Asymptotic notation, divide and conquer, and elementary data structures

Congduan Li

Chinese University of Hong Kong, Shenzhen

congduan.li@gmail.com

Sep 12 & 14, 2017

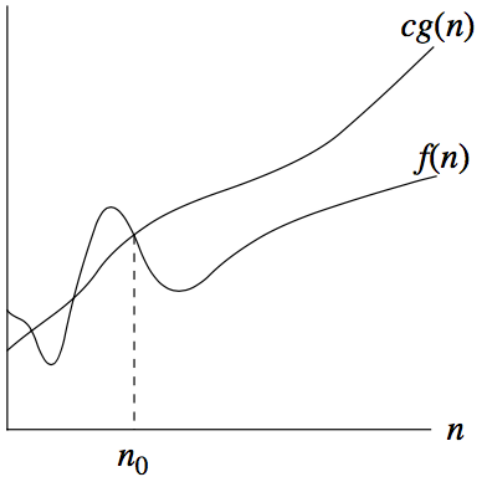
Overview of Chap 3

- A way to describe behavior of functions in the limit. We're studying asymptotic efficiency.
- Describe growth of functions.
- Focus on what's important by abstracting away low-order terms and constant factors.
- How we indicate running times of algorithms.

Asymptotic notation: $O \leftrightarrow \leq$

O-notation

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



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

If $f(n) \in O(g(n))$, we write $f(n) = O(g(n))$ (will precisely explain this soon).

Asymptotic notation: $O \leftrightarrow \leq$

Example: $2n^2 = O(n^3)$, with $c = 1$ and $n_0 = 2$.

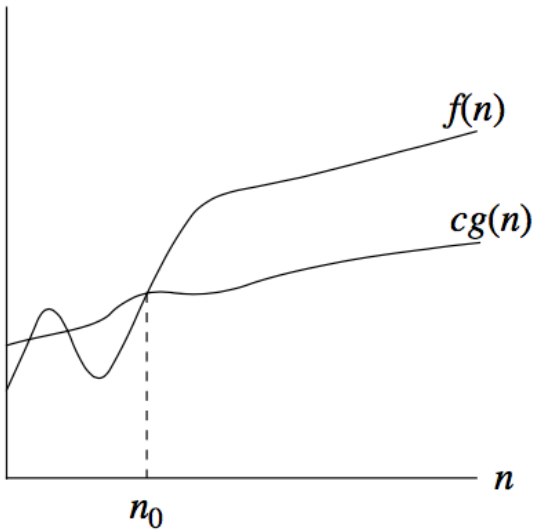
Examples of functions in $O(n^2)$:

n^2
$n^2 + n$
$n^2 + 1000n$
$1000n^2 + 1000n$
Also,
n
$n/1000$
$n^{1.99999}$
$n^2/\lg\lg\lg n$

Asymptotic notation: $\Omega \leftrightarrow \geq$

Ω-notation

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



Asymptotic notation: $\Omega \leftrightarrow \geq$

Example: $\sqrt{n} = \Omega(\lg n)$, with $c = 1$ and $n_0 = 16$.

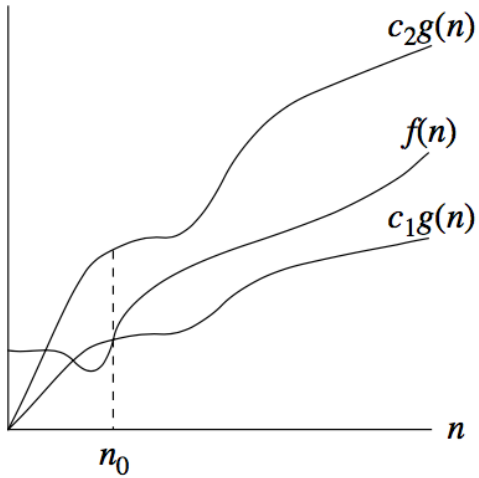
Examples of functions in $\Omega(n^2)$:

n^2
$n^2 + n$
$n^2 - n$
$1000n^2 + 1000n$
$1000n^2 - 1000n$
Also,
n^3
$n^{2.00001}$
$n^2 \lg\lg\lg n$
2^{2^n}

Asymptotic notation: $\Theta \leftrightarrow =$

Θ-notation

$\Theta(g(n)) = \{f(n) :$ there exist positive constants c_1, c_2, and n_0 such that
$0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$.



$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Example: $n^2/2 - 2n = \Theta(n^2)$, with $c_1 = 1/4$, $c_2 = 1/2$, and $n_0 = 8$.

Theorem
$f(n) = \Theta(g(n))$ if and only if $f = O(g(n))$ and $f = \Omega(g(n))$.

Leading constants and low-order terms don't matter.

Asymptotic notation in equations

When on right-hand side: $O(n^2)$ stands for some anonymous function in the set $O(n^2)$.

$2n^2+3n+1 = 2n^2+\Theta(n)$ means $2n^2+3n+1 = 2n^2+f(n)$ *for some* $f(n) \in \Theta(n)$.
In particular, $f(n) = 3n + 1$.

By the way, we interpret # of anonymous functions as = # of times the asymptotic notation appears:

$\sum_{i=1}^{n} O(i)$ OK: 1 anonymous function

$O(1) + O(2) + \cdots + O(n)$ not OK: n hidden constants
$\Rightarrow$ no clean interpretation

Asymptotic notation in equations

When on left-hand side: No matter how the anonymous functions are chosen on the left-hand side, there is a way to choose the anonymous functions on the right-hand side to make the equation valid.

Interpret $2n^2 + \Theta(n) = \Theta(n^2)$ as meaning *for all* functions $f(n) \in \Theta(n)$, there exists a function $g(n) \in \Theta(n^2)$ such that $2n^2 + f(n) = g(n)$.

Can chain together:

$$\begin{aligned} 2n^2 + 3n + 1 &= 2n^2 + \Theta(n) \\ &= \Theta(n^2) \, . \end{aligned}$$

Interpretation:

- First equation: There exists $f(n) \in \Theta(n)$ such that $2n^2 + 3n + 1 = 2n^2 + f(n)$.
- Second equation: For all $g(n) \in \Theta(n)$ (such as the $f(n)$ used to make the first equation hold), there exists $h(n) \in \Theta(n^2)$ such that $2n^2 + g(n) = h(n)$.

o-notation $\leftrightarrow <$

o-notation

$o(g(n)) = \{f(n) :$ for all constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$.

Another view, probably easier to use: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

$n^{1.9999} = o(n^2)$
$n^2/\lg n = o(n^2)$
$n^2 \neq o(n^2)$ (just like $2 \not< 2$)
$n^2/1000 \neq o(n^2)$

ω-notation $\leftrightarrow >$

ω-notation

$\omega(g(n)) = \{f(n) :$ for all constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0\}$.

Another view, again, probably easier to use: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

$n^{2.0001} = \omega(n^2)$
$n^2 \lg n = \omega(n^2)$
$n^2 \neq \omega(n^2)$

$n^2 \lg n = c\, n^2$

always find $\lg n > c$

Comparisons of functions

Relational properties:

Transitivity:
$f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$.
Same for O, Ω, o, and ω.

Reflexivity:
$f(n) = \Theta(f(n))$.
Same for O and Ω.

Symmetry:
$f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.

Transpose symmetry:
$f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.
$f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$.

Comparisons:

- $f(n)$ is ***asymptotically smaller*** than $g(n)$ if $f(n) = o(g(n))$.
- $f(n)$ is ***asymptotically larger*** than $g(n)$ if $f(n) = \omega(g(n))$.

No trichotomy. Although intuitively, we can liken O to $\leq$, Ω to $\geq$, etc., unlike real numbers, where $a < b$, $a = b$, or $a > b$, we might not be able to compare functions.

