EEC7205: Fundamentals of Computer Engineering

Elementary Data Structures

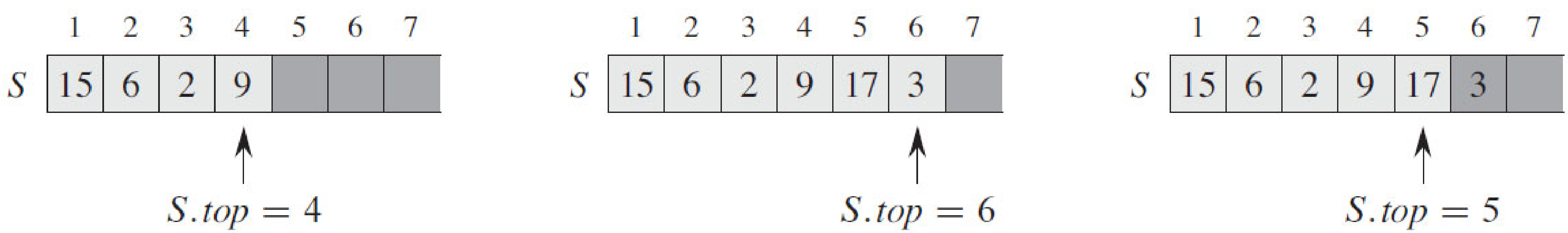
Data Structures

- A ***data structure*** is a way to store and organize data in order to facilitate access and modifications.
- No single data structure works well for all purposes, and so it is important to know the strengths and limitations of several of them.
- We will cover different data structures in this course.

Stacks

- Stacks are dynamic sets in which the element removed 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 is often called POP.
- The following figure shows that 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.
- When *S.top* = 0, the stack contains no elements and is ***empty***.

PUSH[*S*,17] and PUSH[*S*, 3]   POP[*S*]



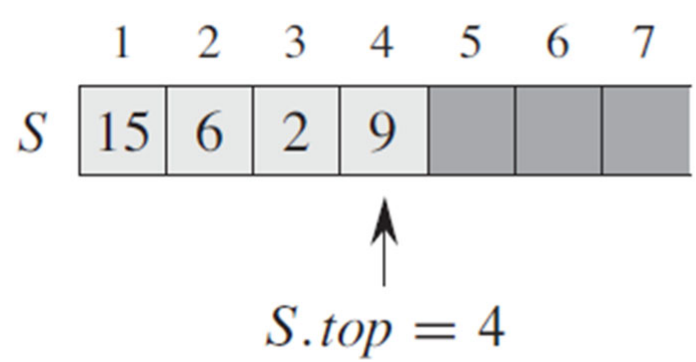
Stack Operations

- Each of the following three stack operations takes *O* (1) time.
- Do we need to check for the array capacity in PUSH?

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

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

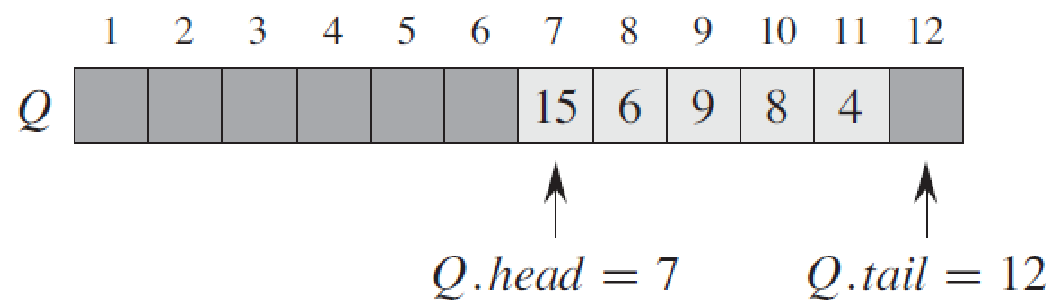
```
POP(S)
if STACK-EMPTY(S)
    error "underflow"
else S.top = S.top - 1
    return S[S.top + 1]
```



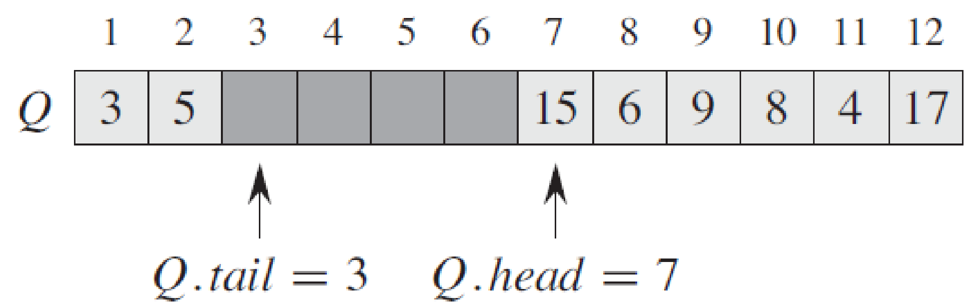
Queues

- Queues are dynamic sets in which the element removed from the set is the one that has been in the set for the longest time.
- The queue implements a ***first-in, first-out***, or ***FIFO***, policy.
- We call the INSERT operation on a queue ENQUEUE, and we call the DELETE operation DEQUEUE.
- The FIFO property of a queue causes it to operate like a line of customers waiting to pay a cashier.
- The queue has a ***head*** and a ***tail***.
- When an element is enqueued, it takes its place at the tail of the queue.
- The element dequeued is always the one at the head of the queue.

