Introduction To Algorithm
Third Edition
Answer

Xia Ding

December 18, 2015

10.1

10.1–1

It's easy to do it by yourself by painting the model.

10.1–2

Stack S1 grows from A[1] to A[n]. Stack S2 grows from A[n] to A[1]. Neither stack overflows unless S1.top = S2.top, indicating the total number of elements in both stacks together is n.

10.1–3

It's easy to do it by yourself by painting the model.

10.1–4

Algorithm 1 and 2.

Algorithm 1 ENQUEUE(Q, x)

if $Q.head == Q.tail + 1$ **then**
  error "overflow"
end if
$Q[Q.tail] = x$
if $Q.tail == Q.length$ **then**
  $Q.tail = 1$
else
  $Q.tail = Q.tail + 1$
end if

Algorithm 2 DEQUEUE(Q)

if $Q.head == Q.tail$ **then**
    error "underflow"
end if
$x = Q[Q.head]$
if $Q.head == Q.length$ **then**
    $Q.head = 1$
else
    $Q.head = Q.head + 1$
end if
return x

10.1–5

Algorithms 3∼6.

Algorithm 3 INSERT-HEAD(Q, x)

if $Q.head == 1$ **then**
    $Q.head = Q.length$
else
    $Q.head = Q.head - 1$
end if
if $Q.head == Q.tail$ **then**
    error "overflow"
end if
$Q[Q.head] = x$

10.1–6

Algorithm 7 and 8. Running time of ENQUEUE is $O(1)$, running time of DEQUEUE is $O(n)$.

10.1–7

Algorithm 9 and 10.Running time of PUSH is $O(1)$, POP is $O(n)$.

10.2

10.2–1

Algorithm 11 and 12. Running time of INSERT is $O(1)$, DELETE is $O(n)$.

Algorithm 4 INSERT-TAIL(Q, x)

```
if Q.tail + 1 == Q.head then
    error "overflow"
end if
Q[Q.tail] = x
if Q.tail == Q.length then
    Q.tail = 1
else
    Q.tail = Q.tail + 1
end if
```

Algorithm 5 DELETE-HEAD(Q)

```
if Q.head == Q.tail then
    error "underflow"
end if
x = Q[Q.head]
if Q.head == Q.length then
    Q.head = 1
else
    Q.head = Q.head + 1
end if
return x
```

Algorithm 6 DELETE-TAIL(Q)

```
if Q.tail == Q.head then
    error "underflow"
end if
if Q.tail == 1 then
    Q.tail = Q.length
else
    Q.tail = Q.tail − 1
end if
x = Q[Q.tail]
return x
```

Algorithm 7 ENQUEUE(S1, S2, x)

```
if STACK-EMPTY(S1) then
    PUSH(S2, x)
else
    PUSH(S1, x)
end if
```

Algorithm 8 DEQUEUE(S1, S2)

```
if STACK-EMPTY(S1) and not STACK-EMPTY(S2) then
    valid-S = S2
    empty-S = S1
else if STACK-EMPTY(S2) and not STACK-EMPTY(S1) then
    valid-S = S1
    empty-S = S2
else
    error "underflow"
end if
while not STACK-EMPTY(valid-S) do
    x = POP(valid-S)
    PUSH(empty-S, x)
end while
return POP(empty-S)
```

Algorithm 9 PUSH(Q1, Q2, x)

```
if QUEUE − EMPTY(Q1) then
    ENQUEUE(Q2, x)
else
    ENQUEUE(Q1, x)
end if
```

Algorithm 10 POP(Q1, Q2, x)

```
if QUEUE-EMPTY(Q1) and not QUEUE-EMPTY(Q2) then
    valid-Q = Q2
    empty-Q = Q1
else if QUEUE-EMPTY(Q2) and not QUEUE-EMPTY(Q1) then
    valid-Q = Q1
    empty-Q = Q2
else
    error "underflow"
end if
while not QUEUE-EMPTY(valid-Q) do
    x = DEQUEUE(valid-Q)
    ENQUEUE(empty-Q, x)
end while
return DEQUEUE(empty-Q)
```

Algorithm 11 LIST-INSERT(L, x)

```
s.next = L.nil.next
L.nil.next = x
```

Algorithm 12 LIST-DELETE(L, x)

$temp = x$
while $temp.next \neq x$ **do**
  $temp = temp.next$
end while
$temp.next = x.next$

10.2–2

Algorithm 13 14.