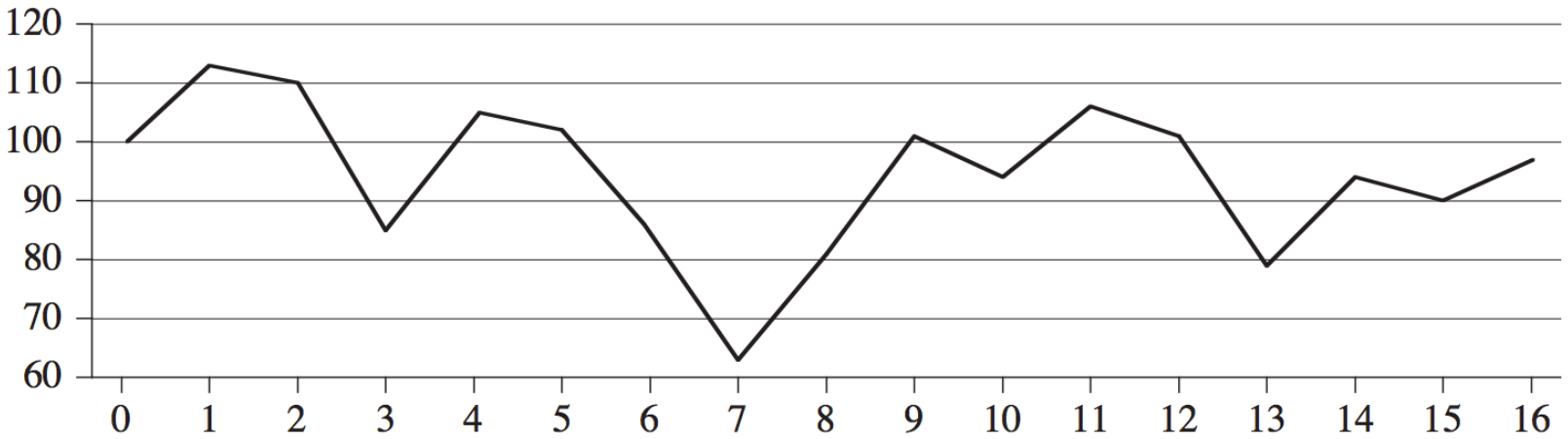
Example: $n^{1+\sin n}$ and n, since $1 + \sin n$ oscillates between 0 and 2.

Preview of Chapter 4

- learn more algorithms of divide and conquer: maximum subarray problem
- learn three methods to solve recurrence equations (inequalities): substitution, recursion tree, master's theorem

Maximum-subarray problem

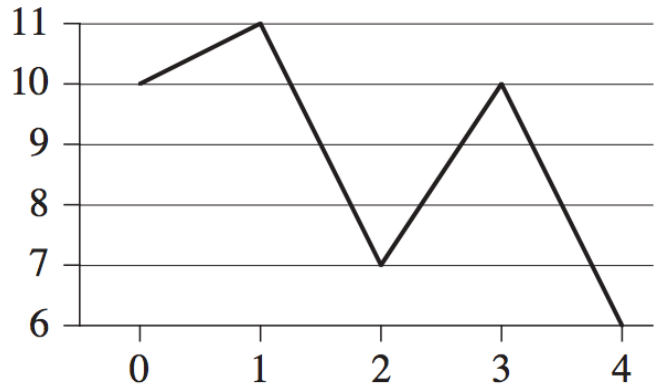
- consider to buy a stock given the price analysis or prediction curve
- to buy low and sell high



Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

Maximum-subarray problem

- Highest price? lowest price?
- might be neither of them



Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	−4	3	−4

A brute-force solution

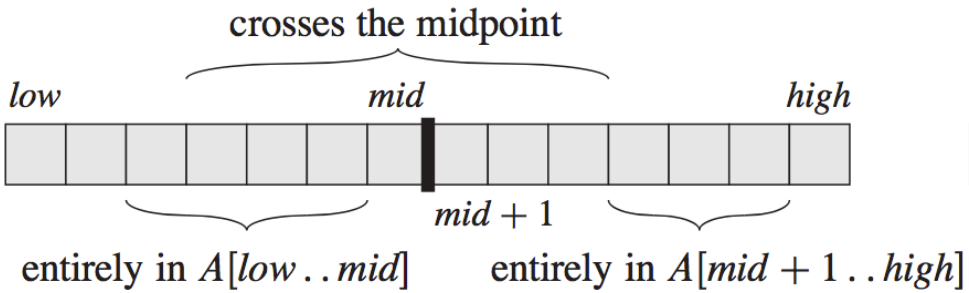
- We can easily devise a brute-force solution to this problem: just try every possible pair of buy and sell dates in which the buy date precedes the sell date.
- A period of n days has n such pairs of dates. Since $\binom{n}{2}$ is $\Theta(n^2)$ and the best we can hope for is to evaluate each pair of dates in constant time.
- this approach would take $\Theta(n^2)$ time.
- Can we do better? Answer is YES!

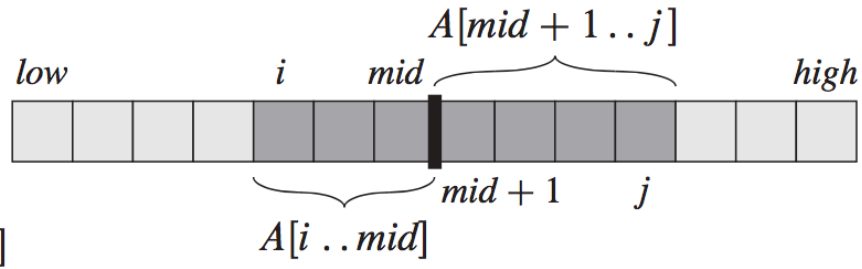
A transformation

- We want to find a sequence of days over which the net change from the first day to the last is maximum
- Instead of looking at the daily prices, let us instead consider the daily change in price and form a new array A
- we now want to find the nonempty, contiguous subarray of A whose values have the largest sum
- We call this contiguous subarray the **maximum subarray**

A divide-and-conquer solution

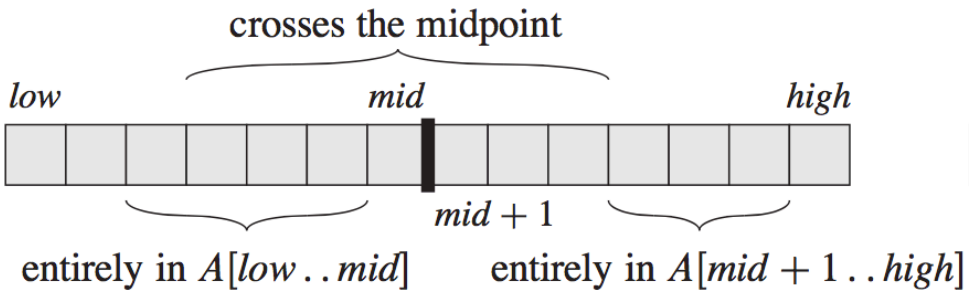
- Suppose we want to find a maximum subarray of the subarray $A[low..high]$.
- we find the midpoint, say mid, of the subarray, and consider the subarrays $A[low..mid]$ and $A[mid + 1..high]$
- any contiguous subarray $A[i..j]$ of $A[low..high]$ must lie in exactly one of the following places:
  - entirely in the subarray $A[low..mid]$, so that $low \le i \le j \le mid$,
  - entirely in the subarray $A[mid + 1..high]$,so that $mid \le i \le j \le high$, or
  - crossing the midpoint, so that $low \le i \le mid < j \le high$

