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 $S[1..6]$.

$\text{PUSH}(S, 4)$	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			

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 $Q[1..6]$.

$\text{ENQUEUE}(Q, 4)$	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		

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 $x \neq L.\text{nil}$ and one for $x.\text{key} \neq k$. Show how to eliminate the test for $x \neq L.\text{nil}$ 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 $x.\text{np}$ per item instead of the usual two (next and prev). Assume all pointer values can be interpreted as k-bit integers, and define $x.\text{np}$ to be $x.\text{np} = x.\text{next} \text{ XOR } x.\text{prev}$, the k-bit "exclusive-or" of $x.\text{next}$ and $x.\text{prev}$. (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 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
```

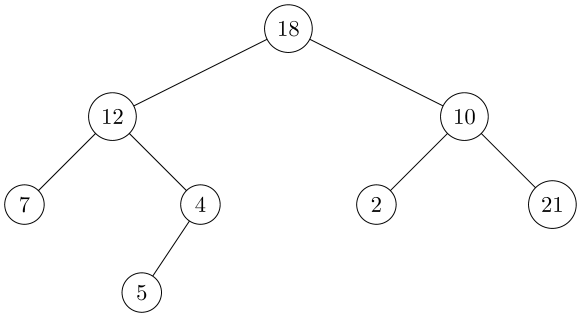
```
TRANSPOSE(a, b)
    SWAP(a.prev.next, b.prev.next)
    SWAP(a.prev, b.prev)
    SWAP(a.next.prev, b.next.prev)
    SWAP(a.next, b.next)
```

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 $x \leq y$ 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.

```
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, $key[i] < key[next[i]]$ for all $i = 1, 2, \ldots, n$ such that $next[i] \neq NIL$. 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 $key[i] \geq k$. In the latter case, if $key[i] = k$, clearly we have found a key with the value k. If, however, $key[i] > k$, 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 $key[j]$ is larger than $key[i]$ 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 COMPACT-LIST-SEARCH directly, we shall analyze a related algorithm, 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 COMPACT-LIST-SEARCH(L, n, k) and COMPACT-LIST-SEARCH′(L, n, k, t), assume that the sequence of integers returned by the calls of RANDOM$(1, n)$ is the same for both algorithms.

a. Suppose that COMPACT-LIST-SEARCH(L, n, k) takes t iterations of the **while** loop of lines 2–8. Argue that 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 COMPACT-LIST-SEARCH′ is at least t.

In the call 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 COMPACT-LIST-SEARCH′(L, n, k, t) is $O(t + E[X_t])$.

c. Show that $E[X_t] \leq \sum_{r=1}^{n} (1 - r/n)^t$. (Hint: Use equation (C.25).)

d. Show that $\sum_{r=0}^{n-1} r^t \leq n^{t+1}/(t + 1)$.

e. Prove that $E[X_t] \leq n/(t + 1)$.

f. Show that COMPACT-LIST-SEARCH′(L, n, k, t) runs in $O(t + n/t)$ expected time.

g. Conclude that COMPACT-LIST-SEARCH runs in $O(\sqrt{n})$ expected time.

h. Why do we assume that all keys are distinct in 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 $E[X_t] = \sum_{i=1}^{\infty} \Pr\{X_t \geq i\}$. 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.

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

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,

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

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 the 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 $X_t \geq i$.

11 Hash Tables

11.1 Direct-address tables

11.1-1

Suppose that a dynamic set S is represented by a direct-address table T of length m. Describe a procedure that finds the maximum element of S. What is the worst-case performance of your procedure?

As the dynamic set S is represented by the direct-address table T, for each key k in S, there is a slot k in T points to it. If no element with key k in S, then $T[k] = NIL$. Using this property, we can find the maximum element of S by traversing down from the highest slot to seek the first non-NIL one.

```
MAXIMUM(S)
    return TABLE-MAXIMUM(T, m - 1)
```

```
TABLE-MAXIMUM(T, l)
    if l < 0
        return NIL
    else if DIRECT-ADDRESS-SEARCH(T, l) != NIL
        return l
    else return TABLE-MAXIMUM(T, l - 1)
```

The TABLE-MAXIMUM procedure gest down and checks 1 sloc at a time, linearly approaches the solution. In the worst case where S is empty, TABLE-MAXIMUM examines m slots. Therefore, the worst-case performance of MAXIMUM is $O(m)$, where m is the length of the direct-address table T.

11.1-2

A ***bit vector*** is simply an array of bits (0s and 1s). A bit vector of length m takes much less space than an array of m pointers. Describe how to use a bit vector to represent a dynamic set of distinct elements with no satellite data. Dictionary operations should run in $O(1)$ time.

Using the bit vector data structure, we can represent keys less than m by a string of m bits, denoted by $V[0..m-1]$, in which each position that occupied by the bit 1, corresponds to a key in the set S. If the set contains no element with key k, then $V[k] = 0$. For instance, we can store the set $\{2, 4, 6, 10, 16\}$ in a bit vector of length 20:

$$00101010001000001000$$

```
BITMAP-SEARCH(V, k)
    if V[k] != 0
        return k
    else return NIL
```

```
BITMAP-INSERT(V, x)
    V[x] = 1
```

```
BITMAP-DELETE(V, x)
    V[x] = 0
```

Each of these operations takes only $O(1)$ time.

