Elementary data structures, Hash tables

Congduan Li

Chinese University of Hong Kong, Shenzhen

congduan.li@gmail.com

Sep 19 & 21, 2017

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.

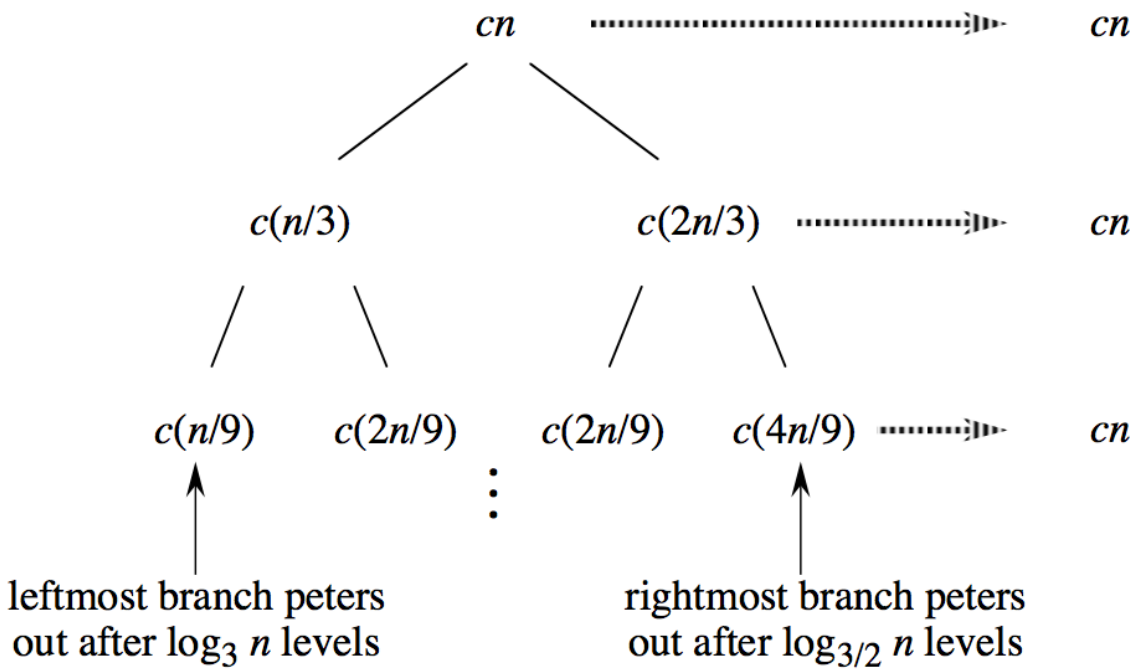
Recursion trees

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.)

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.

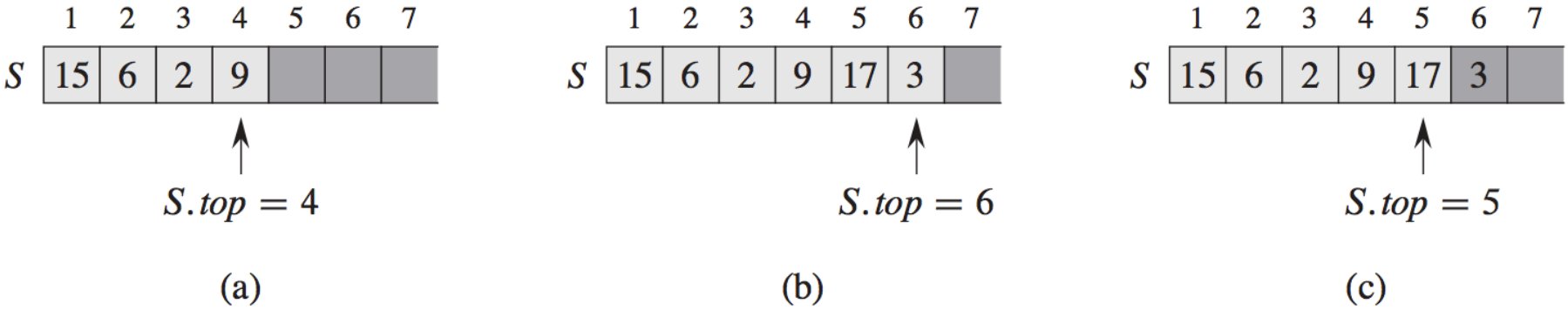
SUCCESSOR(S, x)
: A query that, given an element x whose key is from a totally ordered set S, returns a pointer to the next larger element in S, or NIL if x is the maximum element.

