
In a min-heap, $A[PARENT(i)] \leq A[i]$.

3.1.2 Maintaining the heap property

In order to maintain the max-heap property, we call the procedure MAX-HEAPIFY. Its inputs are an array A and an index i into the array. When it is called, MAX-HEAPIFY assumes that the binary trees rooted at LEFT(i) and RIGHT(i) are max-heaps, but that A[i] be smaller than its children, thus violating the max-heap property.

```
MAX-HEAPIFY(A, i)
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4       largest = l
 5  else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7       largest = r
 8  if largest ≠ i
 9       exchange A[i] with A[largest]
10       MAX-HEAPIFY(A, largest)
```

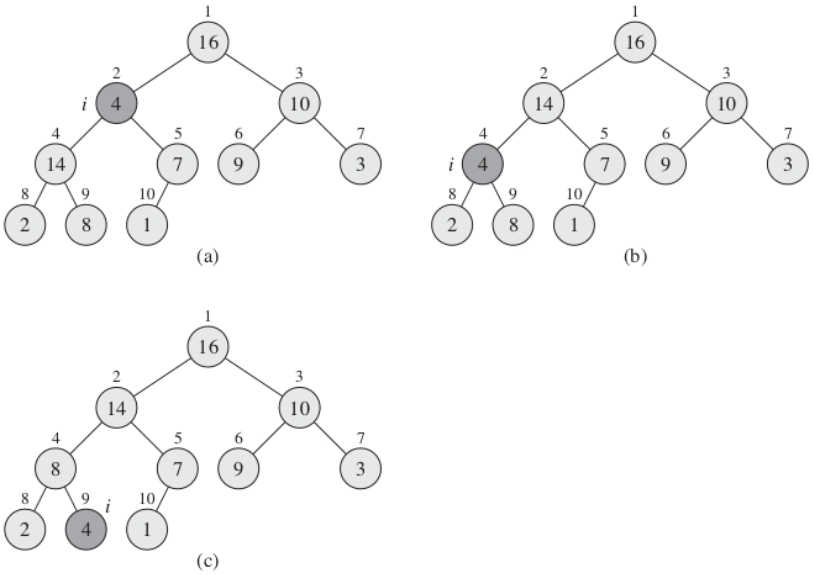


Figure 6.2 The action of MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. **(a)** The initial configuration, with $A[2]$ at node $i = 2$ violating the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in **(b)** by exchanging $A[2]$ with $A[4]$, which destroys the max-heap property for node 4. The recursive call MAX-HEAPIFY$(A, 4)$ now has $i = 4$. After swapping $A[4]$ with $A[9]$, as shown in **(c)**, node 4 is fixed up, and the recursive call MAX-HEAPIFY$(A, 9)$ yields no further change to the data structure.

3.1.3 Building a heap

BUILD-MAX-HEAP(A)

1 $A.heap\text{-}size = A.length$
2 **for** $i = \lfloor A.length/2 \rfloor$ **downto** 1
3 MAX-HEAPIFY(A, i)

3.1.4 The heapsort algorithm

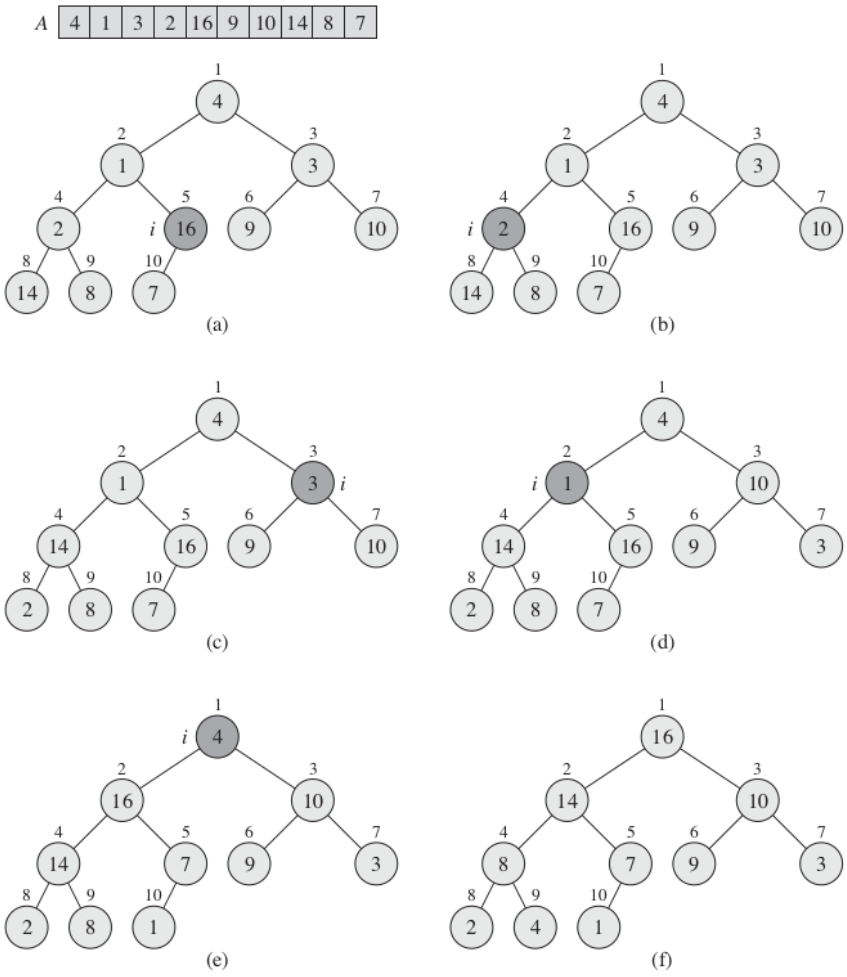


Figure 6.3 The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. **(a)** A 10-element input array A and the binary tree it represents. The figure shows that the loop index i refers to node 5 before the call MAX-HEAPIFY(A, i). **(b)** The data structure that results. The loop index i for the next iteration refers to node 4. **(c)–(e)** Subsequent iterations of the **for** loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. **(f)** The max-heap after BUILD-MAX-HEAP finishes.

3.1.5 The heapsort algorithm

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

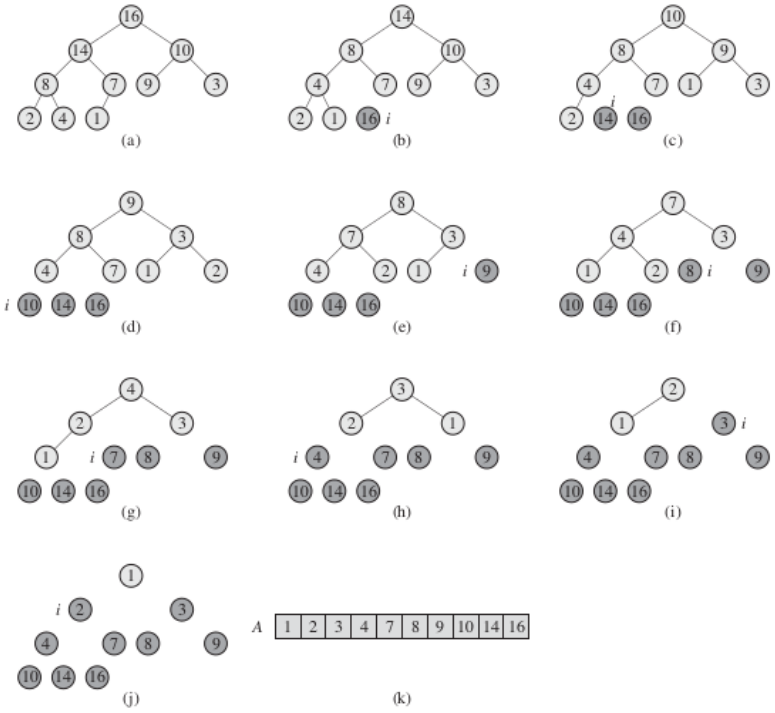


Figure 6.4 The operation of HEAPSORT. **(a)** The max-heap data structure just after BUILD-MAX-HEAP has built it in line 1. **(b)–(j)** The max-heap just after each call of MAX-HEAPIFY in line 5, showing the value of i at that time. Only lightly shaded nodes remain in the heap. **(k)** The resulting sorted array A.

3.1.6 Priority queues

A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a **key**.