11.1-3

Suggest how to implement a direct-address table in which the keys of stored elements do not need to be distinct and the elements can have satellite data. All three dictionary operations (INSERT, DELETE, and SEARCH) should run in $O(1)$ time. (Don't forget that DELETE takes as an argument a pointer to an object to be deleted, not a key.)

Assuming that fetching an element should return the satellite data of all the stored elements, we can have each key map to a doubly linked list.

- INSERT: appends the element to the list in constant time
- DELETE: removes the element from the linked list in constant time (the element contains pointers to the previous and next element)
- SEARCH: returns the first element, which is a node in a linked list, in constant time

11.1-4 ⋆

We wish to implement a dictionary by using direct addressing on a *huge* array. At the start, the array entries may contain garbage, and initializing the entire array is impractical because of its size. Describe a scheme for implementing a direct-address dictionary on a huge array. Each stored object should use $O(1)$ space; the operations SEARCH, INSERT, and DELETE should take $O(1)$ time each; and initializing the data structure should take $O(1)$ time. (Hint: Use an additional array, treated somewhat like a stack whose size is the number of keys actually stored in the dictionary, to help determine whether a given entry in the huge array is valid or not.)

The additional data structure will be a stack S.

Initially, set S to be empty, and do nothing to initialize the huge array. Each object stored in the huge array will have two parts: the key value, and a pointer to an element of S, which contains a pointer back to the object in the huge array.

- To insert x, push an element y to the stack which contains a pointer to position x in the huge array. Update position $A[x]$ in the huge array A to contain a pointer to y in S.
- To search for x, go to position x of A and go to the location stored there. If that location is an element of S which contains a pointer to $A[x]$, then we know x is in A. Otherwise, $x \notin A$.
- To delete x, invalidate the element of S which is pointed to by $A[x]$. Because there may be "holes" in S now, we need to pop an item from S, move it to the position of the "hole", and update the pointer in A accordingly. Each of these takes $O(1)$ time and there are at most as many elements in S as there are valid elements in A.

11.2 Hash tables

11.2-1

Suppose we use a hash function h to hash n distinct keys into an array T of length m. Assuming simple uniform hashing, what is the expected number of collisions? More precisely, what is the expected cardinality of $\{\{k, l\} : k \neq l \text{ and } h(k) = h(l)\}$?

Under the assumption of simple uniform hashing, we will use linearity of expectation to compute this.

Suppose that all the keys are totally ordered $\{k_1, \ldots, k_n\}$. Let X_i be the number of ℓ's such that $\ell > k_i$ and $h(\ell) = h(k_i)$. So X_i is the (expected) number of times that key k_i is collided by those keys hashed afterward. Note, that this is the same thing as $\sum_{j>i} \Pr(h(k_i) = h(k_i)) = \sum_{j>i} 1/m = (n - i)/m$. Then, by linearity of expectation, the number of collisions is the sum of the number of collisions for each possible smallest element in the collision. The expected number of collisions is

$$\sum_{i=1}^{n} \frac{n - i}{m} = \frac{n^2 - n(n + 1)/2}{m} = \frac{n^2 - n}{2m}.$$

11.2-2

Demonstrate what happens when we insert the keys $5, 28, 19, 15, 20, 33, 12, 17, 10$ into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be $h(k) = k \mod 9$.

Let us number our slots $0, 1, \ldots, 8$.

Then our resulting hash table will look like following:

h(k)	keys
0 mod 9	
1 mod 9	10 → 19 → 28
2 mod 9	20
3 mod 9	12
4 mod 9	
5 mod 9	5
6 mod 9	33 → 15
7 mod 9	
8 mod 9	17

11.2-3

Professor Marley hypothesizes that he can obtain substantial performance gains by modifying the chaining scheme to keep each list in sorted order. How does the professor's modification affect the running time for successful searches, unsuccessful searches, insertions, and deletions?

- Successful searches: no difference, $\Theta(1+\alpha)$.
- Unsuccessful searches: faster but still $\Theta(1+\alpha)$.
- Insertions: same as successful searches, $\Theta(1+\alpha)$.
- Deletions: same as before if we use doubly linked lists, $\Theta(1)$.

11.2-4

Suggest how to allocate and deallocate storage for elements within the hash table itself by linking all unused slots into a free list. Assume that one slot can store a flag and either one element plus a pointer or two pointers. All dictionary and free-list operations should run in $O(1)$ expected time. Does the free list need to be doubly linked, or does a singly linked free list suffice?

The flag in each slot of the hash table will be 1 if the element contains a value, and 0 if it is free. The free list must be doubly linked.

- Search is unmodified, so it has expected time $O(1)$.
- To insert an element x, first check if $T[h(x.\text{key})]$ is free. If it is, delete $T[h(x.\text{key})]$ and change the flag of $T[h(x.\text{key})]$ to 1. If it wasn't free to begin with, simply insert $x.\text{key}$ at the start of the list stored there.
- To delete, first check if $x.\text{prev}$ and $x.\text{next}$ are NIL. If they are, then the list will be empty upon deletion of x, so insert $T[h(x.\text{key})]$ into the free list, update the flag of $T[h(x.\text{key})]$ to 0, and delete x from the list it's stored in. Since deletion of an element from a singly linked list isn't $O(1)$, we must use a doubly linked list.
- All other operations are $O(1)$.