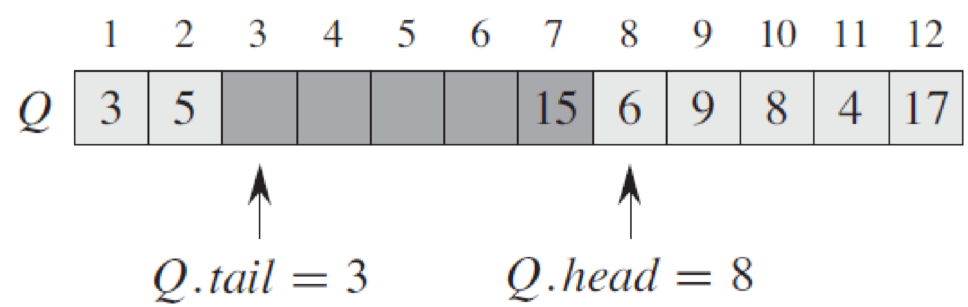
Queue Example



ENQUEUE(Q, 17),
ENQUEUE(Q, 3), and
ENQUEUE(Q, 5)



DEQUEUE(Q)

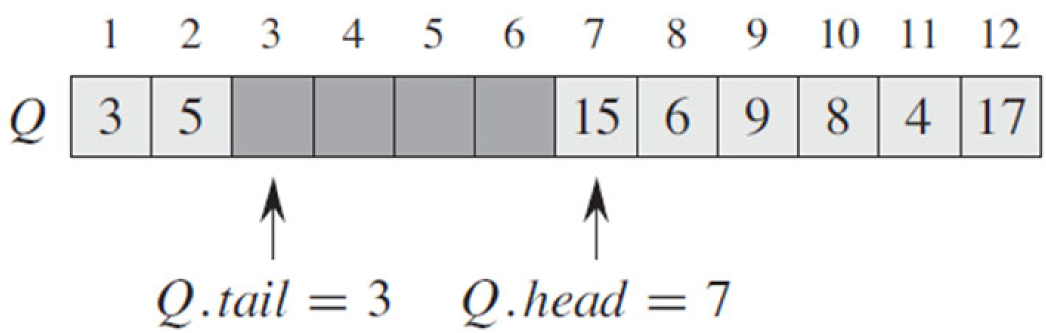


Queue Operations

- Initially, we have *Q.head* = *Q.tail* = 1

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

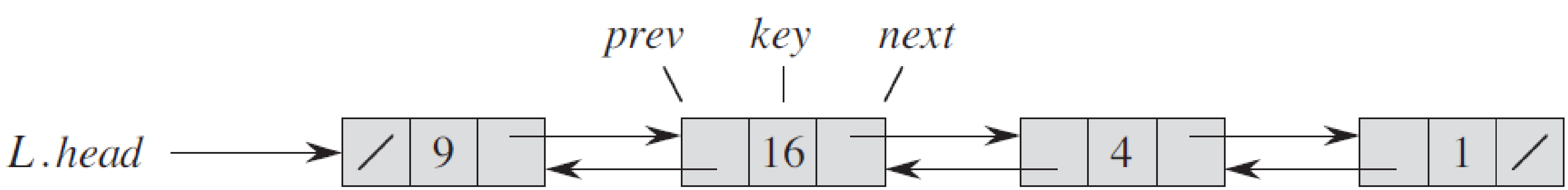
```
DEQUEUE(Q)
x = Q[Q.head]
if Q.head == Q.length
        Q.head = 1
else Q.head = Q.head + 1
return x
```



- When *Q.head* == *Q.tail* , is the queue empty or full?

Linked Lists

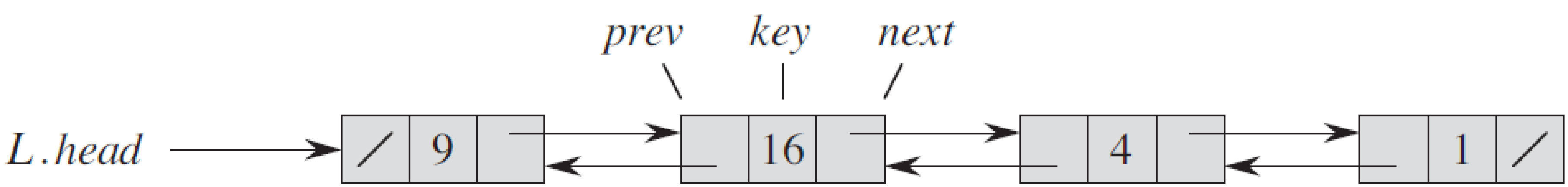
- A ***linked list*** is a data structure in which the objects are arranged in a linear order.
  - Unlike an array, however, in which the linear order is determined by the array indices, the order in a linked list is determined by pointers in each object.
- As shown, each element of a ***doubly linked list*** *L* is an object with an attribute *key* and two other pointer attributes: *next* and *prev*.



