

## 论题 2-8 作业

姓名：陈劭源

学号：161240004

### 1 [TC] Problem 10.1-4

ENQUEUE( $Q, x$ )

```
1  if ( $Q.head == Q.tail + 1$ ) or ( $Q.head == 1$  and  $Q.tail == Q.length$ )
2      error "overflow"
    .....
```

DEQUEUE( $Q, x$ )

```
1  if ( $Q.head == Q.tail$ )
2      error "underflow"
    .....
```

### 2 [TC] Problem 10.1-5

PUSH-FRONT( $Q, x$ )

*//  $Q$  is an array, with two indices  $tail$  and  $head$*

```
1   $Q.head = Q.head - 1$ 
2  if  $Q.head == 0$ 
3       $Q.head = Q.length$ 
4   $Q[Q.head] = x$ 
```

POP-FRONT( $Q$ )

```
1   $x = Q[Q.head]$ 
2  if  $Q.head == Q.length$ 
3       $Q.head = 1$ 
4  else  $Q.head = Q.head + 1$ 
5  return  $x$ 
```

PUSH-BACK( $Q, x$ )

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

POP-BACK( $Q$ )

```
1   $Q.tail = Q.tail - 1$ 
2  if  $Q.tail == 0$ 
3       $Q.tail = Q.length$ 
4   $x = Q[Q.tail]$ 
5  return  $x$ 
```

### 3 [TC] Problem 10.1-6

ENQUEUE( $S1, S2, x$ )

//  $S1$  and  $S2$  are two stacks

```
1  while not STACK-EMPTY( $S2$ )
2      PUSH( $S1, POP(S2)$ )
3  PUSH( $S1, x$ )
```

DEQUEUE( $S1, S2$ )

```
1  while not STACK-EMPTY( $S1$ )
2      PUSH( $S2, POP(S1)$ )
3  return POP( $S2$ )
```

For each operation, the best-case running time is  $O(1)$ , the worst-case running time is  $O(length)$ , and the average-case running time is  $O(length)$ , where  $length$  is the length of the queue.

### 4 [TC] Problem 10.2-1

Operation INSERT can be implemented in  $O(1)$  time.

INSERT( $L, x$ )

```
1   $x.next = L.head$ 
2   $L.head = x$ 
```

Operation DELETE can be implemented in an average-case running time of  $O(1)$ . (assuming that copying the key takes a running time of  $O(1)$ , and no pointers to the elements of the linked list are outside the list itself)

DELETE( $L, x$ )

```
1  if  $x.next \neq \text{NIL}$  // judge whether  $x$  is the last element
2       $x.key = x.next.key$ 
3       $x.next = x.next.next$ 
4  else
5      if  $L.head = x$ 
6           $L.head = \text{NIL}$ 
7      else  $y = L.head$ 
8          while  $y.next \neq x$ 
9               $y = y.next$ 
10          $y.next = \text{NIL}$ 
```

Operation DELETE can be implemented in a worst-case running time of  $O(1)$ , as long as we use a sentinel (a dummy object), as the textbook states in page 238-239.

DELETE'(L,x)

```
1  if  $x.next.next == x$ 
    // the list contains only  $x$  and the dummy object
2       $x.next.next = x.next$ 
3  else
4       $x.key = x.next.key$ 
5       $x.next = x.next.next$ 
```

## 5 [TC] Problem 10.2-2

STACK-EMPTY(L)

```
1  return  $L.head == \text{NIL}$ 
```

PUSH(L,x)

```
1   $x.next = L.head$ 
2   $L.head = x$ 
```

POP(L)

```
1  if  $L.head == \text{NIL}$ 
2      error "underflow"
3   $x = L.head$ 
4   $L.head = L.head.next$ 
5  return  $x$ 
```

## 6 [TC] Problem 10.2-3

We should add an attribute *tail*, pointing to the last element of the list.

