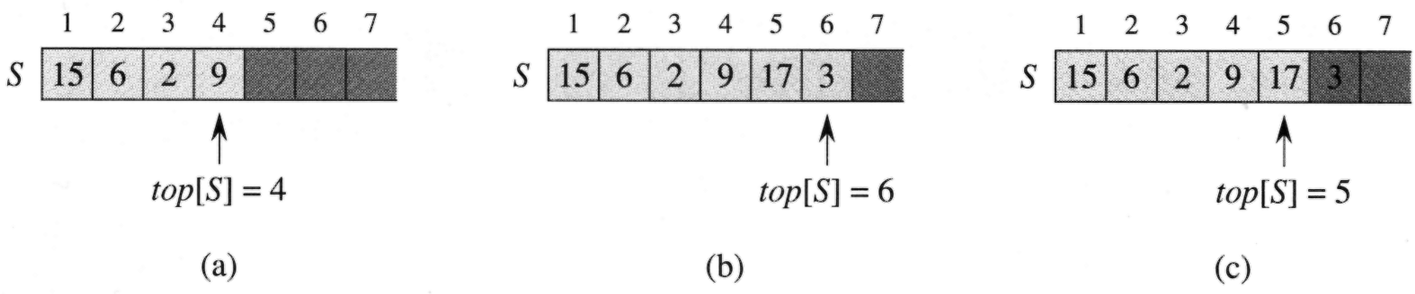
10.Elementary data structures

Hsu, Lih-Hsing

10.1 Stacks and queues

Stacks and queues are dynamic set in which element removed from the set by the DELETE operation is prespecified. In a **stack** the element deleted from the set is the one most recently inserted; the stack implements a ***last-in, first-out***, or **LIFO**, policy. Similarly, in a **queue**, the element deleted is implements a ***first-in, first-out***, or **FIFO**, policy.

An array implementation of a stack S



- empty, underflows, overflows

```
STACK_EMPTY(S)
if top[S] = 0
    then return TRUE
    else return FALSE
```

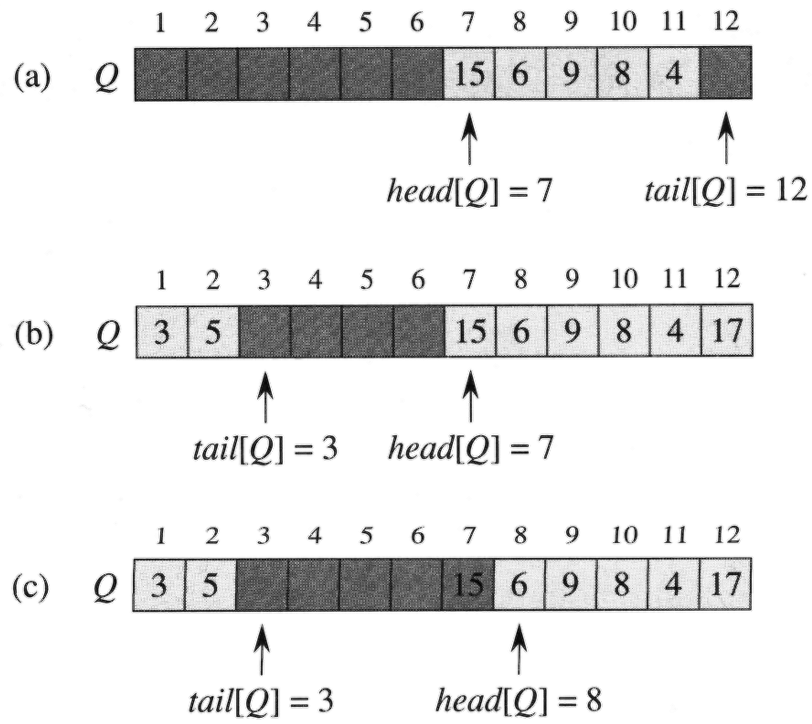
PUSH(*S*,*x*)

```
$top[S] \leftarrow top[S]+1$
$S[top[S]] \leftarrow x$
```