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
```

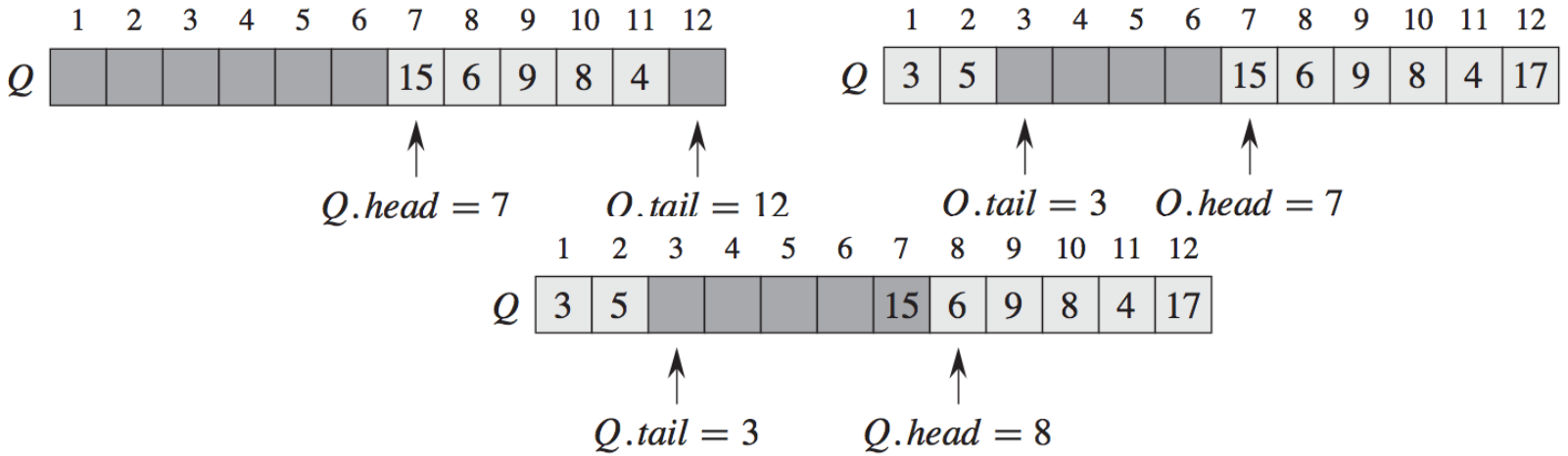
```
POP(S)
1  if STACK-EMPTY(S)
2       error "underflow"
3  else S.top = S.top − 1
4       return S[S.top + 1]
```

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
```

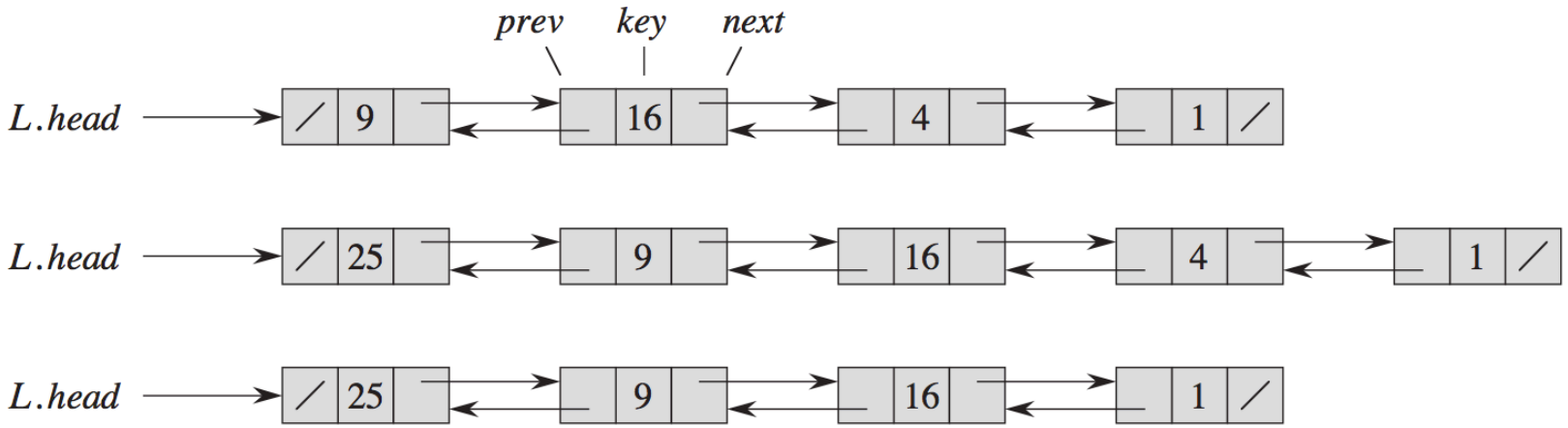
```
DEQUEUE(Q)
1  x = Q[Q.head]
2  if Q.head == Q.length
3        Q.head = 1
4  else Q.head = Q.head + 1
5  return x
```

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)

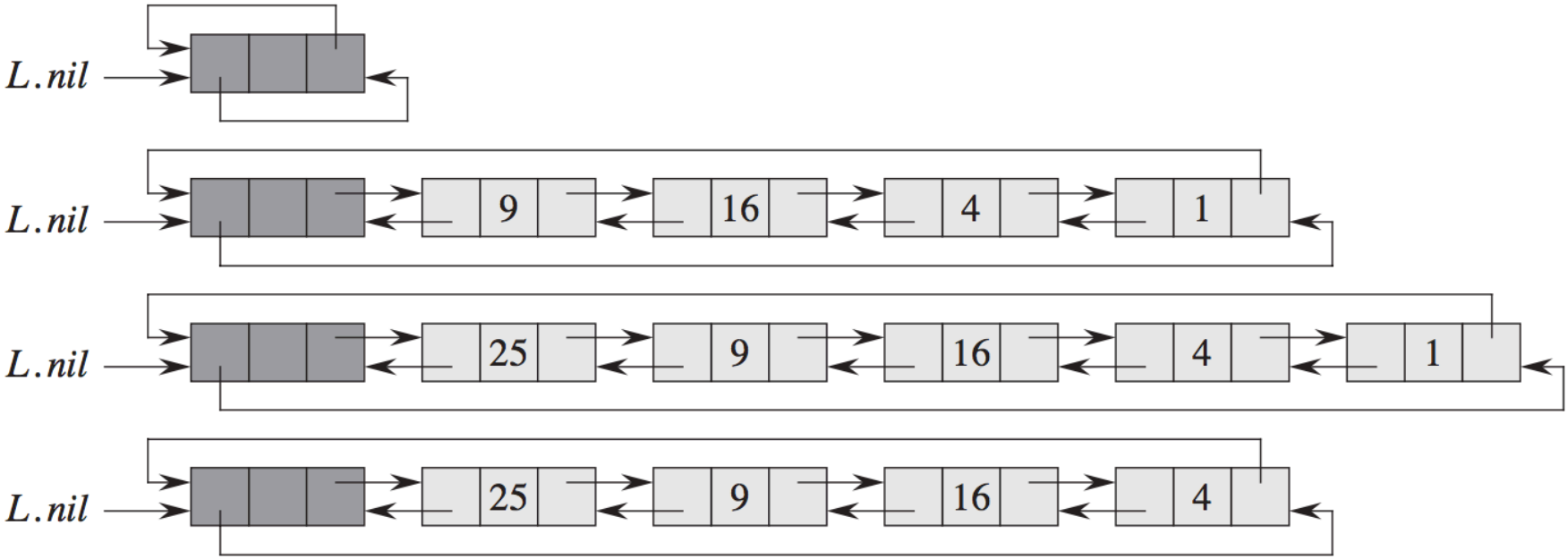
```
1  if x.prev ≠ NIL
2        x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5        x.next.prev = x.prev
```