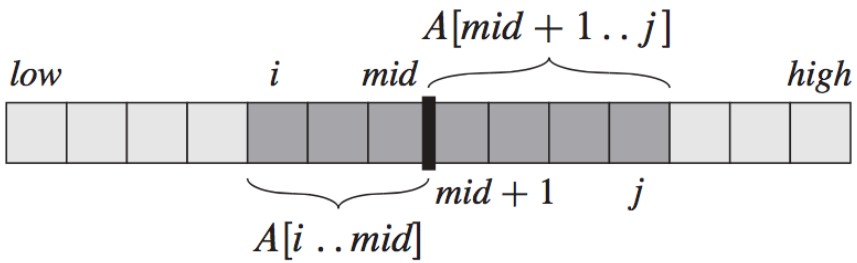




A divide-and-conquer solution

- We can solve the subarrays $A[low..mid]$ and $A[mid+1..high]$ recursively, since they are sub-problems of the original problem
- The only thing left is the case when the max subarray crosses the midpoint, so that $low \le i \le mid < j \le high$
- we then take the largest of these three and return
- The third case is not a smaller instance of our original problem, because it has the added restriction that the subarray it chooses must cross the midpoint
- Any subarray crossing the midpoint is itself made of two subarrays $A[i..mid]$ and $A[mid+1..j]$
- Therefore, we just need to find maximum subarrays of the form $A[i..mid]$ and $A[mid+1..j]$ and then combine them





Find max subarray crossing midpoint

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 1  left-sum = −∞
 2  sum = 0
 3  for i = mid downto low
 4       sum = sum + A[i]
 5       if sum > left-sum
 6            left-sum = sum
 7            max-left = i
 8  right-sum = −∞
 9  sum = 0
10  for j = mid + 1 to high
11       sum = sum + A[j]
12       if sum > right-sum
13            right-sum = sum
14            max-right = j
15  return (max-left, max-right, left-sum + right-sum)
```

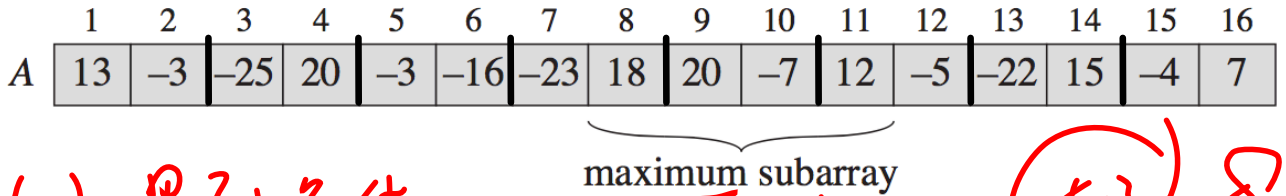
Running time $\Theta(n)$:
$(mid - low + 1) + (high - mid) = high - low + 1 = n$

Find max subarray crossing midpoint

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
 1  if high == low
 2       return (low, high, A[low])          // base case: only one element
 3  else mid = ⌊(low + high)/2⌋
 4       (left-low, left-high, left-sum) =
              FIND-MAXIMUM-SUBARRAY(A, low, mid)
 5       (right-low, right-high, right-sum) =
              FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
 6       (cross-low, cross-high, cross-sum) =
              FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
 7       if left-sum ≥ right-sum and left-sum ≥ cross-sum
 8            return (left-low, left-high, left-sum)
 9       elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
10            return (right-low, right-high, right-sum)
11       else return (cross-low, cross-high, cross-sum)
```

Example

Each iteration, find the midpoint and solve the left subarray, right subarray and crossing midpoint, respectively, and then combine them.



O(n) 遇到负值以正的开始算

Analyze running time

- The base case, when $n = 1$, is easy: line 2 takes constant time, and so $T(1) = \Theta(1)$
- Calculation midpoint, $\Theta(1)$, solve two subarrays $2T(n/2)$, solve the subarray crossing midpoint $\Theta(n)$, return $\Theta(1)$, thus the running time for $n > 1$: $T(n) = 2\Theta(1) + 2T(n/2) + \Theta(n) = 2T(n/2) + \Theta(n)$
- same as merge sort, solution: $T(n) = \Theta(n \lg n)$
- Faster than brute-force solution, divide and conquer is powerful
- See Exercise 4.1-5 for an even faster method for this problem, in linear time

Review recurrence

A ***recurrence*** is a function is defined in terms of

- one or more base cases, and
- itself, with smaller arguments.

Examples:

- $T(n) = \begin{cases} 1 & \text{if } n = 1\,, \\ T(n-1)+1 & \text{if } n > 1\,. \end{cases}$

  Solution: $T(n) = n$.

- $T(n) = \begin{cases} 1 & \text{if } n = 1\,, \\ 2T(n/2)+n & \text{if } n \geq 1\,. \end{cases}$

  Solution: $T(n) = n \lg n + n$.

- $T(n) = \begin{cases} 0 & \text{if } n = 2\,, \\ T(\sqrt{n})+1 & \text{if } n > 2\,. \end{cases}$

  Solution: $T(n) = \lg \lg n$.

- $T(n) = \begin{cases} 1 & \text{if } n = 1\,, \\ T(n/3)+T(2n/3)+n & \text{if } n > 1\,. \end{cases}$

  Solution: $T(n) = \Theta(n \lg n)$.

Substitution method

1. Guess the solution.
2. Use induction to find the constants and show that the solution works.

Example

$$T(n) = \begin{cases} 1 & \text{if } n = 1\,, \\ 2T(n/2) + n & \text{if } n > 1\,. \end{cases}$$

1. *Guess:* $T(n) = n \lg n + n$. *[Here, we have a recurrence with an exact function, rather than asymptotic notation, and the solution is also exact rather than asymptotic. We'll have to check boundary conditions and the base case.]*
2. *Induction:*

   Basis: $n = 1 \Rightarrow n \lg n + n = 1 = T(n)$

   Inductive step: Inductive hypothesis is that $T(k) = k \lg k + k$ for all $k < n$. We'll use this inductive hypothesis for $T(n/2)$.

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + n \\ &= 2\left(\frac{n}{2}\lg\frac{n}{2} + \frac{n}{2}\right) + n \qquad \text{(by inductive hypothesis)} \\ &= n\lg\frac{n}{2} + n + n \\ &= n(\lg n - \lg 2) + n + n \\ &= n\lg n - n + n + n \\ &= n\lg n + n\,. \end{aligned}$$

■

Assumptions

Generally, we use asymptotic notation:

- We would write $T(n) = 2T(n/2) + \Theta(n)$.
- We assume $T(n) = O(1)$ for sufficiently small n.
- We express the solution by asymptotic notation: $T(n) = \Theta(n \lg n)$.
- We don't worry about boundary cases, nor do we show base cases in the substitution proof.
  - $T(n)$ is always constant for any constant n.
  - Since we are ultimately interested in an asymptotic solution to a recurrence, it will always be possible to choose base cases that work.
  - When we want an asymptotic solution to a recurrence, we don't worry about the base cases in our proofs.
  - When we want an exact solution, then we have to deal with base cases.

Another example

For the substitution method:

- Name the constant in the additive term.
- Show the upper (O) and lower (Ω) bounds separately. Might need to use different constants for each.

Example: $T(n) = 2T(n/2)+\Theta(n)$. If we want to show an upper bound of $T(n) = 2T(n/2) + O(n)$, we write $T(n) \leq 2T(n/2) + cn$ for some positive constant c.

Prove upper bound

Guess: $T(n) \le dn \lg n$ for some positive constant d. We are given c in the recurrence, and we get to choose d as any positive constant. It's OK for d to depend on c.

