Design of Algorithms

Data Structures

Lecture 10: Elementary Data Structures

Sultan ALPAR
associate professor, IITU
s.alpar@iitu.edu.kz

Introduction

- Sets manipulated by algorithm can grow, shrink, or change over time.
- Called dynamic set.
- Types of operations to be performed on sets:
  - Insert, delete, search, etc.
  - Extract the smallest element, etc.

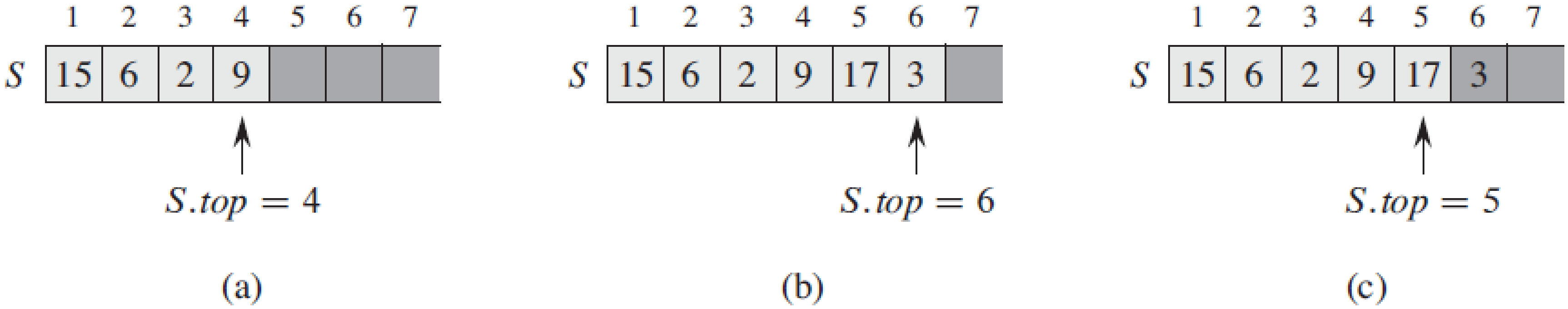
Introduction (continue)

- Operations to be performed on dynamic sets:
  - queries: return information about the set.
    - Search, minimum, maximum
    - Successor, predecessor.
  - modifying operations: change the set.
    - Insert, delete
- Data structures that can support any of these operations on a set of size n: $O(\log n)$

10.1 Stacks and Queues

- Stacks and Queues
  - dynamic set
  - elements removed from the set by the DELETE operation is pre-specified
- Stack
  - LIFO policy: Last-In First-Out
  - Delete the element most recently inserted
  - Push (insert), pop (delete)
- Queue
  - FIFO policy: First-In First-Out
  - Enqueue (insert), dequeue (delete)

An array implementation of a stack S



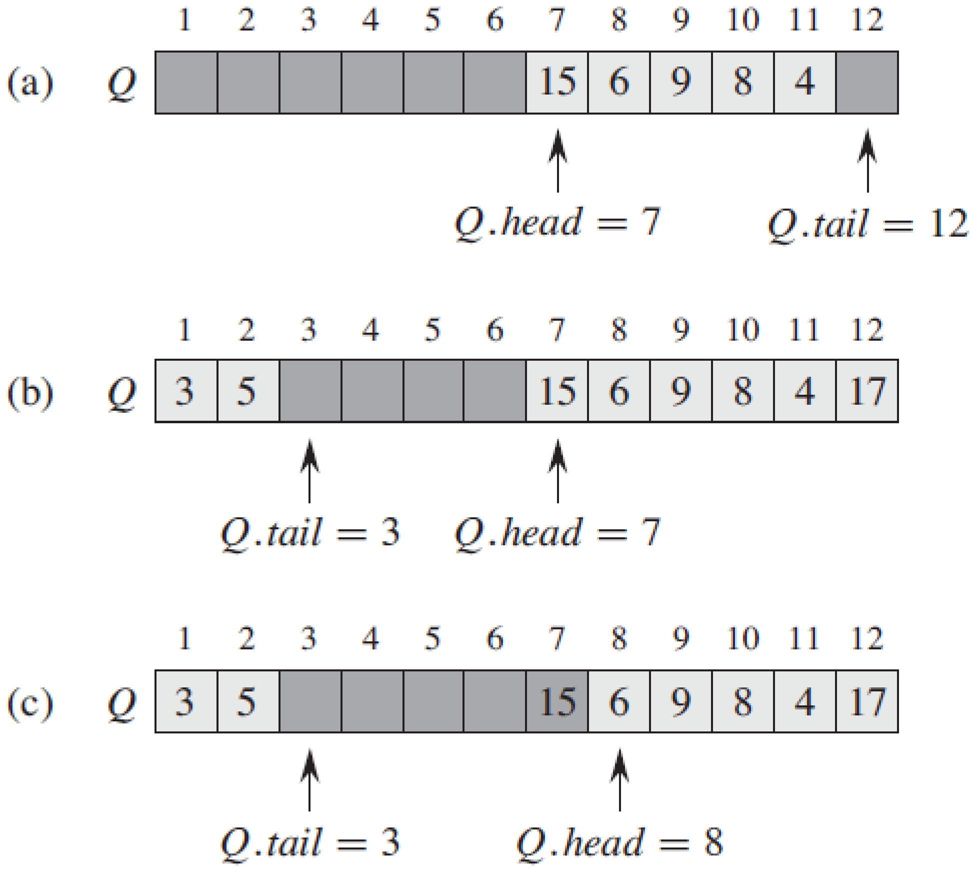
- empty, underflows, overflows

```
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($S$ )
      then error “underflow”
      else $S.top = S.top - 1$
          return $S[S.top + 1]$
```