POP(*S*)

```
if STACK-EMPTY(S )
    then error “underflow”
    else  $top[S] \leftarrow top[s]$ -1
          return $S[top[S] + 1]$
```

An array implementation of a queue Q



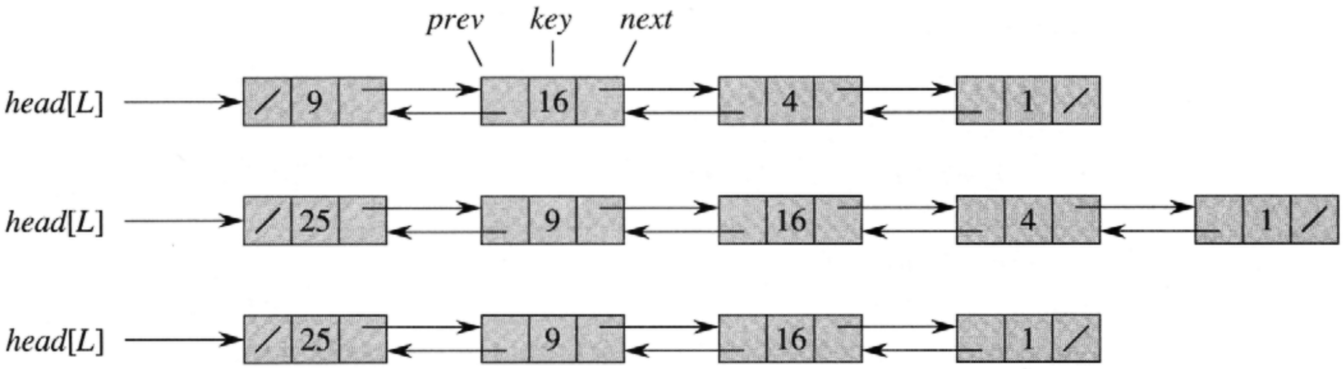
ENQUEUE(Q,S)

```
$Q[tail[Q]] \leftarrow x$
if $tail[Q] = length[Q]$
    then $tail[Q] \leftarrow 1$
    else $tail[Q] \leftarrow tail[Q]+1$
```

DEQUEUE(Q)

```
$x \leftarrow Q[head[Q]]$
if $head[Q] = length[Q]$
    then $head[Q] \leftarrow 1$
    else $head[Q] \leftarrow head[Q] + 1$
return $x$
```

10.2 Linked lists



LIST_SEARCH(*L*,*k*)

```
$x \neq head[L]$
while $x \neq \mathrm{NIL}$ and $key[x] \neq k$
    do $x \neq next[x]$
return $x$
```

$O(n)$

LIST_INSERT(*L*,*x*)

```
$next[x] \leftarrow head[L]$
if $head[L] \neq \mathrm{NIL}$
    then $prev[head[L]] \leftarrow x$
$head[L] \leftarrow x$
$prev[x] \leftarrow \mathrm{NIL}$
```