A max-priority queue supports the following operations:

- INSERT(S,x) inserts the elements x into the set S, which is equivalent to the operations $S = S \cup \{x\}$.
- MAXIMUM(S) returns the element of S with the largest key.
- EXTRACT-MAX(S) removes and returns the element of S with the largest key.
- INCREASE-KEY(S,x,k) increase the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

When we use a heap to implement a priority queue, therefore, we often need to store a handle to the corresponding application object in each heap element.

```
HEAP-MAXIMUM(A)
1  return A[1]
```

```
HEAP-EXTRACT-MAX(A)
1  if A.heap-size < 1
2       error "heap underflow"
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size - 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

```
HEAP-INCREASE-KEY(A, i, key)
1  if key < A[i]
2       error "new key is smaller than current key"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5       exchange A[i] with A[PARENT(i)]
6       i = PARENT(i)
```

```
MAX-HEAP-INSERT(A, key)
1  A.heap-size = A.heap-size + 1
2  A[A.heap-size] = -∞
3  HEAP-INCREASE-KEY(A, A.heap-size, key)
```

3.2 Quicksort

The following procedure implements quicksort:

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

To sort an entire array A, the initial call is QUICKSORT$(A, 1, A.length)$.

Partitioning the array

The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p\,..\,r]$ in place.

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

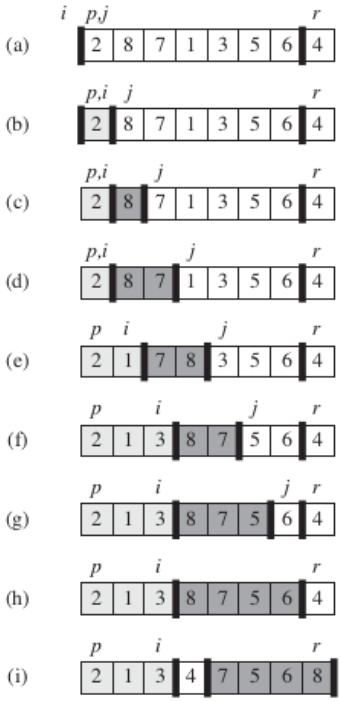


Figure 7.1 The operation of PARTITION on a sample array. Array entry $A[r]$ becomes the pivot element x. Lightly shaded array elements are all in the first partition with values no greater than x. Heavily shaded elements are in the second partition with values greater than x. The unshaded elements have not yet been put in one of the first two partitions, and the final white element is the pivot x. **(a)** The initial array and variable settings. None of the elements have been placed in either of the first two partitions. **(b)** The value 2 is "swapped with itself" and put in the partition of smaller values. **(c)–(d)** The values 8 and 7 are added to the partition of larger values. **(e)** The values 1 and 8 are swapped, and the smaller partition grows. **(f)** The values 3 and 7 are swapped, and the smaller partition grows. **(g)–(h)** The larger partition grows to include 5 and 6, and the loop terminates. **(i)** In lines 7–8, the pivot element is swapped so that it lies between the two partitions.

```
RANDOMIZED-PARTITION(A, p, r)
1  i = RANDOM(p, r)
2  exchange A[r] with A[i]
3  return PARTITION(A, p, r)
```

The new quicksort calls RANDOMIZED-PARTITION in place of PARTITION:

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2      q = RANDOMIZED-PARTITION(A, p, r)
3      RANDOMIZED-QUICKSORT(A, p, q − 1)
4      RANDOMIZED-QUICKSORT(A, q + 1, r)
```

3.3 Sorting in Linear Time

comparison sorts: the sorted order they determine is based only on comparison between the input elements.

Any comparison sort must make $\Omega(n \lg n)$ comparisons in the worst case to sort n elements.

3.3.1 Counting sort

Counting sort assume that each of the input elements is a integer in the range 0 to k, for some integer k. When $k = O(n)$, the sort runs in $\Theta(n)$ time.

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

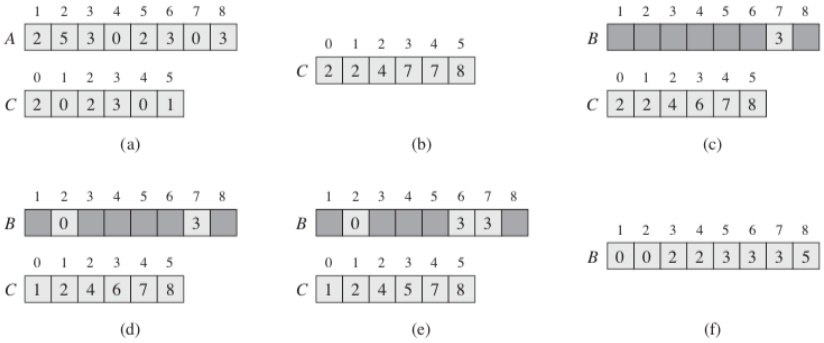


Figure 8.2 The operation of COUNTING-SORT on an input array $A[1..8]$, where each element of A is a nonnegative integer no larger than $k = 5$. **(a)** The array A and the auxiliary array C after line 5. **(b)** The array C after line 8. **(c)–(e)** The output array B and the auxiliary array C after one, two, and three iterations of the loop in lines 10–12, respectively. Only the lightly shaded elements of array B have been filled in. **(f)** The final sorted output array B.

3.3.2 Radix sort

The following procedure assumes that each element in the n-element array A has d digits, where digit 1 is the lowest-order digit and d is the highest-order digit.

```
RADIX-SORT(A, d)
1  for i = 1 to d
2      use a stable sort to sort array A on digit i
```

Figure 8.3 The operation of radix sort on a list of seven 3-digit numbers. The leftmost column is the input. The remaining columns show the list after successive sorts on increasingly significant digit positions. Shading indicates the digit position sorted on to produce each list from the previous one.

Given n d-digit numbers in which each digit can take on up to k possible values, RADIX-SORT correctly sorts these numbers in $\Theta(d(n+k))$ time if the stable sort it uses takes $\Theta(n+k)$ time.

3.3.3 Bucket sort

Bucket sort assumes that the input is generated by a random process that distributes elements uniformly and independently over the interval [0,1). Bucket sort divides the interval [0,1) into n equal-sized subintervals, or buckets, and then distributes the n inputs elements into the buckets.

```
BUCKET-SORT(A)
1  let B[0..n − 1] be a new array
2  n = A.length
3  for i = 0 to n − 1
4      make B[i] an empty list
5  for i = 1 to n
6      insert A[i] into list B[⌊nA[i]⌋]
7  for i = 0 to n − 1
8      sort list B[i] with insertion sort
9  concatenate the lists B[0], B[1], . . . , B[n − 1] together in order
```

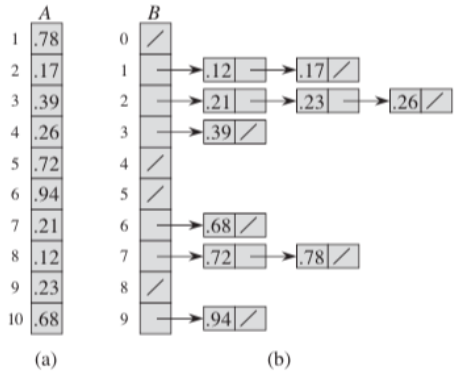


Figure 8.4 The operation of BUCKET-SORT for $n = 10$. **(a)** The input array $A[1..10]$. **(b)** The array $B[0..9]$ of sorted lists (buckets) after line 8 of the algorithm. Bucket i holds values in the half-open interval $[i/10, (i+1)/10)$. The sorted output consists of a concatenation in order of the lists $B[0], B[1], \ldots, B[9]$.

3.3.4 Medians and Order Statistics

The ith order statistic of a set of n elements is the ith smallest element. For example, the minimum of a set of elements is the first order statistics (i=1), and the maximum is the nth order statistics (i=n).
A median is the "halfway point" of the set. When n is odd, the median is unique, ocurring at $i = (n+1)/2$. When n is even, there are two median, ocurring at $i = \lfloor (n+1)/2 \rfloor$ (the lower median) and $i = \lceil (n+1)/2 \rceil$

4 Data Structures

In a typical implementation of a dynamic set, each element is represented by an object whose attributes can be examined and manipulated if we have a pointer to the object.

Operations on a dynamic set can be grouped into two categories: queries, which simply return information about the set, and modifying operations, which change the set.

SEARCH(S,k)
A query that, given a set S and a key value k, returns a pointer x to an

element in S such that x.key = k, or NIL if on such element belongs to S.

INSERT(S,x)
A modifying operation that augements the set S with the element pointed to by x.

DELETE(S,x)
A modifying operation that, given a pointer x to an element in the set S, removes x from S.

MINIMUM(S)
A query on a totally ordered set S that returns a pointer to the element of S with the smallest key.

MAXIMUM(S)
A query on a totally ordered set S that returns a pointer to the element of S with the largest key.

SUCCESSOR(S,x)
A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next larger element in S, or NIL if x is the maximum element.

PREDECESSOR(S,x)
A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next smaller element in S, or NIL if x is the minimum element.

4.1 Elementary Data Structure

4.1.1 Stacks and queues

Stacks and queues are dynamic sets in which the element removed from the set by the DELETE operation is prespecified. In a stack, the element deleted from the set is the one most recently inserted: the stack implements a last-in, first-out, or LIFO, policy. Similarly, in a queue, the element deleted is always the one that has been in the set for the longest time: the queue implements a first-in, first-out, or FIFO, policy.

1. Stacks The INSERT operation on a stack is often called PUSH , and the DELETE operation, which does not take an element argument, is often called POP.

   We can implement a stack of at most n elements with an array S[1..n]. The array has an attribute S.top that indexes the most recently inserted element. The stack consists of elements S[1..S.top], where S[1] is the element at the bottom of the stack and S[S.top] is the element at the top.