$O(1)$

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$'(L, x)$

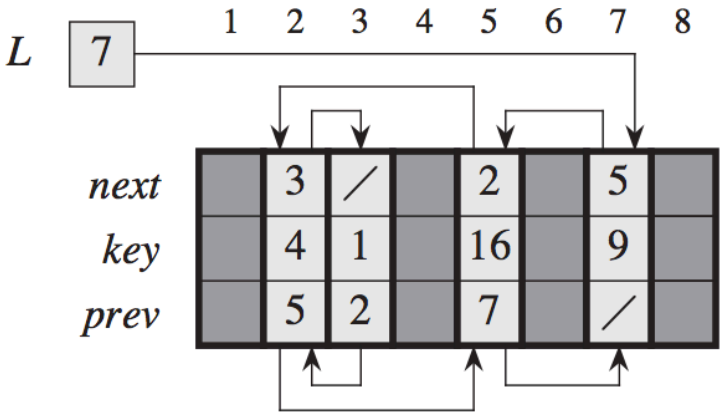
```
1  x.next = L.nil.next
2  L.nil.next.prev = x
3  L.nil.next = x
4  x.prev = L.nil
```

LIST-DELETE$'(L, x)$

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

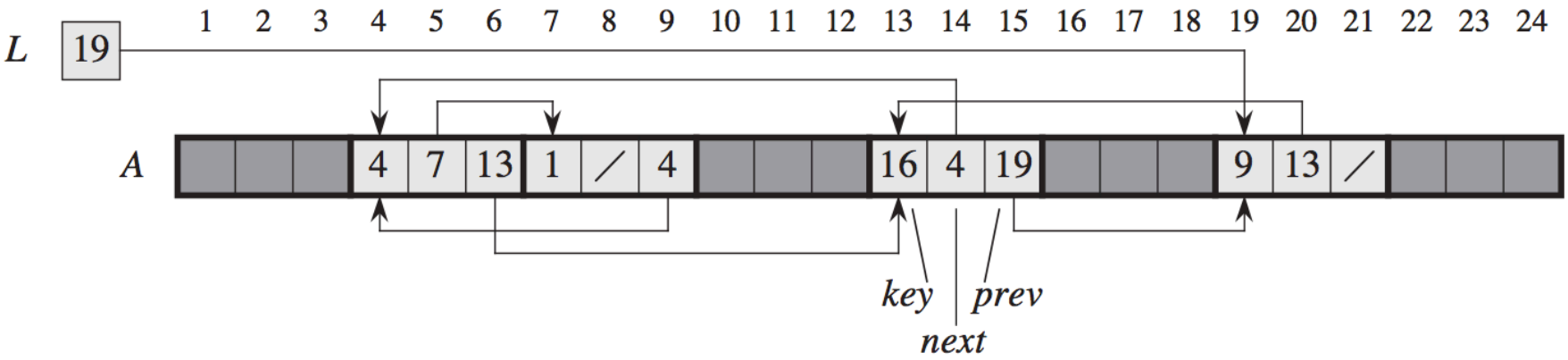
We should use sentinels judiciously. When there are many small lists, the extra storage used by their sentinels can represent significant wasted memory. We use sentinels only when they truly simplify the code.

Implementing pointers and objects: multi-array



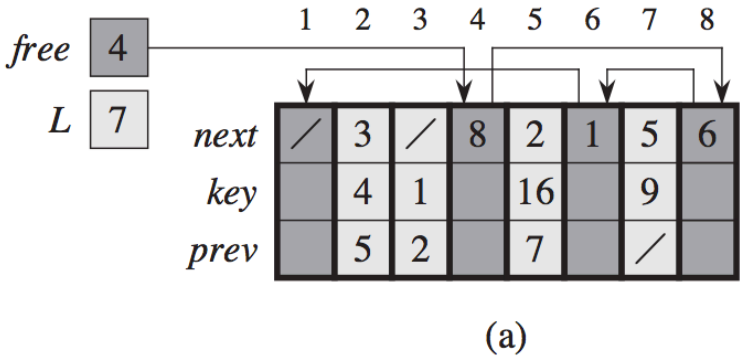
A variable L holds the index of the head of the list.

Implementing pointers and objects: single array

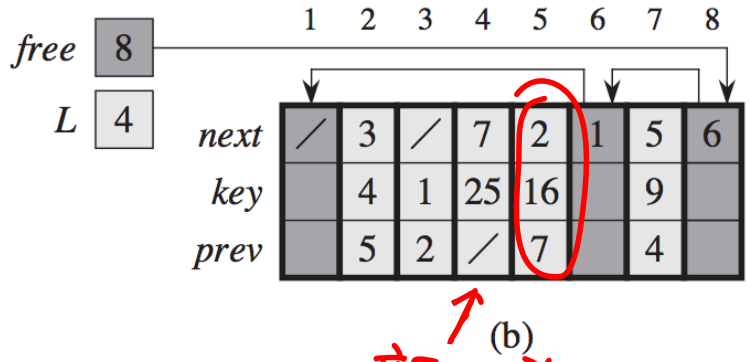


A variable L holds the index of the head of the list.