An array implementation of a queue Q



ENQUEUE(*Q*, *x*)

```
$Q[Q.tail] = x$
if $Q.tail == Q.length$
    $Q.tail = 1$
else $Q.tail = Q.tail + 1$
```

Anything wrong?

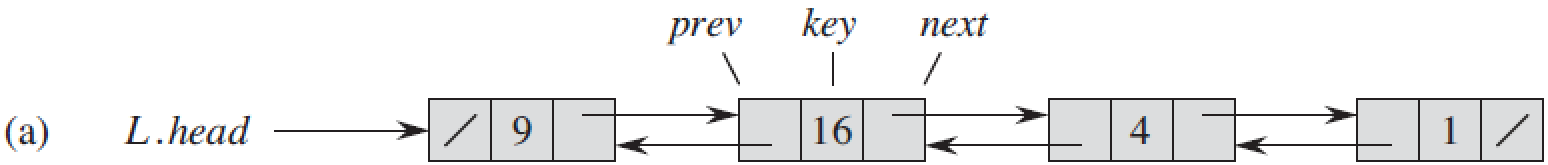
DEQUEUE(*Q*)

```
$x = Q[Q.head]$
if $Q.head == Q.length$
    $Q.head = 1$
else $Q.head = Q.head + 1$
return $x$
```

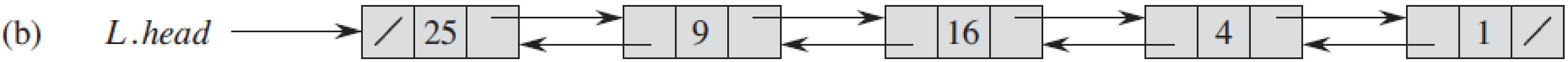
10.2 Linked lists

- Data structure in which the objects are arranged in a linear order.
- The order in a linked list is determined by a pointer in each object.
- The order in an array is determined by the array indices.
- Singly linked list, doubly linked list, circular list.
- Head and tail.

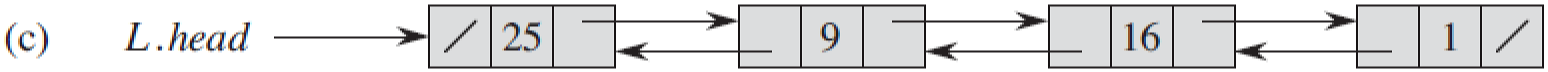
10.2 Linked lists



List-Insert(L, x), *x.key*=25



List-Delete(L, x), *x.key*=4



LIST_SEARCH(*L*,*k*)

```
$x = L.head$
while $x \neq \text{NIL}$ and $x.key \neq k$
    $x = x.next$
return $x$
```

$O(n)$ time in the worst case

LIST_INSERT(*L*,*x*)

```
1  $x.next = L.head$
2  **if** $L.head \neq$ NIL
3        $L.head$.prev = $x$
4  $L.head = x$
5  $x.prev$ = NIL
```

$O(1)$

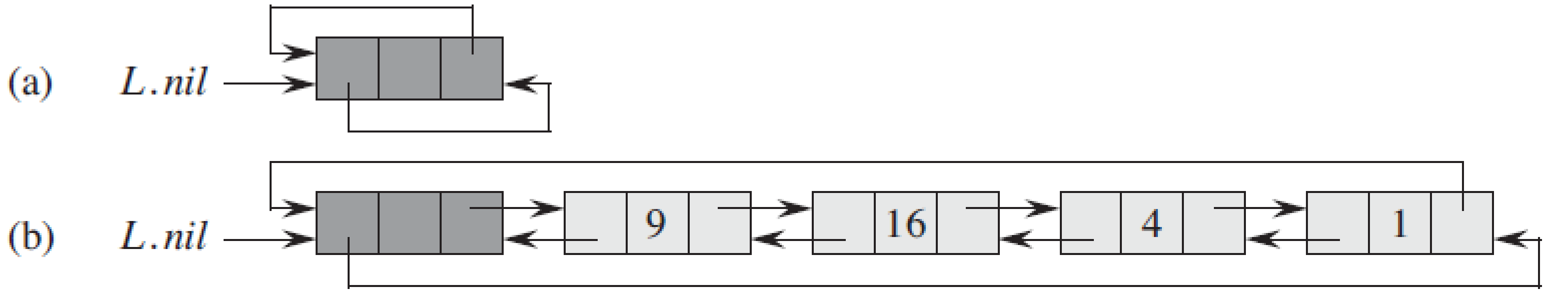
LIST_DELETE(*L*,*x*)