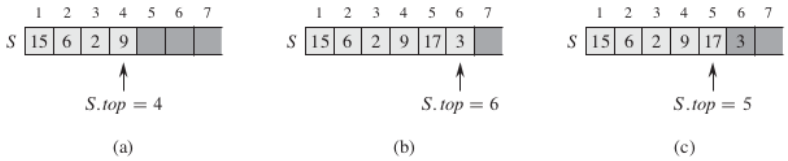


Figure 10.1 An array implementation of a stack S. Stack elements appear only in the lightly shaded positions. **(a)** Stack S has 4 elements. The top element is 9. **(b)** Stack S after the calls PUSH$(S, 17)$ and PUSH$(S, 3)$. **(c)** Stack S after the call POP(S) has returned the element 3, which is the one most recently pushed. Although element 3 still appears in the array, it is no longer in the stack; the top is element 17.

   When S.top = 0, the stack contains no elements and is empty. If we attempt to pop an empty stack, we say the stack underflows. If S.top exceeds n, the stack overflows.

```
STACK-EMPTY(S)
1  if S.top == 0
2       return TRUE
3  else return FALSE

PUSH(S, x)
1  S.top = S.top + 1
2  S[S.top] = x

POP(S)
1  if STACK-EMPTY(S)
2       error "underflow"
3  else S.top = S.top - 1
4       return S[S.top + 1]
```

2. Queues We call the INSERT operation on a queue ENQUEUE , and we call the DELETE operation DEQUEUE.

   The queue has a head and a tail. When an element is enqueued, it takes its place at the tail of the queue, takes a place at the end of the line. The element dequeued is always the one at the head of the queue.

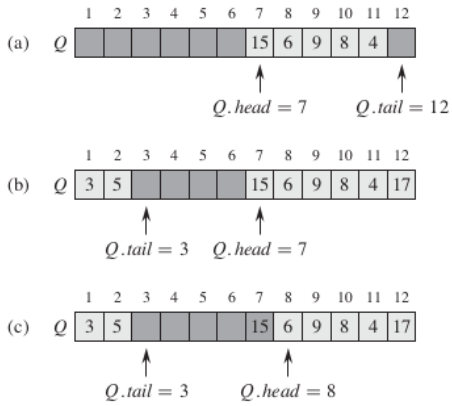


Figure 10.2 A queue implemented using an array $Q[1..12]$. Queue elements appear only in the lightly shaded positions. **(a)** The queue has 5 elements, in locations $Q[7..11]$. **(b)** The configuration of the queue after the calls ENQUEUE$(Q, 17)$, ENQUEUE$(Q, 3)$, and ENQUEUE$(Q, 5)$. **(c)** The configuration of the queue after the call DEQUEUE(Q) returns the key value 15 formerly at the head of the queue. The new head has key 6.

```
ENQUEUE(Q, x)
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3       Q.tail = 1
4  else Q.tail = Q.tail + 1

DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3       Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

4.1.2 Linked lists

A linked list is a data structure in which the objects are arranged in a linear order.

Unlike an array, however, in which the linear order is determined by the array indices, the order in a linked list is determined by a pointer in each object.

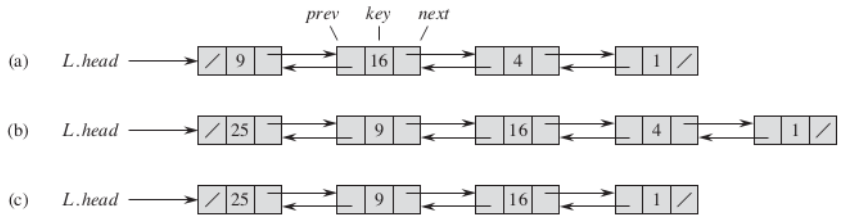


Figure 10.3 **(a)** A doubly linked list L representing the dynamic set $\{1, 4, 9, 16\}$. Each element in the list is an object with attributes for the key and pointers (shown by arrows) to the next and previous objects. The *next* attribute of the tail and the *prev* attribute of the head are NIL, indicated by a diagonal slash. The attribute $L.head$ points to the head. **(b)** Following the execution of LIST-INSERT(L, x), where $x.key = 25$, the linked list has a new object with key 25 as the new head. This new object points to the old head with key 9. **(c)** The result of the subsequent call LIST-DELETE(L, x), where x points to the object with key 4.

```
LIST-SEARCH(L, k)
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3       x = x.next
4  return x
```

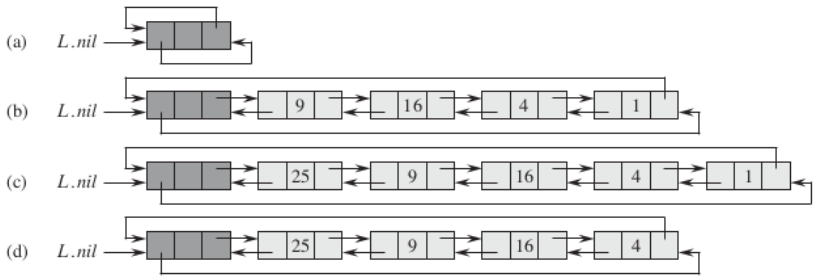
```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head ≠ NIL
3        L.head.prev = x
4  L.head = x
5  x.prev = NIL
```

```
LIST-DELETE(L, x)
1  if x.prev ≠ NIL
2        x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5        x.next.prev = x.prev
```

The code for LIST-DELETE would be simpler if we could ignore the boundary conditions at the head and the tail of the list:

```
LIST-DELETE'(L, x)
1   x.prev.next = x.next
2   x.next.prev = x.prev
```

A sentinel is a dummy object that allows us to simplify boundary conditions.




```
LIST-SEARCH'(L, k)
1   x = L.nil.next
2   while x ≠ L.nil and x.key ≠ k
3       x = x.next
4   return x
```

$$
\begin{array}{ll}
\multicolumn{2}{l}{\textsc{List-Insert}'(L, x)} \\
1 & x.next = L.nil.next \\
2 & L.nil.next.prev = x \\
3 & L.nil.next = x \\
4 & x.prev = L.nil
\end{array}
$$

Sentinels rarely reduce the asymptotic time bounds of data structure operations, but they can reduce constant factors.

selection probelm:
Input: A set of n (distinct) numbers and an interger i, with $1 \leq i \geq n$.
Output: The elements $x \in A$ that is larger than exactly i-1 other elements of A.

1. Minimum and maximum

```
MINIMUM(A)
    min = A[1]
    for i = 2 to A.length
        if min > A[i]
            min = A[i]
    return min

MIN_MAX(A)
    # init the min and max and start index
    if A.lenght is odd
        min = A[1]
        max = A[1]
```

```
    start = 2
else
    if A[1] > A[2]
        min = A[2]
        max = A[1]
    else
        min = A[1]
        max = A[2]
    start = 3

for i = start to A.length by 2
    if A[i] > A[i + 1]
        if A[i] > max
            max = A[i]
        if A[i + 1] < min
            min = A[i + 1]
    else
        if A[i + 1] > max
            max = A[i + 1]
        if A[i] < min
            min = A[i]

return min, max
```

2. Selection in expected linear time

```
RANDOMIZED-SELECT(A, p, r, i)
1  if p == r
2      return A[p]
3  q = RANDOMIZED-PARTITION(A, p, r)
4  k = q − p + 1
5  if i == k            // the pivot value is the answer
6      return A[q]
7  elseif i < k
8      return RANDOMIZED-SELECT(A, p, q − 1, i)
9  else return RANDOMIZED-SELECT(A, q + 1, r, i − k)
```

3. Selection in worst-case linear time

The SELECT algorithm determines the ith smallest of an input array of $n > 1$ distinct elements by executing the following steps. (If $n = 1$, then SELECT merely returns its only input value as the ith smallest.)

1. Divide the n elements of the input array into $\lfloor n/5 \rfloor$ groups of 5 elements each and at most one group made up of the remaining n mod 5 elements.
2. Find the median of each of the $\lceil n/5 \rceil$ groups by first insertion-sorting the elements of each group (of which there are at most 5) and then picking the median from the sorted list of group elements.
3. Use SELECT recursively to find the median x of the $\lceil n/5 \rceil$ medians found in step 2. (If there are an even number of medians, then by our convention, x is the lower median.)
4. Partition the input array around the median-of-medians x using the modified version of PARTITION. Let k be one more than the number of elements on the low side of the partition, so that x is the kth smallest element and there are $n-k$ elements on the high side of the partition.
5. If $i = k$, then return x. Otherwise, use SELECT recursively to find the ith smallest element on the low side if $i < k$, or the $(i-k)$th smallest element on the high side if $i > k$.

4.1.3 Representing rooted trees

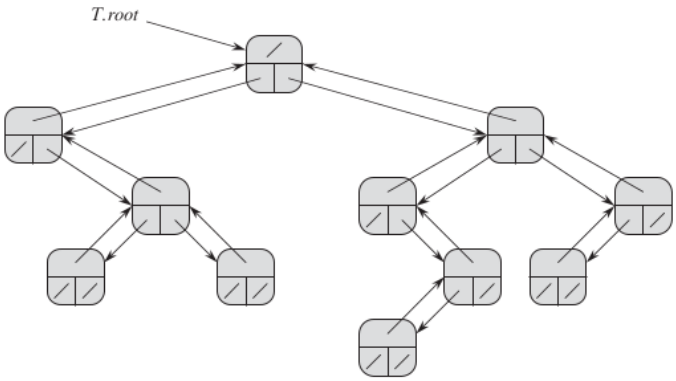


Figure 10.9 The representation of a binary tree T. Each node x has the attributes $x.p$ (top), $x.left$ (lower left), and $x.right$ (lower right). The *key* attributes are not shown.

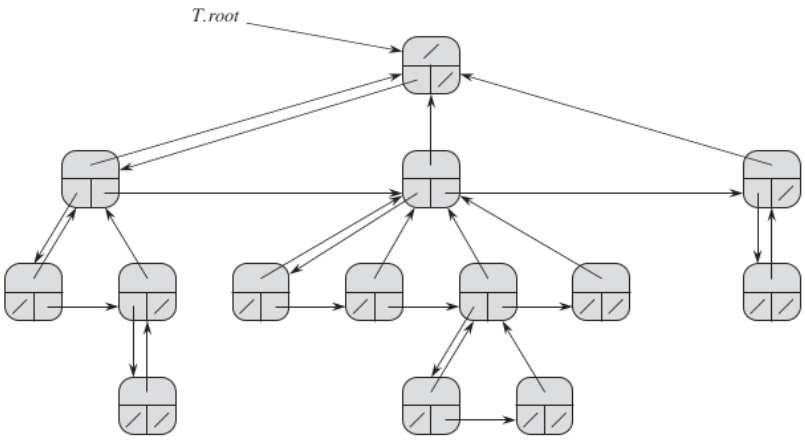


Figure 10.10 The left-child, right-sibling representation of a tree T. Each node x has attributes $x.p$ (top), $x.left\text{-}child$ (lower left), and $x.right\text{-}sibling$ (lower right). The *key* attributes are not shown.

4.2 Hash Tables

Many applications require a dynamic set that supports only the dictionary operations INSERT, SEARCH, and DELETE.

Directly addressing into an ordinary array makes effective use of our ability to examine an arbitrary position in an array in O(1) time.

When the number of keys actually stored is small relative to the total number of possible keys, hash tables become an effective alternative to directly addressing an array, since a hash table typically uses an array of size proportional to the number of keys actually stored. Instead of using the key as an array index directly, the array index is computed from the key.