Allocating and freeing objects

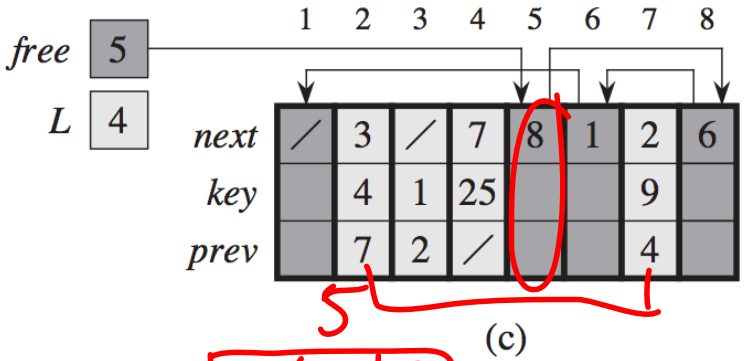


(a)



(b)

新的头



(c)

prev nxt

5 nxt = 8

free = 5

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)

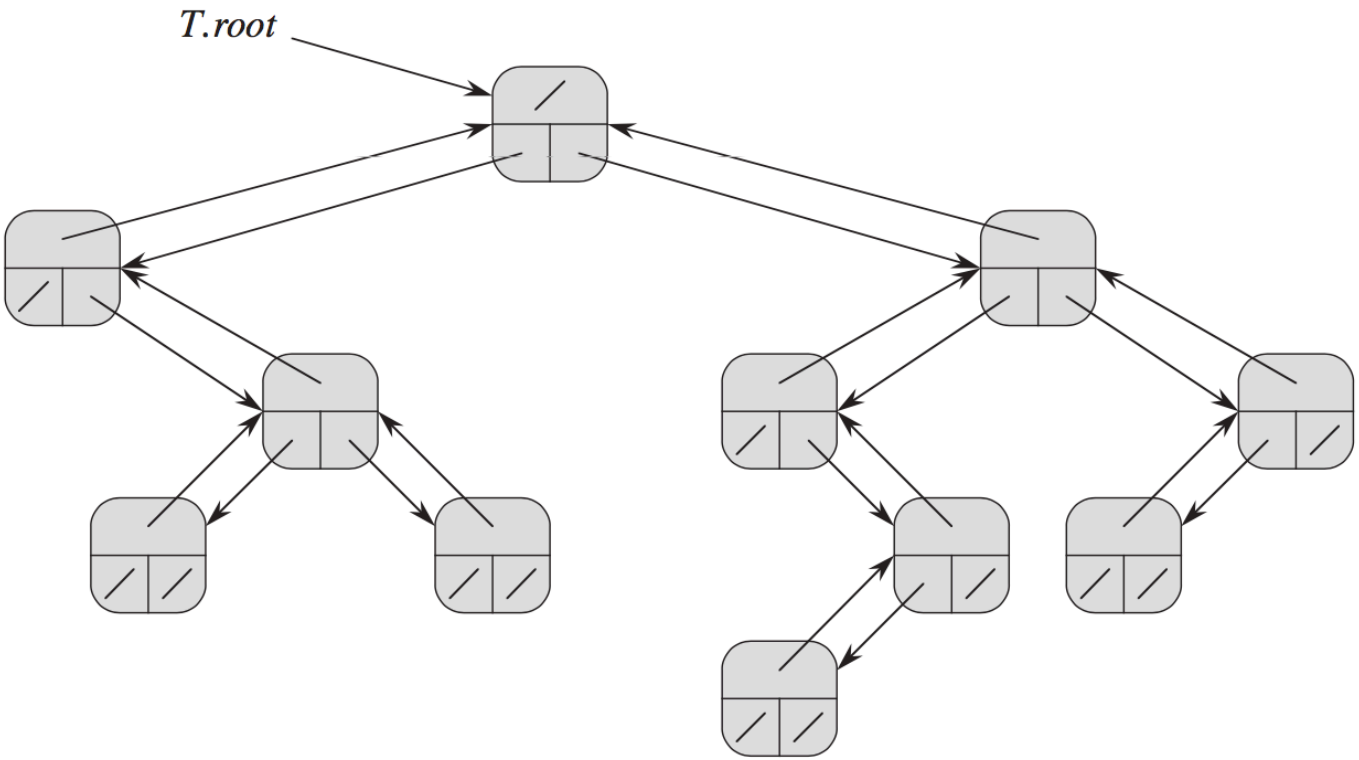
```
1  x.next = free
2  free = x
```

x.prev.nxt = x.nxt

x.nxt.prev = x.prev

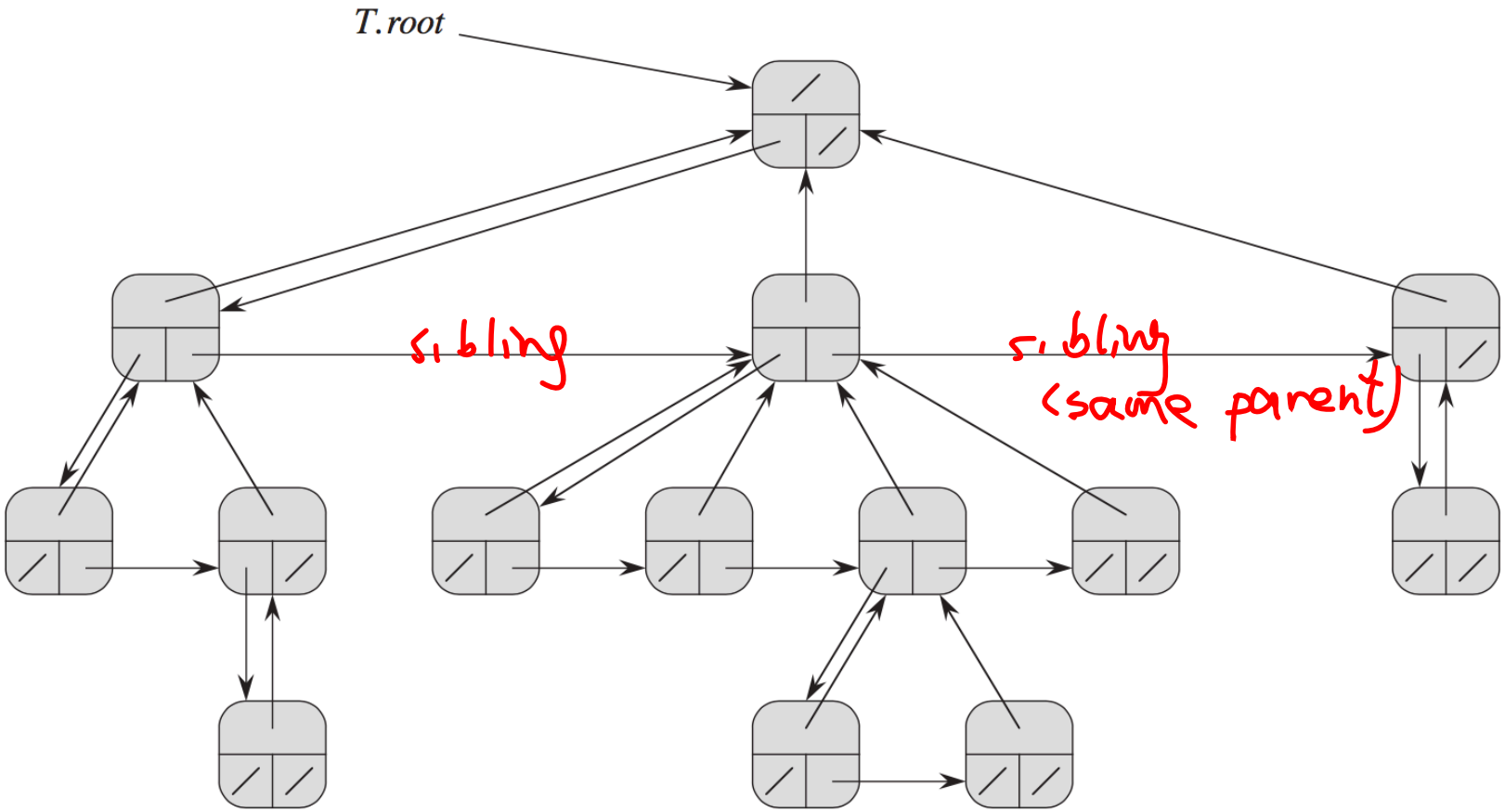
Free list: single linked, acts like a stack

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.

in the same level

Hash Tables