11.2-5

Suppose that we are storing a set of n keys into a hash table of size m. Show that if the keys are drawn from a universe U with $|U| > nm$, then U has a subset of size n consisting of keys that all hash to the same slot, so that the worst-case searching time for hashing with chaining is $\Theta(n)$.

Suppose the $m-1$ slots contains at most $n-1$ elements, then the remaining slot should have

$$|U| - (m-1)(n-1) > nm - (m-1)(n-1) = n + m - 1 \geq n$$

elements, thus U has a subset of size n.

11.2-6

Suppose we have stored n keys in a hash table of size m, with collisions resolved by chaining, and that we know the length of each chain, including the length L of the longest chain. Describe a procedure that selects a key uniformly at random from among the keys in the hash table and returns it in expected time $O(L \cdot (1 + 1/\alpha))$.

☑ = p(1 + \alpha) + (1 − p)(1 + E[X])sincewe′dexpecttotake1 + \alpha stepstoreachanelementonthelist, andsinceweknowhowmanyelementsareoneachlist, it′theelementdoesn′texistwe′llknowrightaway. Thenwehave E[X] = \alpha + 1 / p. Theprobabilityof′pickingaparticularelementisn / mL = \alpha / L$. Therefore, we have

$$ \begin{aligned} ☑ & = \alpha + L / \alpha \\ & = L(\alpha/L + 1/\alpha) \\ & = O(L(1 + 1/\alpha)) \end{aligned} $$

since $\alpha \leq L$.

11.3 Hash functions

11.3-1

Suppose we wish to search a linked list of length n, where each element contains a key k along with a hash value $h(k)$. Each key is a long character string. How might we take advantage of the hash values when searching the list for an element with a given key?

If every element also contained a hash of the long character string, when we are searching for the desired element, we'll first check if the hashvalue of the node in the linked list, and move on if it disagrees. This can increase the runtime by a factor proportional to the length of the long character strings.

11.3-2

Suppose that we hash a string of r characters into m slots by treating it as a radix-128 number and then using the division method. We can easily represent the number m as a 32-bit computer word, but the string of r characters, treated as a radix-128 number, takes many words. How can we apply the division method to compute the hash value of the character string without using more than a constant number of words of storage outside the string itself?

```
sum = 0
for i = 1 to r
    sum = (sum * 128 + s[i]) % m
```

Use `sum` as the key.

11.3-3

Consider a version of the division method in which $h(k) = k \mod m$, where $m = 2^p - 1$ and k is a character string interpreted in radix 2^p. Show that if we can derive string x from string y by permuting its characters, then x and y hash to the same value. Give an example of an application in which this property would be undesirable in a hash function.

We will show that each string hashes to the sum of it's digits $\mod 2^p - 1$. We will do this by induction on the length of the string.

- Base case

  Suppose the string is a single character, then the value of that character is the value of k which is then taken $\mod m$.

- Inductive step.

  Let $w = w_1 w_2$ where $|w_1| \geq 1$ and $|w_2| = 1$. Suppose $h(w_1) = k_1$. Then, $h(w) = h(w_1)2^p + h(w_2) \mod 2^p - 1 = h(w_1) + h(w_2) \mod 2^p - 1$. So, since $h(w_1)$ was the sum of all but the last digit $\mod m$, and we are adding the last digit $\mod m$, we have the desired conclusion.

11.3-4

Consider a hash table of size $m = 1000$ and a corresponding hash function $h(k) = \lfloor m(kA \mod 1) \rfloor$ for $A = (\sqrt{5} - 1)/2$. Compute the locations to which the keys 61, 62, 63, 64, and 65 are mapped.

- $h(61) = \lfloor 1000(61 \cdot \frac{\sqrt{5}-1}{2} \mod 1) \rfloor = 700$.
- $h(62) = \lfloor 1000(62 \cdot \frac{\sqrt{5}-1}{2} \mod 1) \rfloor = 318$.
- $h(63) = \lfloor 1000(63 \cdot \frac{\sqrt{5}-1}{2} \mod 1) \rfloor = 936$.
- $h(64) = \lfloor 1000(64 \cdot \frac{\sqrt{5}-1}{2} \mod 1) \rfloor = 554$.
- $h(65) = \lfloor 1000(65 \cdot \frac{\sqrt{5}-1}{2} \mod 1) \rfloor = 172$.

11.3-5 ⋆

Define a family $\mathcal{H}$ of hash functions from a finite set U to a finite set B to be ***ϵ-universal*** if for all pairs of distinct elements k and l in U,

$$\Pr\{h(k) = h(l)\} \leq \epsilon,$$

where the probability is over the choice of the hash function h drawn at random from the family $\mathcal{H}$. Show that an ϵ-universal family of hash functions must have

$$\epsilon \geq \frac{1}{|B|} - \frac{1}{|U|}.$$

As a simplifying assumption, assume that $|B|$ divides $|U|$. It's just a bit messier if it doesn't divide evenly.