```
Hashing is an extremely effective and practicaltechnique:
the basic dictionary operations require only O(1) time on the average.
```

4.2.1 Direct-address tables

Direct addressing is a simple technique that works well when the universe U of keys is reasonably small.

To represent the dynamic set, we use an array, or direct-address table, denoted by T[0..m-1], in which each position, or slot, corresponds to a key in the universe U.

DIRECT-ADDRESS-SEARCH(T, k)

1 **return** $T[k]$

DIRECT-ADDRESS-INSERT(T, x)

1 $T[x.key] = x$

DIRECT-ADDRESS-DELETE(T, x)

1 $T[x.key] = \text{NIL}$

Each of these operations takes only $O(1)$ time.

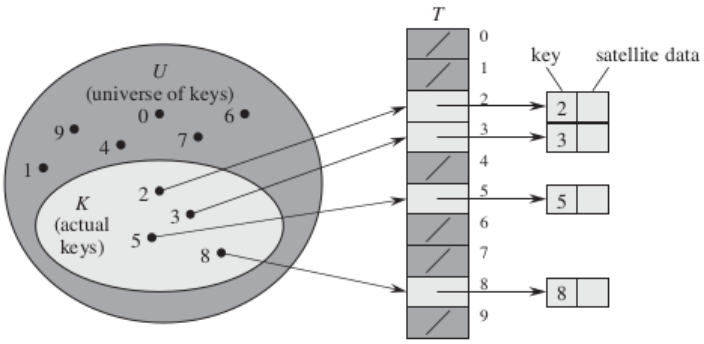


Figure 11.1 How to implement a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded, contain NIL.

4.2.2 Hash Tables

The downside of direct addressing is obvious:

1. if the universe U is large, storing a table T of size |U| may be impractical, or even impossible.

2. the set K of keys actually stored may be so small relative to U that most of the space allocated for T would be wasted.

When the set K of keys stored in a dictionary is much smaller than the universe U of all possible keys, a hash table reduces the storage requirement

to $\Theta(|K|)$ while maintains the benefit that searching for an element in the hash table still requires only O(1) time.

we use a hash function h to compute the slot from the key k.

$$h : U \to \{0, 1, ..., m-1\} \tag{11}$$

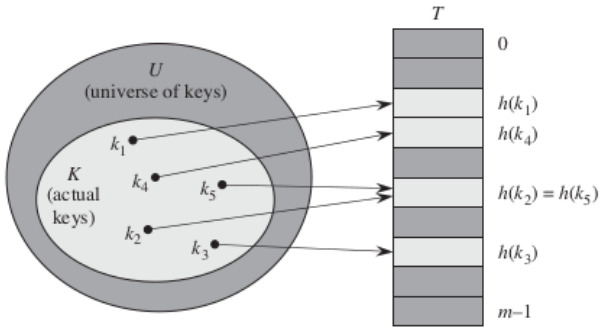


Figure 11.2 Using a hash function h to map keys to hash-table slots. Because keys k_2 and k_5 map to the same slot, they collide.

1. Collision resolution by chaining In chaining, we place all the elements that hash to the same slot into the same linked list.

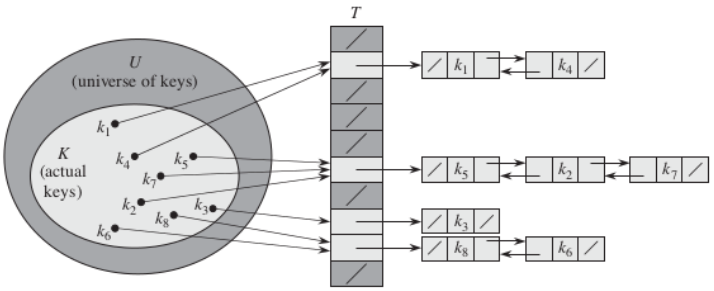


Figure 11.3 Collision resolution by chaining. Each hash-table slot $T[j]$ contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_7) = h(k_2)$. The linked list can be either singly or doubly linked; we show it as doubly linked because deletion is faster that way.

CHAINED-HASH-INSERT(T, x)

1 insert x at the head of list $T[h(x.key)]$

CHAINED-HASH-SEARCH(T, k)

1 search for an element with key k in list $T[h(k)]$

CHAINED-HASH-DELETE(T, x)

1 delete x from the list $T[h(x.key)]$

Given a hash table T with m slots that stores n elements, we define the load factor α for T as n/m, that is, the average number of elements stored in a chain.

simple uniform hashing:
Any given element is equally likely to hash into any of the m slots, independently of where any other element has hashed to.

4.2.3 Hash Functions

A good hash function satisfies (approximately) the assumption of simple uniform hashing: each key is equally likely to hash to any of the m slots, independently of where any other key has hashed to.

In practice, we can often employ heuristic techniques to create a hash function that performs well.

A good approach derives the hash value in a way that we expect to be independent of any patterns that might exist in the data.

Most hash functions assume that the universe of keys is the set $N = 0, 1, 2, ...$ of natural numbers. Thus, if the keys are not natural numbers, we find a way to interpret them as natural numbers.

1. The division method h(k) = k mod m

```
m should not be a power of 2, since if m = 2^p , then h(k) is just the p lowest-or
A prime not too close to an exact power of 2 is often a good choice for m.
```

2. The multiplication method $h(k) = \lfloor m(kA \gtrdot\!\!\!\bowtie 1) \rfloor$
   $A \approx (\sqrt{5} - 1)/2$

4.2.4 Open addressing

4.3 Binary Search Trees

4.3.1 What is a binary search tree?

A binary search tree is organized in a binary tree.

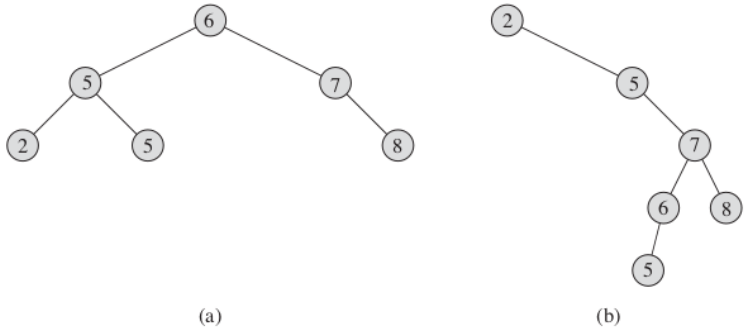


Figure 12.1 Binary search trees. For any node x, the keys in the left subtree of x are at most $x.key$, and the keys in the right subtree of x are at least $x.key$. Different binary search trees can represent the same set of values. The worst-case running time for most search-tree operations is proportional to the height of the tree. **(a)** A binary search tree on 6 nodes with height 2. **(b)** A less efficient binary search tree with height 4 that contains the same keys.

binary-search-tree property:
Lets x be the node in a binary search tree. If y is a node in the left subtree of x, then $x.key \leq x.key$. If y is a node in the right subtree of x, then $y.key \geq x.key$.

The binary-search-tree property allows us to print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an inorder tree walk. This algorithm is so named because it prints the key of the root of a subtree between printing the values in its left subtree and

printing those in its right subtree.

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

If x is the root of an n-node subtree, then the call INODER-TREE-WALK(x) takes $\Theta(n)$ time.

4.3.2 Querying a binary search time

SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR run in O(h) time. (h is the height)

