10 Elementary Data Structures

10.1 Stacks and queues

10.1-1

> Using Figure 10.1 as a model, illustrate the result of each operation in the sequence $\text{PUSH}(S, 4)$, $\text{PUSH}(S, 1)$, $\text{PUSH}(S, 3)$, $\text{POP}(S)$, $\text{PUSH}(S, 8)$, and $\text{POP}(S)$ on an initially empty stack S stored in array $S[1..6]$.

$$
\begin{array}{l|lll}
\text{PUSH}(S, 4) & 4 & & \\
\text{PUSH}(S, 1) & 4 & 1 & \\
\text{PUSH}(S, 3) & 4 & 1 & 3 \\
\text{POP}(S) & 4 & 1 & \\
\text{PUSH}(S, 8) & 4 & 1 & 8 \\
\text{POP}(S) & 4 & 1 &
\end{array}
$$

10.1-2

> Explain how to implement two stacks in one array $A[1..n]$ in such a way that neither stack overflows unless the total number of elements in both stacks together is n. The PUSH and POP operations should run in $O(1)$ time.

The first stack starts at 1 and grows up towards n, while the second starts form n and grows down towards 1. Stack overflow happens when an element is pushed when the two stack pointers are adjacent.

10.1-3

> Using Figure 10.2 as a model, illustrate the result of each operation in the sequence $\text{ENQUEUE}(Q, 4)$, $\text{ENQUEUE}(Q, 1)$, $\text{ENQUEUE}(Q, 3)$, $\text{DEQUEUE}(Q)$, $\text{ENQUEUE}(Q, 8)$, and $\text{DEQUEUE}(Q)$ on an initially empty queue Q stored in array $Q[1..6]$.

$$
\begin{array}{l|llll}
\text{ENQUEUE}(Q, 4) & 4 & & & \\
\text{ENQUEUE}(Q, 1) & 4 & 1 & & \\
\text{ENQUEUE}(Q, 3) & 4 & 1 & 3 & \\
\text{DEQUEUE}(Q) & & 1 & 3 & \\
\text{ENQUEUE}(Q, 8) & & 1 & 3 & 8 \\
\text{DEQUEUE}(Q) & & & 3 & 8
\end{array}
$$

10.1-4

> Rewrite ENQUEUE and DEQUEUE to detect underflow and overflow of a queue.

To detect underflow and overflow of a queue, we can implement QUEUE-EMPTY and QUEUE-FULL first.

```
QUEUE-EMPTY(Q)
    if Q.head == Q.tail
        return true
    else return false
```

```
QUEUE-FULL(Q)
    if Q.head == Q.tail + 1 or (Q.head == 1 and Q.tail == Q.length)
        return true
    else return false
```

```
ENQUEUE(Q, x)
    if QUEUE-FULL(Q)
        error "overflow"
    else
        Q[Q.tail] = x
        if Q.tail == Q.length
            Q.tail = 1
        else Q.tail = Q.tail + 1
```

```
DEQUEUE(Q)
    if QUEUE-EMPTY(Q)
        error "underflow"
    else
        x = Q[Q.head]
        if Q.head == Q.length
            Q.head = 1
        else Q.head = Q.head + 1
        return x
```

10.1-5

> Whereas a stack allows insertion and deletion of elements at only one end, and a queue allows insertion at one end and deletion at the other end, a ***deque*** (double-ended queue) allows insertion and deletion at both ends. Write four $O(1)$-time procedures to insert elements into and delete elements from both ends of a deque implemented by an array.

The procedures QUEUE-EMPTY and QUEUE-FULL are implemented in Exercise 10.1-4.

```
HEAD-ENQUEUE(Q, x)
    if QUEUE-FULL(Q)
        error "overflow"
    else
        if Q.head == 1
            Q.head = Q.length
        else Q.head = Q.head - 1
        Q[Q.head] = x
```

```
TAIL-ENQUEUE(Q, x)
    if QUEUE-FULL(Q)
        error "overflow"
    else
        Q[Q.tail] = x
        if Q.tail == Q.length
            Q.tail = 1
        else Q.tail = Q.tail + 1
```

```
HEAD-DEQUEUE(Q)
    if QUEUE-EMPTY(Q)
        error "underflow"
    else
        x = Q[Q.head]
        if Q.head == Q.length
            Q.head = 1
        else Q.head = Q.head + 1
        return x
```

```
TAIL-DEQUEUE(Q)
    if QUEUE-EMPTY(Q)
        error "underflow"
    else
        if Q.tail == 1
            Q.tail = Q.length
        else Q.tail = Q.tail - 1
        x = Q[Q.tail]
        return x
```

10.1-6

> Show how to implement a queue using two stacks. Analyze the running time of the queue operations.