- If a list is ***singly linked***, we omit the *prev* pointer in each element.

Doubly Linked List

- In a ***doubly linked list*** *L*:
  - If *x.prev* = 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.



Doubly Linked List Node

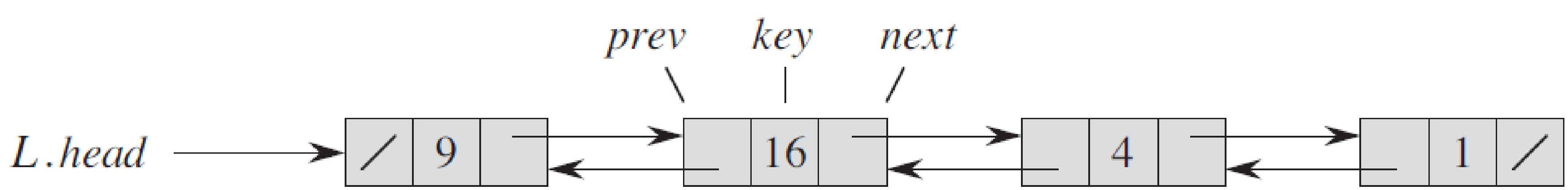
- Each node in a ***doubly linked list*** contains not only the values of the elements we need to store in the list, like a Queue data structure, but also the `prev` and `next` pointers.
- A node in a list of integers can be defined as:

```
struct Node          {
        Node*  prev;
        int    item;
        Node*  next;  };
```

- A node is created dynamically as:

```
        Node* x = new Node;
```

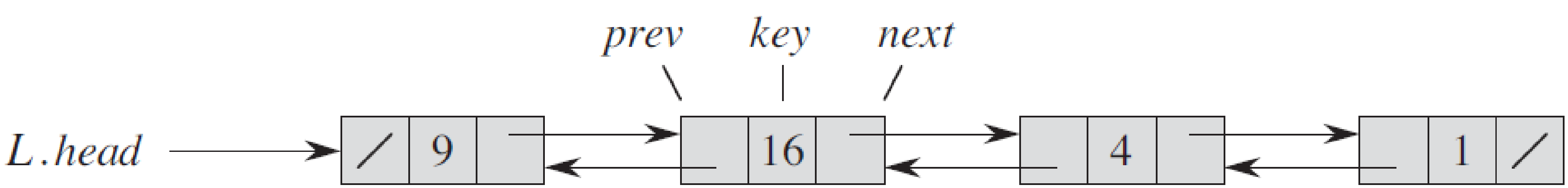
- Where x is used to insert the new node in the list.



Searching a Linked List

- The procedure LIST-SEARCH(L, k) finds the first element with key k in list L by a simple linear search, returning a pointer to this element.
- If no object with key k appears in the list, then the procedure returns NIL.

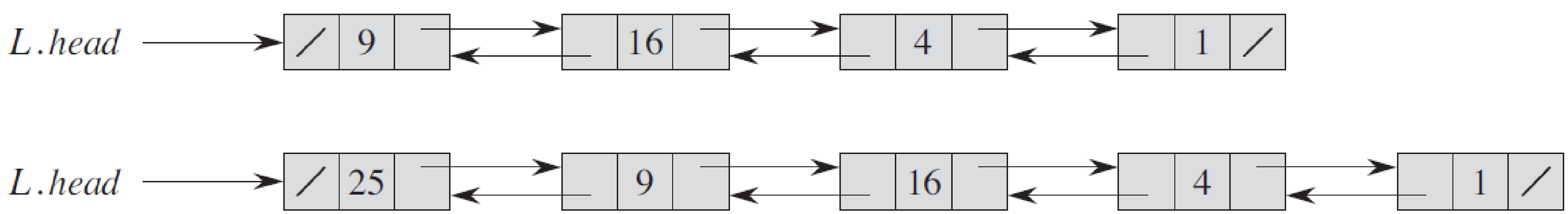
```
LIST-SEARCH(L, k)
  x = L.head
  while x ≠ NIL and x.key ≠ k
      x = x.next
  return x
```



Inserting Into a Linked List

- Given an element *x* whose *key* attribute has already been set, the LIST-INSERT procedure inserts *x* onto the front of the linked list.

```
LIST-INSERT(L, x)
x.next = L.head
if L.head ≠ NIL
    L.head.prev = x
L.head = x
x.prev = NIL
```