$O(1)$

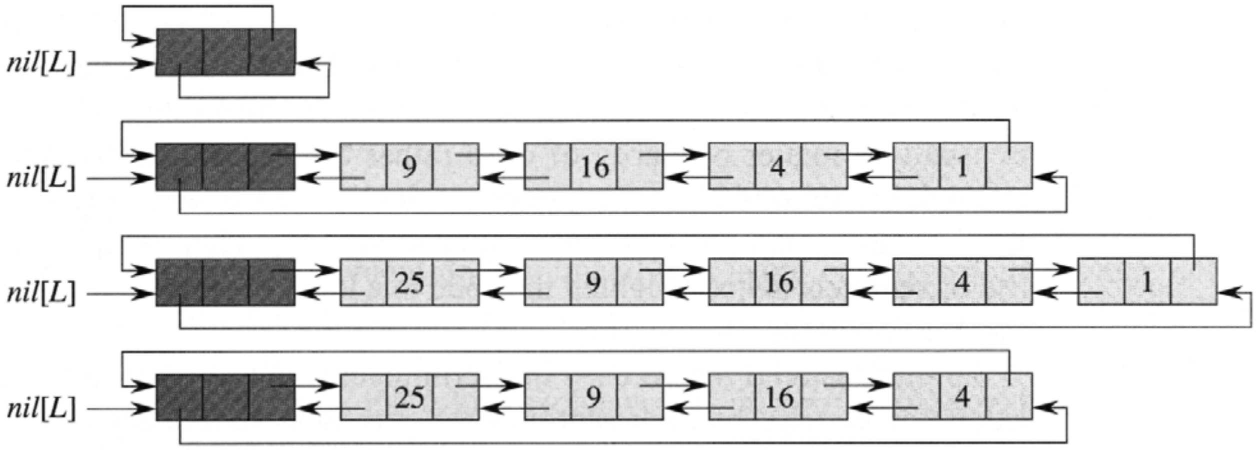
LIST_DELETE(*L*,*x*)

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

```
if $prev[x] \neq \mathrm{NIL}$
    then $next[prev[x]] \leftarrow next[x]$
    else $head[L] \leftarrow next[x]$
if $next[x] \neq \mathrm{NIL}$
then $prev[next[x]] \leftarrow prev[x]$
```

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

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



LIST_DELETE'(*L*,*x*)

```
$next[prev[x]] \leftarrow \mathrm{next}[x]$
$prev[next[x]] \leftarrow \mathrm{prev}[x]$
```

LIST_SEARCH’(*L*,*k*)

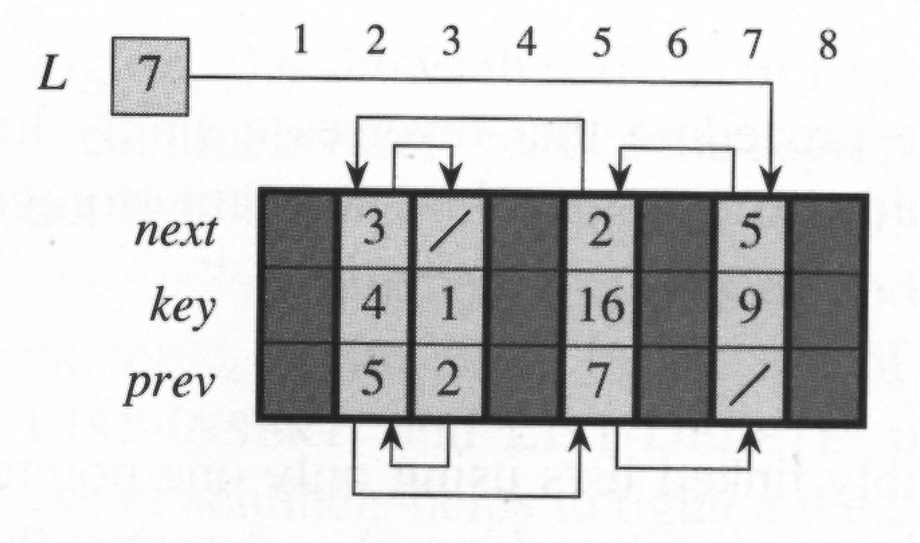
```
$x \leftarrow next[nil[L]]$
while $x \neq nil[L]$ and $key[x] \neq k$
      do $x \leftarrow next[x]$
return $x$
```

LIST_INSERT’(*L*,*x*)