- ENQUEUE: $\Theta(1)$.
- DEQUEUE: worst $O(n)$, amortized $\Theta(1)$.

Let the two stacks be A and B.

ENQUEUE pushes elements on B. DEQUEUE pops elements from A. If A is empty, the contents of B are transfered to A by popping them out of B and pushing them to A. That way they appear in reverse order and are popped in the original.

A DEQUEUE operation can perform in $\Theta(n)$ time, but that will happen only when A is empty. If many ENQUEUEs and DEQUEUEs are performed, the total time will be linear to the number of elements, not to the largest length of the queue.

10.1-7

> Show how to implement a stack using two queues. Analyze the running time of the stack operations.

- PUSH: $\Theta(1)$.
- POP: $\Theta(n)$.

We have two queues and mark one of them as active. PUSH queues an element on the active queue. POP should dequeue all but one element of the active queue and queue them on the inactive. The roles of the queues are then reversed, and the final element left in the (now) inactive queue is returned.

The PUSH operation is $\Theta(1)$, but the POP operation is $\Theta(n)$ where n is the number of elements in the stack.

10.2 Linked lists

10.2-1

> Can you implement the dynamic-set operation INSERT on a singly linked list in $O(1)$ time? How about DELETE?

- INSERT: can be implemented in constant time by prepending it to the list.

  ```
  LIST-INSERT(L, x)
      x.next = L.head
      L.head = x
  ```

- DELETE: you can copy the value from the successor to element you want to delete, and then you can delete the successor in $O(1)$ time. This solution is not good in situations when you have a large object, in that case copying the whole object will be a bad idea.

10.2-2

> Implement a stack using a singly linked list L. The operations PUSH and POP should still take $O(1)$ time.

```
STACK-EMPTY(L)
    if L.head == NIL
        return true
    else return false
```

- PUSH: adds an element in the beginning of the list.

  ```
  PUSH(L, x)
      x.next = L.head
      L.head = x
  ```

- POP: removes the first element from the list.

  ```
  POP(L)
      if STACK-EMPTY(L)
          error "underflow"
      else
          x = L.head
          L.head = L.head.next
          return x
  ```

10.2-3

> Implement a queue by a singly linked list L. The operations ENQUEUE and DEQUEUE should still take $O(1)$ time.

```
QUEUE-EMPTY(L)
    if L.head == NIL
        return true
    else return false
```

- ENQUEUE: inserts an element at the end of the list. In this case we need to keep track of the last element of the list. We can do that with a sentinel.

```
ENQUEUE(L, x)
    if QUEUE-EMPTY(L)
        L.head = x
    else L.tail.next = x
    L.tail = x
    x.next = NIL
```

- DEQUEUE: removes an element from the beginning of the list.

```
DEQUEUE(L)
    if QUEUE-EMPTY(L)
        error "underflow"
    else
        x = L.head
        if L.head == L.tail
            L.tail = NIL
        L.head = L.head.next
        return x
```

10.2-4

> As written, each loop iteration in the $\text{LIST-SEARCH}'$ procedure requires two tests: one for $x \ne L.\text{nil}$ and one for $x.key \ne k$. Show how to eliminate the test for $x \ne L.\text{nil}$ in each iteration.

```
LIST-SEARCH'(L, k)
    x = L.nil.next
    L.nil.key = k
    while x.key != k
        x = x.next
    return x
```

10.2-5

> Implement the dictionary operations INSERT, DELETE, and SEARCH using singly linked, circular lists. What are the running times of your procedures?