- Many applications require a dynamic set that supports only the **dictionary operations** INSERT, SEARCH, and DELETE.
- With an ordinary array, we store the element whose key is k in position k of the array.
- Given a key k, we find the element whose key is k by just looking in the kth position of the array. This is called **direct addressing**.

Direct-address tables

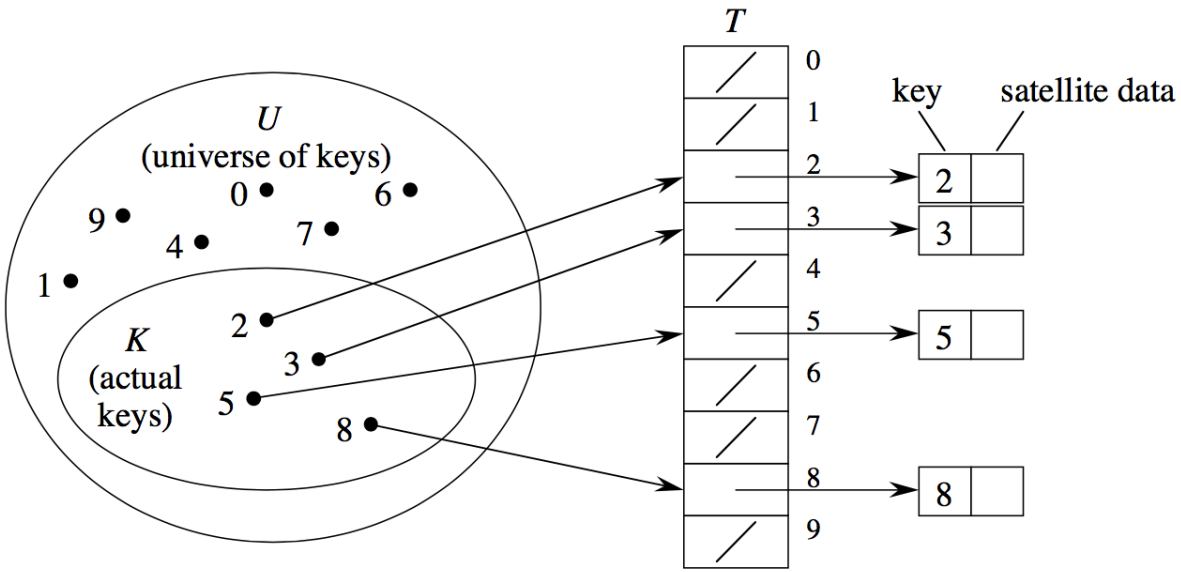
Scenario:

- Maintain a dynamic set.
- Each element has a key drawn from a universe $U = \{0, 1, \ldots, m-1\}$ where m isn't too large.
- No two elements have the same key.