```
TREE-SEARCH(x, k)
1  if x == NIL or k == x.key
2      return x
3  if k < x.key
4      return TREE-SEARCH(x.left, k)
5  else return TREE-SEARCH(x.right, k)
```

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ x.key
2      if k < x.key
3          x = x.left
4      else x = x.right
5  return x
```

```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2      x = x.left
3  return x
```

```
TREE-MAXIMUM(x)
1  while x.right ≠ NIL
2      x = x.right
3  return x
```

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2      return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5      x = y
6      y = y.p
7  return y
```

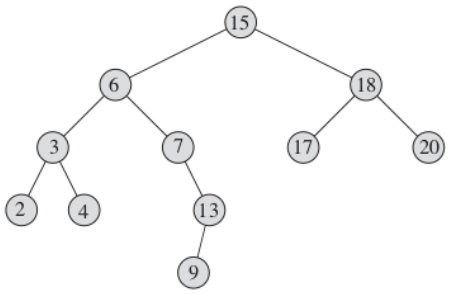


Figure 12.2 Queries on a binary search tree. To search for the key 13 in the tree, we follow the path $15 \to 6 \to 7 \to 13$ from the root. The minimum key in the tree is 2, which is found by following *left* pointers from the root. The maximum key 20 is found by following *right* pointers from the root. The successor of the node with key 15 is the node with key 17, since it is the minimum key in the right subtree of 15. The node with key 13 has no right subtree, and thus its successor is its lowest ancestor whose left child is also an ancestor. In this case, the node with key 15 is its successor.

```
TREE-PREDECESSOR(x)
  if x.left != NIL
    return TREE-MINIMUM(x.left)
  y = x.p
  while y != NIL and x == y.left
    x = y
    y = y.p
  return y
```

```
If a node in a binary search tree has two children, then its successor has no left chi
(Ohterwise it will be successor or predecessor)
```

4.3.3 Insertion and deletion

INSERTION and DELETION run in O(h) time.

```
TREE-INSERT(T, z)
 1   y = NIL
 2   x = T.root
 3   while x ≠ NIL
 4         y = x
 5         if z.key < x.key
 6               x = x.left
 7         else x = x.right
 8   z.p = y
 9   if y == NIL
10         T.root = z          // tree T was empty
11   elseif z.key < y.key
12         y.left = z
13   else y.right = z
```

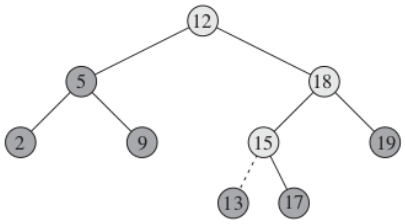
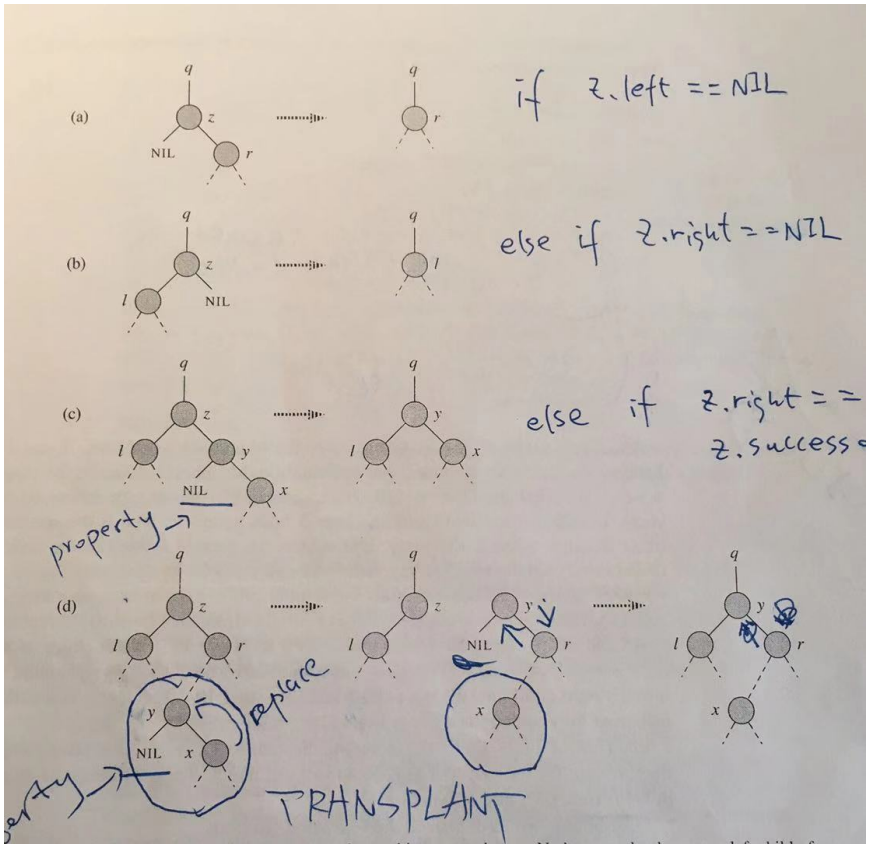


Figure 12.3 Inserting an item with key 13 into a binary search tree. Lightly shaded nodes indicate the simple path from the root down to the position where the item is inserted. The dashed line indicates the link in the tree that is added to insert the item.

deletetion auxilary function:

```
TRANSPLANT(T, u, v)
1   if u.p == NIL
2       T.root = v
3   elseif u == u.p.left
4       u.p.left = v
5   else u.p.right = v
6   if v ≠ NIL
7       v.p = u.p
```




```
TREE-DELETE(T, z)
 1  if z.left == NIL
 2      TRANSPLANT(T, z, z.right)
 3  elseif z.right == NIL
 4      TRANSPLANT(T, z, z.left)
 5  else y = TREE-MINIMUM(z.right)        successor
 6      if y.p ≠ z                        not right child
 7          TRANSPLANT(T, y, y.right)
 8          y.right = z.right
 9          y.right.p = y
10      TRANSPLANT(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

4.4 Red-Black Trees

Red-black trees are one of many search-tree schemes that are "balanced" in order to guarantee that basic dynamic-set operations take $O(\lg n)$ time in the worst case.

4.4.1 Properties of red-black trees

A red-black tree is a binary search tree with one extra bit of storage per node: its color, which can be either RED or BLACK . By constraining the node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other, so that the tree is approximately balanced.

```
Red-black properties:
1. Every node is either red or black.
2. The root is black.
3. Every leaf ( NIL ) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same
```

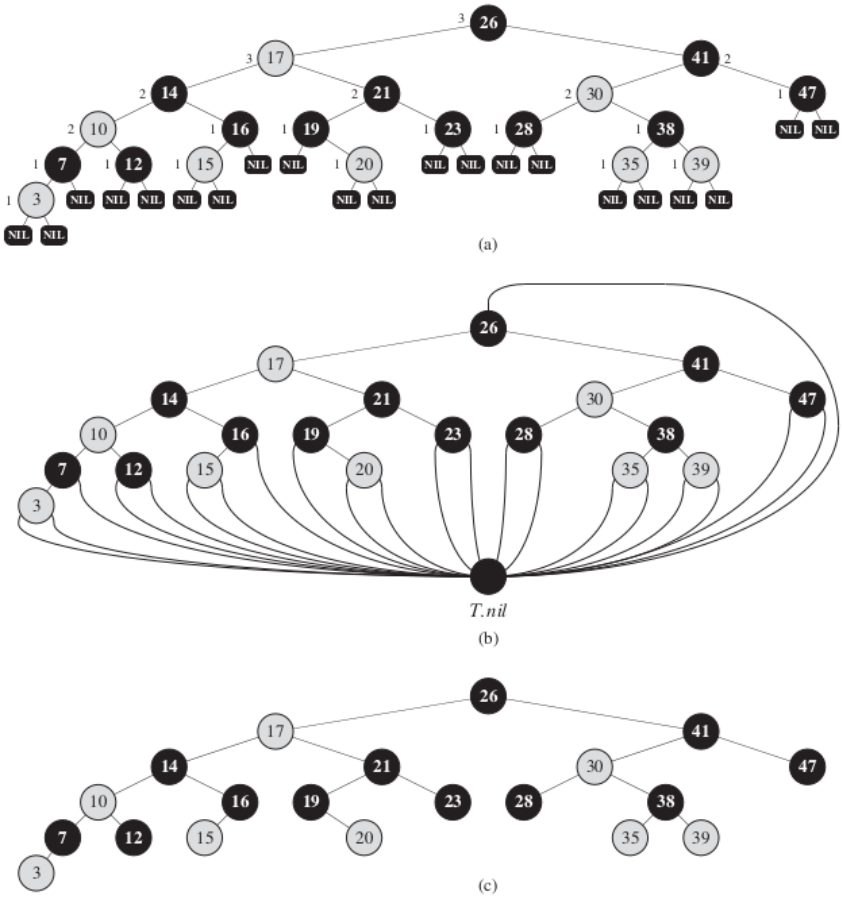


Figure 13.1 A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. **(a)** Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height; NILs have black-height 0. **(b)** The same red-black tree but with each NIL replaced by the single sentinel *T.nil*, which is always black, and with black-heights omitted. The root's parent is also the sentinel. **(c)** The same red-black tree but with leaves and the root's parent omitted entirely. We shall use this drawing style in the remainder of this chapter.