- INSERT: $O(1)$.

```
LIST-INSERT''(L, x)
    x.next = L.nil.next
    L.nil.next = x
```

- DELETE: $O(n)$.

```
LIST-DELETE''(L, x)
    prev = L.nil
    while prev.next != x
        if prev.next == L.nil
            error "element not exist"
        prev = prev.next
    prev.next = x.next
```

- SEARCH: O(n).

```
LIST-SEARCH''(L, k)
    x = L.nil.next
    while x != L.nil and x.key != k
        x = x.next
    return x
```

10.2-6

> The dynamic-set operation UNION takes two disjoint sets S_1 and S_2 as input, and it returns a set $S = S_1 \cup S_2$ consisting of all the elements of S_1 and S_2. The sets S_1 and S_2 are usually destroyed by the operation. Show how to support UNION in $O(1)$ time using a suitable list data structure.

If both sets are a doubly linked lists, we just point link the last element of the first list to the first element in the second. If the implementation uses sentinels, we need to destroy one of them.

```
LIST-UNION(L[1], L[2])
    L[2].nil.next.prev = L[1].nil.prev
    L[1].nil.prev.next = L[2].nil.next
    L[2].nil.prev.next = L[1].nil
    L[1].nil.prev = L[2].nil.prev
```

10.2-7

> Give a $\Theta(n)$-time nonrecursive procedure that reverses a singly linked list of n elements. The procedure should use no more than constant storage beyond that needed for the list itself.

```
LIST-REVERSE(L)
    p[1] = NIL
    p[2] = L.head
    while p[2] != NIL
        p[3] = p[2].next
        p[2].next = p[1]
        p[1] = p[2]
        p[2] = p[3]
    L.head = p[1]
```

10.2-8 ⋆

> Explain how to implement doubly linked lists using only one pointer value $x.np$ per item instead of the usual two ($next$ and $prev$). Assume all pointer values can be interpreted as k-bit integers, and define $x.np$ to be $x.np = x.next \text{ XOR } x.prev$, the k-bit "exclusive-or" of $x.next$ and $x.prev$. (The value NIL is represented by 0.) Be sure to describe what information you need to access the head of the list. Show how to implement the SEARCH, INSERT, and DELETE operations on such a list. Also show how to reverse such a list in $O(1)$ time.

```
LIST-SEARCH(L, k)
    prev = NIL
    x = L.head
    while x != NIL and x.key != k
        next = prev XOR x.np
        prev = x
        x = next
    return x
```

```
LIST-INSERT(L, x)
    x.np = NIL XOR L.tail
    if L.tail != NIL
        L.tail.np = (L.tail.np XOR NIL) XOR x   // tail.prev XOR x
    if L.head == NIL
        L.head = x
    L.tail = x
```

```
LIST-DELETE(L, x)
    y = L.head
    prev = NIL
    while y != NIL
        next = prev XOR y.np
        if y != x
            prev = y
            y = next
        else
            if prev != NIL
                prev.np = (prev.np XOR y) XOR next  // prev.prev XOR next
            else L.head = next
            if next != NIL
                next.np = prev XOR (y XOR next.np)  // prev XOR next.next
            else L.tail = prev
```

```
LIST-REVERSE(L)
    tmp = L.head
    L.head = L.tail
    L.tail = tmp
```

10.3 Implementing pointers and objects

10.3-1

> Draw a picture of the sequence $\langle 13, 4, 8, 19, 5, 11 \rangle$ stored as a doubly linked list using the multiple-array representation. Do the same for the single-array representation.

- A multiple-array representation with $L = 2$,

index	1	2	3	4	5	6	7
next		3	4	5	6	7	/
key		13	4	8	19	5	11
prev		/	2	3	4	5	6

- A single-array version with $L = 1$,

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
key	13	4	/	4	7	1	8	10	4	19	13	7	5	16	10	11	/	13

10.3-2

> Write the procedures ALLOCATE-OBJECT and FREE-OBJECT for a homogeneous collection of objects implemented by the single-array representation.

```
ALLOCATE-OBJECT()
    if free == NIL
        error "out of space"
    else x = free
        free = A[x + 1]
        return x
```

```
FREE-OBJECT(x)
    A[x + 1] = free
    free = x
```

10.3-3

> Why don't we need to set or reset the $prev$ attributes of objects in the implementation of the ALLOCATE-OBJECT and FREE-OBJECT procedures?