Represent by a ***direct-address table***, or array, $T[0 \ldots m-1]$:

- Each ***slot***, or position, corresponds to a key in U.
- If there's an element x with key k, then $T[k]$ contains a pointer to x.
- Otherwise, $T[k]$ is empty, represented by NIL.

Direct-address tables



Dictionary operations are trivial and take $O(1)$ time each:

DIRECT-ADDRESS-SEARCH(T, k)
return $T[k]$

DIRECT-ADDRESS-INSERT(T, x)
$T[key[x]] \leftarrow x$

DIRECT-ADDRESS-DELETE(T, x)
$T[key[x]] \leftarrow$ NIL

Hash tables

The problem with direct addressing is if the universe U is large, storing a table of size $|U|$ may be impractical or impossible.

Often, the set K of keys actually stored is small, compared to U, so that most of the space allocated for T is wasted.

- When K is much smaller than U, a hash table requires much less space than a direct-address table.
- Can reduce storage requirements to $\Theta(|K|)$.
- Can still get $O(1)$ search time, but in the *average case*, not the *worst case*.

查找一个sequence，库中的sequence都根据 hasn func 算好，算sequence的hash value，然后找对应的

Idea: Instead of storing an element with key k in slot k, use a function h and store the element in slot $h(k)$.

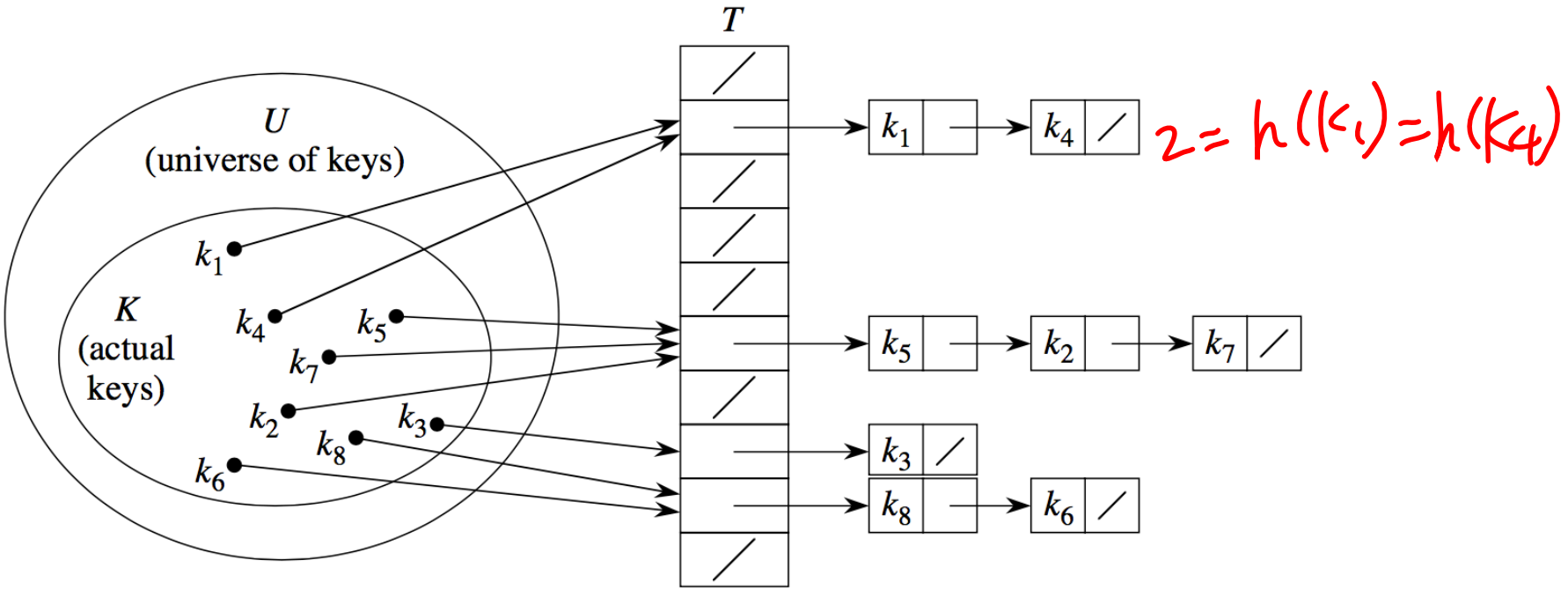
- We call h a ***hash function***.
- $h : U \to \{0, 1, \ldots, m-1\}$, so that $h(k)$ is a legal slot number in T.
- We say that k ***hashes*** to slot $h(k)$.

Collisions

Collisions: When two or more keys hash to the same slot.

- Can happen when there are more possible keys than slots ($|U| > m$).
- For a given set K of keys with $|K| \leq m$, may or may not happen. Definitely happens if $|K| > m$.
- Therefore, must be prepared to handle collisions in all cases.
- Use two methods: chaining and open addressing.
- Chaining is usually better than open addressing. We'll examine both.

Collision resolution by chaining



[This figure shows singly linked lists. If we want to delete elements, it's better to use doubly linked lists.]

- Slot j contains a pointer to the head of the list of all stored elements that hash to j *[or to the sentinel if using a circular, doubly linked list with a sentinel]* ,
- If there are no such elements, slot j contains NIL.

Operations with chaining

- ***Insertion:***

  CHAINED-HASH-INSERT(T, x)
  insert x at the head of list $T[h(key[x])]$

  - Worst-case running time is $O(1)$.
  - Assumes that the element being inserted isn't already in the list.
  - It would take an additional search to check if it was already inserted.