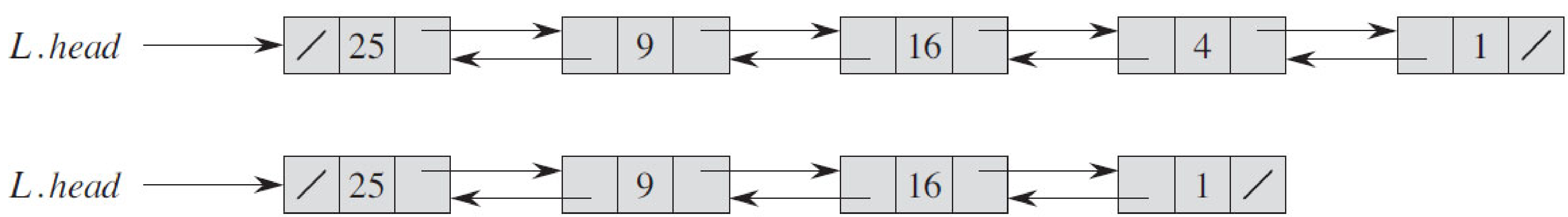
Following the execution of LIST-INSERT(L, x), where $x.key = 25$

Deleting From a Linked List

- The procedure LIST-DELETE removes an element *x* from a linked list *L*. It must be given a pointer to x, and it then delete *x* out of the list by updating pointers.

LIST-DELETE(L, x)

if $x.prev \neq$ NIL
    $x.prev.next = x.next$
else $L.head = x.next$
if $x.next \neq$ NIL
    $x.next.prev = x.prev$



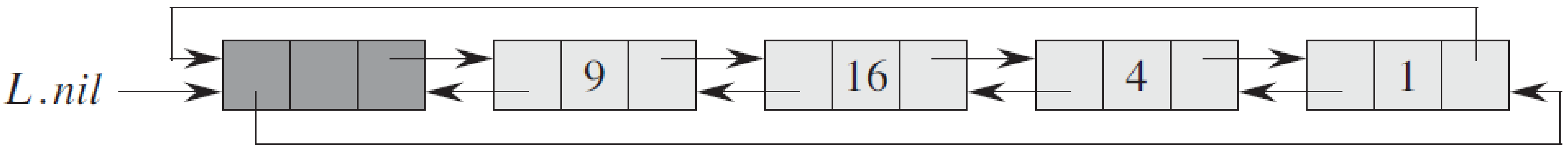
Following the execution of LIST-DELETE(*L*, *x*), where *x.key* = 4

Linked List Sentinels

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

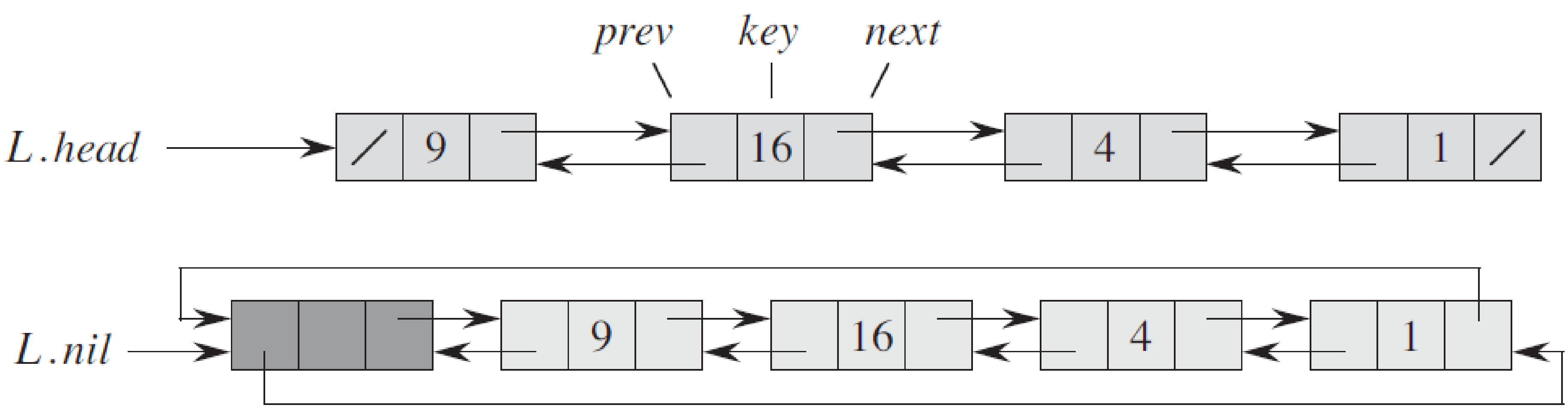
```
LIST-DELETE'(L, x)
1  x.prev.next = x.next
2  x.next.prev = x.prev
```

- One way to achieve that is to use a circular, doubly linked list with a sentinel.
  - A ***sentinel*** is a dummy object that allows us to simplify boundary conditions. The ***sentinel*** is *L.nil* in the following example:



A Circular, Doubly Linked List With a Sentinel

- In a circular, doubly linked list with a sentinel, 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.prev* points to the tail.
- Similarly, both the *next* attribute of the tail and the *prev* attribute of the head point to *L.nil*.



Searching and Inserting

- The gain from using sentinels within loops is usually a matter of clarity of code rather than speed.
- The linked list code, for example, becomes simpler when we use sentinels, but we save only $O(1)$ time in the LIST-INSERT' and LIST-DELETE' procedures.
- We should use sentinels judiciously. When there are many small lists, the extra storage used by their sentinels can represent significant wasted memory.

