Chapter 10

Michelle Bodnar, Andrew Lohr

April 12, 2016

Exercise 10.1-1

4		
4	1	
4	1	3
4	1	
4	1	8
4	1	

Exercise 10.1-2

We will call the stacks T and R. Initially, set $T.top = 0$ and $R.top = n + 1$. Essentially, stack T uses the first part of the array and stack R uses the last part of the array. In stack T, the top is the rightmost element of T. In stack R, the top is the leftmost element of R.

Algorithm 1 PUSH(S,x)

```
1: if S == T then
2:     if T.top + 1 == R.top then
3:         error "overflow"
4:     else
5:         T.top = T.top + 1
6:         T[T.top] = x
7:     end if
8: end if
9: if S == R then
10:    if R.top − 1 == T.top then
11:        error "overflow"
12:    else
13:        R.top = R.top − 1
14:        T[T.top] = x
15:    end if
16: end if
```

Algorithm 2 POP(S)

```
if S == T then
    if T.top == 0 then
        error "underflow"
    else
        T.top = T.top − 1.
        return T[T.top + 1]
    end if
end if
if S == R then
    if R.top == n + 1 then
        error "underflow"
    else
        R.top = R.top + 1.
        return R[R.top − 1]
    end if
end if
```

Exercise 10.1-3

4			
4	1		
4	1	3	
	1	3	
	1	3	8
		3	8

Exercise 10.1-4

Algorithm 3 ENQUEUE

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

Exercise 10.1-5

As in the example code given in the section, we will neglect to check for overflow and underflow errors.

Algorithm 4 DEQUEUE

if $Q.tail == Q.head$ **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 5 HEAD-ENQUEUE(Q,x)

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

Algorithm 6 TAIL-ENQUEUE(Q,x)

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

Algorithm 7 HEAD-DEQUEUE(Q,x)

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

Algorithm 8 TAIL-DEQUEUE(Q,x)

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

Exercise 10.1-6

The operation enqueue will be the same as pushing an element on to stack 1. This operation is $O(1)$. To dequeue, we pop an element from stack 2. If stack 2 is empty, for each element in stack 1 we pop it off, then push it on to stack 2. Finally, pop the top item from stack 2. This operation is $O(n)$ in the worst case.

Exercise 10.1-7

The following is a way of implementing a stack using two queues, where pop takes linear time, and push takes constant time. The first of these ways, consists of just enqueueing each element as you push it. Then, to do a pop, you dequque each element from one of the queues and place it in the other, but stopping just before the last element. Then, return the single element left in the original queue.

Exercise 10.2-1

To insert an element in constant time, just add it to the head by making it point to the old head and have it be the head. To delete an element, it needs linear time because there is no way to get a pointer to the previous element in the list without starting at the head and scanning along.

Exercise 10.2-2

The PUSH(L,x) operation is exactly the same as LIST-INSERT(L,x). The POP operation sets x equal to $L.head$, calls LIST-DELETE(L,L.head), then returns x.

Exercise 10.2-3

In addition to the head, also keep a pointer to the last element in the linked list. To enqueue, insert the element after the last element of the list, and set it to be the new last element. To dequeue, delete the first element of the list and return it.

Exercise 10.2-4

First let $L.nil.key = k$. Then run LIST-SEARCH' as usual, but remove the check that $x \neq L.nil$.

Exercise 10.2-5

To insert, just do list insert before the current head, in constant time. To search, start at the head, check if the element is the current node being inspected, check the next element, and so on until at the end of the list or you

found the element. This can take linear time in the worst case. To delete, again linear time is used because there is no way to get to the element immediately before the current element without starting at the head and going along the list.

Exercise 10.2-6

Let L_1 be a doubly linked list containing the elements of S_1 and L_2 be a doubly linked list containing the elements of S_2. We implement UNION as follows: Set $L_1.nil.prev.next = L_2.nil.next$ and $L_2.nil.next.prev = L_1.nil.prev$ so that the last element of L_1 is followed by the first element of L_2. Then set $L_1.nil.prev = L_2.nil.prev$ and $L_2.nil.prev.next = L_1.nil$, so that $L_1.nil$ is the sentinel for the doubly linked list containing all the elements of L_1 and L_2.