As a matter of convenience in dealing with boundary conditions in red-black tree code, we use a single sentinel to represent NIL. We use the sentinel so that we can treat a NIL child of a node x as an ordinary node whose parent is x.

black-height of the node: (denoted bh(x))

the number of black nodes on any simple path from, but not including, a node x down to a leaf

Lemma:
A red-black tree with n internal nodes has height at most $2\lg(n+1)$.

4.4.2 Rotations

ratation: a local operation in a search tree that preserves the binary-search-tree property. (run in O(1) time)

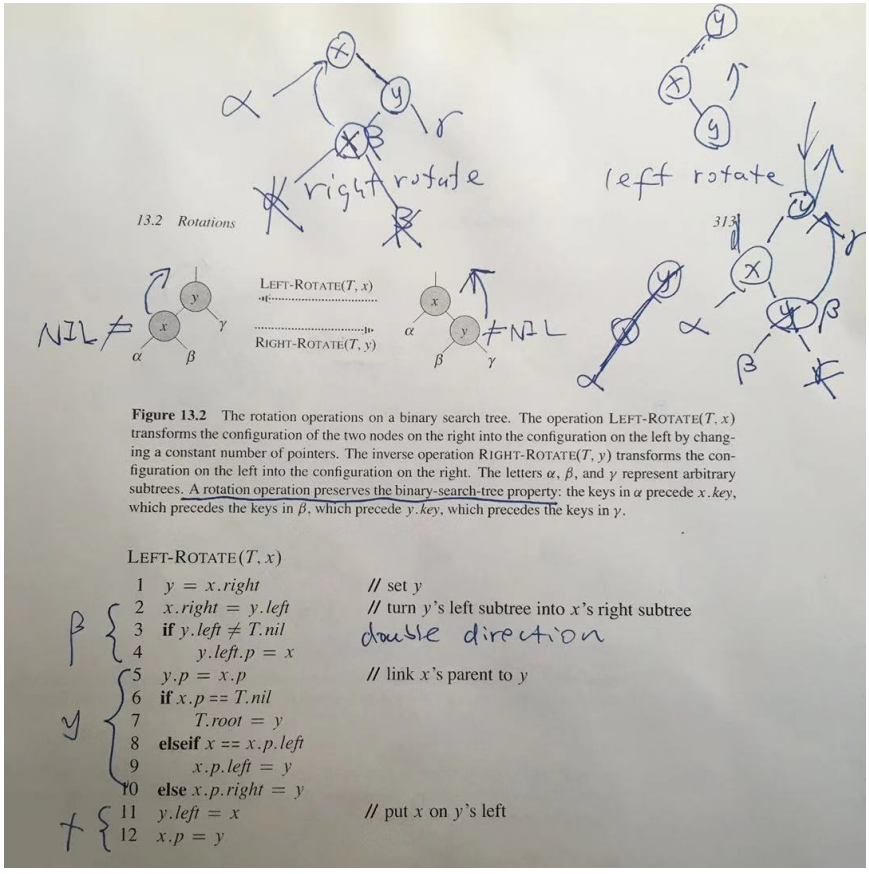
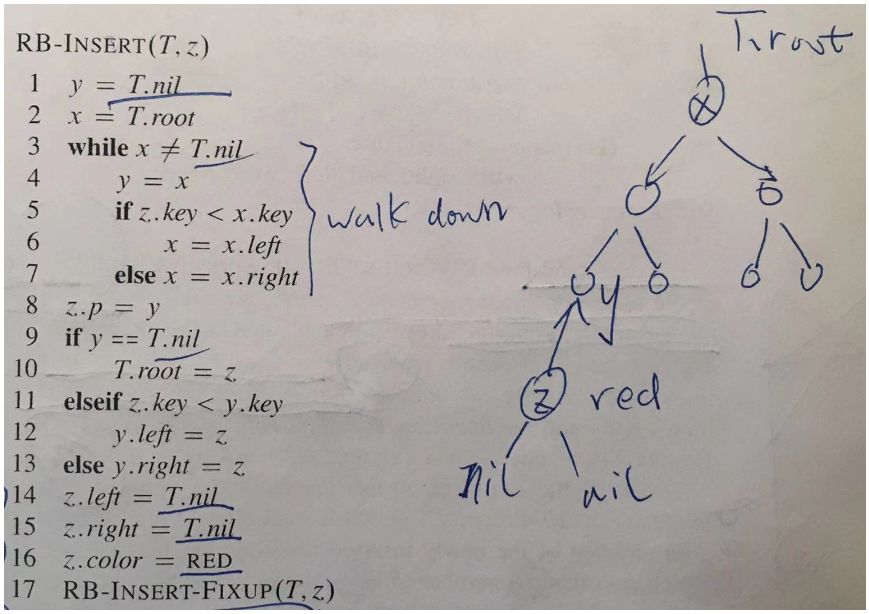


Figure 13.2 The rotation operations on a binary search tree. The operation LEFT-ROTATE(T, x) transforms the configuration of the two nodes on the right into the configuration on the left by changing a constant number of pointers. The inverse operation RIGHT-ROTATE(T, y) transforms the configuration on the left into the configuration on the right. The letters α, β, and γ represent arbitrary subtrees. A rotation operation preserves the binary-search-tree property: the keys in α precede $x.key$, which precedes the keys in β, which precede $y.key$, which precedes the keys in γ.

```
LEFT-ROTATE(T, x)
 1  y = x.right              // set y
 2  x.right = y.left         // turn y's left subtree into x's right subtree
 3  if y.left ≠ T.nil
 4      y.left.p = x
 5  y.p = x.p                // link x's parent to y
 6  if x.p == T.nil
 7      T.root = y
 8  elseif x == x.p.left
 9      x.p.left = y
10  else x.p.right = y
11  y.left = x               // put x on y's left
12  x.p = y
```

4.4.3 Insertion




```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2      if z.p == z.p.p.left
 3          y = z.p.p.right
 4          if y.color == RED
 5              z.p.color = BLACK                    // case 1
 6              y.color = BLACK                      // case 1
 7              z.p.p.color = RED                    // case 1
 8              z = z.p.p                            // case 1
 9          else if z == z.p.right
10                  z = z.p                          // case 2
11                  LEFT-ROTATE(T, z)                // case 2
12              z.p.color = BLACK                    // case 3
13              z.p.p.color = RED                    // case 3
14              RIGHT-ROTATE(T, z.p.p)               // case 3
15      else (same as then clause
                with "right" and "left" exchanged)
16  T.root.color = BLACK
```

To understand how RB-INSERT-FIXUP works, we break the code into 3 major steps.