Substitution:

$$\begin{aligned} T(n) &\le 2T(n/2) + cn \\ &= 2\left(d\frac{n}{2}\lg\frac{n}{2}\right) + cn \\ &= dn\lg\frac{n}{2} + cn \\ &= dn\lg n - dn + cn \\ &\le dn\lg n \qquad \text{if } -dn + cn \le 0\,, \\ &\qquad\qquad\qquad\qquad\quad d \ge c \end{aligned}$$

Therefore, $T(n) = O(n \lg n)$.

Prove lower bound

Guess: $T(n) \geq dn \lg n$ for some positive constant d.

Substitution:

$$\begin{aligned} T(n) &\geq 2T(n/2) + cn \\ &= 2\left(d\frac{n}{2}\lg\frac{n}{2}\right) + cn \\ &= dn\lg\frac{n}{2} + cn \\ &= dn\lg n - dn + cn \\ &\geq dn\lg n \qquad \text{if } -dn + cn \geq 0\,, \\ &\qquad\qquad\qquad\qquad\quad d \leq c \end{aligned}$$

Therefore, $T(n) = \Omega(n \lg n)$.

Caution: pitfalls!

Make sure you show the same *exact* form when doing a substitution proof.

Consider the recurrence

$$T(n) = 8T(n/2) + \Theta(n^2) .$$

For an upper bound:

$$T(n) \leq 8T(n/2) + cn^2 .$$

Guess: $T(n) \leq dn^3$.

$$\begin{aligned} T(n) &\leq 8d(n/2)^3 + cn^2 \\ &= 8d(n^3/8) + cn^2 \\ &= dn^3 + cn^2 \\ &\not\leq dn^3 \qquad \text{doesn't work!} \end{aligned}$$

Caution: pitfalls!

Remedy: *Subtract off* a lower-order term.

Guess: $T(n) \leq dn^3 - d'n^2$.

$$
\begin{aligned}
T(n) &\leq 8(d(n/2)^3 - d'(n/2)^2) + cn^2 \\
&= 8d(n^3/8) - 8d'(n^2/4) + cn^2 \\
&= dn^3 - 2d'n^2 + cn^2 \\
&= dn^3 - d'n^2 - d'n^2 + cn^2 \\
&\leq dn^3 - d'n^2 \qquad \text{if } -d'n^2 + cn^2 \leq 0\,, \\
&\qquad\qquad\qquad\qquad\qquad\quad d' \geq c
\end{aligned}
$$

Caution: pitfalls!

Be careful when using asymptotic notation.

The false proof for the recurrence $T(n) = 4T(n/4) + n$, that $T(n) = O(n)$:

$$\begin{aligned} T(n) &\leq 4(c(n/4)) + n \\ &\leq cn + n \\ &= O(n) \qquad \text{wrong!} \end{aligned}$$

Because we haven't proven the *exact form* of our inductive hypothesis (which is that $T(n) \leq cn$), this proof is false.

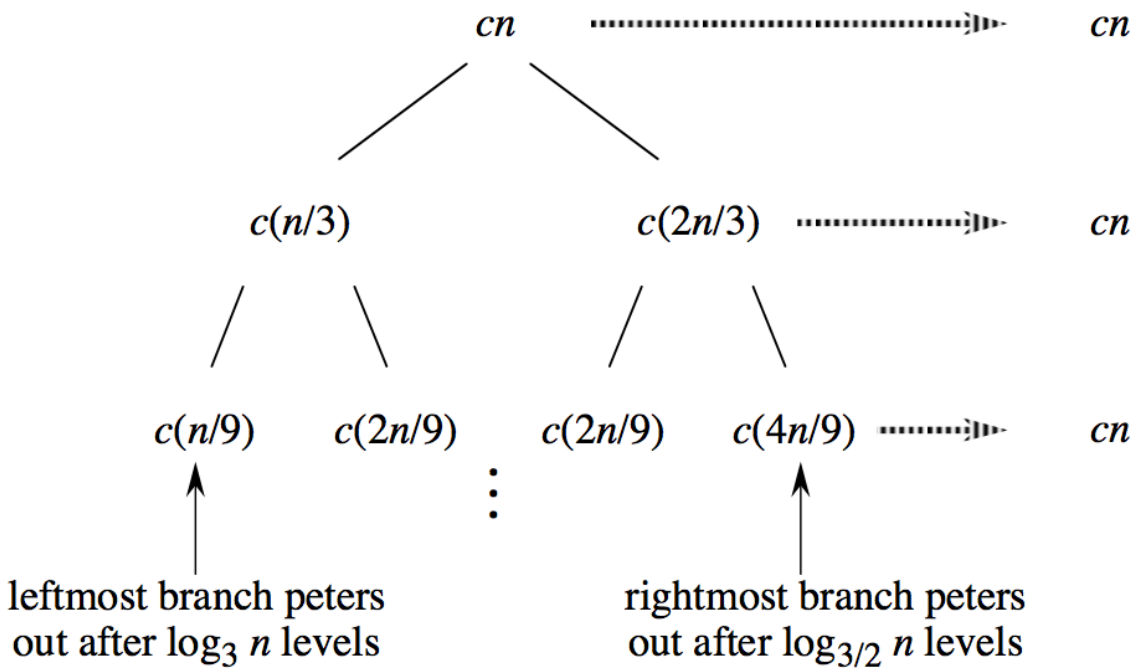
Recursion trees

Use to generate a guess. Then verify by substitution method.

Example: $T(n) = T(n/3)+T(2n/3)+\Theta(n)$. For upper bound, rewrite as $T(n) \leq T(n/3) + T(2n/3) + cn$; for lower bound, as $T(n) \geq T(n/3) + T(2n/3) + cn$.

By summing across each level, the recursion tree shows the cost at each level of recursion (minus the costs of recursive calls, which appear in subtrees):

Recursion trees



- There are $\log_3 n$ full levels, and after $\log_{3/2} n$ levels, the problem size is down to 1.
- Each level contributes $\leq cn$.
- Lower bound guess: $\geq dn \log_3 n = \Omega(n \lg n)$ for some positive constant d.
- Upper bound guess: $\leq dn \log_{3/2} n = O(n \lg n)$ for some positive constant d.
- Then *prove* by substitution.

Verification: formal proof

1. ***Upper bound:***

   Guess: $T(n) \le dn \lg n$.

   Substitution:

$$\begin{aligned}
T(n) &\le T(n/3) + T(2n/3) + cn \\
&\le d(n/3)\lg(n/3) + d(2n/3)\lg(2n/3) + cn \\
&= (d(n/3)\lg n - d(n/3)\lg 3) \\
&\qquad + (d(2n/3)\lg n - d(2n/3)\lg(3/2)) + cn \\
&= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg(3/2)) + cn \\
&= dn\lg n - d((n/3)\lg 3 + (2n/3)\lg 3 - (2n/3)\lg 2) + cn \\
&= dn\lg n - dn(\lg 3 - 2/3) + cn \\
&\le dn\lg n \qquad \text{if } -dn(\lg 3 - 2/3) + cn \le 0, \\
&\qquad\qquad\qquad\qquad\qquad\qquad d \ge \frac{c}{\lg 3 - 2/3}.
\end{aligned}$$

   Therefore, $T(n) = O(n \lg n)$.

   Note: Make sure that the symbolic constants used in the recurrence (e.g., c) and the guess (e.g., d) are different.

Verification: formal proof

2. ***Lower bound:***

   Guess: $T(n) \geq dn \lg n$.

   Substitution: Same as for the upper bound, but replacing $\leq$ by $\geq$. End up needing

   $$0 < d \leq \frac{c}{\lg 3 - 2/3} .$$

   Therefore, $T(n) = \Omega(n \lg n)$.