Exercise 10.2-7

Algorithm 9 REVERSE(L)

```
a = L.head.next
b = L.head
while a ≠ NIL do
    tmp = a.next
    a.next = b
    b = a
    a = tmp
end while
L.head = b
```

Exercise 10.2-8

We will store the pointer value for $L.head$ separately, for convenience. In general, A XOR (A XOR C) = C, so once we know one pointer's true value we can recover all the others (namely $L.head$) by applying this rule. Assuming there are at least two elements in the list, the first element will contain exactly the address of the second.

Algorithm 10 LISTnp-SEARCH(L,k)

```
p = NIL
x = L.head
while x ≠ NIL and x.key ≠ k do
    temp = x
    x = pXORx.np
    p = temp
end while
```

To reverse the list, we simply need to make the head be the "last" ele-

Algorithm 11 LISTnp-INSERT(L,x)

$x.np = L.head$
$L.nil.np = xXOR(L.nil.npXORL.head)$
$L.head = x$

Algorithm 12 LISTnp-Delete(L,x)

$L.nil.np = L.nil.npXORL.headXORL.head.np$
$L.head.np.np = L.head.np.npXORL.head$

ment before $L.nil$ instead of the first one after this. This is done by setting $L.head = L.nil.npXORL.head$.

Exercise 10.3-1

A multiple array version could be $L = 2$,

/	3	4	5	6	7	/
	12	4	8	19	5	11
	/	2	3	4	5	6

A single array version could be $L = 4$,

			12	7	/	4	10	4	8	13	7	19	16	10	5	19	13	11	/	16

Exercise 10.3-2

Algorithm 13 Allocate-Object()

if $free == NIL$ **then**
  error "out of space"
else
  $x = free$
  $free = A[x + 1]$
end if

Exercise 10.3-3

Allocate object just returns the index of some cells that it's guaranteed to not give out again until they've been freed. The prev attribute is not modified because only the next attribute is used by the memory manager, it's up to the code that calls allocate to use the prev and key attributes as it sees fit.

Exercise 10.3-4

For ALLOCATE-OBJECT, we will keep track of the next available spot in the array, and it will always be one greater than the number of elements being stored. For FREE-OBJECT(x), when a space is freed, we will decrement the

Algorithm 14 Free-Object(x)

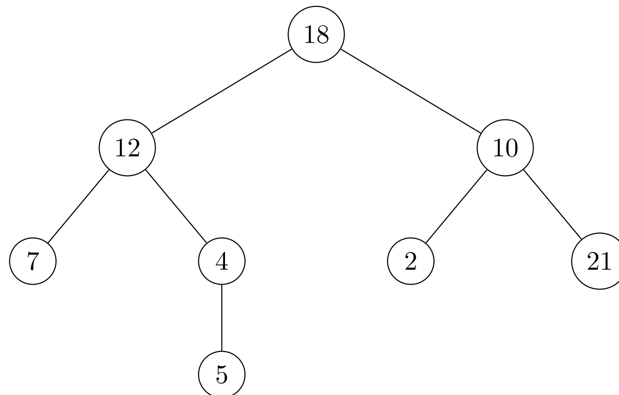
$A[x+1] = free$
$free = x$

position of each element in a position greater than that of x by 1 and update pointers accordingly. This takes linear time.

Exercise 10.3-5

See the algorithm $COMPACTIFY - LIST(L, F)$

Exercise 10.4-1



Note that indices 8 and 2 in the array do not appear, and, in fact do not represent a valid tree.

Exercise 10.4-2

See the algorithm PRINT-TREE.

Exercise 10.4-3

Exercise 10.4-4

See the algorithm PRINT-TREE.

Exercise 10.4-5

See the algorithm INORDER-PRINT'(T)

Exercise 10.4-6

Our two pointers will be $left$ and $right$. For a node x, $x.left$ will point to the leftmost child of x and $x.right$ will point to the sibling of x immediately to its right, if it has one, and the parent of x otherwise. Our boolean value b, stored at x, will be such that $b = depth(x) \mod 2$. To reach the parent of a node, simply keep following the "right" pointers until the parity of the boolean value changes. To find all the children of a node, start by finding $x.left$, then follow

Algorithm 15 COMPACTIFY-LIST(L,F)

```
if n=m then
    return
end if
e = max{max_{i∈[m]}{|key[i]|}, max_{i∈L}{|key[i]|}}
increase every element of key[1..m] by 2e
for every element of L, if its key is greater than e, reduce it by 2e
f = 1
while key[f] < e do
    f++
end while
a = L.head
if a > m then
    next[prev[f]] = next[f]
    prev[next[f]] = prev[f]
    next[f] = next[a]
    key[f] = key[a]
    prev[f] = prev[a]
    FREE−OBJECT(a)
    f++
    while key[f] < e do
        f++
    end while
end if
while a ≠ L.head do
    if a > m then
        next[prev[f]] = next[f]
        prev[next[f]] = prev[f]
        next[f] = next[a]
        key[f] = key[a]
        prev[f] = prev[a]
        FREE−OBJECT(a)
        f++
        while key[f] < e do
            f++
        end while
    end if
end while
```