- ***Search:***

  CHAINED-HASH-SEARCH(T, k)
  search for an element with key k in list $T[h(k)]$

  Running time is proportional to the length of the list of elements in slot $h(k)$.

Operations with chaining

- ***Deletion:***

  CHAINED-HASH-DELETE(T, x)
  delete x from the list $T[h(key[x])]$

  - Given pointer x to the element to delete, so no search is needed to find this element.
  - Worst-case running time is $O(1)$ time if the lists are doubly linked.
  - If the lists are singly linked, then deletion takes as long as searching, because we must find x's predecessor in its list in order to correctly update *next* pointers.

Analysis of hashing with chaining

Given a key, how long does it take to find an element with that key, or to determine that there is no element with that key?

- Analysis is in terms of the ***load factor*** $\alpha = n/m$:
  - $n =$ # of elements in the table.
  - $m =$ # of slots in the table $=$ # of (possibly empty) linked lists.
  - Load factor is average number of elements per linked list.
  - Can have $\alpha < 1$, $\alpha = 1$, or $\alpha > 1$.
- Worst case is when all n keys hash to the same slot $\Rightarrow$ get a single list of length n $\Rightarrow$ worst-case time to search is $\Theta(n)$, plus time to compute hash function.
- Average case depends on how well the hash function distributes the keys among the slots.

Analysis of hashing with chaining

We focus on average-case performance of hashing with chaining.

- Assume ***simple uniform hashing***: any given element is equally likely to hash into any of the m slots.
- For $j = 0, 1, \ldots, m-1$, denote the length of list $T[j]$ by n_j. Then $n = n_0 + n_1 + \cdots + n_{m-1}$.
- Average value of n_j is $\mathrm{E}[n_j] = \alpha = n/m$.
- Assume that we can compute the hash function in $O(1)$ time, so that the time required to search for the element with key k depends on the length $n_{h(k)}$ of the list $T[h(k)]$.

We consider two cases:

- If the hash table contains no element with key k, then the search is unsuccessful.
- If the hash table does contain an element with key k, then the search is successful.

Unsuccessful search

Theorem

An unsuccessful search takes expected time $\Theta(1+\alpha)$.

Proof Simple uniform hashing $\Rightarrow$ any key not already in the table is equally likely to hash to any of the m slots.

To search unsuccessfully for any key k, need to search to the end of the list $T[h(k)]$. This list has expected length $\mathrm{E}[n_{h(k)}] = \alpha$. Therefore, the expected number of elements examined in an unsuccessful search is α.

Adding in the time to compute the hash function, the total time required is $\Theta(1+\alpha)$. ■

Successful search

Theorem

A successful search takes expected time $\Theta(1+\alpha)$.

Proof Assume that the element x being searched for is equally likely to be any of the n elements stored in the table.

The number of elements examined during a successful search for x is 1 more than the number of elements that appear before x in x's list. These are the elements inserted *after* x was inserted (because we insert at the head of the list).

So we need to find the average, over the n elements x in the table, of how many elements were inserted into x's list after x was inserted.

For $i = 1, 2, \ldots, n$, let x_i be the ith element inserted into the table, and let $k_i = key[x_i]$.

For all i and j, define indicator random variable $X_{ij} = \mathrm{I}\{h(k_i) = h(k_j)\}$. $= \begin{cases} 1 \\ 0 \end{cases}$

Simple uniform hashing $\Rightarrow \Pr\{h(k_i) = h(k_j)\} = 1/m \Rightarrow \mathrm{E}[X_{ij}] = 1/m$

Successful search

Expected number of elements examined in a successful search is

$$\mathrm{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\left(\sum_{j=i+1}^{n}X_{ij}\right)\right)\right]$$

All the elements after i

$$\begin{aligned}
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\mathrm{E}[X_{ij}]\right) && \text{(linearity of expectation)}\\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right)\\
&= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)\\
&= 1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right)\\
&= 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) && \text{(equation (A.1))}\\
&= 1+\frac{n-1}{2m}\\
&= 1+\frac{\alpha}{2}-\frac{\alpha}{2n}\,.
\end{aligned}$$

Adding in the time for computing the hash function, we get that the expected total time for a successful search is $\Theta(2+\alpha/2-\alpha/2n)=\Theta(1+\alpha)$.

Successful search

Interpretation: If $n = O(m)$, then $\alpha = n/m = O(m)/m = O(1)$, which means that searching takes constant time on average.

Since insertion takes $O(1)$ worst-case time and deletion takes $O(1)$ worst-case time when the lists are doubly linked, all dictionary operations take $O(1)$ time on average.

Hash functions

What makes a good hash function?

- Ideally, the hash function satisfies the assumption of simple uniform hashing.
- In practice, it's not possible to satisfy this assumption, since we don't know in advance the probability distribution that keys are drawn from, and the keys may not be drawn independently.
- Often use heuristics, based on the domain of the keys, to create a hash function that performs well.

Keys as natural numbers

- Hash functions assume that the keys are natural numbers.
- When they're not, have to interpret them as natural numbers.
- ***Example:*** Interpret a character string as an integer expressed in some radix notation. Suppose the string is `CLRS`:
  - ASCII values: `C` = 67, `L` = 76, `R` = 82, `S` = 83.
  - There are 128 basic ASCII values.
  - So interpret `CLRS` as $(67 \cdot 128^3) + (76 \cdot 128^2) + (82 \cdot 128^1) + (83 \cdot 128^0) =$ 141,764,947.

Hash functions

Division method

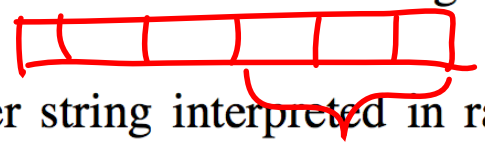
$h(k) = k \bmod m$. 取余

Example: $m = 20$ and $k = 91 \Rightarrow h(k) = 11$.