Algorithm 13 PUSH(L, x)

LIST-INSERT(L, x)

Algorithm 14 POP(L)

$x = L.nil.next$
$L.nil.next = L.nil.next.next$
return x

10.2–3

Algorithm 15 16.

0.1 10.2–4

Set $L.nil.key = x.key$ first.

0.2 10.2–5

All of their running time is $O(n)$.

0.3 10.2–6

Algorithm 17. Using doubly linked list to represent set.

0.4 10.2–7

Algorithm 18.

10.2–8

I need L.nil and L.nil.prev's memory location–**L.m**(k-bit integers). Thus the head of list's location is $L.nil.np$ **XOR** $L.m$.
Algorithm $19sim21$.

Algorithm 15 ENQUEUE(L, x)

$L.tail.next = x$
$L.tail = x$

Algorithm 16 DEQUEUE(L)

$x = L.head$
$L.head = L.head.next$
return x

Algorithm 17 UNION(S1, S2)

$L2.nil.next.prev = L1.nil.prev$
$L1.nil.prev.next = L2.nil.next$
$L1.nil.prev = L2.nil.prev$
$L2.nil.prev.next = L1.nil.next$

Algorithm 18 REVERSE(L)

$pt = L.nil.next$
$ptprev = L.nil$
$ptnext = pt.next$
while $ptnext \neq L.nil$ **do**
    $pt.next = ptprev$
    $ptprev = pt$
    $pt = ptnext$
    $ptnext = pt.next$
end while
$pt.next = ptprev$
$L.nil.next = pt$

Algorithm 19 SEARCH(L, k)

$L.nil.key = k$
$next = L.nil.np$ **XOR** L.m {pointer to next node}
$prev = L.nil$
while $x.key \neq k$ **do**
    $current = x$
    $x = next$
    $next = x.np$ **XOR** prev
    $prev = current$
end while
return x

Algorithm 20 INSERT(L, x)

$next = L.nil.np$ **XOR** L.m {pointer to next node}
$prev = L.nil$
$x.np = next$ **XOR** prev
nextNext = next.np **XOR** prev
next.np = nextNext **XOR** x
L.nil.np = x **XOR** L.m

Algorithm 21 DELETE(L, x)

$next = L.nil.np$ **XOR** L.m {pointer to next node}
$prev = L.nil$
while next $\neq$ x **do**
  current = next
  next = next.np **XOR** prev
  prev = current
end while
prev = current.np **XOR** next
{now: prev$\rightarrow$current$\rightarrow$next(x) $\rightarrow$Xnext$\rightarrow$Xnextnext}
Xnext = next.np **XOR** current
Xnextnext = Xnext.np **XOR** X
Xnext.np = Xnextnext **XOR** current
current.np = Xnext **XOR** prev

10.3

10.3–1

It's easy to do by yourself.

10.3–2

Algorithms 22 and 23.

Algorithm 22 ALLOCATE-OBJECT())

if $free == NIL$ **then**
  error "out of space"
else
  $x = free$
  $free = x.next$
  return x
end if

Algorithm 23 FREE-OBJECT(x)

1: $x.next = free$
2: $free = x$

10.3–3

Because when we search the free list, we needn't *prev*.

10.3–4

Let every blocks' *next* point to the right block and *prev* point to the left block(according to the picture in the book). *free* point to the leftmost block which isn't allocated. When call ALLOCATE-OBJECT, allocate the block pointed by *free*, and *free* move to the right block. When call FREE-OBJECT on position x, then free the object in x, then move all objects between x and position pointed by *free* left by one step, then *free* points to it's left block. Just like array-stack's INSERT and DELETE.

10.3–5

Algorithm 24.