Suppose to a contradiction that $\epsilon > \frac{1}{|B|} - \frac{1}{|U|}$. This means that $\forall$ pairs $k, \ell \in U$, we have that the number $n_{k,\ell}$ of hash functions in $\mathcal{H}$ that have a collision on those two elements satisfies $n_{k,\ell} \leq \frac{|\mathcal{H}|}{|B|} - \frac{|\mathcal{H}|}{|U|}$. So, summing over all pairs of elements in U, we have that the total number is $\leq \frac{|\mathcal{H}||U|^2}{2|B|} - \frac{|\mathcal{H}||U|}{2}$.

Any particular hash function must have that there are at least $|B|\left(\frac{|U|\cdot|B|}{2}\right) = |B|\left(\frac{|U|^2 - |U||B|}{2|B|}\right) = \frac{|U|^2}{2|B|} - \frac{|U|}{2}$ colliding pairs for that hash function, summing over all hash functions, we get that there are at least $|\mathcal{H}|\left(\frac{|U|^2}{2|B|} - \frac{|U|}{2}\right)$ colliding pairs total. Since we have that there are at most some number less than this many, we have a contradiction, and so must have the desired restriction on ϵ.

11.3-6 ⋆

Let U be the set of n-tuples of values drawn from $\mathbb{Z}_p$, and let $B = \mathbb{Z}_p$, where p is prime. Define the hash function $h_b : U \to B$ for $b \in \mathbb{Z}_p$ on an input n-tuple $\langle a_0, a_1, \ldots, a_{n-1} \rangle$ from U as

$$h_b(\langle a_0, a_1, \ldots, a_{n-1} \rangle) = \left(\sum_{j=0}^{n-1} a_j b^j\right) \mod p,$$

and let $\mathcal{H} = \{h_b : b \in \mathbb{Z}_p\}$. Argue that $\mathcal{H}$ is $((n-1)/p)$-universal according to the definition of ϵ-universal in Exercise 11.3-5. (Hint: See Exercise 31.4-4.)

Fix $b \in \mathbb{Z}_p$. By exercise 31.4-4, $h_b(x)$ collides with $h_b(y)$ for at most $n-1$ other $y \in U$. Since there are a total of p possible values that h_b takes on, the probability that $h_b(x) = h_b(y)$ is bounded from above by $\frac{n-1}{p}$, since this holds for any value of b, $\mathcal{H}$ is $((n-1)/p)$-universal.

11.4 Open addressing

11.4-1

Consider inserting the keys $10, 22, 31, 4, 15, 28, 17, 88, 59$ into a hash table of length $m = 11$ using open addressing with the auxiliary hash function $h'(k) = k$. Illustrate the result of inserting these keys using linear probing, using quadratic probing with $c_1 = 1$ and $c_2 = 3$, and using double hashing with $h_1(k) = k$ and $h_2(k) = 1 + (k \mod (m-1))$.

We use T_i to represent each time stamp t starting with $i = 0$, and if encountering a collision, then we iterate i from $i = 1$ to $i = m - 1 = 10$ until there is no collision.

- **Linear probing**:

$h(k,i) = (k+i) \mod 11$	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
0 mod 11		22	22	22	22	22	22	22	22
1 mod 11								88	88
2 mod 11									
3 mod 11									
4 mod 11					4	4	4	4	4
5 mod 11						15	15	15	15
6 mod 11							28	28	28
7 mod 11								17	17
8 mod 11									59
9 mod 11				31	31	31	31	31	31
10 mod 11	10	10	10	10	10	10	10	10	10

- **Quadradic probing**, it will look identical until there is a collision on inserting the fifth element:

$h(k,i) = (k+i+3i^2) \mod 11$	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
0 mod 11		22	22	22	22	22	22	22	22
1 mod 11									
2 mod 11								88	88
3 mod 11							17	17	17
4 mod 11					4	4	4	4	4
5 mod 11									
6 mod 11							28	28	28
7 mod 11									59
8 mod 11						15	15	15	15
9 mod 11				31	31	31	31	31	31
10 mod 11	10	10	10	10	10	10	10	10	10

Note that there is no way to insert the element 59 now, because the offsets coming from $c_1 = 1$ and $c_2 = 3$ can only be even, and an odd offset would be required to insert 59 because $59 \mod 11 = 4$ and all the empty positions are at odd indices.

- **Double hashing**:

$h(k,i) = (k + i(1 + k \mod 10)) \mod 11$	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
0 mod 11		22	22	22	22	22	22	22	22
1 mod 11									
2 mod 11									59
3 mod 11								17	17
4 mod 11					4	4	4	4	4
5 mod 11						15	15	15	15
6 mod 11							28	28	28
7 mod 11								88	88
8 mod 11									
9 mod 11				31	31	31	31	31	31
10 mod 11	10	10	10	10	10	10	10	10	10

11.4-2

Write pseudocode for HASH-DELETE as outlined in the text, and modify HASH-INSERT to handle the special value DELETED.

```
HASH-DELETE(T, k)
    i = 0
    repeat
        j = h(k, i)
        if T[j] == k
            T[j] = DELETE
            return j
        else i = i + 1
    until T[j] == NIL or i == m
    error "element not exist"
```

By implementing HASH-DELETE in this way, the HASH-INSERT need to be modified to treat NIL slots as empty ones.

```
HASH-INSERT(T, k)
    i = 0
    repeat
        j = h(k, i)
        if T[j] == NIL or T[j] == DELETE
            T[j] = k
            return j
        else i = i + 1
    until i == m
    error "hash table overflow"
```

11.4-3

Consider an open-address hash table with uniform hashing. Give upper bounds on the expected number of probes in an unsuccessful search and on the expected number of probes in a successful search when the load factor is 3/4 and when it is 7/8.