1. determine the violations of the red-black properties

2. examine the overall goal of the while loop

3. explore each of the three cases

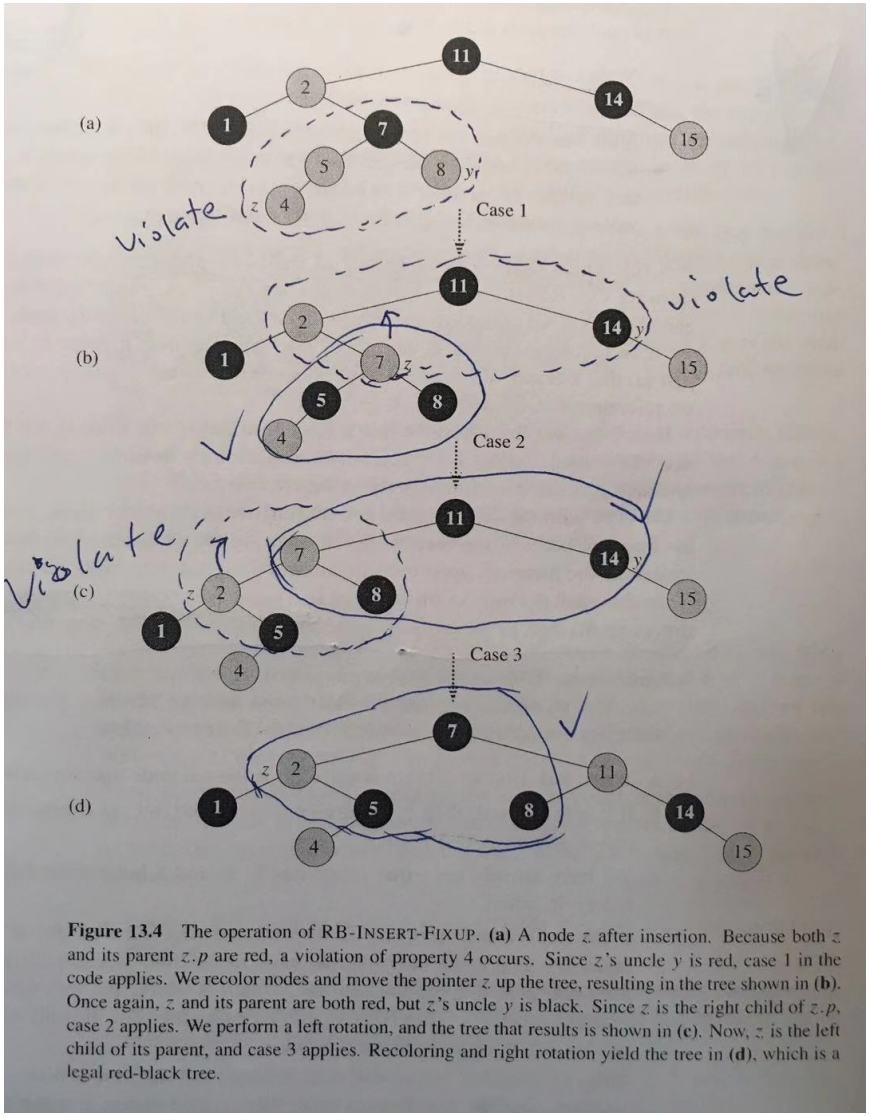


Figure 13.4 The operation of RB-INSERT-FIXUP. **(a)** A node z after insertion. Because both z and its parent $z.p$ are red, a violation of property 4 occurs. Since z's uncle y is red, case 1 in the code applies. We recolor nodes and move the pointer z up the tree, resulting in the tree shown in **(b)**. Once again, z and its parent are both red, but z's uncle y is black. Since z is the right child of $z.p$, case 2 applies. We perform a left rotation, and the tree that results is shown in **(c)**. Now, z is the left child of its parent, and case 3 applies. Recoloring and right rotation yield the tree in **(d)**, which is a legal red-black tree.

4.4.4 Deletion

RB-TRANSPLANT(T, u, v)

```
1  if u.p == T.nil
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  v.p = u.p
```

```
RB-DELETE(T, z)
 1  y = z
 2  y-original-color = y.color
 3  if z.left == T.nil
 4       x = z.right
 5       RB-TRANSPLANT(T, z, z.right)
 6  elseif z.right == T.nil
 7       x = z.left
 8       RB-TRANSPLANT(T, z, z.left)
 9  else y = TREE-MINIMUM(z.right)
10       y-original-color = y.color
11       x = y.right
12       if y.p == z
13            x.p = y
14       else RB-TRANSPLANT(T, y, y.right)
15            y.right = z.right
16            y.right.p = y
17       RB-TRANSPLANT(T, z, y)
18       y.left = z.left
19       y.left.p = y
20       y.color = z.color
21  if y-original-color == BLACK
22       RB-DELETE-FIXUP(T, x)
```

y: as the node either removed from the tree or moved within the tree;
x: moves into node y's original position;
node y's color might change, the variable y-original-color stores y's color before any

```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ T.root and x.color == BLACK
 2      if x == x.p.left
 3          w = x.p.right
 4          if w.color == RED
 5              w.color = BLACK                          // case 1
 6              x.p.color = RED                          // case 1
 7              LEFT-ROTATE(T, x.p)                      // case 1
 8              w = x.p.right                            // case 1
 9          if w.left.color == BLACK and w.right.color == BLACK
10              w.color = RED                            // case 2
11              x = x.p                                  // case 2
12          else if w.right.color == BLACK
13                  w.left.color = BLACK                 // case 3
14                  w.color = RED                        // case 3
15                  RIGHT-ROTATE(T, w)                   // case 3
16                  w = x.p.right                        // case 3
17              w.color = x.p.color                      // case 4
18              x.p.color = BLACK                        // case 4
19              w.right.color = BLACK                    // case 4
20              LEFT-ROTATE(T, x.p)                      // case 4
21              x = T.root                               // case 4
22      else (same as then clause with "right" and "left" exchanged)
23  x.color = BLACK
```

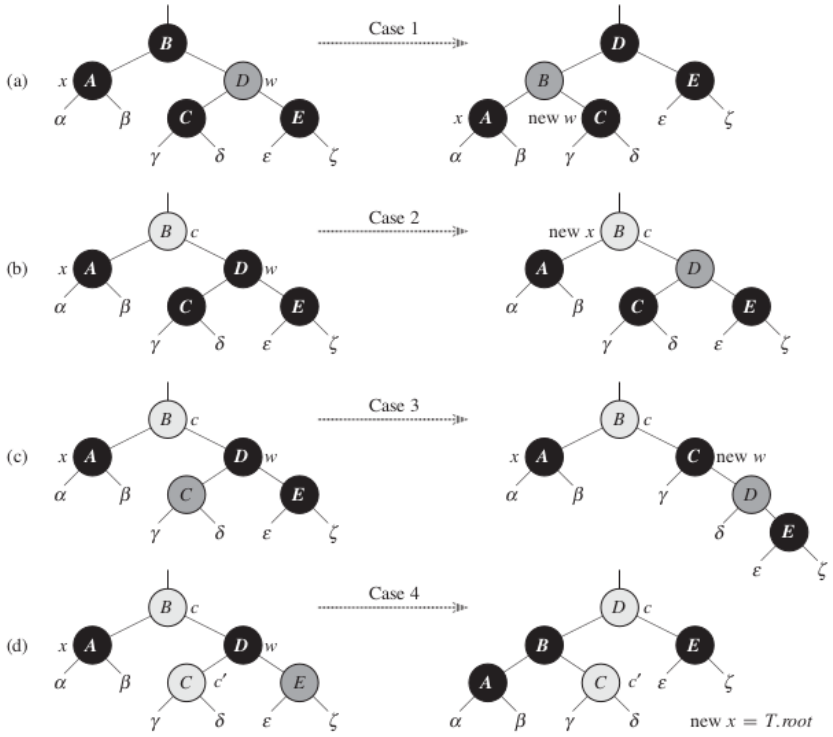


Figure 13.7 The cases in the **while** loop of the procedure RB-DELETE-FIXUP. Darkened nodes have *color* attributes BLACK, heavily shaded nodes have *color* attributes RED, and lightly shaded nodes have *color* attributes represented by c and c', which may be either RED or BLACK. The letters $\alpha, \beta, \ldots, \zeta$ represent arbitrary subtrees. Each case transforms the configuration on the left into the configuration on the right by changing some colors and/or performing a rotation. Any node pointed to by x has an extra black and is either doubly black or red-and-black. Only case 2 causes the loop to repeat. **(a)** Case 1 is transformed to case 2, 3, or 4 by exchanging the colors of nodes B and D and performing a left rotation. **(b)** In case 2, the extra black represented by the pointer x moves up the tree by coloring node D red and setting x to point to node B. If we enter case 2 through case 1, the **while** loop terminates because the new node x is red-and-black, and therefore the value c of its *color* attribute is RED. **(c)** Case 3 is transformed to case 4 by exchanging the colors of nodes C and D and performing a right rotation. **(d)** Case 4 removes the extra black represented by x by changing some colors and performing a left rotation (without violating the red-black properties), and then the loop terminates.

If y is red, the red-black properties still hold when y is removed or moved

If node y was black, three problems may arise:
1. y had been the root and a red child of y because the new root (property 2 violated)
2. if both x and x.p are red (property 4 violated)
3. moving y within the tree causes any simple path that privously contained y to have

We can correct the violation of property 5 by saying that node x, now occupying y's original position, has an "extra" black. That is, if we add 1 to the count of black nodes on any simple path that contains x, then under this interpretation, property 5 holds. When we remove or move the black node y, we "push" its blackness onto node x. The problem is that now node x is neither red nor black, thereby violating property 1.

4.5 Augmenting Data Structures

4.5.1 Dynamic order statistics

An order-statistic tree T is simply a red-black tree with additional information stored in each node.

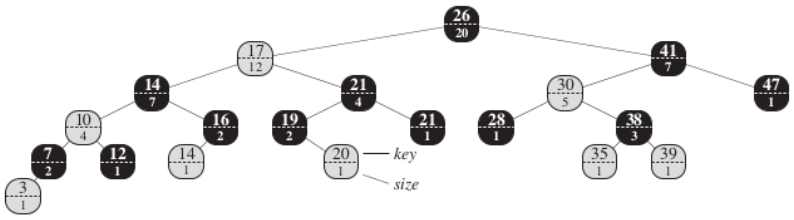


Figure 14.1 An order-statistic tree, which is an augmented red-black tree. Shaded nodes are red, and darkened nodes are black. In addition to its usual attributes, each node x has an attribute $x.size$, which is the number of nodes, other than the sentinel, in the subtree rooted at x.

Besides the usual red-black tree attributes x.key, x.color, x.p, x.left, and x.right in a node x, we have another attribute, x.size. This attribute contains the number of (internal) nodes in the subtree rooted at x (including x itself), that is, the size of the subtree.

```
OS-SELECT(x, i)
1  r = x.left.size + 1
2  if i == r
3       return x
4  elseif i < r
5       return OS-SELECT(x.left, i)
6  else return OS-SELECT(x.right, i −
```

1. Retrieving an element with a given rank

```
OS-RANK(T, x)
1  r = x.left.size + 1
2  y = x
3  while y ≠ T.root
4       if y == y.p.right
5            r = r + y.p.left.size + 1
6       y = y.p
7  return r
```

2. Determing the rank of an element

3. Maintaing subtree sizes

4.5.2 How to augment a data structure

We can break the process of augmenting a data structure into four steps:

1. Choose an underlying data structure.

2. Determine additional information to maintain in the underlying data structure.

3. Verify that we can maintain the additional information for the basic modifying operations on the underlying data structure.