LIST-SEARCH'(L, k)

```
x = L.nil.next
while x ≠ L.nil and x.key ≠ k
    x = x.next
return x
```

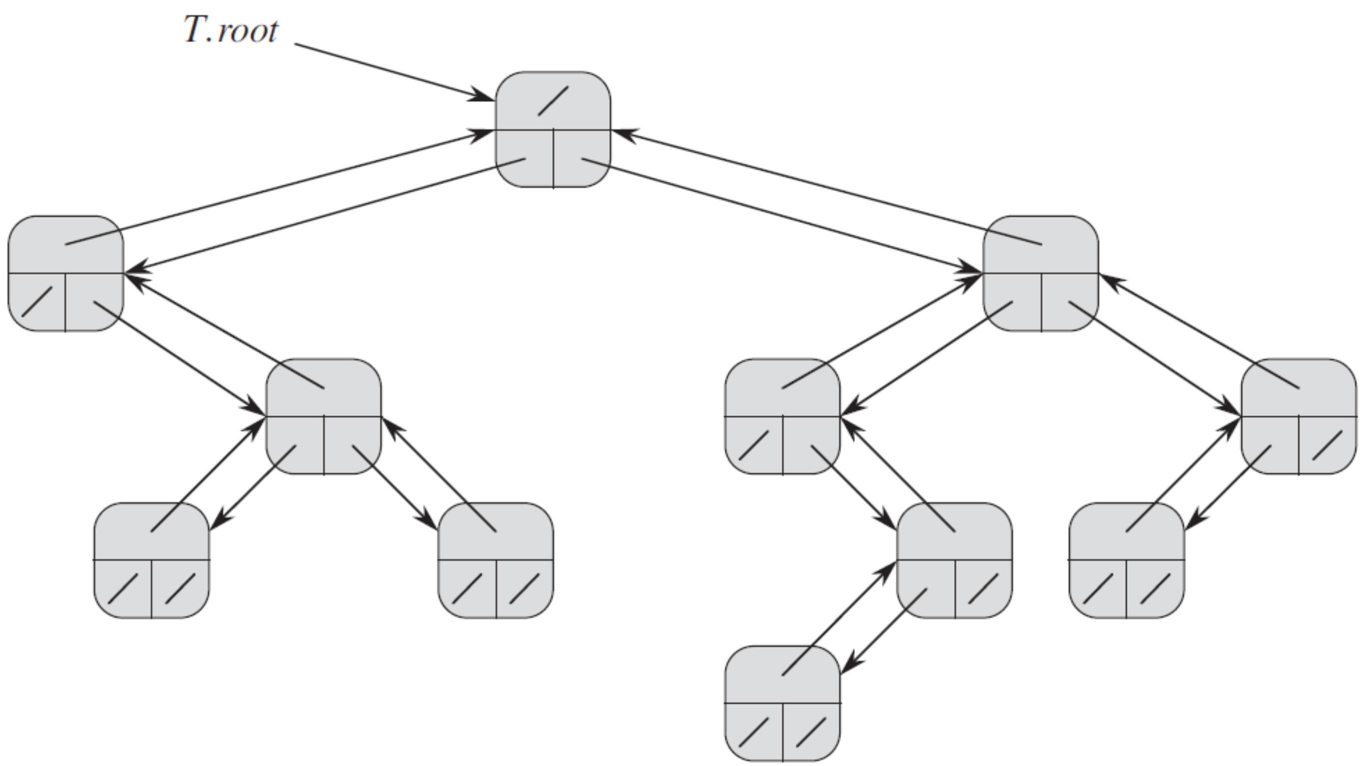
LIST-INSERT'(L, x)

```
x.next = L.nil.next
L.nil.next.prev = x
L.nil.next = x
x.prev = L.nil
```