Since $T(n) = O(n \lg n)$ and $T(n) = \Omega(n \lg n)$, we conclude that $T(n) = \Theta(n \lg n)$. ■

Master method

Used for many divide-and-conquer recurrences of the form

$$T(n) = aT(n/b) + f(n) ,$$

where $a \geq 1$, $b > 1$, and $f(n) > 0$.

Based on the ***master theorem*** (Theorem 4.1).

Theorem 4.1 (Master theorem)
Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the nonnegative integers by the recurrence

$$T(n) = aT(n/b) + f(n) ,$$

where we interpret n/b to mean either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. 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 $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$. ■

Master method

Case 1: $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$.
($f(n)$ is polynomially smaller than $n^{\log_b a}$.)
Solution: $T(n) = \Theta(n^{\log_b a})$.
(Intuitively: cost is dominated by leaves.)

cost dominated by leaves

Case 2: $f(n) = \Theta(n^{\log_b a} \lg^k n)$, where $k \geq 0$.
[This formulation of Case 2 is more general than in Theorem 4.1, and it is given in Exercise 4.4-2.]
($f(n)$ is within a polylog factor of $n^{\log_b a}$, but not smaller.)
Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.
(Intuitively: cost is $n^{\log_b a} \lg^k n$ at each level, and there are $\Theta(\lg n)$ levels.)
Simple case: $k = 0 \Rightarrow f(n) = \Theta(n^{\log_b a}) \Rightarrow T(n) = \Theta(n^{\log_b a} \lg n)$.

Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$ and $f(n)$ satisfies the regularity condition $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.
($f(n)$ is polynomially greater than $n^{\log_b a}$.)
Solution: $T(n) = \Theta(f(n))$.
(Intuitively: cost is dominated by root.)

Whats with the Case 3 regularity condition?

Generally not a problem.
It always holds whenever $f(n) = n^k$ and $f(n) = (n \log_b a + \epsilon)$ for constant $\epsilon > 0$.
Proof: Since $f(n) = (n \log_b a + \epsilon)$ and $f(n) = n^k$, we have that $k > \log_b a$. Using a base of b and treating both sides as exponents, we have $b^k > b^{\log_b a} = a$, and so $a/b^k < 1$. Since a, b, and k are constants, if we let $c = a/b^k$, then c is a constant strictly less than 1. We have that $af(n/b) = a(n/b)^k = (a/b^k)n^k = cf(n)$, and so the regularity condition is satisfied.

Master method examples

- $T(n) = 5T(n/2) + \Theta(n^2)$
  $n^{\log_2 5}$ vs. n^2
  Since $\log_2 5 - \epsilon = 2$ for some constant $\epsilon > 0$, use Case 1 $\Rightarrow T(n) = \Theta(n^{\lg 5})$
- $T(n) = 27T(n/3) + \Theta(n^3 \lg n)$
  $n^{\log_3 27} = n^3$ vs. $n^3 \lg n$
  Use Case 2 with $k = 1 \Rightarrow T(n) = \Theta(n^3 \lg^2 n)$
- $T(n) = 5T(n/2) + \Theta(n^3)$
  $n^{\log_2 5}$ vs. n^3
  Now $\lg 5 + \epsilon = 3$ for some constant $\epsilon > 0$
  Check regularity condition (don't really need to since $f(n)$ is a polynomial):
  $af(n/b) = 5(n/2)^3 = 5n^3/8 \le cn^3$ for $c = 5/8 < 1$
  Use Case 3 $\Rightarrow T(n) = \Theta(n^3)$
- $T(n) = 27T(n/3) + \Theta(n^3/\lg n)$
  $n^{\log_3 27} = n^3$ vs. $n^3/\lg n = n^3 \lg^{-1} n \neq \Theta(n^3 \lg^k n)$ for any $k \ge 0$.
  Cannot use the master method.

Summary of Chap 4

Learnt algorithms for solving the max subarray problem: divide and conquer runs faster.
Learnt three methods for solving recursions: substitution, recursion tree, master.
Technical issues in solving recurrences

- Floors and ceilings: Floors and ceilings can easily be removed and don't affect the solution to the recurrence
- Exact vs. asymptotic functions: In algorithm analysis, we usually express both the recurrence and its solution using asymptotic notation.
- Boundary conditions:
  - The boundary conditions are usually expressed as "$T(n) = O(1)$ for sufficiently small n".
  - When we desire an exact, rather than an asymptotic, solution, we need to deal with boundary conditions.
  - In practice, we just use asymptotics most of the time, and we ignore boundary conditions.

Data Structures

- Dynamic sets: whereas mathematical sets are unchanging, the sets manipulated by algorithms can grow, shrink, or otherwise change over time.
- Dictionary: algorithms may require several different types of operations to be performed on sets, like insert elements into, delete elements from, and test membership in a set, we call a dynamic set that supports these operations a dictionary
- Key: one of the object?s attributes identifying it
- Satellite data: carried around in other object attributes but are otherwise unused by the set implementation; may also have attributes that are manipulated by the set operations; these attributes may contain data or pointers to other objects in the set

Operations on dynamic sets

- Queries: which simply return information about the set
- Modifying operations: 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 no such element belongs to S.

INSERT(S, x)

A modifying operation that augments the set S with the element pointed to by x. We usually assume that any attributes in element x needed by the set implementation have already been initialized.

DELETE(S, x)

A modifying operation that, given a pointer x to an element in the set S, removes x from S. (Note that this operation takes a pointer to an element x, not a key value.)

Operations on dynamic sets

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.

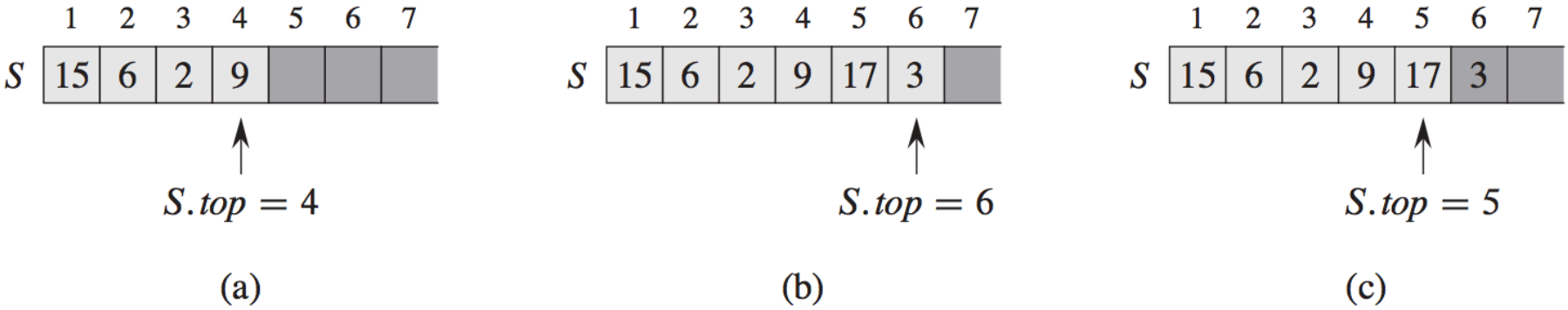
SUCCESSOR(S, x)
: A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next larger element in S, or NIL if x is the maximum element.

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.

Elementary Data Structures: Stacks

- 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.
- 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.