ENQUEUE(L,x)

```
1  if  $L.head == \text{NIL}$ 
2       $L.head = x$ 
3       $L.tail = x$ 
4  else
5       $L.tail.next = x$ 
6       $L.tail = x$ 
```

DEQUEUE(*L*)

```
1  if L.head == NIL
2      error "underflow"
3  elseif L.head == L.tail
4      x = L.head
5      L.head = NIL
6      L.tail = NIL
7  else
8      x = L.head
9      L.head = L.head.next
10 return x
```

## 7 [TC] Problem 10.3-4

When we allocate an object, we store the object in position *free*, and increment *free* by 1.

When we free an object, we first copy the last element (both key and pointers), denoted by *P1*, to the position which the object we want to free is in, denoted by *P2*. Then adjust the corresponding pointers (*P2.next.prev* = *P2* if *P2.next* ≠ NIL, and *P2.prev.next* = *P2* if *P2.prev* ≠ NIL). Finally, decrement *free* by 1.

## 8 [TC] Problem 10.3-5

SWAP-ELEMENTS(*i, j*)

// this procedure swaps two elements in doubly linked list(s) without changing the logical structure

```
1  if i == j // judge whether i == j
2      return
3  if i.next == j or j.next == i // judge whether i and j are adjacent
4      Exchange key[i] with key[j]
5      return
6  // i and j are neither the same nor adjacent
7  Exchange key[i] with key[j], prev[i] with prev[j], next[i] with next[j]
8  if next[i] ≠ NIL
9      prev[next[i]] = i
10 if next[j] ≠ NIL
11     prev[next[j]] = j
12 if prev[i] ≠ NIL
13     next[prev[i]] = i
14 if prev[j] ≠ NIL
15     next[prev[j]] = j
```

COMPACTIFY-LIST( $L, F$ )

```
// make the free list a doubly linked list
1   $x = F.head$ 
2  if  $x \neq \text{NIL}$ 
3       $x.prev = \text{NIL}$ 
4  while  $x.next \neq \text{NIL}$ 
5       $x.next.prev = x$ 
6       $x = x.next$ 
// move the elements of  $L$ 
7   $i = 1$ 
8   $x = L.head$ 
9  while  $x \neq \text{NIL}$ 
10      $y = x.next$  // note that 'x' will point to the wrong element after swapping
11     SWAP-ELEMENTS( $x, i$ )
12      $i = i + 1$ 
13      $x = y$ 
```

The correctness of the first loop is obvious. We use a loop invariant to prove the partial correctness of the second loop:

Prior to the  $i$ -th iteration, the first  $i - 1$  elements of  $L$  occupy array positions  $1, 2, \dots, i - 1$ .

**Initialization** Prior to the first iteration,  $i - 1 = 0$ , the invariant holds.

**Maintenance** Prior to the  $k$ -th iteration, the first  $k - 1$  elements of  $L$  occupy array positions  $1, 2, \dots, k - 1$ . After the iteration, the  $k$ -th element has been moved to position  $k$ , so the loop invariant still holds.

**Termination** Finally, we get  $i = n + 1$ . Therefore, all the elements in  $L$  occupy array positions  $1, 2, \dots, n$ .

Note that the data and the logical structure of the two linked lists are not changed during the whole procedure, so the rest  $m - n$  positions must store the free list. In each loop, every element in the free list or object list is visited exactly once, so the procedure can terminate. Hence, the procedure is totally correct.

## 9 [TC] Problem 10.4-2

PRINT-KEYS( $T$ )

```
1  if  $T \neq \text{NIL}$ 
2      print  $T.key$ 
3      PRINT-KEYS( $T.left$ )
4      PRINT-KEYS( $T.right$ )
```

## 10 [TC] Problem 10.4-3

PRINT-KEYS( $T$ )

```
1  let  $S$  be a new stack
2  PUSH( $S, T$ )
3  while not EMPTY( $S$ )
4       $x = \text{POP}(S)$ 
5      if  $x \neq \text{NIL}$ 
6          print  $T.\text{key}$ 
7          PUSH( $S, T.\text{left}$ )
8          PUSH( $S, T.\text{right}$ )
```