Binary Trees

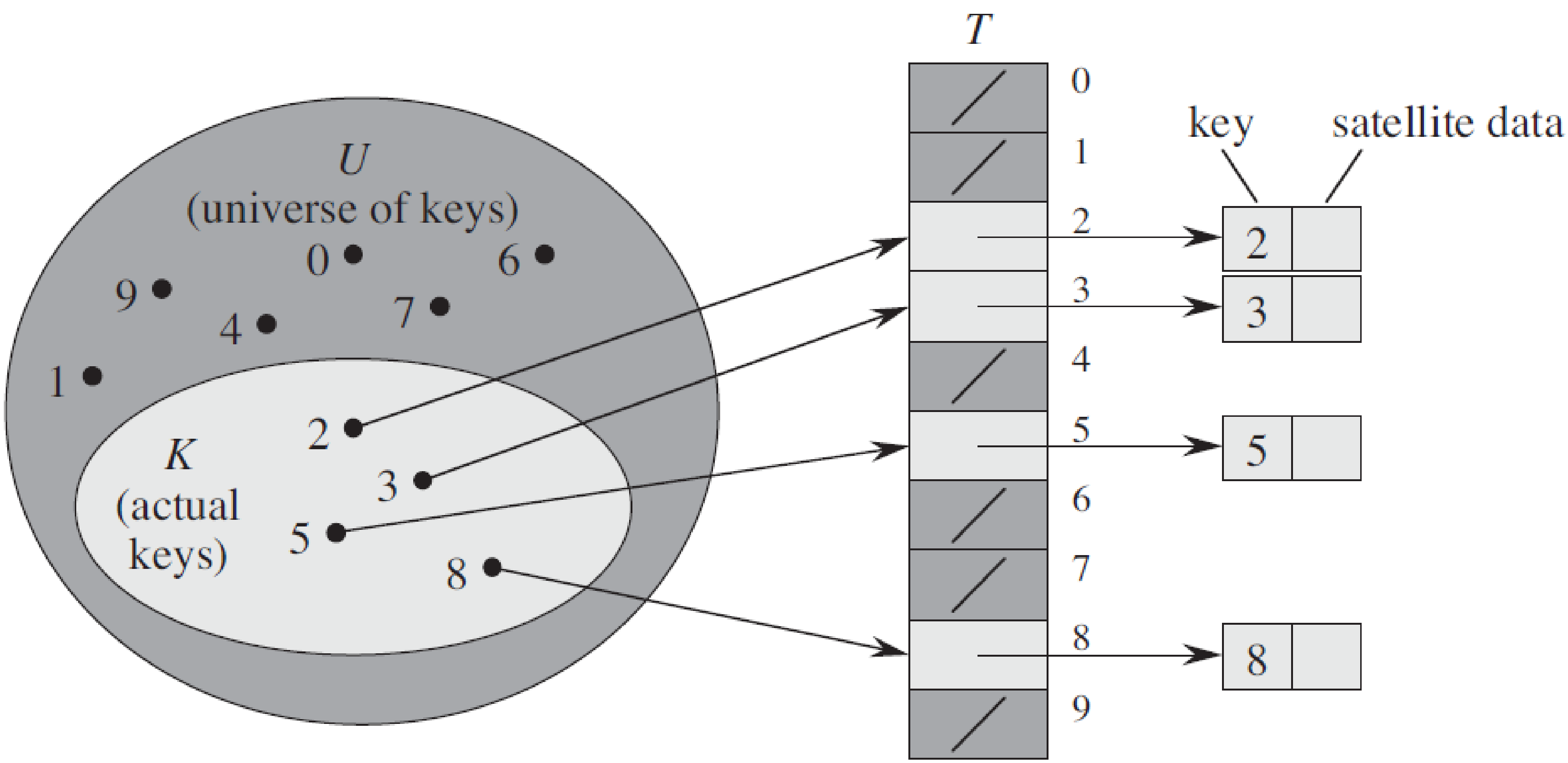
- The methods we have studied for representing lists can be extended to represent rooted trees data structure.
- In the following representation of a binary tree *T*, each node *x* has the attributes *x.p* (top), *x.left* (lower left), and *x.right* (lower right). The key attributes are not shown.

  - The root of the entire tree *T* is pointed to by the attribute *T.root*.
  - If *T.root* = NIL, then the tree is empty.
  - If *x.p* = NIL, then x is the root.



Direct-Address Tables

- Direct addressing is a simple technique that works well when the universe *U* of keys is reasonably small assuming that no two elements have the same key.



Direct-Address Table Operations

- Each of the following operations takes only $O(1)$ time.

DIRECT-ADDRESS-SEARCH(T, k)

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

DIRECT-ADDRESS-INSERT(T, x)

1 $T[x.key] = x$

DIRECT-ADDRESS-DELETE(T, x)