4. Develop new operations.

As with any prescriptive design method, you should not blindly follow the steps in the order given. Most design work contains an element of trial and error, and progress on all steps usually proceeds in parallel. There is no point, for example, in determining additional information and developing new operations (steps 2 and 4) if we will not be able to maintain the additional information efficiently. Nevertheless, this four-step method provides a good focus for your efforts in augmenting a data structure, and it is also a good way to organize the documentation of an augmented data structure.

Theroem (augmenting a red-black tree)

Let f be an attribute that augments a red-black tree T of n nodes, and suppose that the value of f for each node x depends on only the information in nodes x, x.left, and x.right, possibly including x.left.f and x.right.f. Then, we can maintain the values of f in all nodes of T during insertion and deletion without asymptotically affecting the O(lgn) performance of these operations.

4.5.3 Interval trees

A closed interval is an ordered pair of real numbers $[t_1, t_2]$ with $t_1 \leq t_2$. The interval $[t_1, t_2]$ represents the set $\{t \in \mathbb{R} : t_1 \leq t \leq t_2\}$.

Any two intervals i and i' satisfy the interval trichotomy; that is, exactly one of the following three properties holds:

1. i and i' overlap

2. i is to the left of i'

3. i is to the right of i'

An interval tree is a red-black tree that maintains a dynamic set of elements, with each element x containing an interval x.int.

Interval trees support the following operations:

INTERVAL-INSERT(T, x) adds the element x, whose *int* attribute is assumed to contain an interval, to the interval tree T.

INTERVAL-DELETE(T, x) removes the element x from the interval tree T.

INTERVAL-SEARCH(T, i) returns a pointer to an element x in the interval tree T such that $x.int$ overlaps interval i, or a pointer to the sentinel $T.nil$ if no such element is in the set.

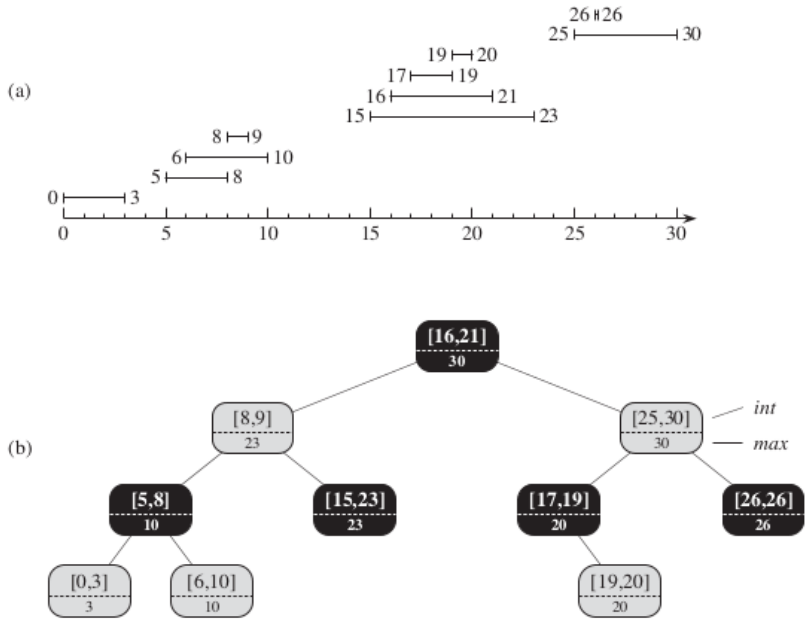


Figure 14.4 An interval tree. **(a)** A set of 10 intervals, shown sorted bottom to top by left endpoint. **(b)** The interval tree that represents them. Each node x contains an interval, shown above the dashed line, and the maximum value of any interval endpoint in the subtree rooted at x, shown below the dashed line. An inorder tree walk of the tree lists the nodes in sorted order by left endpoint.

$$x.max = \max(x.int.high, x.left.max, x.right.max) .$$

Thus, by Theorem 14.1, insertion and deletion run in $O(\lg n)$ time. In fact, we can update the *max* attributes after a rotation in $O(1)$ time, as Exercises 14.2-3 and 14.3-1 show.

```
INTERVAL-SEARCH(T, i)
1  x = T.root
2  while x ≠ T.nil and i does not overlap x.int
3      if x.left ≠ T.nil and x.left.max ≥ i.low
4          x = x.left
5      else x = x.right
6  return x
```

5 Advanced Design and Analysis Techniques

Dynamic programming typically applies to optimization problems in which we make a set of choices in order to arrive at an optimal solution. As we make each choice, subproblems of the same form often arise. Dynamic programming is effective when a given subproblem may arise from more than one partial set of choices; the key technique is to store the solution to each such subproblem in case it should reappear.

Greedy algorithms typically apply to optimization problems in which we make a set of choices in order to arrive at an optimal solution. The idea of a greedy algorithm is to make each choice in a locally optimal manner.

We use amortized analysis to analyze certain algorithms that perform a sequence of similar operations. Instead of bounding the cost of the sequence of operations by bounding the actual cost of each operation separately, an amortized analysis provides a bound on the actual cost of the entire sequence. One advantage of this approach is that although some operations might be expensive, many others might be cheap.

5.1 Dynamic Programming

Dynamic programming solves problems by combining the solutions to subproblems. ("programming" in this context refers to a tabular method, not

to writing computer code.)
Dynamic programming applies when the subproblems overlap.
A dynamic programming algorithm solve each subproblem just once and then save its answer in a table, thereby avoiding the work of recomputing the answer every time is solves each subproblem.

We typically apply dynamic programming to optimization problems.

When developing a dynamic programming algorithm, we follow a sequence of four steps:

1. characterize the structure of an optimal solution

2. recursivly define the value of an optimal solution

3. compute the value of an optimal solution, typically in a bottom-up fashion

4. construct an optimal solution from computed information

5.1.1 Rod cutting

The rod-cutting problem:
Given a rod of length n inches and a table of price p_i for $i = 1, 2, ..., n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

We can cut up a rod of length n in 2^{n-1} different ways.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Figure 15.1 A sample price table for rods. Each rod of length i inches earns the company p_i dollars of revenue.

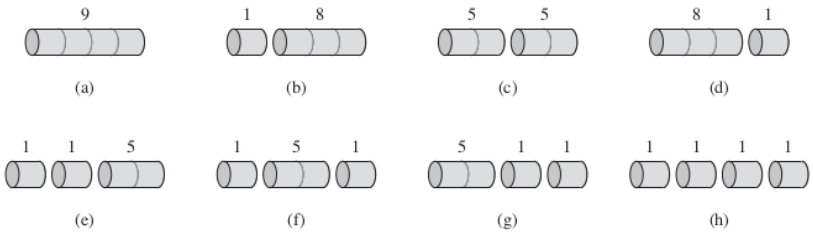


Figure 15.2 The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart of Figure 15.1. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

If an optimal solution cuts the rod into k pieces, for some $1 \leq k \leq n$, then an optimal decomposition
$n = i_1 + i_2 + ... + i_k$
of the rod into pieces of lengths $i_1, i_2, ..., i_k$ provides maximum corresponding revenue
$r_n = p_{i_1} + p_{i_2} + ... + p_{i_k}$.

Frame the values r_n for $n \geq 1$ in terms of optimal revenues from shorter rods:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, ..., r_{n-1} + r_1) \tag{12}$$

To solve the original problem of size n, we solve smaller problems of the same type, but of smaller sizes. The overall optimal solution incorporates optimal solution to the two related subproblems, maximizing revenue from each of those two pieces.

We say that the rod-cutting problem exhibits **optimal substructure**: optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently.

In a related, but slightly simpler, way to arrange a recursive structure for the rod-cutting problem, we view a decomposition as consisting of a first piece of length i cut off the left-hand end, and then a right-hand remainder of length $n - i$. Only the remainder, and not the first piece, may be further divided. We thus obtain the following simpler version:

$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i}). \tag{13}$$

In this formulation, an optimal solution embodies the solution to only one related subproblem—the remainder—rather than two.

1. Recursive top-down implementation

```
CUT-ROD(p, n)
1  if n == 0
2       return 0
3  q = -∞
4  for i = 1 to n
5       q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

Why is CUT-ROD so inefficient?
The problem is that CUT-ROD calls itself recursively over and over again with the same parameter values; it solves the same subproblems repeatedly.

2. Using dynamic programming for optimal rod cutting The dynamic-programming method works as follows. Having observed that a naive recursive solution is inefficient because it solves the same subproblems repeatedly, we arrange for each subproblem to be solved only once, saving its solution. Dynamic programming thus uses additional memory to save computation time; it serves an example of a time-memory trade-off. The savings may be dramatic: an exponential-time solution may be transformed into a polynomial-time solution. A dynamic-programming approach runs in polynomial time when the number of distinct subproblems involved is polynomial in the input size and we can solve each such subproblem in polynomial time.