Algorithm 16 PRINT-TREE(T.root)

```
if T.root == NIL then
    return
else
    Print T.root.key
    PRINT-TREE(T.root.left)
    PRINT-TREE(T.root.right)
end if
```

Algorithm 17 INORDER-PRINT(T)

```
let S be an empty stack
push(S, T)
while S is not empty do
    U = pop(S)
    if U ≠ NIL then
        print U.key
        push(S, U.left)
        push(S, U.right)
    end if
end while
```

Algorithm 18 PRINT-TREE(T.root)

```
if T.root == NIL then
    return
else
    Print T.root.key
    x = T.root.left − child
    while x ≠ NIL do
        PRINT-TREE(x)
        x = x.right − sibling
    end while
end if
```

Algorithm 19 INORDER-PRINT'(T)

```
a = T.left
prev = T
while a ≠ T do
    if prev = a.left then
        print a.key
        prev = a
        a = a.right
    else if prev = a.right then
        prev = a
        a = a.p
    else if prev = a.p then
        prev = a
        a = a.left
    end if
end while
print T.key
a = T.right
while a ≠ T do
    if prev = a.left then
        print a.key
        prev = a
        a = a.right
    else if prev = a.right then
        prev = a
        a = a.p
    else if prev = a.p then
        prev = a
        a = a.left
    end if
end while
```

the “right” pointers until the parity of the boolean value changes, ignoring this last node since it will be x.

Problem 10-1

For each, we assume sorted means sorted in ascending order

	$unsorted, single$	$sorted, single$	$unsorted, double$	$sorted, double$
$SEARCH(L, k)$	n	n	n	n
$INSERT(L, x)$	1	1	1	1
$DELETE(L, x)$	n	n	1	1
$SUCCESSOR(L, x)$	n	1	n	1
$PREDECESSOR(L, x)$	n	n	n	1
$MINIMUM(L, x)$	n	1	n	1
$MAXIMUM(L, x)$	n	n	n	1

Problem 10-2

In all three cases, MAKE-HEAP simply creates a new list L, sets $L.head = NIL$, and returns L in constant time. Assume lists are doubly linked. To realize a linked list as a heap, we imagine the usual array implementation of a binary heap, where the children of the ith element are $2i$ and $2i + 1$.

a. To insert, we perform a linear scan to see where to insert an element such that the list remains sorted. This takes linear time. The first element in the list is the minimum element, and we can find it in constant time. Extract-min returns the first element of the list, then deletes it. Union performs a merge operation between the two sorted lists, interleaving their entries such that the resulting list is sorted. This takes time linear in the sum of the lengths of the two lists.

b. To insert an element x into the heap, begin linearly scanning the list until the first instance of an element y which is strictly larger than x. If no such larger element exists, simply insert x at the end of the list. If y does exist, replace yt by x. This maintains the min-heap property because $x \leq y$ and y was smaller than each of its children, so x must be as well. Moreover, x is larger than its parent because y was the first element in the list to exceed x. Now insert y, starting the scan at the node following x. Since we check each node at most once, the time is linear in the size of the list. To get the minimum element, return the key of the head of the list in constant time.

   To extract the minimum element, we first call MINIMUM. Next, we'll replace the key of the head of the list by the key of the second smallest element y in the list. We'll take the key stored at the end of the list and use it to replace the key of y. Finally, we'll delete the last element of the list, and call MIN-HEAPIFY on the list. To implement this with linked lists, we need to step through the list to get from element i to element $2i$. We omit this detail from the code, but we'll consider it for runtime analysis. Since the value of i on which MIN-HEAPIFY is called is always increasing and we never need

to step through elements multiple times, the runtime is linear in the length of the list.