We implement ALLOCATE-OBJECT and FREE-OBJECT in the hope of managing the storage of currently non-used object in the free list so that one can be allocated for reusing. As the free list acts like a stack, to maintain this stack-like collection, we merely remember its first pointer and set the $next$ attribute of objects. There is no need to worry the $prev$ attribute, for it hardly has any impact on the resulting free list.

10.3-4

> It is often desirable to keep all elements of a doubly linked list compact in storage, using, for example, the first m index locations in the multiple-array representation. (This is the case in a paged, virtual-memory computing environment.) Explain how to implement the procedures ALLOCATE-OBJECT and FREE-OBJECT so that the representation is compact. Assume that there are no pointers to elements of the linked list outside the list itself. (Hint: Use the array implementation of a stack.)

```
ALLOCATE-OBJECT()
    if STACK-EMPTY(F)
        error "out of space"
    else x = POP(F)
        return x
```

```
FREE-OBJECT(x)
    p = F.top - 1
    p.prev.next = x
    p.next.prev = x
    x.key = p.key
    x.prev = p.prev
    x.next = p.next
    PUSH(F, p)
```

10.3-5

> Let L be a doubly linked list of length n stored in arrays key, $prev$, and $next$ of length m. Suppose that these arrays are managed by ALLOCATE-OBJECT and FREE-OBJECT procedures that keep a doubly linked free list F. Suppose further that of the m items, exactly n are on list L and $m - n$ are on the free list. Write a procedure $\text{COMPACTIFY-LIST}(L, F)$ that, given the list L and the free list F, moves the items in L so that they occupy array positions $1, 2, \ldots, n$ and adjusts the free list F so that it remains correct, occupying array positions $n + 1, n + 2, \ldots, m$. The running time of your procedure should be $\Theta(n)$, and it should use only a constant amount of extra space. Argue that your procedure is correct.

We represent the combination of arrays key, $prev$, and $next$ by a multible-array A. Each object of A's is either in list L or in the free list F, but not in both. The procedure COMPACTIFY-LIST transposes the first object in L with the first object in A, the second objects until the list L is exhausted.

```
COMPACTIFY-LIST(L, F)
    TRANSPOSE(A[L.head], A[1])
    if F.head == 1
        F.head = L.head
    L.head = 1
    l = A[L.head].next
    i = 2
    while l != NIL
        TRANSPOSE(A[l], A[i])
        if F == i
            F = l
        l = A[l].next
        i = i + 1
```

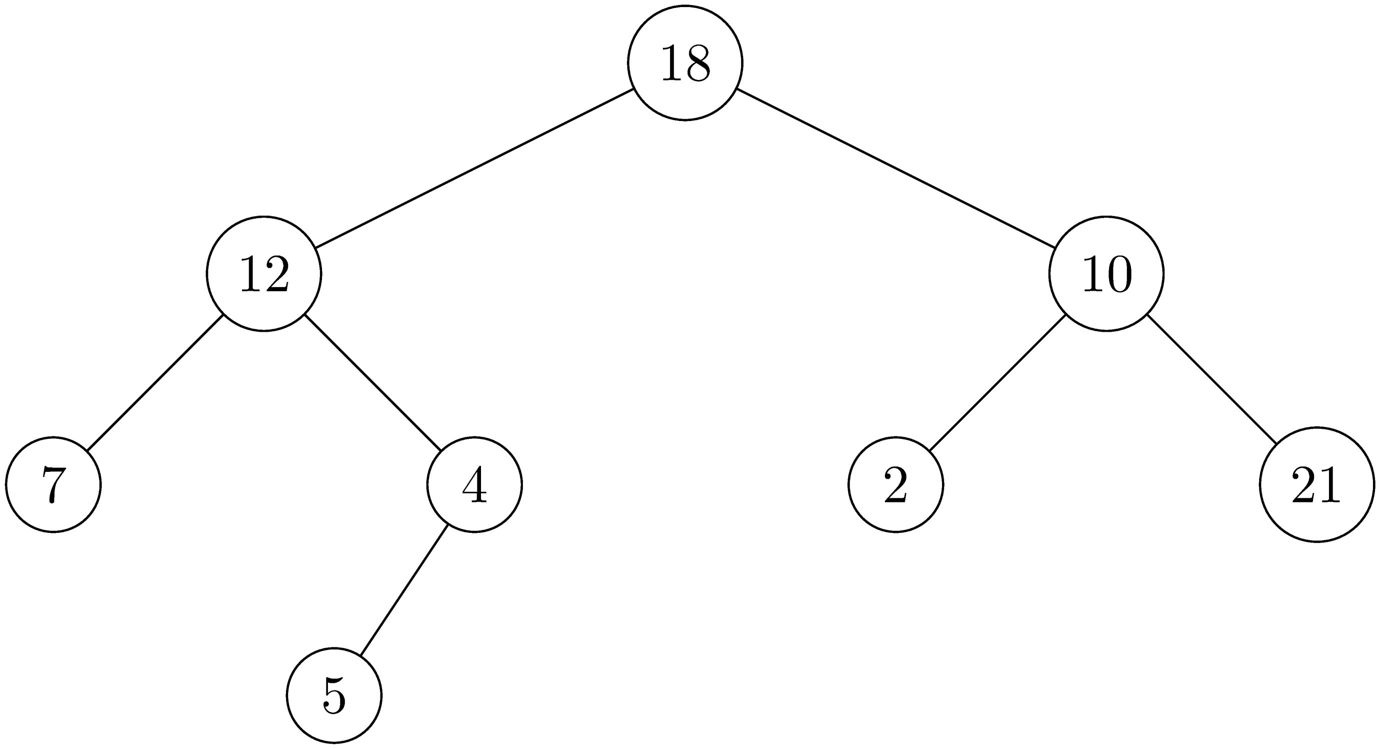
```
TRANSPOSE(a, b)
    SWAP(a.prev.next, b.prev.next)
    SWAP(a.prev, b.prev)
    SWAP(a.next.prev, b.next.prev)
    SWAP(a.next, b.next)
```

