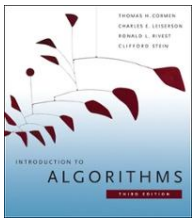
Introduction to Algorithms

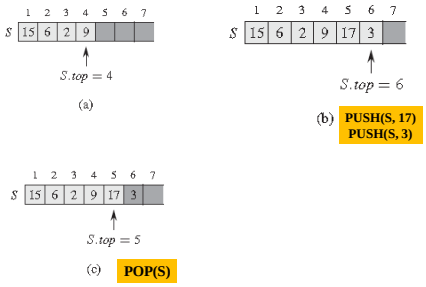


Chapter 10: Elementary data structures

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**

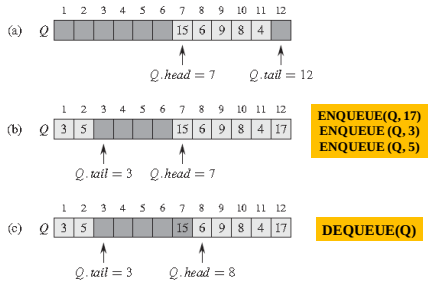
```
STACK_EMPTY(S )
1   if S.top == 0
2       return TRUE
3   else return FALSE
```

$O(1)$

PUSH(*S*,*x*)
1 S.*top* = *S.top*+1
2 *S*[S.*top*] = *x* *O*(1)

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

An array implementation of a queue Q



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

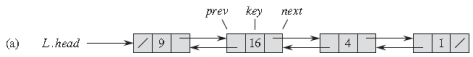
O(*1*)

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*

O(*1*)

10.2 Linked lists

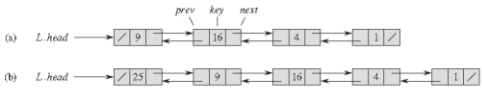
- **Doubly linked list** L: *key*, *next*, *prev*
- If $x.prev$ = NIL, the element has no predecessor and is the first element, or head, of the list.
- If $x.next$ = NIL, the element has no successor and is the last element, or tail, of the list.
- $L.head$ points to the first element of the list.
- If $L.head$ = NIL, the list is empty.



LIST-SEARCH(L, 4) returns a pointer to the third element
LIST-SEARCH(L, 7) returns NIL

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

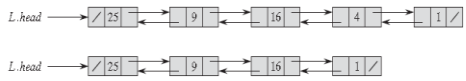
$\Theta(n)$ in worst case



LIST-INSERT(L, 25)

```
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
```

$O(1)$

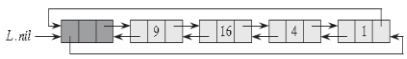


LIST-DELETE(L, 4)

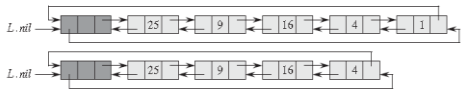
```
LIST_DELETE(L, x)
1  if x.prev ≠ NIL
2       x.prev.next = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5       x.next.prev = x.prev          O(1) or O(n)
```

Delete a specified element (Call LIST_SEARCH first $O(n)$)

Circular, doubly linked list with a sentinel



- A Sentinel is a dummy object to simplify boundary conditions.
- The sentinel *L.nil* is placed between the head and the tail.
- *L.nil.next* points to the head of the list and *L.nil.prev* points to the tail.
- Both the *next* field of the tail and the *prev* field of the head point to *L.nil*.
- We can eliminate the attribute *L.head*, since *L.nil.next* points to the head.



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

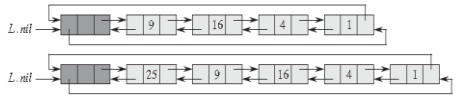
```
LIST_DELETE'(L, x)
1  x.prev.next  = x.next
2  x.next.prev  = x.prev
```

LIST-DELETE(L, 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) or O(n)
```

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

```
LIST-SEARCH'(L, k)
1  x = L.nil.next
2  while x ≠ NIL  and x.key ≠ k
3        x = x.next
4  return  x
```

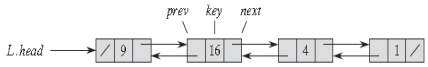