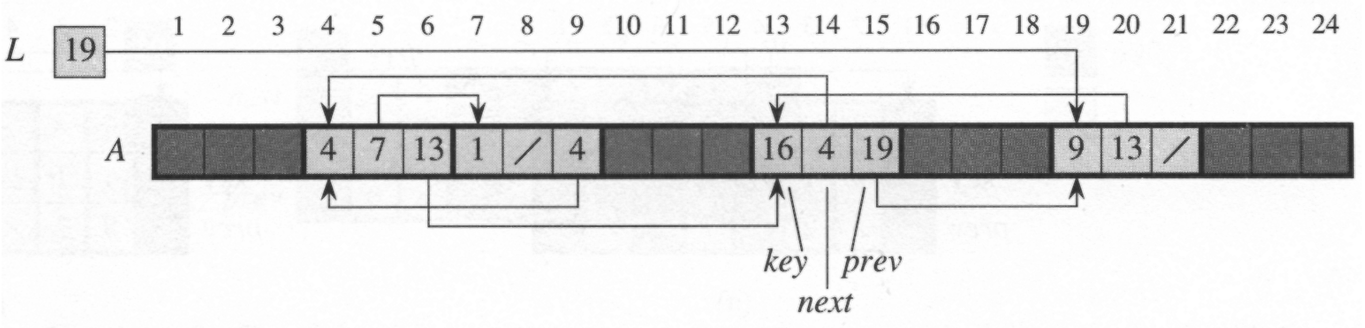
```
$next[x] \leftarrow next[nil[L]]$
$prev[next[nil[L]]] \leftarrow x$
$next[nil[L]] \leftarrow x$
$prev[x] \leftarrow nil[L]$
```

11.3 Implementing pointers and objects

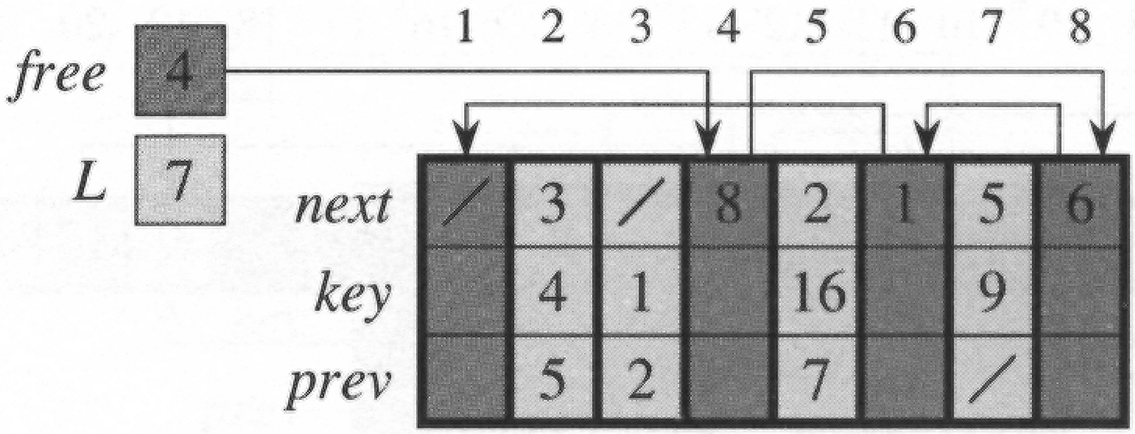
- A multiple-array representation of objects



A single array representation of objects



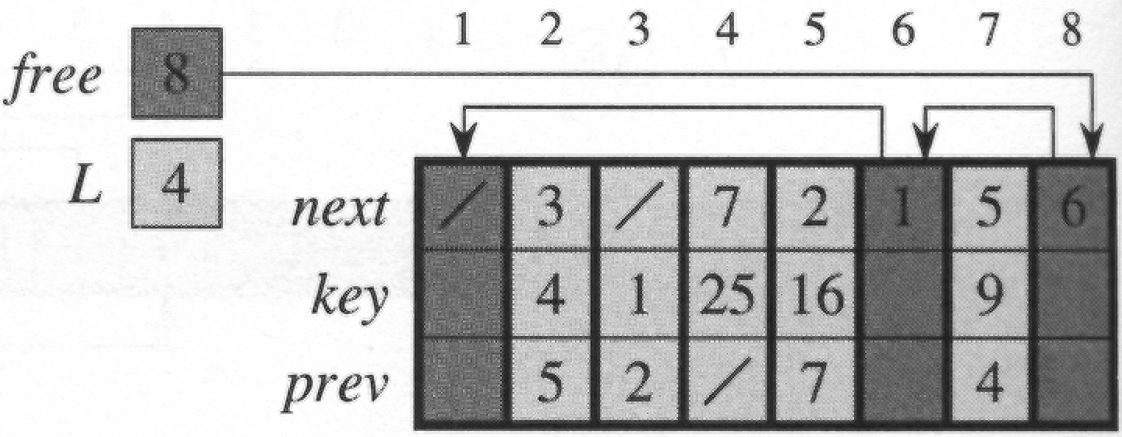
Allocating and freeing objects--garbage collector