10.4 Representing rooted trees

10.4-1

> Draw the binary tree rooted at index 6 that is represented by the following attributes:

index	key	left	right
1	12	7	3
2	15	8	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL



10.4-2

> Write an $O(n)$-time recursive procedure that, given an n-node binary tree, prints out the key of each node in the tree.

```
PRINT-BINARY-TREE(T)
    x = T.root
    if x != NIL
        PRINT-BINARY-TREE(x.left)
        print x.key
        PRINT-BINARY-TREE(x.right)
```

10.4-3

> Write an $O(n)$-time nonrecursive procedure that, given an n-node binary tree, prints out the key of each node in the tree. Use a stack as an auxiliary data structure.

```
PRINT-BINARY-TREE(T, S)
    PUSH(S, T.root)
    while !STACK-EMPTY(S)
        x = S[S.top]
        while x != NIL      // store all nodes on the path towards the leftmost leaf
            PUSH(S, x.left)
            x = S[S.top]
        POP(S)              // S has NIL on its top, so pop it
        if !STACK-EMPTY(S)  // print this nodes, leap to its in-order successor
            x = POP(S)
            print x.key
            PUSH(S, x.right)
```

10.4-4

> Write an $O(n)$-time procedure that prints all the keys of an arbitrary rooted tree with n nodes, where the tree is stored using the left-child, right-sibling representation.

```
PRINT-LCRS-TREE(T)
    x = T.root
    if x != NIL
        print x.key
        lc = x.left-child
        if lc != NIL
            PRINT-LCRS-TREE(lc)
            rs = lc.right-sibling
            while rs != NIL
                PRINT-LCRS-TREE(rs)
                rs = rs.right-sibling
```

10.4-5 ⋆

> Write an $O(n)$-time nonrecursive procedure that, given an n-node binary tree, prints out the key of each node. Use no more than constant extra space outside of the tree itself and do not modify the tree, even temporarily, during the procedure.

```
PRINT-KEY(T)
    prev = NIL
    x = T.root
    while x != NIL
        if prev = x.parent
            print x.key
            prev = x
            if x.left
                x = x.left
            else
                if x.right
                    x = x.right
                else
                    x = x.parent
        else if prev == x.left and x.right != NIL
                prev = x
                x = x.right
        else
```

```
                prev = x
                x = x.parent
```

10.4-6 ⋆

> The left-child, right-sibling representation of an arbitrary rooted tree uses three pointers in each node: *left-child*, *right-sibling*, and *parent*. From any node, its parent can be reached and identified in constant time and all its children can be reached and identified in time linear in the number of children. Show how to use only two pointers and one boolean value in each node so that the parent of a node or all of its children can be reached and identified in time linear in the number of children.