## 11 [TC] Problem 10.4-4

PRINT-KEYS( $T$ )

```
1  if  $T \neq \text{NIL}$ 
2      print  $T.\text{key}$ 
3      PRINT-KEYS( $T.\text{left-child}$ )
4      PRINT-KEYS( $T.\text{right-sibling}$ )
```

## 12 [TC] Problem 10-3

- a.* For COMPACT-LIST-SEARCH', it randomly chooses at most  $t$  indices. If there is a key with value  $k$ , the procedure returns the index. Otherwise, it picks the largest key which is smaller than  $k$ . Then, it searches from the key. Since the list is sorted, COMPACT-LIST-SEARCH' works correctly, and it returns the same answer as COMPACT-LIST-SEARCH.

We only have to consider the case that the **for** loop was executed less than  $t$  times. For each iteration of **while** loop, the pointer  $i$  advances, and the distance from  $i$  to the desired key is decremented by 1. However, for each **while** iteration in COMPACT-LIST-SEARCH, it not only advances the pointer but also randomly picks an element and judge whether it is closer to the desired key, therefore, the distance from  $i$  to the desired key is decremented by at least 1. Hence, the total number of iterations of both the **for** and **while** loops within COMPACT-LIST-SEARCH' is at least  $t$ .

- b.* The procedure ends in either line 7 or line 10-12, which takes an expected running time of  $O(t)$  and  $O(t + E[X_t])$  respectively. So the expected running time of COMPACT-LIST-SEARCH'( $L, n, k, t$ ) is  $O(t + E[X_t])$ .
- c.* When  $r$  is not less than the distance from the first element to  $k$ , the probability of  $X_t \geq r$  is  $(1 - r/n)^t$ . When  $r$  is less than the the distance from the first element to  $k$ ,  $X_t \geq r$  is impossible. Therefore, we have

$$P(X_t \geq r) \leq (1 - r/n)^t$$

By Equation (C.25), we get

$$E[X_t] = \sum_{r=1}^n P(X_t \geq r) \leq \sum_{r=1}^n (1 - r/n)^t$$

**d.** Since  $r^t \leq \int_r^{r+1} r^t dr$  when  $r, t \geq 0$ , we have

$$\sum_{r=0}^{n-1} r^t \leq \int_0^n r^t dr = n^{t+1}/(t+1)$$

**e.** By **c.** and **d.**, we have

$$\begin{aligned} E[X_t] &\leq \sum_{r=1}^n (1 - r/n)^t \\ &= \sum_{r=1}^n (n-r)^t / n^t \\ &= \sum_{r=0}^{n-1} (r)^t / n^t \\ &\leq n^{t+1}/(t+1)/n^t \\ &= n/(t+1) \end{aligned}$$

**f.** By **b.** and **e.**, the expected running time of COMPACT-LIST-SEARCH'( $L, n, k, t$ ) is

$$O(t + E[X_t]) = O(t + n/(t+1)) = O(t + n/t)$$

**g.** Let  $f(n)$  denote the expected running time of COMPACT-LIST-SEARCH',  $g(n)$  denote the expected running time of COMPACT-LIST-SEARCH. From **a.** and **f.** we know that for every  $t$ ,  $g(n) = O(f(n)) = O(t + n/t)$  holds. Take  $t = \lfloor \sqrt{n} \rfloor$ , we obtain  $g(n) = O(\sqrt{n})$ , i.e. COMPACT-LIST-SEARCH runs in  $O(\sqrt{n})$  expected time.

**h.** When the keys is not distinct, the probability of  $X_t \geq r$  in **c.** might not be  $(1 - r/n)^t$ . For example, if all the keys in the list are the same, and the desired key is larger than the keys in the list, then the pointer  $i$  only advances, but never skips, so the running time is  $\Theta(n)$ , and the bound we obtained in **g.** no longer applies.