1. We traverse the free list and set each element's *prev* pointer to a special value to identify later.
2. We start two pointers, one from the beginning of the memory and one from the end. We increase the left pointer until it reaches an empty block **x** and decrease the right until it reaches a non-empty block **y**.Then copy contents of **y** to **x** and set $\mathbf{y}.next = x$.This terminates when the two pointers catch up. At this time, memory in the left of pointer is allocated, and in the right of pointer is free. Set the current position of pointer as threshold.
3. Linearly scan the memory from the begin to threshold. Update all pointer *next* that point beyond the threshold, by using the *prev* in the block pointed by *next*.
4. Finally, we organize the memory beyond the threshold in a free list. [1]

10.4

10.4–1

It's easy to do it by yourself.

[1]reference to "http://clrs.skanev.com/10/03/05.html"

Algorithm 24 COMPACTIFY-LIST(L, F)

```
while F ≠ NIL do
   F.prev = 0
   F = F.next
end while
left = 1
right = MAX-SIZE-OF-MEMORY
while true do
   while MEMORY[left].prev ≠ 0 do
      left = left + 1
   end while
   while MEMORY[right].prev == 0 do
      right = right − 1
   end while
   if left ≥ right then
      break
   end if
   MEMORY[left].prev = MEMORY[right].prev
   MEMORY[left].next = MEMORY[right].next
   MEMORY[left].key = MEMORY[right].key
   MEMORY[right].next = left
   right = right − 1
   left = left − 1
end while
right = right + 1
for i = 1; i < right; i++ do
   if MEMORY[i].prev ≥ right then
      MEMORY[i].prev = MEMORY[MEMORY[i].prev].next
   end if
   if MEMORY[i].next ≥ right then
      MEMORY[i].next = MEMORY[MEMORY[i].next].next
   end if
end for
L = MEMORY[1]
F = MEMORY[right]
```

10.4–2

Algorithm 25 and 26. Using pre-order traverse.

Algorithm 25 PRINT(T)

```
root = T.root
REC-PRINT(root)
```

Algorithm 26 REC-PRINT(root)

```
if root ≠ NIL then
    print(root.key)
    REC-PRINT(root.left − child)
    REC-PRINT(root.right − child)
end if
```

10.4–3

Algorithm 27. Pre-order.

Algorithm 27 NONREC-PRINT(T)

```
root = T.root
S = ∅ {use ∅ to initialize stack}
PUSH(S, root)
while !STACK-EMPTY(S) do
    current = POP(S)
    print(current.key)
    if current.right − child ≠ NIL then
        PUSH(S, current.right − child)
    end if
    if current.left − child ≠ NIL then
        PUSH(S, current.left − child)
    end if
end while
```

10.4–4

Algorithm 28.

10.4–5

Algorithm 29. we need a pointer to keep track of previous node we visit(NIL at the begin). We set NIL as parent of the root. Then we have three case.

Algorithm 28 PRINT'(T)

```
current = T.root
while current != NIL do
    print(current.key)
    sibling = current.right − sibling
    while sibling ≠ NIL do
        print(sibling.key)
        sibling = sibling.right − sibling
    end while
    current = current.left − child
end while
```

1. come from current node's parent, then we go to left-child. If no left-child, then to right-child, if no right-child either, then go to parent.

2. come from current node's left-child, then go to right-child. If no right-child, then go to parent.

3. come from current node's right-child, then go to parent.

10.4–6

The two pointers will be left-child and next. The boolean should be called last-sibling. If it's **FALSE**, then next point to it's right-sibling, otherwise next point to it's parent.

Problem

10.1

It's easy to do it by yourself.

10.2

Algorithm 29 PRINT(T)

```
prev = NIL
current = T.root
while current ≠ NIL do
    if prev == current.parent then
        print(current.key)
        prev = current
        if current.left − child ≠ NIL then
            current = current.left − chile
        else if current.right − chile ≠ NIL then
            current = current.right − child
        else
            current = current.parent
        end if
    else if prev == current.left − child and current.right − child ≠ NIL then
        prev = current
        current = current.right − child
    else
        prev = current
        current = current.parent
    end if
end while
```