- $\alpha = 3/4$,
  - unsuccessful: $\frac{1}{1-\frac{3}{4}} = 4$ probes,
  - successful: $\frac{1}{\frac{3}{4}} \ln \frac{1}{1-\frac{3}{4}} \approx 1.848$ probes.
- $\alpha = 7/8$,
  - unsuccessful: $\frac{1}{1-\frac{7}{8}} = 8$ probes,
  - successful: $\frac{1}{\frac{7}{8}} \ln \frac{1}{1-\frac{7}{8}} \approx 2.377$ probes.

11.4-4 ⋆

Suppose that we use double hashing to resolve collisions—that is, we use the hash function $h(k,i) = (h_1(k) + ih_2(k)) \mod m$. Show that if m and $h_2(k)$ have greatest common divisor $d \geq 1$ for some key k, then an unsuccessful search for key k examines $(1/d)$th of the hash table before returning to slot $h_1(k)$. Thus, when $d = 1$, so that m and $h_2(k)$ are relatively prime, the search may examine the entire hash table. (Hint: See Chapter 31.)

Suppose $d = \gcd(m, h_2(k))$, the $\text{LCM}\ l = m \cdot h_2(k)/d$.

Since $d|h_2(k)$, then $m \cdot h_2(k)/d \mod m = 0 \cdot (h_2(k)/d \mod m) = 0$, therefore $(l + ih_2(k)) \mod m = ih_2(k) \mod m$, which means $ih_2(k) \mod m$ has a period of m/d.

11.4-5 ⋆

Consider an open-address hash table with a load factor α. Find the nonzero value α for which the expected number of probes in an unsuccessful search equals twice the expected number of probes in a successful search. Use the upper bounds given by Theorems 11.6 and 11.8 for these expected numbers of probes.

$$\frac{1}{1-\alpha} = 2 \cdot \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$
$$\alpha = 0.71533.$$

11.5 Perfect hashing

11.5-1 ⋆

Suppose that we insert n keys into a hash table of size m using open addressing and uniform hashing. Let $p(n, m)$ be the probability that no collisions occur. Show that $p(n, m) \leq e^{-n(n-1)/2m}$. (Hint: See equation (3.12).) Argue that when n exceeds $\sqrt{m}$, the probability of avoiding collisions goes rapidly to zero.

$$\begin{aligned} p(n,m) &= \frac{m}{m} \cdot \frac{m-1}{m} \cdots \frac{m-n+1}{m} \\ &= \frac{m \cdot (m-1) \cdots (m-n+1)}{m^n} \end{aligned}$$

$$\begin{aligned} (m-i) \cdot (m-n+i) &= (m - \frac{n}{2} + \frac{n}{2} - i) \cdot (m - \frac{n}{2} - \frac{n}{2} + i) \\ &= (m - \frac{n}{2})^2 - (i - \frac{n}{2})^2 \\ &\leq (m - \frac{n}{2})^2. \end{aligned}$$

$$\begin{aligned} p(n,m) &\leq \frac{m \cdot (m - \frac{n}{2})^{n-1}}{m^n} \\ &= (1 - \frac{n}{2m})^{n-1}. \end{aligned}$$

Based on equation (3.12), $e^x \geq 1 + x$,

$$\begin{aligned} p(n,m) &\leq (e^{-n/2m})^{n-1} \\ &= e^{-n(n-1)/2m}. \end{aligned}$$

Problem 11-1 Longest-probe bound for hashing

Suppose that we use an open-addressed hash table of size m to store $n \leq m/2$ items.

a. Assuming uniform hashing, show that for $i = 1, 2, \ldots, n$, the probability is at most 2^{-k} that the ith insertion requires strictly more than k probes.

b. Show that for $i = 1, 2, \ldots, n$, the probability is $O(1/n^2)$ that the ith insertion requires more than $2 \lg n$ probes.

Let the random variable X_i denote the number of probes required by the ith insertion. You have shown in part (b) that $\Pr\{X_i > 2 \lg n\} = O(1/n^2)$. Let the random variable $X = \max_{1 \leq i \leq n} X_i$ denote the maximum number of probes required by any of the n insertions.

c. Show that $\Pr\{X > 2 \lg n\} = O(1/n)$.

d. Show that the expected length $E[X]$ of the longest probe sequence is $O(\lg n)$.

(Removed)

Problem 11-2 Slot-size bound for chaining

Suppose that we have a hash table with n slots, with collisions resolved by chaining, and suppose that n keys are inserted into the table. Each key is equally likely to be hashed to each slot. Let M be the maximum number of keys in any slot after all the keys have been inserted. Your mission is to prove an $O(\lg n / \lg \lg n)$ upper bound on $E[M]$, the expected value of M.

a. Argue that the probability Q_k that exactly k keys hash to a particular slot is given by

$$Q_k = \left(\frac{1}{n}\right)^k \left(1 - \frac{1}{n}\right)^{n-k} \binom{n}{k}.$$

b. Let P_k be the probability that $M = k$, that is, the probability that the slot containing the most keys contains k keys. Show that $P_k \leq nQ_k$.

c. Use Stirling's approximation, equation (3.18), to show that $Q_k < e^k/k^k$.

d. Show that there exists a constant $c > 1$ such that $Q_{k_0} < 1/n^3$ for $k_0 = c \lg n / \lg \lg n$. Conclude that $P_k < 1/n^2$ for $k \geq k_0 = c \lg n / \lg \lg n$.

e. Argue that

$$E[M] \leq \Pr\left\{M > \frac{c \lg n}{\lg \lg n}\right\} \cdot n + \Pr\left\{M \leq \frac{c \lg n}{\lg \lg n}\right\} \cdot \frac{c \lg n}{\lg \lg n}.$$

Conclude that $E[M] = O(\lg n / \lg \lg n)$.