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

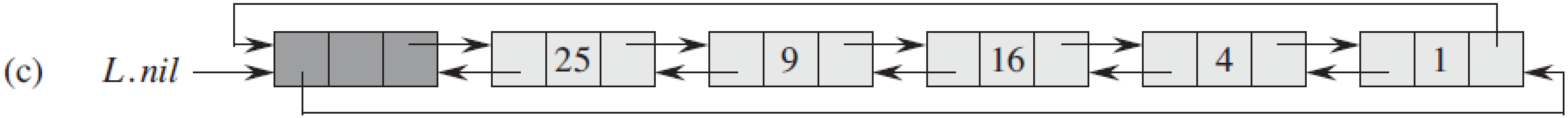
- (Call LIST_SEARCH first O(*n*))

$O(1)$ or $O(n)$

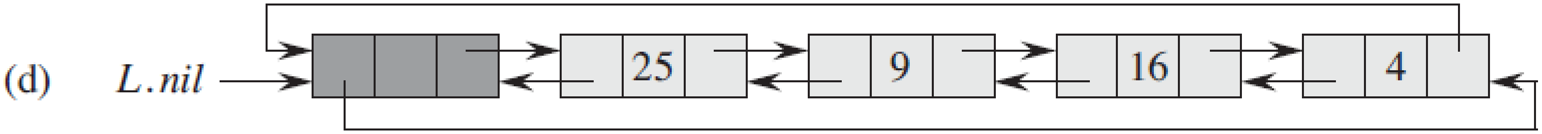
A Sentinel is a dummy object that allows us to simplify boundary conditions,



List-Insert'(L, x), *x.key*=25



List-Delete'(L, x), *x.key*=4



LIST_DELETE'(*L*,*x*)

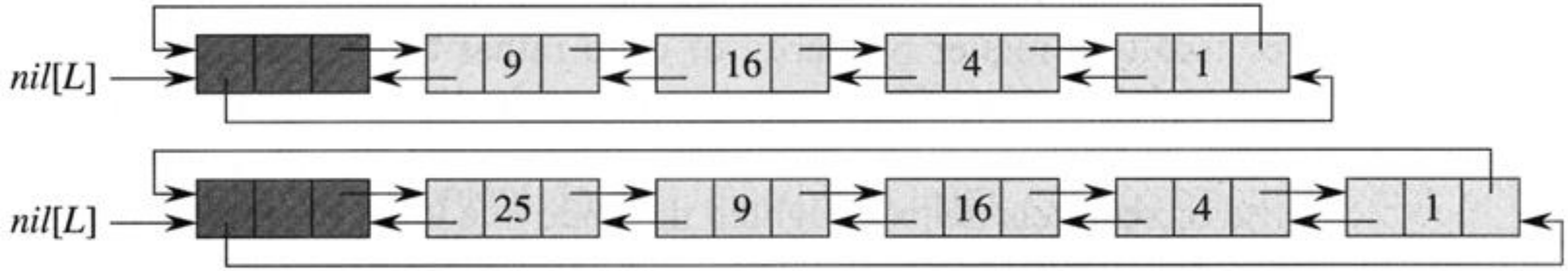
```
$x.prev.next = x.next$
$x.next.prev = x.prev$
```

LIST_SEARCH'(*L*,*k*)

```
$x = L.nil.next$
while $x \neq L.nil$ and $x.key \neq k$
    $x = x.next$
return $x$
```

LIST_INSERT'(*L*,*x*)