Elementary Data Structures: Stacks



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

Elementary Data Structures: Stacks

```
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
```

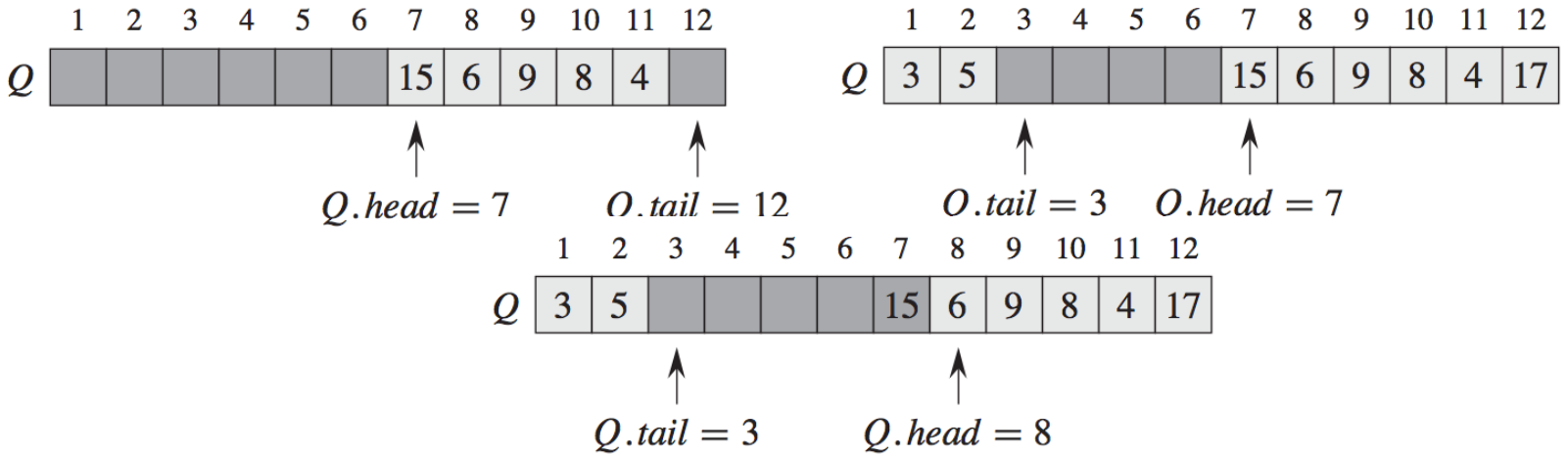
```
POP(S)
1  if STACK-EMPTY(S)
2       error "underflow"
3  else S.top = S.top − 1
4       return S[S.top + 1]
```

Each of the three stack operations takes $O(1)$ time.

Elementary Data Structures: Queues

- 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.
- The INSERT operation on a queue ENQUEUE, and we call the DELETE operation DEQUEUE; like the stack operation POP, DEQUEUE takes no element argument.
- The queue has a head and a tail.

Elementary Data Structures: Queues



- The queue has an attribute $Q.head$ that indexes, or points to, its head. The attribute $Q.tail$ indexes the next location at which a newly arriving element will be inserted into the queue. Wrap around is allowed.
- When $Q.head = Q.tail$, the queue is empty
- If we attempt to dequeue an element from an empty queue, the queue underflows.
- When $Q.head = Q.tail + 1$, the queue is full, and if we attempt to enqueue an element, then the queue overflows.

Elementary Data Structures: Queues

```
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
```

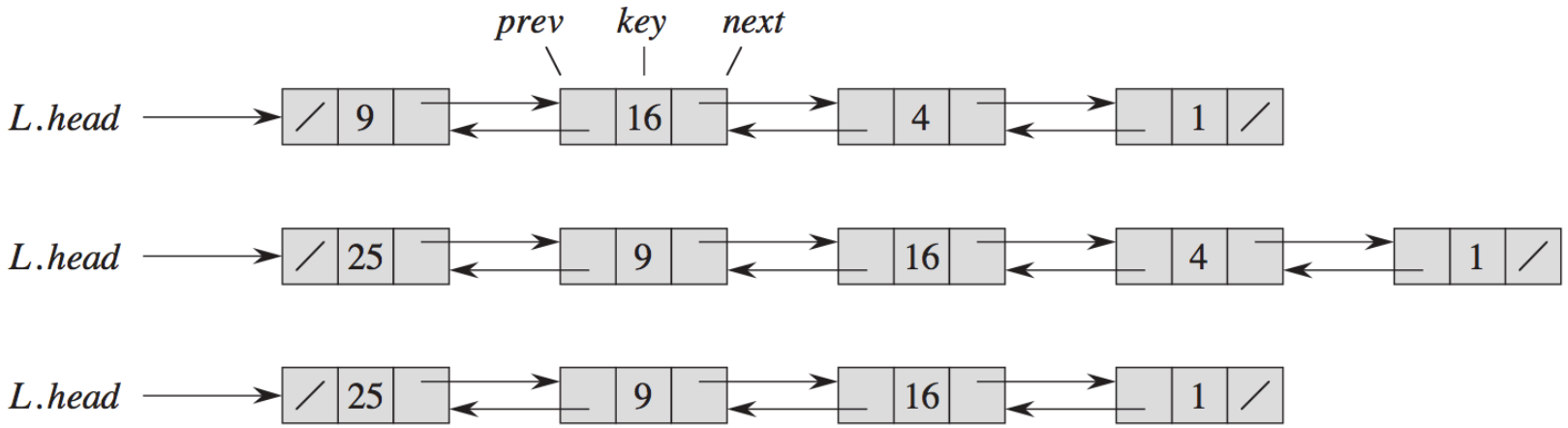
```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3        Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

Each of the queue operations takes $O(1)$ time.

Elementary Data Structures: 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
- each element of a doubly linked list L is an object with an attribute key and two other pointer attributes: $next$ and pre

Elementary Data Structures: Linked lists



- If $x.pre = NIL$, the element x has no predecessor and is therefore the first element, or head, of the list.
- If $x.next = NIL$, the element x has no successor and is therefore the last element, or tail, of the list.
- An attribute $L.head$ points to the first element of the list. If $L.head = NIL$, the list is empty.
- Lists: sorted vs unsorted, doubly linked vs singly linked, circular vs uncircular

Elementary Data Structures: Linked lists

LIST-SEARCH(L, k)

```
1  x = L.head
2  while x ≠ NIL and x.key ≠ k
3        x = x.next
4  return x
```

$\Theta(n)$

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
```

$O(1)$

LIST-DELETE(L, x)

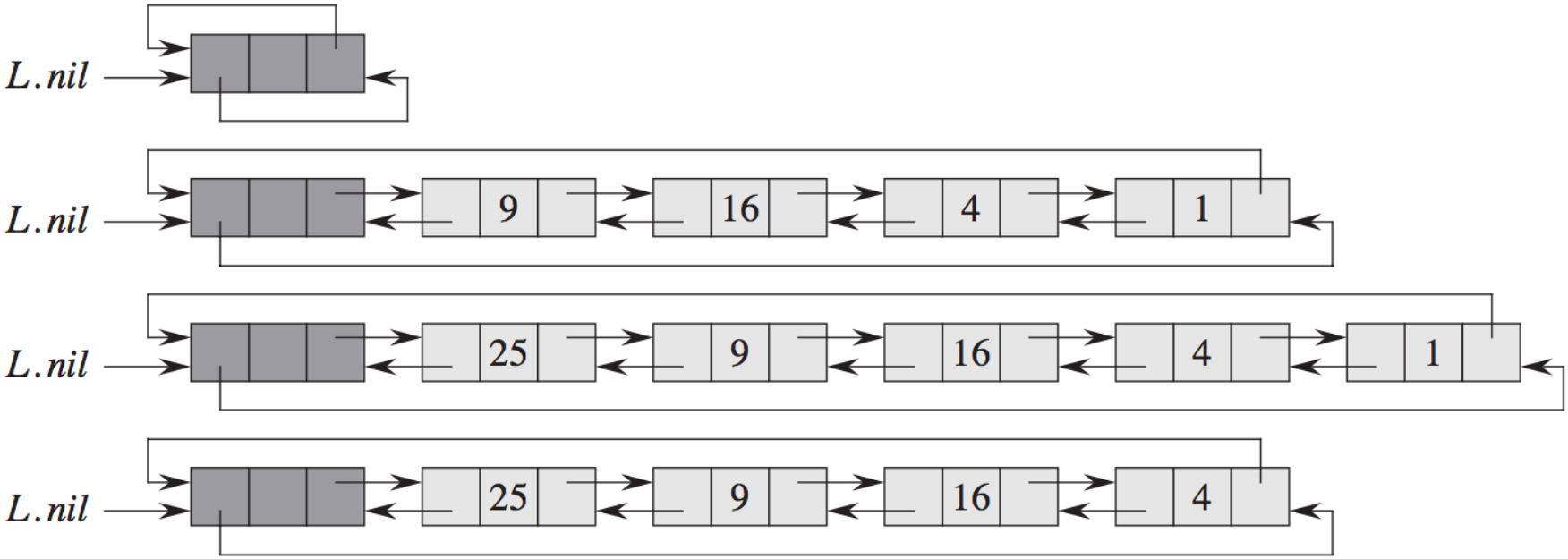
```
1  if x.prev ≠ NIL
2        x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5        x.next.prev = x.prev
```