Advantage: Fast, since requires just one division operation.

Disadvantage: Have to avoid certain values of m:

- Powers of 2 are bad. If $m = 2^p$ for integer p, then $h(k)$ is just the least significant p bits of k.
- If k is a character string interpreted in radix 2^p (as in CLRS example), then $m = 2^p - 1$ is bad: permuting characters in a string does not change its hash value (Exercise 11.3-3).



$(2^n + 2^{p+1} + 2^3 +) - 2^3$

p bits

除很的

p bits

Good choice for m: A prime not too close to an exact power of 2.

$m = 12$

$h(CLRS) = h(CRLS) = (67 \times (12)+11)^n +$

除3都取1 $\Rightarrow$ 67 + 72 + 82 + 83 都能除尽 (有12)

Hash functions

Multiplication method

1. Choose constant A in the range $0 < A < 1$.
2. Multiply key k by A.
3. Extract the fractional part of kA.
4. Multiply the fractional part by m.
5. Take the floor of the result.

Put another way, $h(k) = \lfloor m\,(k\,A \bmod 1)\rfloor$, where $k\,A \bmod 1 = kA - \lfloor kA\rfloor =$ fractional part of kA.

Disadvantage: Slower than division method.

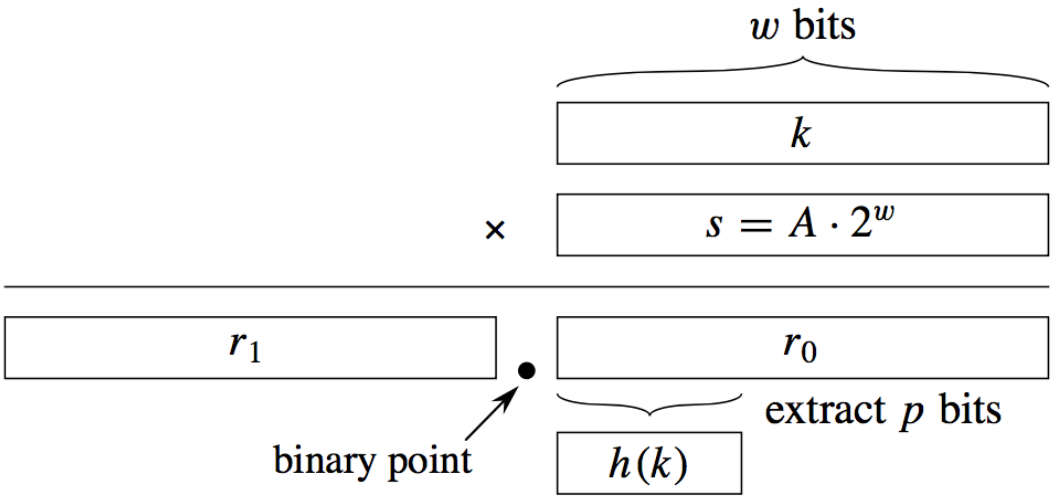
Advantage: Value of m is not critical.

(Relatively) easy implementation:

- Choose $m = 2^p$ for some integer p.
- Let the word size of the machine be w bits.
- Assume that k fits into a single word. (k takes w bits.)
- Let s be an integer in the range $0 < s < 2^w$. (s takes w bits.)

Hash functions

- Restrict A to be of the form $s/2^w$.



- Multiply k by s.
- Since we're multiplying two w-bit words, the result is $2w$ bits, $r_1 2^w + r_0$, where r_1 is the high-order word of the product and r_0 is the low-order word.
- r_1 holds the integer part of kA ($\lfloor kA \rfloor$) and r_0 holds the fractional part of kA ($k\ A \bmod 1 = kA - \lfloor kA \rfloor$). Think of the "binary point" (analog of decimal point, but for binary representation) as being between r_1 and r_0. Since we don't care about the integer part of kA, we can forget about r_1 and just use r_0.
- Since we want $\lfloor m\ (k\ A \bmod 1) \rfloor$, we could get that value by shifting r_0 to the left by $p = \lg m$ bits and then taking the p bits that were shifted to the left of the binary point.

Hash functions

- We don't need to shift. The p bits that would have been shifted to the left of the binary point are the p most significant bits of r_0. So we can just take these bits after having formed r_0 by multiplying k by s.
- ***Example:*** $m = 8$ (implies $p = 3$), $w = 5$, $k = 21$. Must have $0 < s < 2^5$; choose $s = 13 \Rightarrow A = 13/32$.
  - Using just the formula to compute $h(k)$: $kA = 21 \cdot 13/32 = 273/32 = 8\frac{17}{32}$ $\Rightarrow k\ A \bmod 1 = 17/32 \Rightarrow m\ (k\ A \bmod 1) = 8 \cdot 17/32 = 17/4 = 4\frac{1}{4} \Rightarrow \lfloor m\ (k\ A \bmod 1) \rfloor = 4$, so that $h(k) = 4$.
  - Using the implementation: $ks = 21 \cdot 13 = 273 = 8 \cdot 2^5 + 17 \Rightarrow r_1 = 8$, $r_0 = 17$. Written in $w = 5$ bits, $r_0 = 10001$. Take the $p = 3$ most significant bits of r_0, get 100 in binary, or 4 in decimal, so that $h(k) = 4$.

How to choose A:

- The multiplication method works with any legal value of A.
- But it works better with some values than with others, depending on the keys being hashed.
- Knuth suggests using $A \approx (\sqrt{5} - 1)/2$.

Home work 2

- Problem 4-1
- Problem 4-3
- Problem 4-5
- Exercise 11.2-1
- Exercise 11.3-3
- Problem 11-2

Due on Oct 12, Thursday