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

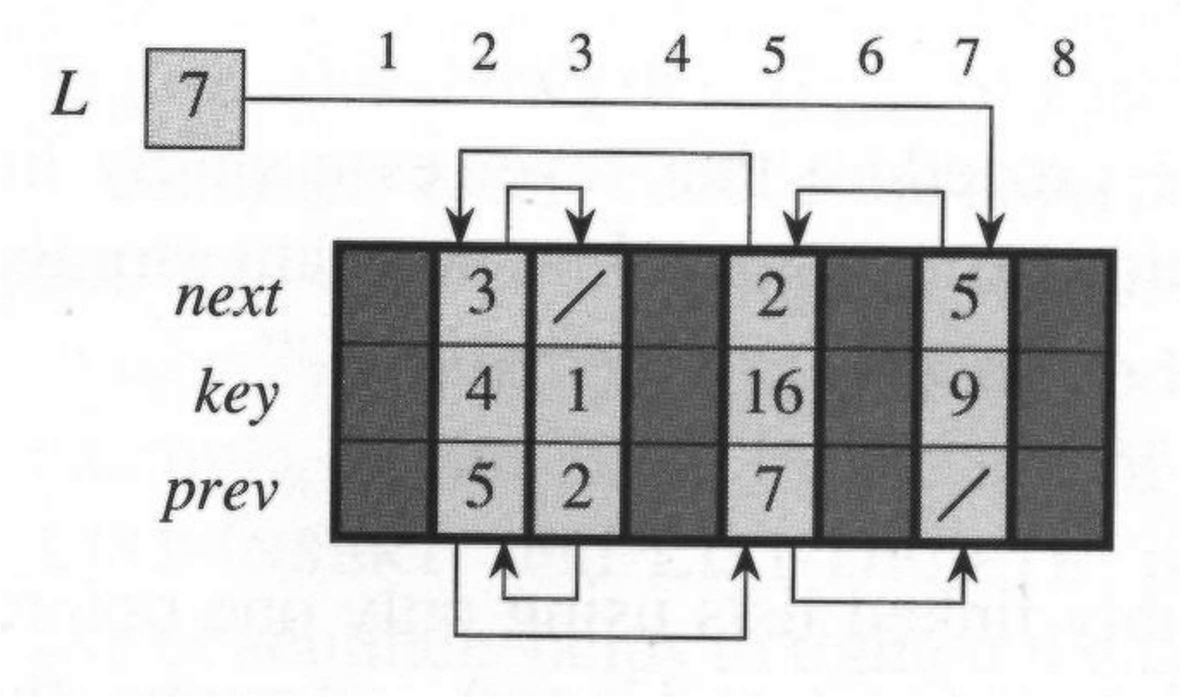


Remarks on sentinels

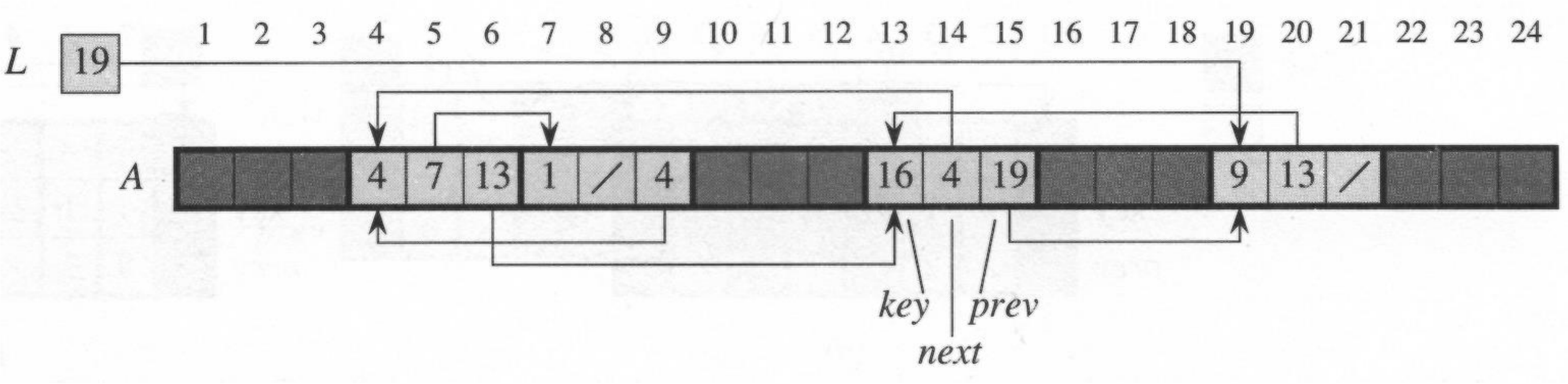
- Rarely reduce the asymptotic time bounds of data structure operations.
- Can only reduce constant factors.
- Can improve the clarity of code rather than speed.
- The extra storage used by the sentinels, for small lists, can represent significant wasted memory.
- Use sentinels only when they truly simplify the code.

11.3 Implementing pointers and objects

- A multiple-array representation of objects



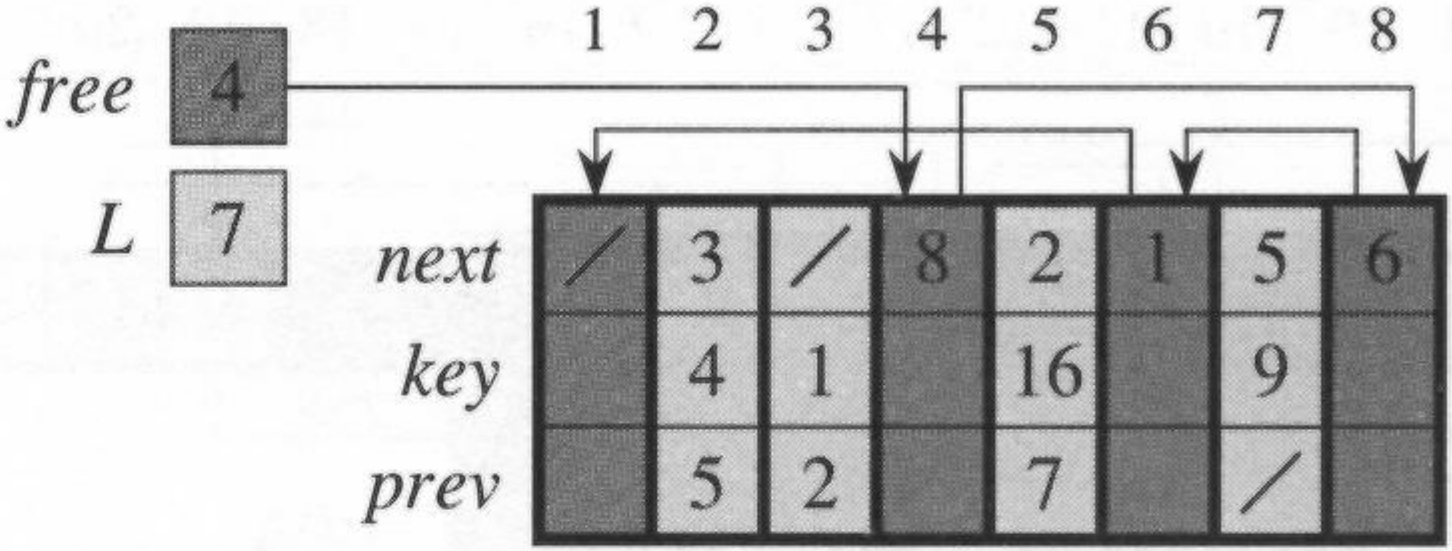
A single array representation of objects