Algorithm 20 EXTRACT-MIN(L)

$min = MINIMUM(L)$
Linearly scan for the second smallest element, located in position i.
$L.head.key = L[i]$
$L[i].key = L[L.length].key$
DELETE(L, L[L.length])
MIN-HEAPIFY$(L[i], i)$
return min

Algorithm 21 MIN-HEAPIFY(L[i],i)

1: $l = L[2i].key$
2: $r = L[2i + 1].key$
3: $p = L[i].key$
4: $smallest = i$
5: **if** $L[2i] \neq NIL$ and $l < p$ **then**
6: $\quad smallest = 2i$
7: **end if**
8: **if** $L[2i + 1] \neq NIL$ and $r < L[smallest]$ **then**
9: $\quad smallest = 2i + 1$
10: **end if**
11: **if** $smallest \neq i$ **then**
12: $\quad$ exchange $L[i]$ with $L[smallest]$
13: $\quad$ MIN-HEAPIFY(L[smallest],smallest)
14: **end if**

Union is implemented below, where we assume A and B are the two list representations of heaps to be merged. The runtime is again linear in the lengths of the lists to be merged.

c. Since the algorithms in part b didn't depend on the elements being distinct, we can use the same ones.

Problem 10-3

a. If the original version of the algorithm takes only t iterations, then, we have that it was only at most t random skips though the list to get to the desired value, since each iteration of the original while loop is a possible random jump followed by a normal step through the linked list.

b. The for loop on lines 2-7 will get run exactly t times, each of which is constant runtime. After that, the while loop on lines 8-9 will be run exactly X_t times. So, the total runtime is $O(t + E[X_t])$.

Algorithm 22 UNION(A,B)

```
1: if A.head = NIL then
2:     return B
3: end if
4: i = 1
5: x = A.head
6: while B.head ≠ NIL do
7:     if B.head.key ≤ x.key then
8:         Insert a node at the end of list B with key x.key
9:         x.key = B.head.key
10:        Delete(B, B.head)
11:    end ifx = x.next
12: end while
13: return A
```

c. Using equation C.25, we have that $E[X_t] = \sum_{i=1}^{\infty} Pr(X_t \geq i)$. So, we need to show that $Pr(X_t \geq i) \leq (1 - i/n)^t$. This can be seen because having X_t being greater than i means that each random choice will result in an element that is either at least i steps before the desired element, or is after the desired element. There are $n - i$ such elements, out of the total n elements that we were pricking from. So, for a single one of the choices to be from such a range, we have a probability of $(n - i)/n = (1 - i/n)$. Since each of the selections was independent, the total probability that all of them were is $(1 - i/n)^t$, as desired. Lastly, we can note that since the linked list has length n, the probability that X_t is greater than n is equal to zero.

d. Since we have that $t > 0$, we know that the function $f(x) = x^t$ is increasing, so, that means that $\lfloor x \rfloor^t \leq f(x)$. So,

$$\sum_{r=0}^{n-1} r^t = \int_0^n \lfloor r \rfloor^t dr \leq \int_0^n f(r)dr = \frac{n^{t+1}}{t+1}$$

e.

$$E[X_t] \leq \sum_{r=1}^{n} (1 - r/n)^t = \sum_{r=1}^{n} \sum_{i=0}^{t} \binom{t}{i} (-r/n)^i = \sum_{i=0}^{t} \sum_{r=1}^{n} \binom{t}{i} (-r/n)^i$$

$$= \sum_{i=0}^{t} \binom{t}{i} (-1)^i \left(n^i - 1 + \sum_{r=0}^{n-1} (r)^t \right) /n \leq \sum_{i=0}^{t} \binom{t}{i} (-1)^i \left(n^i - 1 + \frac{n^{i+1}}{i+1} \right) /n$$

$$\leq \sum_{i=0}^{t} \binom{t}{i} (-1)^i \frac{n^i}{i+1} = \frac{1}{t+1} \sum_{i=0}^{t} \binom{t+1}{i+1} (-n)^i \leq \frac{(1-n)^{t+1}}{t+1}$$

f. We just put together parts b and e to get that it runs in time $O(t + n/(t+1))$. But, this is the same as $O(t + n/t)$.

---

g. Since we have that for any number of iterations t that the first algorithm takes to find its answer, the second algorithm will return it in time $O(t + n/t)$. In particular, if we just have that $t = \sqrt{n}$. The second algorithm takes time only $O(\sqrt{n})$. This means that tihe first list search algorithm is $O(\sqrt{n})$ as well.

h. If we don't have distinct key values, then, we may randomly select an element that is further along than we had been before, but not jump to it because it has the same key as what we were currently at. The analysis will break when we try to bound the probability that $X_t \geq i$.