(Removed)

Problem 11-3 Quadratic probing

Suppose that we are given a key k to search for in a hash table with positions $0, 1, \ldots, m-1$, and suppose that we have a hash function h mapping the key space into the set $\{0, 1, \ldots, m-1\}$. The search scheme is as follows:

1. Compute the value $j = h(k)$, and set $i = 0$.
2. Probe in position j for the desired key k. If you find it, or if this position is empty, terminate the search.
3. Set $i = i + 1$. If i now equals m, the table is full, so terminate the search. Otherwise, set $j = (i + j) \mod m$, and return to step 2.

Assume that m is a power of 2.

a. Show that this scheme is an instance of the general "quadratic probing" scheme by exhibiting the appropriate constants c_1 and c_2 for equation (11.5).

b. Prove that this algorithm examines every table position in the worst case.

(Removed)

Problem 11-4 Hashing and authentication

Let $\square$ be a class of hash functions in which each hash function $h \in \square$ maps the universe U of keys to $\{0, 1, \ldots, m-1\}$. We say that $\square$ is ***k-universal*** if, for every fixed sequence of k distinct keys $\langle x^{(1)}, x^{(2)}, \ldots, x^{(k)} \rangle$ and for any h chosen at random from $\square$, the sequence $\langle h(x^{(1)}), h(x^{(2)}), \ldots, h(x^{(k)}) \rangle$ is equally likely to be any of the m^k sequences of length k with elements drawn from $\{0, 1, \ldots, m-1\}$.

a. Show that if the family $\square$ of hash functions is 2-universal, then it is universal.

b. Suppose that the universe U is the set of n-tuples of values drawn from $\mathbb{Z}_p = \{0, 1, \ldots, p-1\}$, where p is prime. Consider an element $x = \langle x_0, x_1, \ldots, x_{n-1} \rangle \in U$. For any n-tuple $a = \langle a_0, a_1, \ldots, a_{n-1} \rangle \in U$, define the hash function h_a by

$$h_a(x) = \left(\sum_{j=0}^{n-1} a_j x_j \right) \mod p.$$

Let $\square = \{h_a\}$. Show that $\square$ is universal, but not 2-universal. (Hint: Find a key for which all hash functions in $\square$ produce the same value.)

c. Suppose that we modify $\square$ slightly from part (b): for any $a \in U$ and for any $b \in \mathbb{Z}_p$, define

$$h'_{ab}(x) = \left(\sum_{j=0}^{n-1} a_j x_j + b \right) \mod p$$

and $\square' = \{h'_{ab}\}$. Argue that $\square'$ is 2-universal. (Hint: Consider fixed n-tuples $x \in U$ and $y \in U$, with $x_i \neq y_i$ for some i. What happens to $h'_{ab}(x)$ and $h'_{ab}(y)$ as a_i and b range over $\mathbb{Z}_p$?)

d. Suppose that Alice and Bob secretly agree on a hash function h form 2-universal family $\square$ of hash functions. Each $h \in \square$ maps from a universe of keys u to $\mathbb{Z}_p$, where p is aprime. Later, Alice sends a message m to Bob over the Internet, where $m \in U$. She authenticates this message to Bob by also sending an authentication tag $t = h(m)$, and Bob checks that the pair (m, t) he receives indeed satisfies $t = h(m)$. Suppose that an adversary intercepts (m, t) en route and tries to fool Bob by replacing the pair (m, t) with a different pair (m', t'). Argue that the probability that the adversary succeeds in fooling Bob into accepting (m', t') is at most $1/p$, no matter how much computing power the adversary has, and even if the adversary knows the family $\square$ of hash functions used.

a. The number of hash functions for which $h(k) = h(l)$ is $\frac{m}{m^2}|\square| = \frac{1}{m}|\square|$, therefore the family is universal.

b. For $x = \langle 0, 0, \ldots, 0 \rangle$, $\square$ could not be 2-universal.

c. Let $x, y \in U$ be fixed, distinct n-tuples. As a_i and b range over $\mathbb{Z}_p$, $h'_{ab}(x)$ is equally likely to achieve every value from 1 to p since for any sequence a, we can let b vary from 1 to $p-1$.

Thus, $\langle h'_{ab}(x), h'_{ab}(y) \rangle$ is equally likely to be any of the p^2 sequences, so $\square$ is 2-universal.

d. Since $\square$ is 2-universal, every pair of $\langle t, t' \rangle$ is equally likely to appear, thus t' could be any value from $\mathbb{Z}_p$. Even the adversary knows $\square$, since $\square$ is 2-universal, then $\square$ is universal, the probability of choosing a hash function that $h(k) = h(l)$ is at most $1/p$, therefore the probability is at most $1/p$.