Allocating and freeing objects:

- Garbage collector
  - Determining which objects are unused.
- Free list
  - Singly linked list that keeps the free objects
  - Uses only the “next” pointer.
  - Stack.

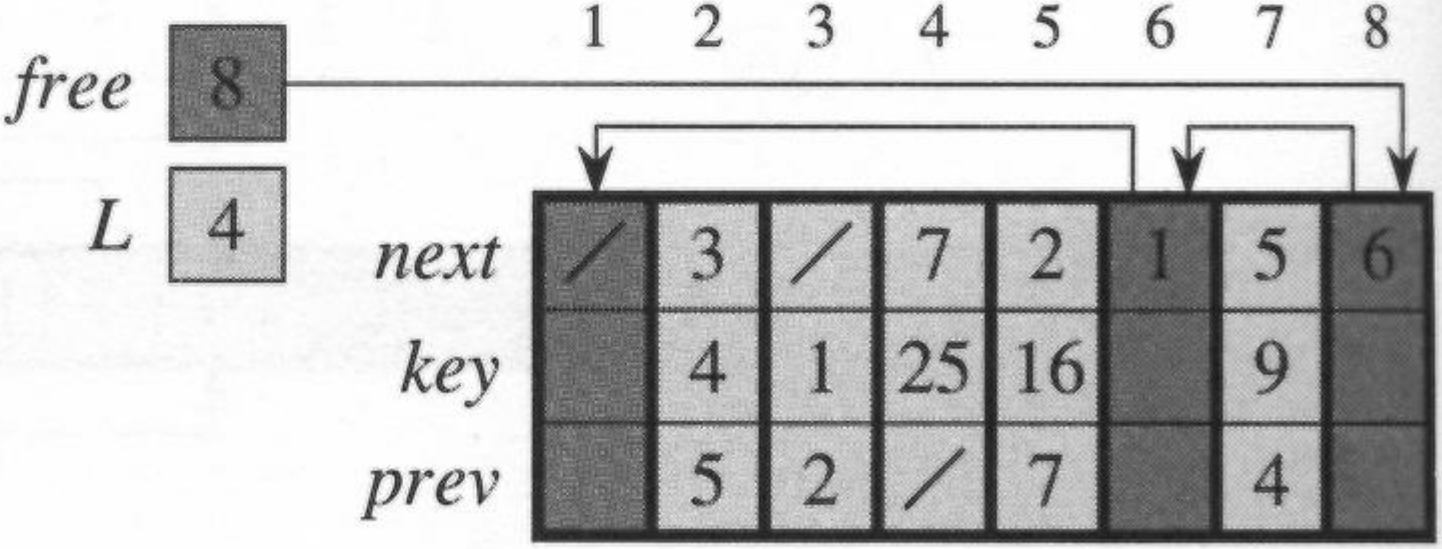
Allocating and freeing objects--garbage collector



	1	2	3	4	5	6	7	8
next	/	3	/	8	2	1	5	6
key		4	1		16		9	
prev		5	2		7		/	

(a)