Use boolean to identify the last sibling, and the last sibling's right-sibling points to the parent.

Problem 10-1 Comparisons among lists

> For each of the four types of lists in the following table, what is the asymptotic worst-case running time for each dynamic-set operation listed?
>
> | | unsorted, singly linked | sorted, singly linked | unsorted, doubly linked | sorted, doubly linked |
> |---|---|---|---|---|
> | SEARCH(L, k) | | | | |
> | INSERT(L, x) | | | | |
> | DELETE(L, x) | | | | |
> | SUCCESSOR(L, x) | | | | |
> | PREDECESSOR(L, x) | | | | |
> | MINIMUM(L) | | | | |
> | MAXIMUM(L) | | | | |

	unsorted, singly linked	sorted, singly linked	unsorted, doubly linked	sorted, doubly linked
SEARCH(L, k)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
INSERT(L, x)	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
DELETE(L, x)	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
SUCCESSOR(L, x)	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
PREDECESSOR(L, x)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
MINIMUM(L)	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
MAXIMUM(L)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$

Problem 10-2 Mergeable heaps using linked lists

> A ***mergeable heap*** supports the following operations: MAKE-HEAP (which creates an empty mergeable heap), INSERT, MINIMUM, EXTRACT-MIN, and UNION. Show how to implement mergeable heaps using linked lists in each of the following cases. Try to make each operation as efficient as possible. Analyze the running time of each operation in terms of the size of the dynamic set(s) being operated on.
>
> **a.** Lists are sorted.
>
> **b.** Lists are unsorted.
>
> **c.** Lists are unsorted, and dynamic sets to be merged are disjoint.

In all three cases, MAKE-HEAP simply creates a new list L, sets $L.\text{head} = \text{NIL}$, 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 $x \le y$ 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.

```
EXTRACT-MIN(L)
    min = MINIMIM(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
```

```
MIN-HEAPIFY(L[i], i)
    l = L[2i].key
    r = L[2i + 1].key
    p = L[i].key
    smallest = i
    if L[2i] != NIL and l < p
        smallest = 2i
    if L[2i + 1] != NIL and r < L[smallest]
        smallest = 2i + 1
    if smallest != i
        exchange L[i] with L[smallest]
        MIN-HEAPIFY(L[smallest], smallest])
```

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.

```
UNION(A, B)
    if A.head == NIL
        return B
    x = A.head
    while B.head != NIL
        if B.head.key ≤ x.key
            INSERT(B, x.key)
            x.key = B.head.key
            DELETE(B, B.head)
        x = x.next
    return A
```

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

Problem 10-3 Searching a sorted compact list

> Exercise 10.3-4 asked how we might maintain an n-element list compactly in the first n positions of an array. We shall assume that all keys are distinct and that the compact list is also sorted, that is, $\text{key}[i] < \text{key}[\text{next}[i]]$ for all $i = 1, 2, \ldots, n$ such that $\text{next}[i] \neq \text{NIL}$. We will also assume that we have a variable L that contains the index of the first element on the list. Under these assumptions, you will show that we can use the following randomized algorithm to search the list in $O(\sqrt{n})$ expected time.
>
> ```
> COMPACT-LIST-SEARCH(L, n, k)
>     i = L
>     while i != NIL and key[i] < k
>         j = RANDOM(1, n)
>         if key[i] < key[j] and key[j] ≤ k
>             i = j
>             if key[i] == k
>                 return i
>         i = next[i]
>     if i == NIL or key[i] > k
>         return NIL
>     else return i
> ```
>
> If we ignore lines 3–7 of the procedure, we have an ordinary algorithm for searching a sorted linked list, in which index i points to each position of the list in turn. The search terminates once the index i "falls off" the end of the list or once $\text{key}[i] \geq k$. In the latter case, if $\text{key}[i] = k$, clearly we have found a key with the value k. If, however, $\text{key}[i] > k$, then we will never find a key with the value k, and so terminating the search was the right thing to do.
>
> Lines 3–7 attempt to skip ahead to a randomly chosen position j. Such a skip benefits us if $\text{key}[j]$ is larger than $\text{key}[i]$ and no larger than k; in such a case, j marks a position in the list that i would have to reach during an ordinary list search. Because the list is compact, we know that any choice of j between 1 and n indexes some object in the list rather than a slot on the free list.
>
> Instead of analyzing the performance of $\text{COMPACT-LIST-SEARCH}$ directly, we shall analyze a related algorithm, $\text{COMPACT-LIST-SEARCH}'$, which executes two separate loops. This algorithm takes an additional parameter t which determines an upper bound on the number of iterations of the first loop.
>
> ```
> COMPACT-LIST-SEARCH'(L, n, k, t)
>     i = L
>     for q = 1 to t
>         j = RANDOM(1, n)
>         if key[i] < key[j] and key[j] ≤ k
>             i = j
>             if key[i] == k
>                 return i
>     while i != NIL and key[i] < k
>         i = next[i]
> ```

```
    if i == NIL or key[i] > k
        return NIL
    else return i
```