$O(1)$

Linked lists with sentinel

- A sentinel is a dummy object that allows us to simplify boundary conditions.
- This change turns a regular doubly linked list into a circular, doubly linked list with a sentinel, in which the sentinel $L.nil$ lies between the head and tail.
- The attribute $L.nil.next$ points to the head of the list, and $L.nil.pre$ points to the tail.
- Both the $next$ attribute of the tail and the pre attribute of the head point to $L.nil$.

Elementary Data Structures: Linked lists



Codes of operations become simpler.

Elementary Data Structures: Linked lists

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
```

LIST-INSERT$'(L, x)$

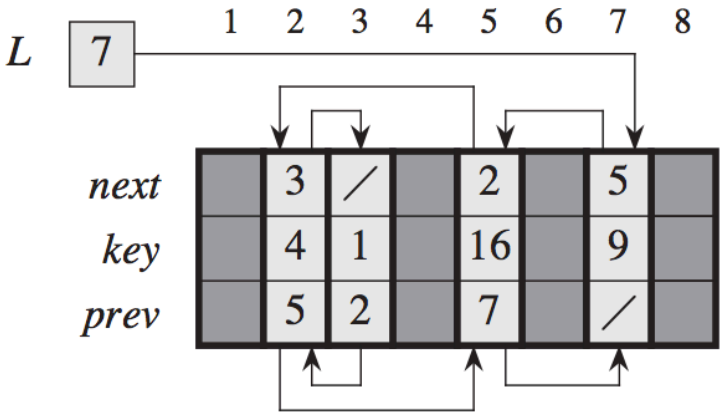
```
1  x.next = L.nil.next
2  L.nil.next.prev = x
3  L.nil.next = x
4  x.prev = L.nil
```

LIST-DELETE$'(L, x)$

```
1  x.prev.next = x.next
2  x.next.prev = x.prev
```

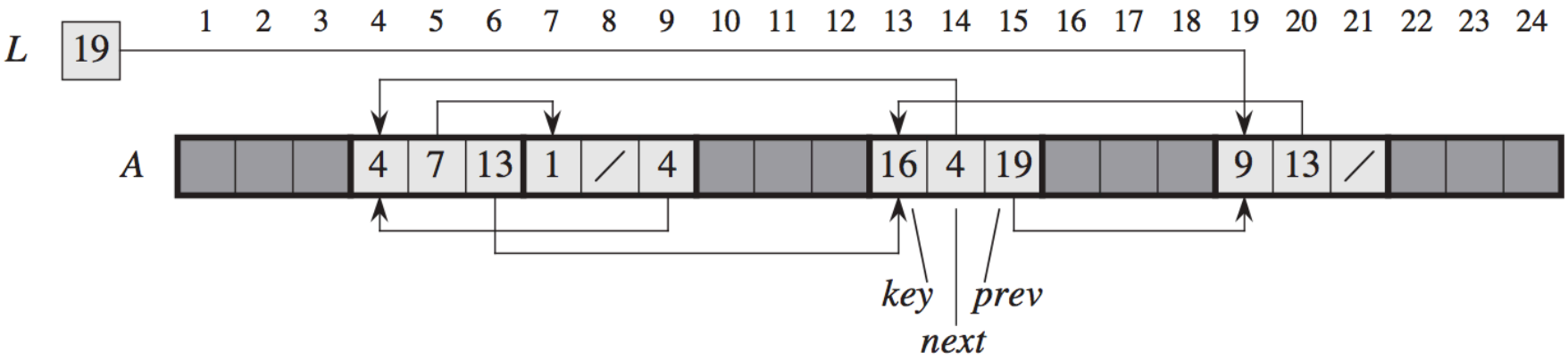
We should use sentinels judiciously. When there are many small lists, the extra storage used by their sentinels can represent significant wasted memory. We use sentinels only when they truly simplify the code.

Implementing pointers and objects: multi-array



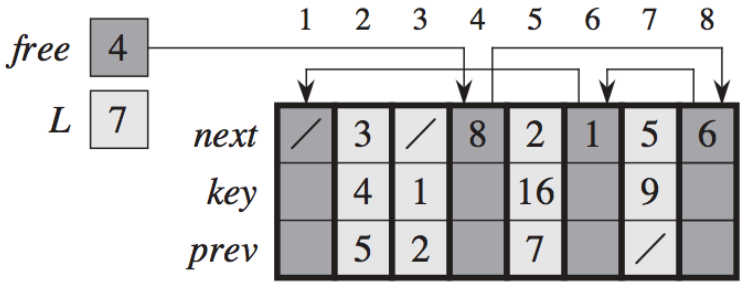
A variable L holds the index of the head of the list.

Implementing pointers and objects: single array

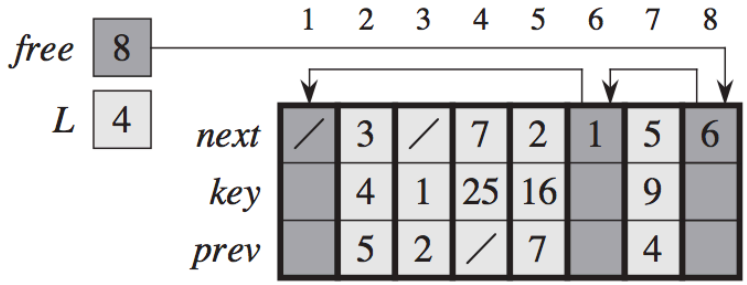


A variable L holds the index of the head of the list.

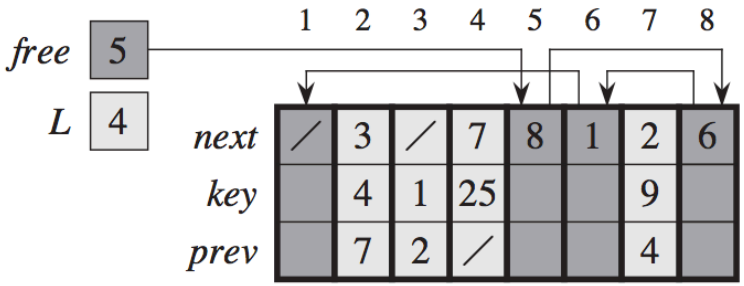
Allocating and freeing objects



(a)



(b)



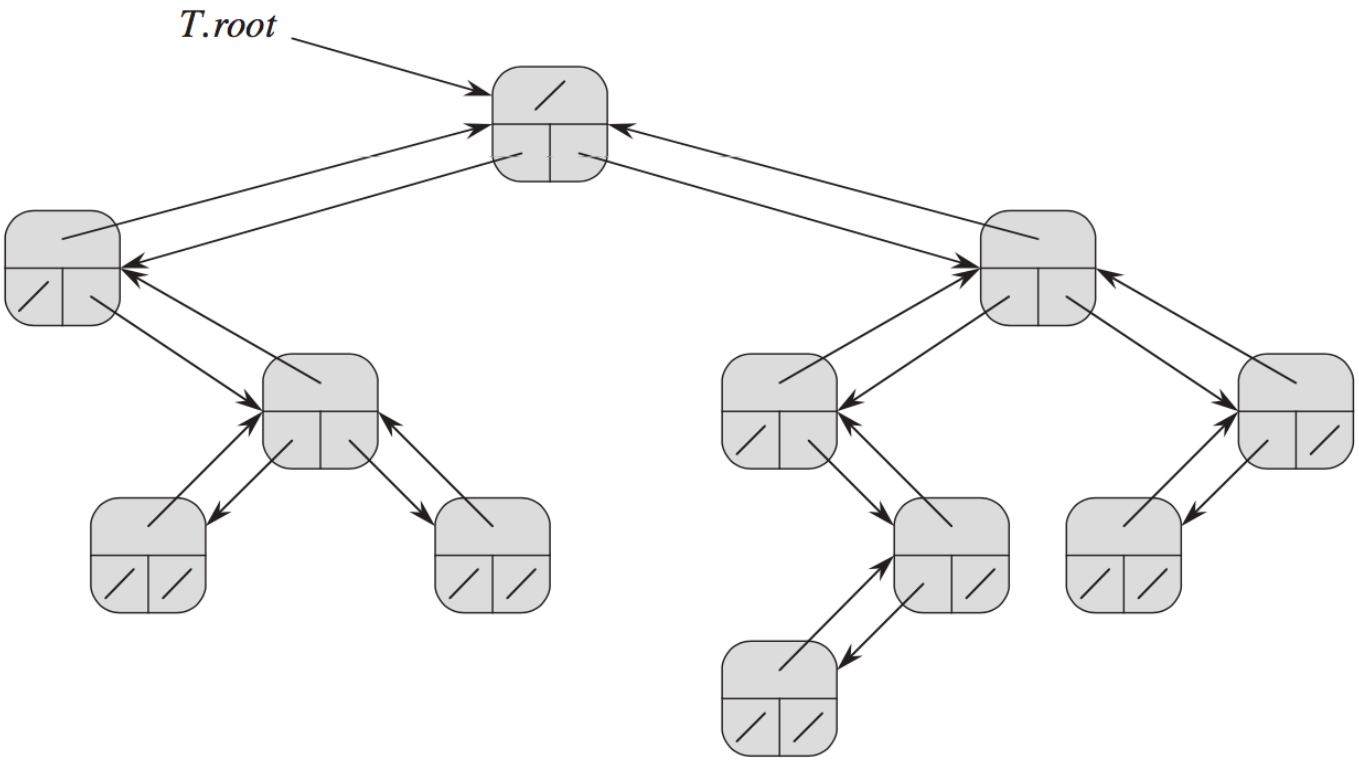
(c)