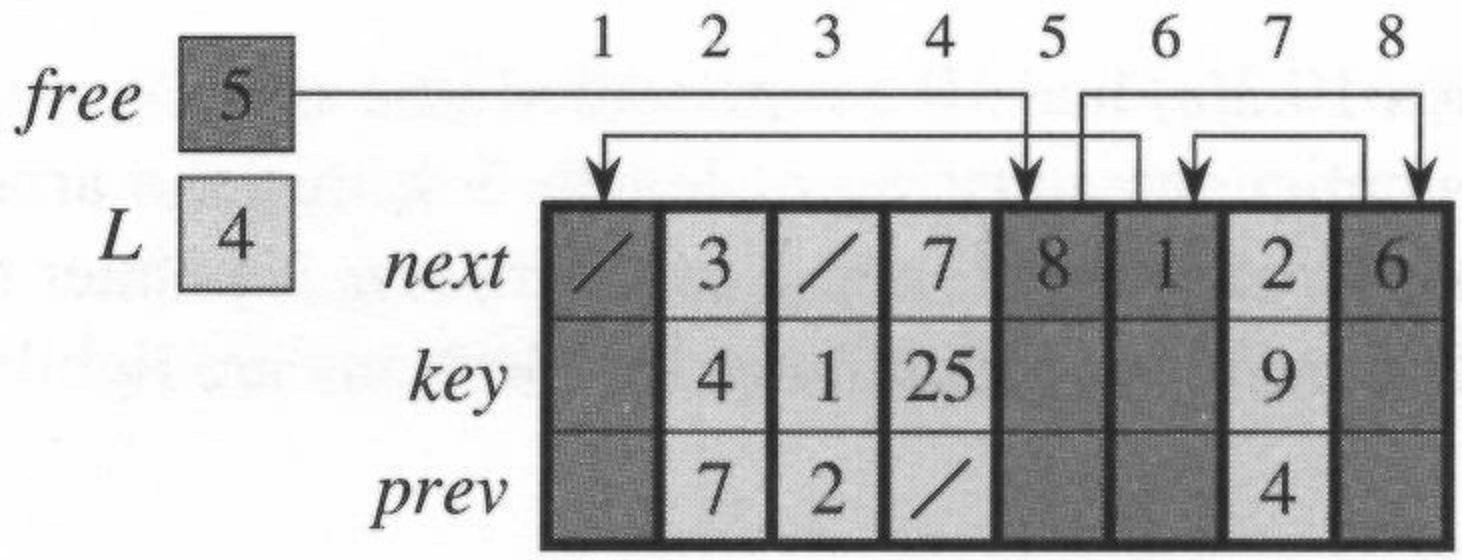
Allocate_object( ),LIST_INSERT(L,4),Key(4)=25



	1	2	3	4	5	6	7	8
next	/	3	/	7	2	1	5	6
key		4	1	25	16		9	
prev		5	2	/	7		4	

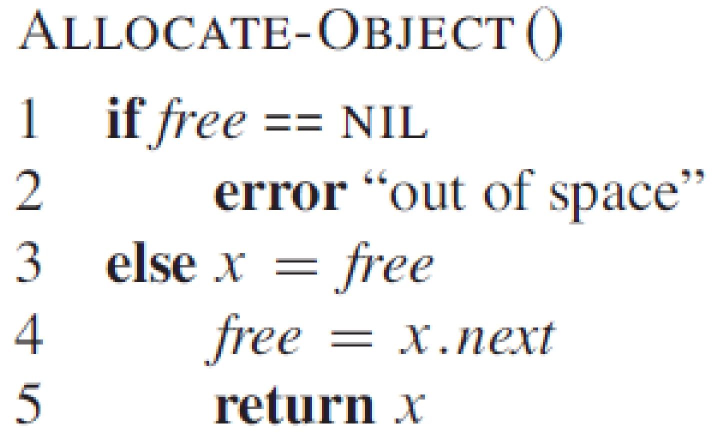
(b)

LIST_DELETE(*L*,5), FREE_OBJECT(5)

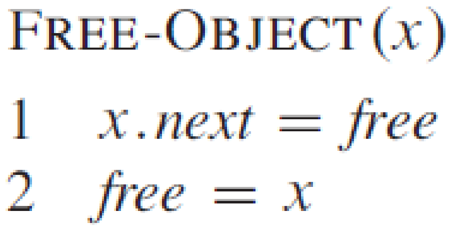


	1	2	3	4	5	6	7	8
next	/	3	/	7	8	1	2	6
key		4	1	25			9	
prev		7	2	/			4	

(c)

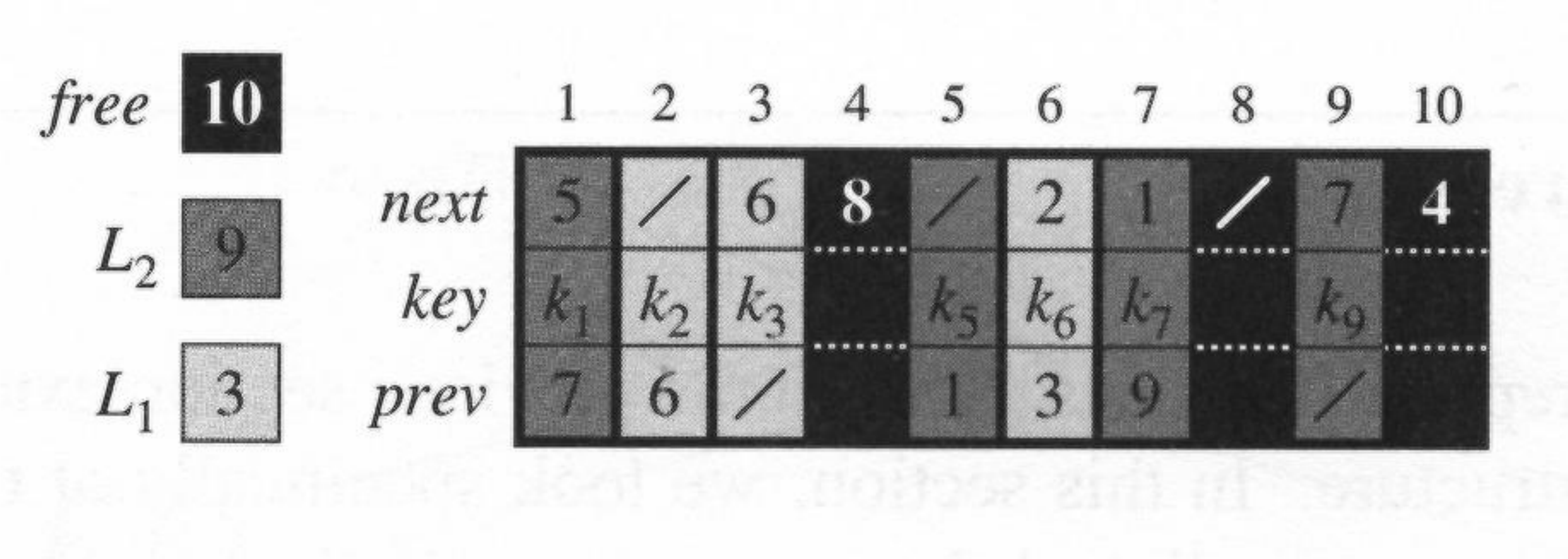


```
ALLOCATE-OBJECT()
if $free$ == NIL
    error "out of space"
else $x = free$
    $free = x.next$
    return $x$
```



```
FREE-OBJECT($x$)
$x.next = free$
$free = x$
```