	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)

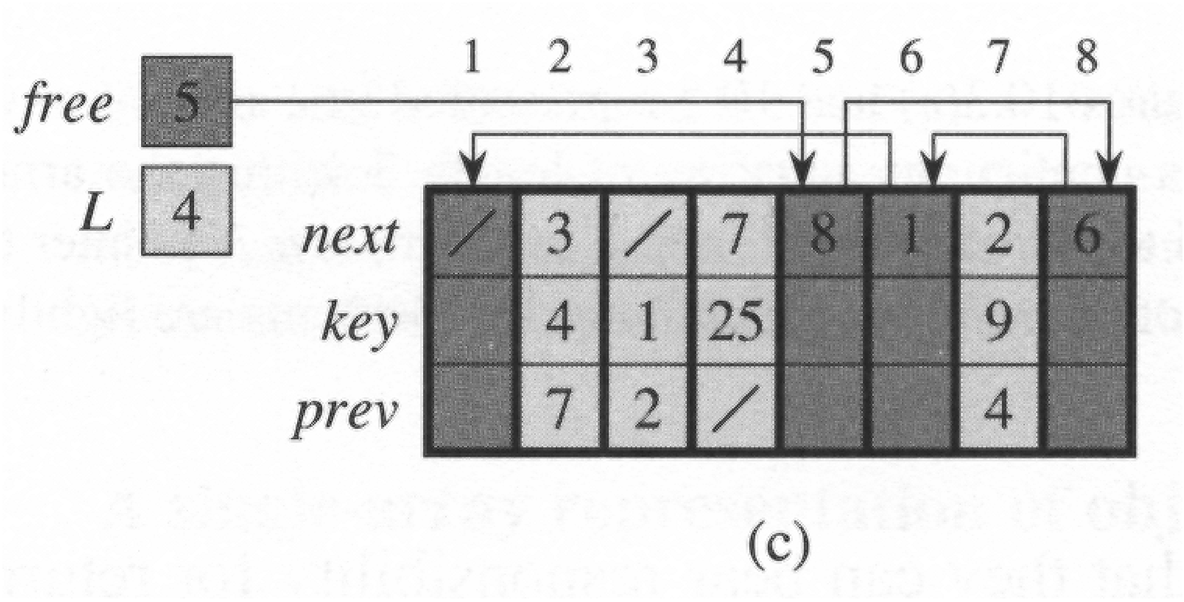
Allocate_object( ),LIST_INSERT(L,4),Key(4)=25



	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)