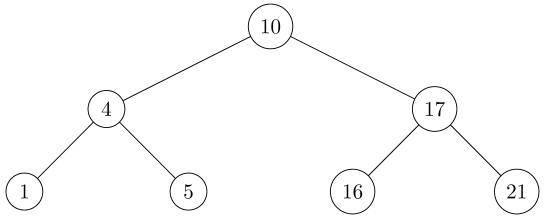
12 Binary Search Trees

12.1 What is a binary search tree?

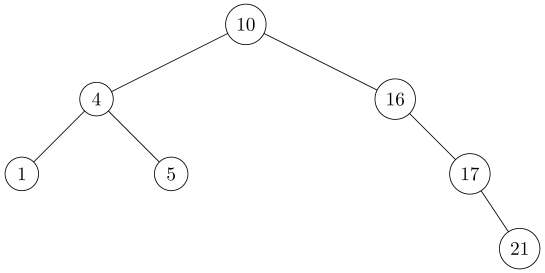
12.1-1

For the set of $\{1, 4, 5, 10, 16, 17, 21\}$ of keys, draw binary search trees of heights 2, 3, 4, 5, and 6.

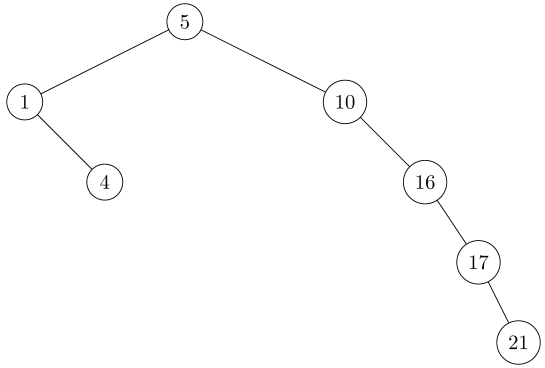
- height = 2:



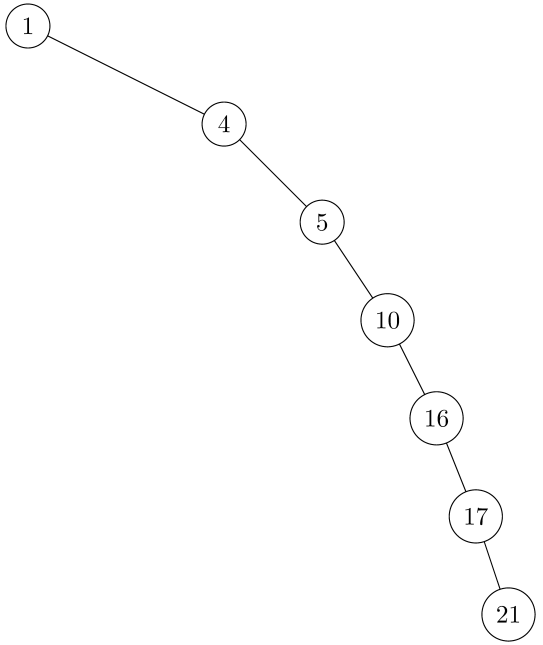
- height = 3:

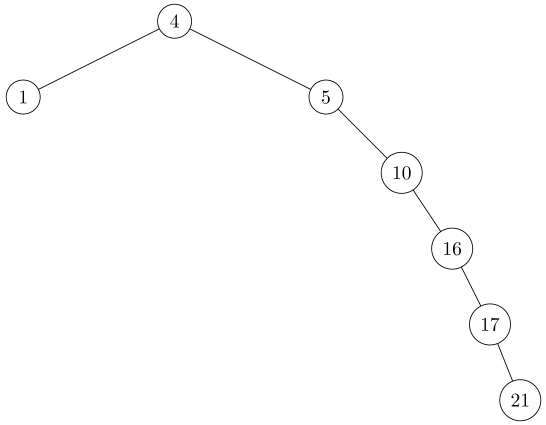


- height = 4:



- height = 5:





- height = 6:

12.1-2

What is the difference between the binary-search-tree property and the min-heap property (see page 153)? Can the min-heap property be used to print out the keys of an n-node tree in sorted order in $O(n)$ time? Show how, or explain why not.

- The binary-search-tree property guarantees that all nodes in the left subtree are smaller, and all nodes in the right subtree are larger.
- The min-heap property only guarantees the general child-larger-than-parent relation, but doesn't distinguish between left and right children. For this reason, the min-heap property can't be used to print out the keys in sorted order in linear time because we have no way of knowing which subtree contains the next smallest element.