Two link lists



free 10

L_2 9

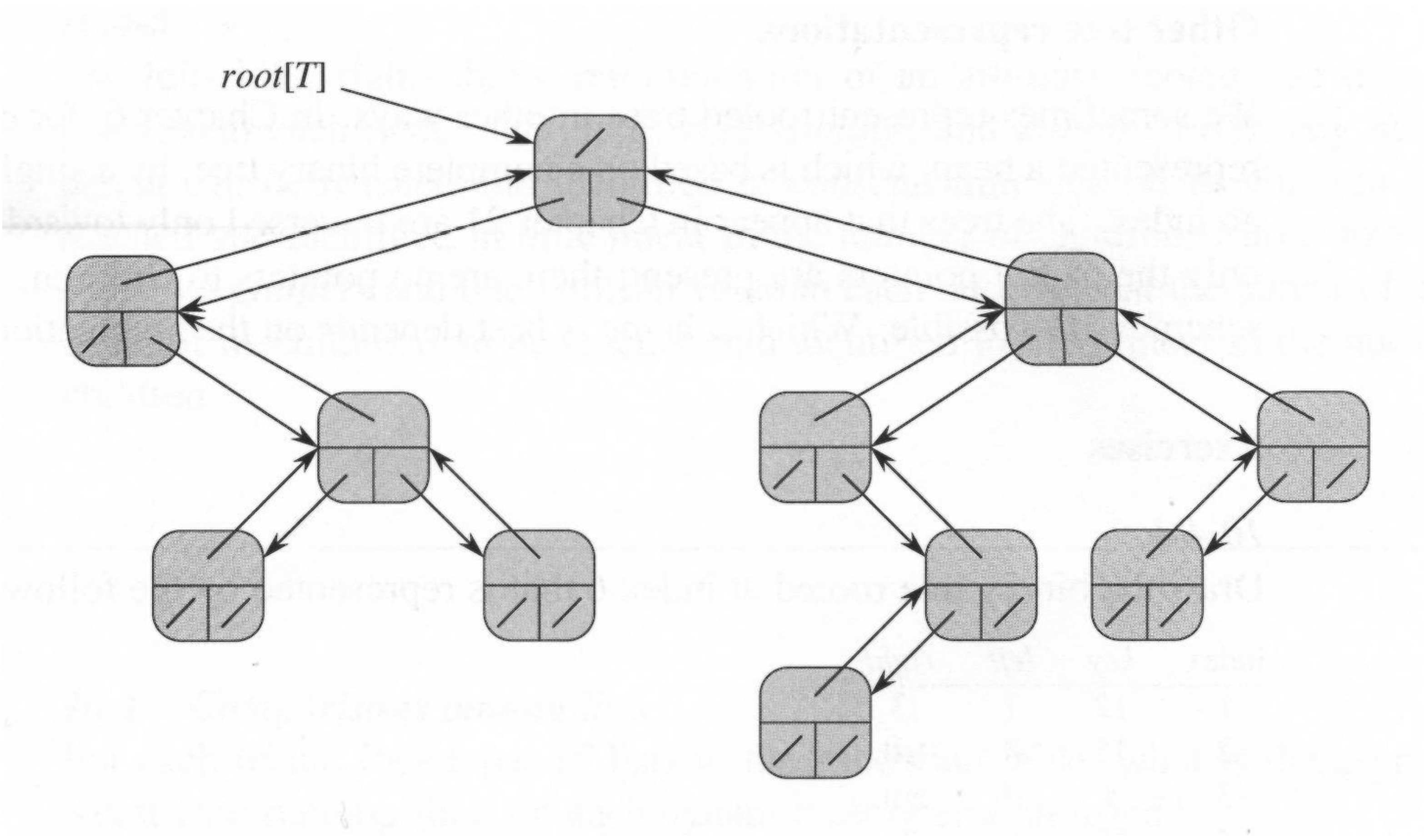
L_1 3

	1	2	3	4	5	6	7	8	9	10
next	5	/	6	8	/	2	1	/	7	4
key	k_1	k_2	k_3		k_5	k_6	k_7		k_9	
prev	7	6	/		1	3	9		/	