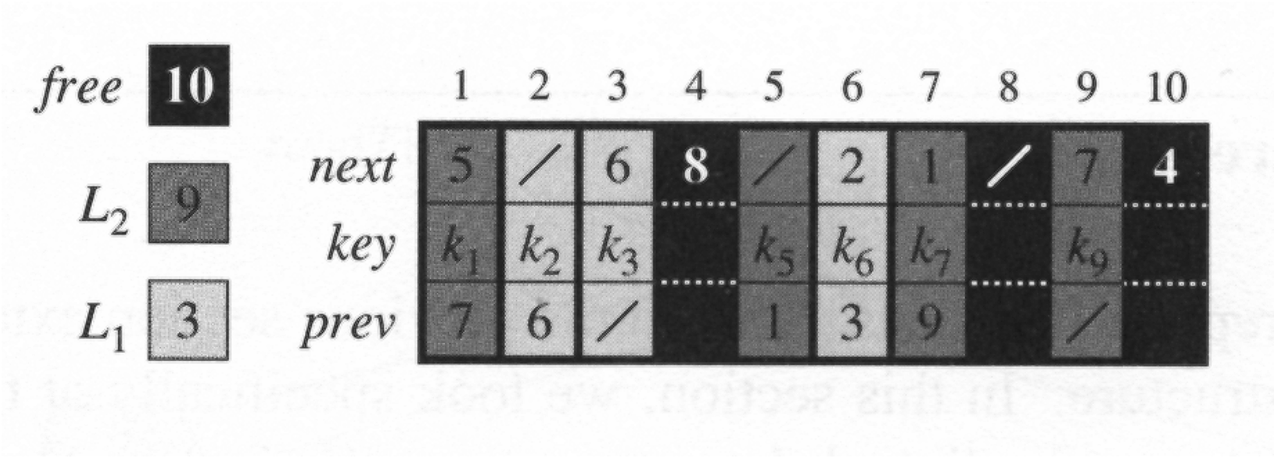
ALLOCATE_OBJECT( )

```
if $free$ = NIL
    then error "out of space"
    else $x \leftarrow free$
        $free \leftarrow next[x]$
        return $x$
```