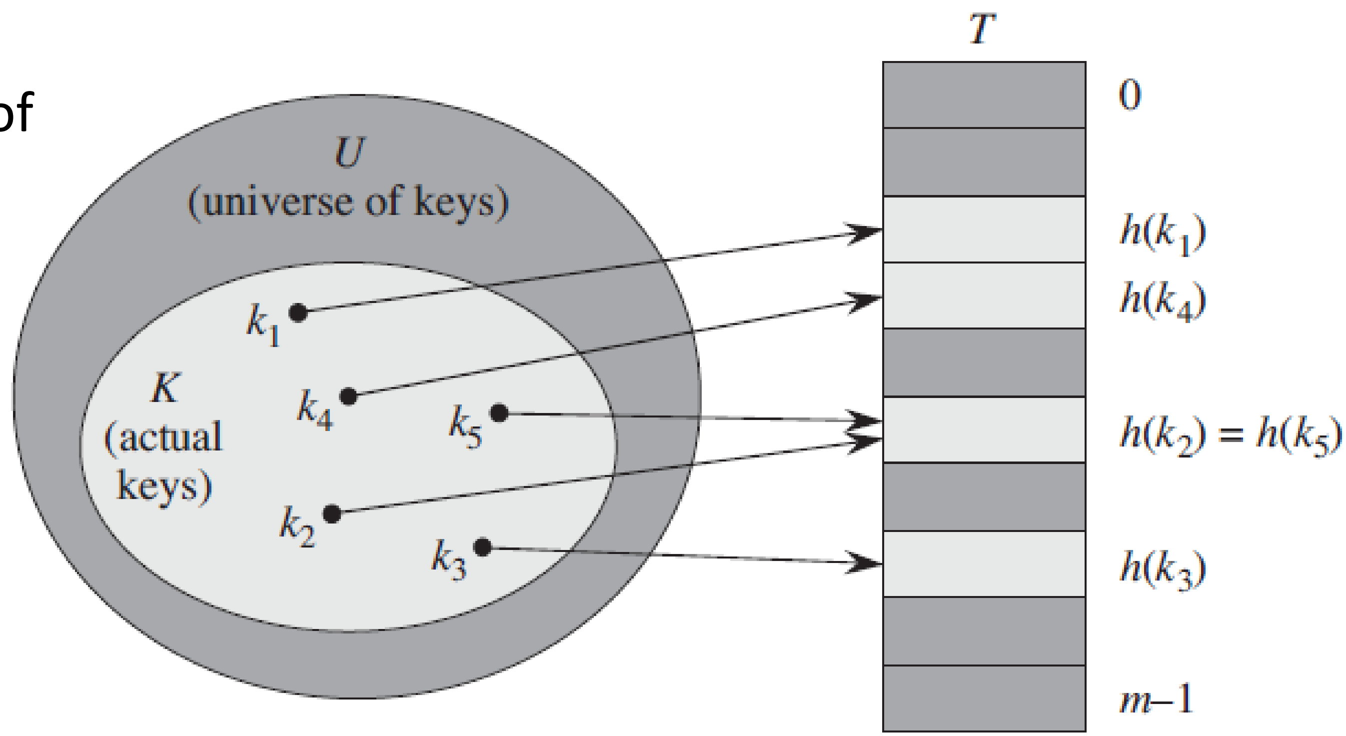
1 $T[x.key] = \text{NIL}$

Hash Tables

- The downside of direct addressing is obvious: if the universe U is large, storing a table T of size $|U|$ may be impractical, or even impossible, given the memory available on a typical computer.
- Furthermore, 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 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.
- A hash table is an effective data structure for implementing ***dictionaries*** operations INSERT, SEARCH, and DELETE.
- Although searching for an element in a hash table can take as long as searching for an element in a linked list – $\Theta(n)$ time in the worst case - in practice, hashing performs extremely well.

Hash Functions

- With direct addressing, an element with key *k* is stored in slot *k*.
- With hashing, this element is stored in slot *h*(*k*); that is, we use a ***hash function*** *h* to compute the slot from the key *k*.
- A hash function *h* must be **deterministic** in that a given input *k* should always produce the same output *h*(*k*).
- In the shown example, *h* maps the universe *U* of keys into the slots of a ***hash table*** *T* [0 .. *m* - 1] where the size *m* of the hash table is typically much less than |*U*|