```
ALLOCATE-OBJECT()
1  if free == NIL
2       error "out of space"
3  else x = free
4       free = x.next
5       return x
```

```
FREE-OBJECT(x)
1  x.next = free
2  free = x
```

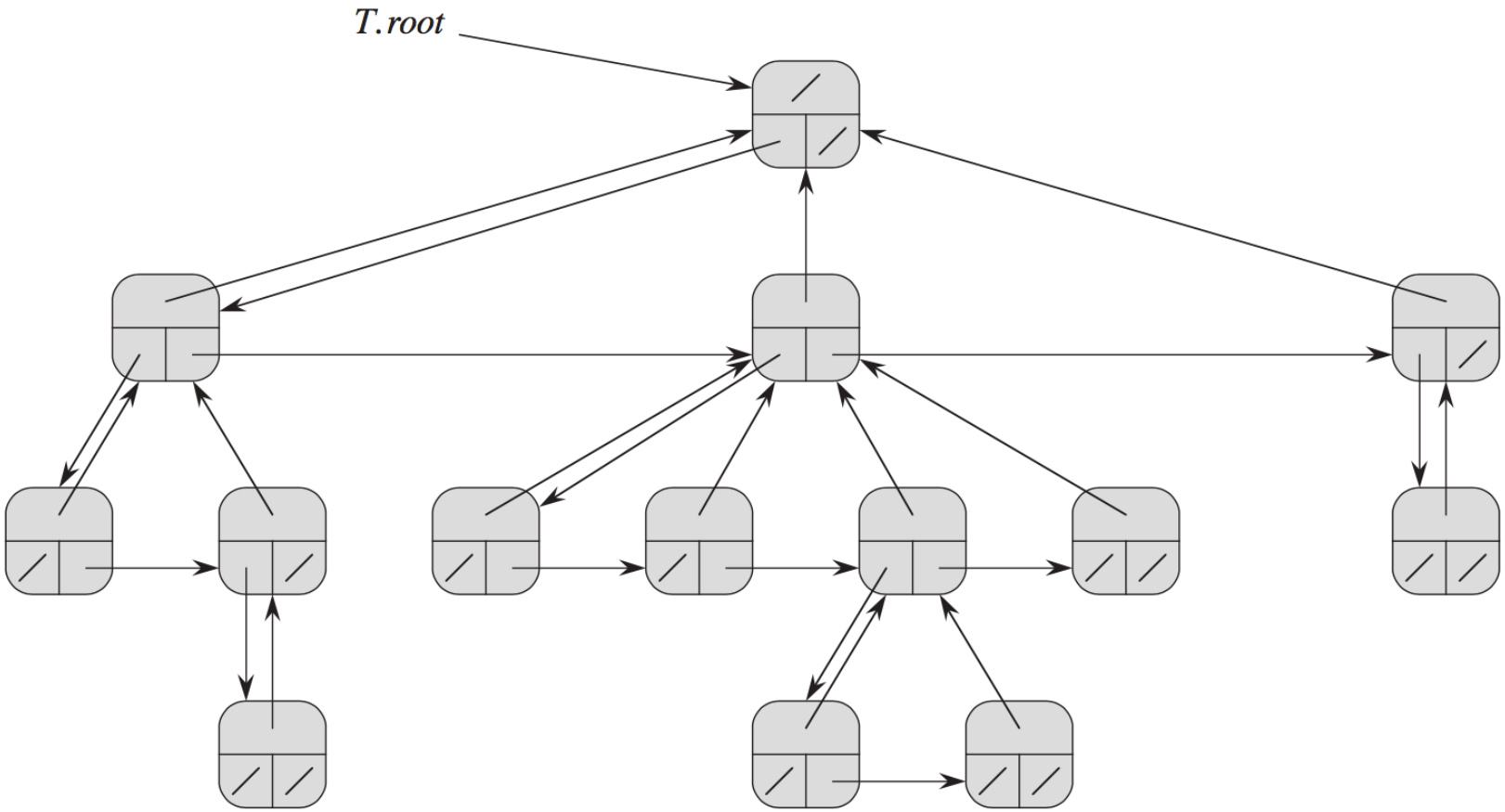
Free list: single linked, acts like a stack

Representing rooted trees: binary



Each node x has the attributes $x.p$ (top), $x.left$ (lower left), and $x.right$ (lower right). The key attributes are not shown.

Representing rooted trees: unbounded branching



Each node x has attributes $x.p$ (top), $x.left-child$ (lower left), and $x.right-sibling$ (lower right). The *key* attributes are not shown.