```
LIST-INSERT(L, x)
1  x.next = L.head
2  if L.head  ≠ NIL
3     L.head.prev = x
4  L.head  = x
5  x.prev  = NIL
```

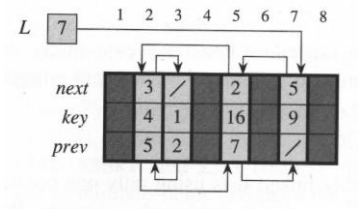
```
LIST-INSERT'(L, x)
1  x.next = L.nil.next
2     L.nil.next.prev = x
4  L.nil.next  = x
5  x.prev  = L.nil
```

11.3 Implementing pointers and objects

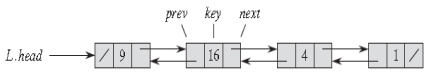
- How to implement pointers and objects in languages, such as Fortran, that do not provide them?

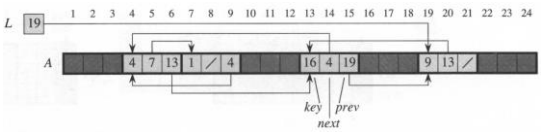
Multiple-array representation of objects



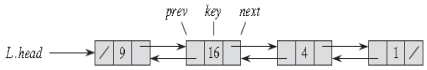


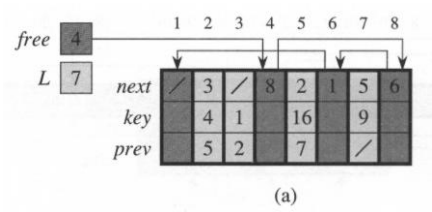
A single array representation of objects



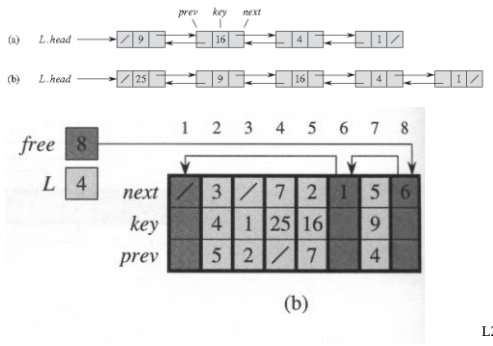


Allocating and freeing objects--garbage collector

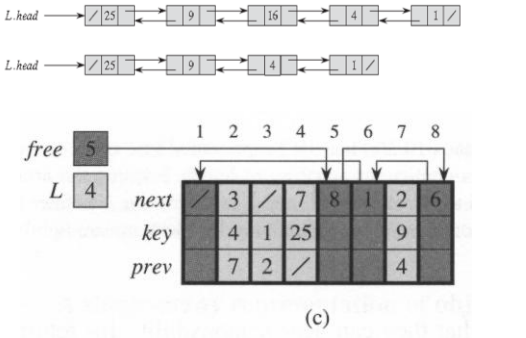




Allocate_object( ), LIST_INSERT(L,4), Key(4)=25



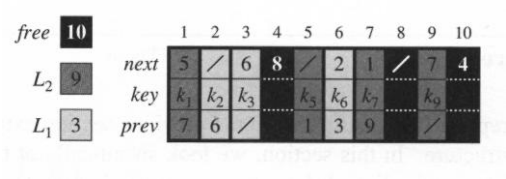
LIST_DELETE(*L*,5), FREE_OBJECT(5)




```
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
```

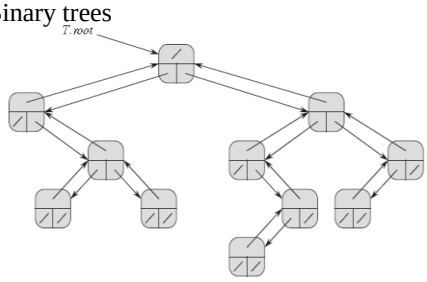

Two link lists



Two linked lists, L_1 (lightly shaded) and L_2 (heavily shaded), and a free list (darkened) intertwined.

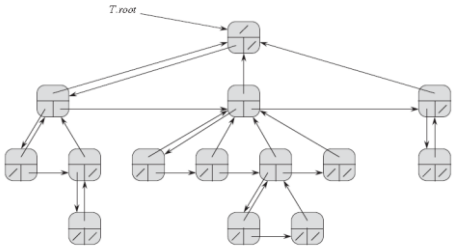
10.4 Representing rooted trees

Binary trees



Each node x has the attributes x:*p (top)*, x:*left* (lower left), and x:*right (lower right).*

Rooted tree with unbounded branching



Each node x has attributes x:*p*(top), x:*left-child (lower left), and x:right-sibling (lower right)*