FREE_OBJECT(x)

```
$next[x] \leftarrow free$
$free \leftarrow x$
```

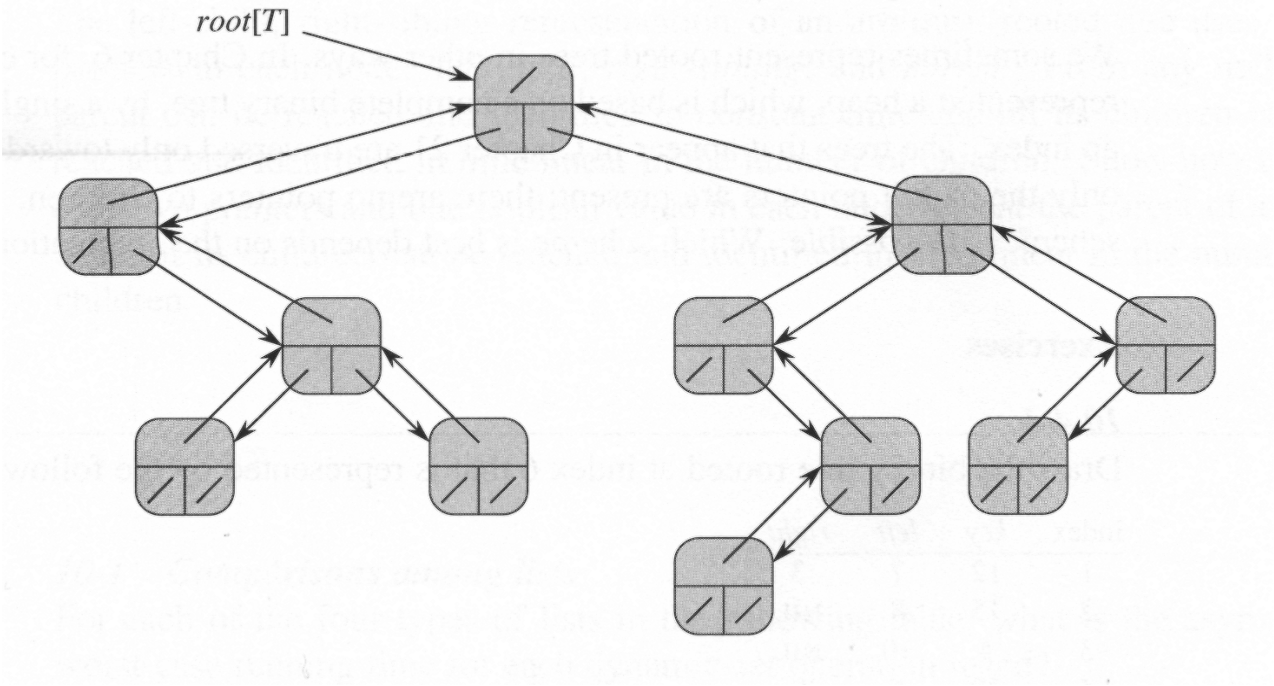
Two link lists



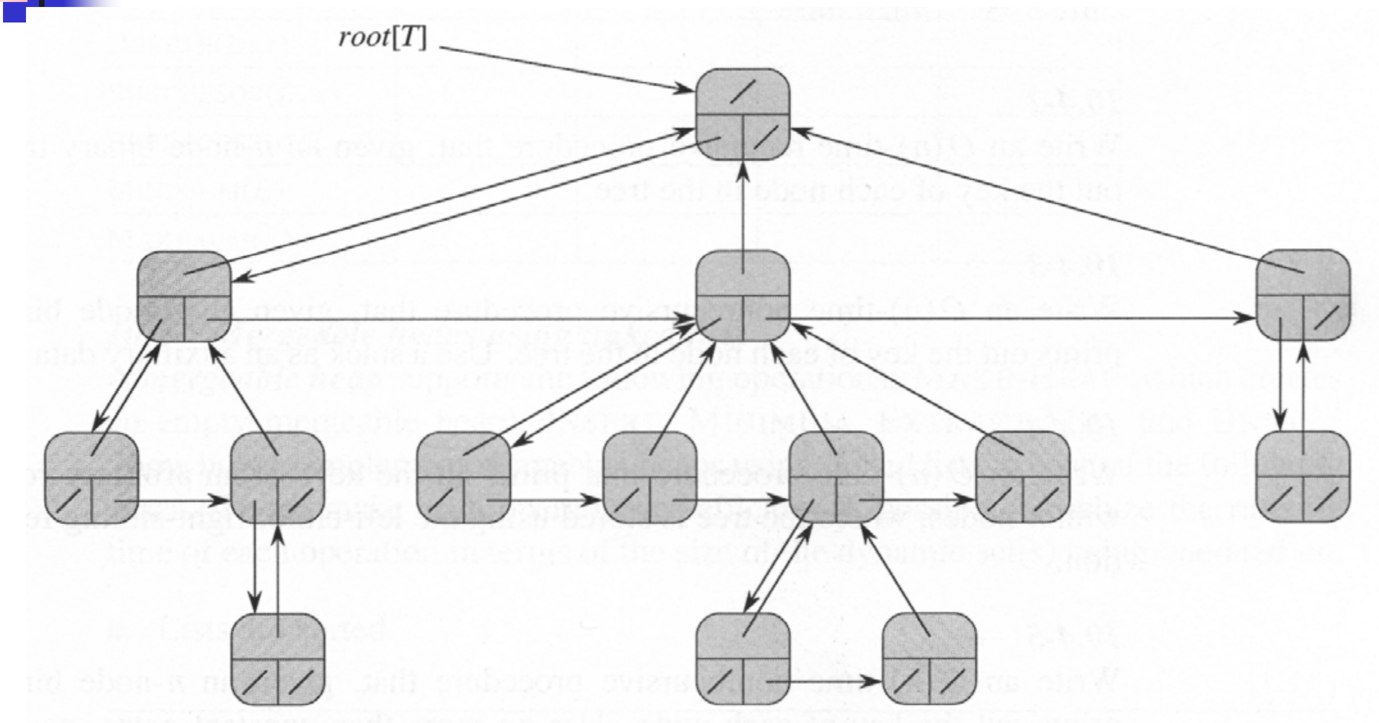
	1	2	3	4	5	6	7	8	9	10
next	5	/	6	8	/	2	1	/	7	4
key	k_1	k_2	k_3		k_5	k_6	k_7		k_9	
prev	7	6	/		1	3	9		/	

10.4 Representing rooted trees

Binary trees



Rooted tree with unbounded branching



- Other tree representation