> To compare the execution of the algorithms $\text{COMPACT-LIST-SEARCH}(L, n, k)$ and $\text{COMPACT-LIST-SEARCH}'(L, n, k, t)$, assume that the sequence of integers returned by the calls of $\text{RANDOM}(1, n)$ is the same for both algorithms.
>
> **a.** Suppose that $\text{COMPACT-LIST-SEARCH}(L, n, k)$ takes t iterations of the **while** loop of lines 2–8. Argue that $\text{COMPACT-LIST-SEARCH}'(L, n, k, t)$ returns the same answer and that the total number of iterations of both the **for** and **while** loops within $\text{COMPACT-LIST-SEARCH}'$ is at least t.
>
> In the call $\text{COMPACT-LIST-SEARCH}'(L, n, k, t)$, let X_t be the random variable that describes the distance in the linked list (that is, through the chain of $next$ pointers) from position i to the desired key k after t iterations of the **for** loop of lines 2–7 have occurred.
>
> **b.** Argue that the expected running time of $\text{COMPACT-LIST-SEARCH}'(L, n, k, t)$ is $O(t + E[X_t])$.
>
> **c.** Show that $E[X_t] \le \sum_{r=1}^{n} (1 - r/n)^t$. (Hint: Use equation $(C.25)$.)
>
> **d.** Show that $\sum_{r=0}^{n-1} r^t \le n^{t+1}/(t+1)$.
>
> **e.** Prove that $E[X_t] \le n/(t+1)$.
>
> **f.** Show that $\text{COMPACT-LIST-SEARCH}'(L, n, k, t)$ runs in $O(t + n/t)$ expected time.
>
> **g.** Conclude that $\text{COMPACT-LIST-SEARCH}$ runs in $O(\sqrt{n})$ expected time.
>
> **h.** Why do we assume that all keys are distinct in $\text{COMPACT-LIST-SEARCH}$? Argue that random skips do not necessarily help asymptotically when the list contains repeated key values.

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 $O(t + E[X_t])$.

c. Using equation $C.25$, we have that $E[X_t] = \sum_{i=1}^{\infty} \Pr\{X_t \ge i\}$. So, we need to show that $\Pr\{X_t \ge i\} \le (1 - i/n)^t$. 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 $\lfloor x \rfloor^t \le f(x)$. So,

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

e.

$$\begin{aligned} E[X_t] &\le \sum_{r=1}^{n}(1 - r/n)^t \qquad \text{from part (c)} \\ &= \sum_{r=1}^{n} \frac{(n-r)^t}{n^t} \\ &= \frac{1}{n^t}\sum_{r=1}^{n}(n-r)^t, \end{aligned}$$

and

$$\begin{aligned} \sum_{r=1}^{n}(n-r)^t &= (n-1)^t + (n-2)^t + \cdots + 1^t + 0^t \\ &= \sum_{r=0}^{n-1} r^t. \end{aligned}$$

So,

$$\begin{aligned} E[X_t] &= \frac{1}{n^t}\sum_{r=0}^{n-1} r^t \\ &\le \frac{1}{n^t}\cdot\frac{n^{t+1}}{t+1} \qquad \text{from part (d)} \\ &= \frac{n}{t+1}. \end{aligned}$$

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 $X_t \ge i$.