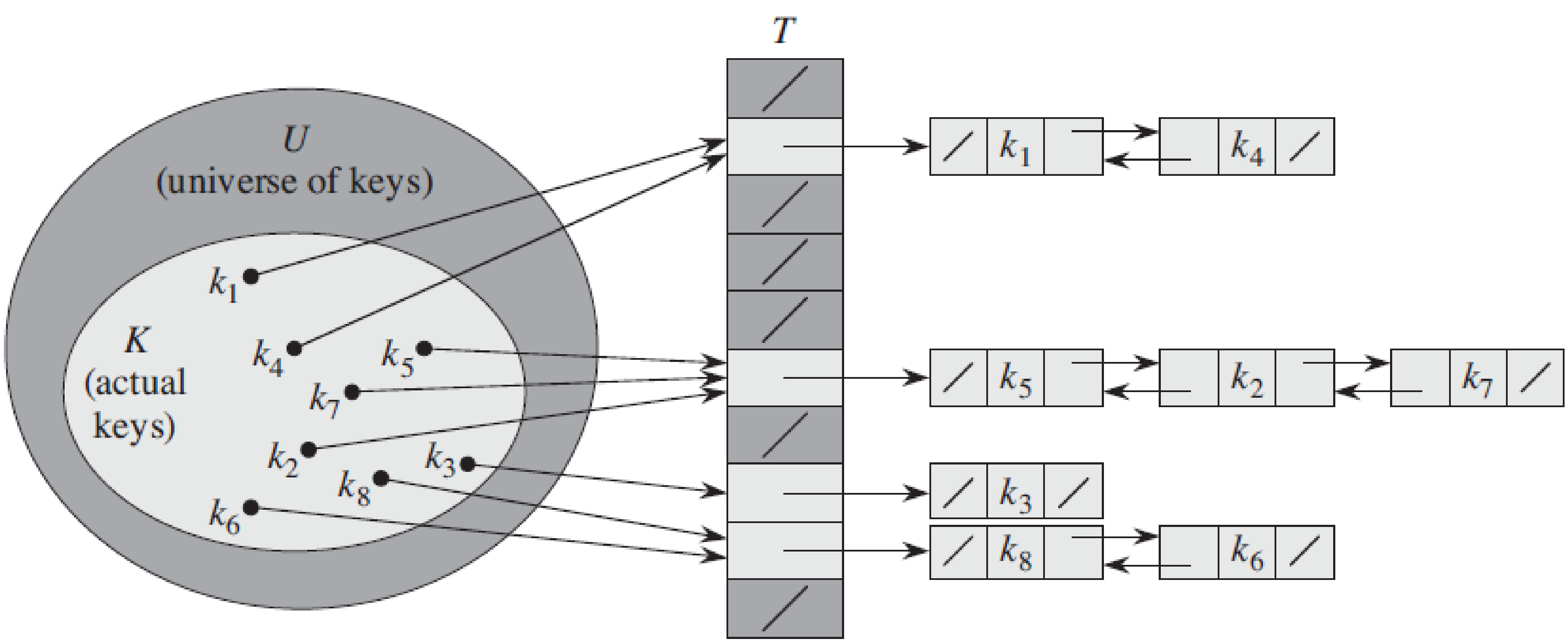


Hash Collision

- Two keys may hash to the same slot. We call this situation a ***collision***.
- The ideal solution would be to avoid collisions altogether or at least minimizing their number.
- We might try to achieve this goal by choosing a suitable hash function h.
- Because $|U| > m$, however, there must be at least two keys that have the same hash value; avoiding collisions altogether is therefore impossible.

Collision Resolution by Chaining

- In ***chaining***, we place all the elements that hash to the same slot into the same linked list, as shown.



Hash Table Operations

- The dictionary operations on a hash table *T* are easy to implement when collisions are resolved by chaining:

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 $T[h(x.key)]$

Hash Functions (Cont'd)

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

Division Hash Function

- In the ***division method*** for creating hash functions, we map a key k into one of m slots by taking the remainder of k divided by m. That is, the hash function is $h(k) = k \bmod m$.
- For example, if the hash table has size $m = 12$ and the key is $k = 100$, then $h(k) = 4$.
- Example of bad choice of m is 2^p, as $h(k) = p$ lowest-order bits of k.
- We are better off designing the hash function to depend on all the bits of the key.
- A prime not too close to an exact power of 2 is often a good choice for m. Example:
  - Suppose we wish to allocate a hash table, with collisions resolved by chaining, to hold elements with keys range from 0 to 2000. We don't mind examining an average of 3 elements in an unsuccessful search, and so we allocate a hash table of size $m = 701$ because it is a prime near 2000/3 but not near any power of 2.

Resolving Collisions by Open Addressing

- In ***open addressing***, all elements occupy the hash table itself. That is, each table entry contains either an element of the dynamic set or NIL.
- When searching for an element, we systematically examine table slots until either we find the desired element, or we have ascertained that the element is not in the table.
- Unlike chaining, no lists and no elements are stored outside the table.
- Thus, in open addressing, the hash table can "**fill up**" so that no further insertions can be made.
- the advantage of open addressing is that it **avoids pointers** altogether.
- Instead of following pointers, we *compute* the sequence of slots to be examined.

Open Addressing (Cont'd)

- To perform insertion using open addressing, we successively examine, or ***probe***, the hash table until we find an empty slot in which to put the key.
- To determine which slots to probe, we extend the hash function to include the probe number (starting from 0) as a second input.
- For every key k, The probe sequence

$$\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$$

  should be a permutation of $\{0, 1, \ldots, m-1\}$.
- So that every hash-table position is eventually considered as a slot for a new key as the table fills up.

Open Addressing - Insert

- The following HASH-INSERT procedure takes as input a hash table *T* and a key *k*. It either returns the slot number where it stores key *k* or flags an error because the hash table is already full.

```
HASH-INSERT(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == NIL
        T[j] = k
        return j
    else i = i + 1
until i == m
error "hash table overflow"
```

Open Addressing - Search

- The algorithm for searching for key *k* probes the same sequence of slots that the insertion algorithm examined when key *k* was inserted.
- Therefore, the search can terminate (unsuccessfully) when it finds an empty slot.
  (This argument assumes that keys **were not deleted** from the hash table.)

```
HASH-SEARCH(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == k
        return j
    i = i + 1
until T[j] == NIL or i == m
return NIL
```

Open Addressing - Delete

- Deletion from an open-address hash table is difficult. When we delete a key from slot *i* , we cannot simply mark that slot as empty by storing NIL in it. If we did, we might be unable to retrieve any key *k* during its insertion we had probed slot *i* and found it occupied.
- We can solve this problem by marking the slot, storing in it the **special value DELETED** instead of **NIL**.
- We would then modify the procedure HASH-INSERT to treat such a slot as if it were empty so that we can insert a new key there.
- We do not need to modify HASH-SEARCH, since it will pass over DELETED values while searching.

Linear Probing

- Given an ordinary hash function $h'(k)$, linear probing uses the hash function:

$$h(k,i) = (h'(k) + i) \bmod m \text{ for } i = 0,1, \ldots, m\text{-}1.$$

- Given key k, we first probe $T[h'(k)]$, i.e., the slot given by the auxiliary hash function.
- We next probe slot $T[h'(k) + 1]$, and so on up to slot $T[m - 1]$ and then we wrap around to slots $T[0]$, $T[1]$, … until we finally probe slot $T[h'(k) - 1]$. Because the initial probe determines the entire probe sequence, there are only m distinct probe sequences.