10.4 Representing rooted trees

- Binary trees:
  - parent, left(-child), right(-child)
  - p[x] = NIL ➔ x is the root
  - x has no left child ➔ left[x] = NIL
  - x has no right child ➔ right[x] = NIL

Binary trees

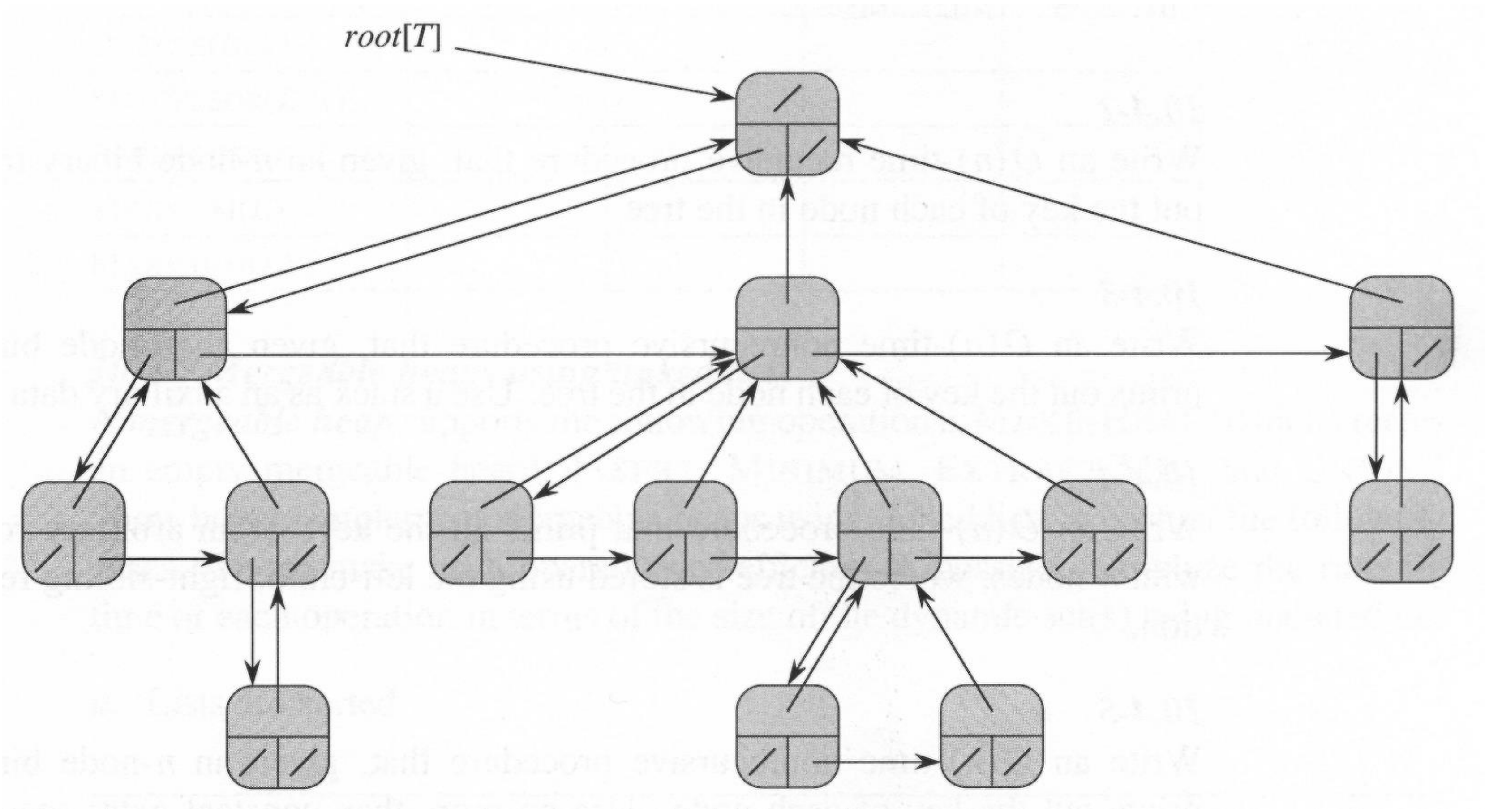


10.4 Representing rooted trees

- Rooted trees with unbounded branching:
  - parent, left-child, right-sibling
  - p[x] = NIL ➔ x is the root
  - left-child[x]: points to the leftmost child of x.
  - right-sibling[x]: points to the sibling of x immediately to the right.
  - x has no children ➔ left-child[x] = NIL
  - x is the rightmost child of its parent

    ➔ right-sibling[x] = NIL

Rooted tree with unbounded branching



- Thank u for Attention!