12.1-3

Give a nonrecursive algorithm that performs an inorder tree walk. (Hint: An easy solution uses a stack as an auxiliary data structure. A more complicated, but elegant, solution uses no stack but assumes that we can test two pointers for equality.)

```
INORDER-TREE-WALK(T)
    let S be an empty stack
    current = T.root
    done = 0
    while !done
        if current != NIL
            PUSH(S, current)
            current = current.left
        else
            if !S.EMPTY()
                current = POP(S)
                print current
                current = current.right
            else done = 1
```

12.1-4

Give recursive algorithms that perform preorder and postorder tree walks in $\Theta(n)$ time on a tree of n nodes.

```
PREORDER-TREE-WALK(x)
    if x != NIL
        print x.key
        PREORDER-TREE-WALK(x.left)
        PREORDER-TREE-WALK(x.right)
```

```
POSTORDER-TREE-WALK(x)
    if x != NIL
        POSTORDER-TREE-WALK(x.left)
```

```
        POSTORDER-TREE-WALK(x.right)
        print x.key
```

12.1-5

> Argue that since sorting n elements takes $\Omega(n \lg n)$ time in the worst case in the comparison model, any comparison-based algorithm for constructing a binary search tree from an arbitrary list of n elements takes $\Omega(n \lg n)$ time in the worst case.

Assume, for the sake of contradiction, that we can construct the binary search tree by comparison-based algorithm using less than $\Omega(n \lg n)$ time, since the inorder tree walk is $\Theta(n)$, then we can get the sorted elements in less than $\Omega(n \lg n)$ time, which contradicts the fact that sorting n elements takes $\Omega(n \lg n)$ time in the worst case.

12.2 Querying a binary search tree

12.2-1

> Suppose that we have numbers between 1 and 1000 in a binary search tree, and we want to search for the number 363. Which of the following sequences could *not* be the sequence of nodes examined?
>
> **a.** $2, 252, 401, 398, 330, 344, 397, 363$.
>
> **b.** $924, 220, 911, 244, 898, 258, 362, 363$.
>
> **c.** $925, 202, 911, 240, 912, 245, 363$.
>
> **d.** $2, 399, 387, 219, 266, 382, 381, 278, 363$.
>
> **e.** $935, 278, 347, 621, 299, 392, 358, 363$.

- **c.** could not be the sequence of nodes explored because we take the left child from the 911 node, and yet somehow manage to get to the 912 node which cannot belong the left subtree of 911 because it is greater.
- **e.** is also impossible because we take the right subtree on the 347 node and yet later come across the 299 node.

12.2-2

> Write recursive versions of TREE-MINIMUM and TREE-MAXIMUM.

```
TREE-MINIMUM(x)
    if x.left != NIL
        return TREE-MINIMUM(x.left)
    else return x
```

```
TREE-MAXIMUM(x)
    if x.right != NIL
        return TREE-MAXIMUM(x.right)
    else return x
```

12.2-3

> Write the TREE-PREDECESSOR procedure.

```
TREE-PREDECESSOR(x)
    if x.left != NIL
        return TREE-MAXIMUM(x.left)
    y = x.p
    while y != NIL and x == y.left
        x = y
        y = y.p
    return y
```

12.2-4

> Professor Bunyan thinks he has discovered a remarkable property of binary search trees. Suppose that the search for key k in a binary search tree ends up in a leaf. Consider three sets: A, the keys to the left of the search path; B, the keys on the search path; and C, the keys to the right of the search path. Professor Bunyan claims that any three keys $a \in A$, $b \in B$, and $c \in C$ must satisfy $a \le b \le c$. Give a smallest possible counterexample to the professor's claim.

Search for 9 in this tree. Then $A = \{7\}$, $B = \{5, 8, 9\}$ and $C = \{\}$. So, since $7 > 5$ it breaks professor's claim.

12.2-5

> Show that if a node in a binary search tree has two children, then its successor has no left child and its predecessor has no right child.

Suppose the node x has two children. Then it's successor is the minimum element of the BST rooted at $x.\text{right}$. If it had a left child then it wouldn't be the minimum element. So, it must not have a left child. Similarly, the predecessor must be the maximum element of the left subtree, so cannot have a right child.

12.2-6

> Consider a binary search tree T whose keys are distinct. Show that if the right subtree of a node x in T is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. (Recall that every node is its own ancestor.)

First we establish that y must be an ancestor of x. If y weren't an ancestor of x, then let z denote the first common ancestor of x and y. By the binary-search-tree property, $x < z < y$, so y cannot be the successor of x.

Next observe that $y.\text{left}$ must be an ancestor of x because if it weren't, then $y.\text{right}$ would be an ancestor of x, implying that $x > y$. Finally, suppose that y is not the lowest ancestor of x whose left child is also an ancestor of x. Let z denote this lowest ancestor. Then z must be in the left subtree of y, which implies $z < y$, contradicting the fact that y is the successor of x.

12.2-7

> An alternative method of performing an inorder tree walk of an n-node binary search tree finds the minimum element in the tree by calling TREE-MINIMUM and then making $n - 1$ calls to TREE-SUCCESSOR. Prove that this algorithm runs in $\Theta(n)$ time.

To show this bound on the runtime, we will show that using this procedure, we traverse each edge twice. This will suffice because the number of edges in a tree is one less than the number of vertices.

Consider a vertex of a BST, say x. Then, we have that the edge between $x.p$ and x gets used when successor is called on $x.p$ and gets used again when it is called on the largest element in the subtree rooted at x. Since these are the only two times that that edge can be used, apart from the initial finding of tree minimum. We have that the runtime is $O(n)$. We trivially get the runtime is $\Omega(n)$ because that is the size of the output.

12.2-8

> Prove that no matter what node we start at in a height-h binary search tree, k successive calls to TREE-SUCCESSOR take $O(k + h)$ time.

Suppose x is the starting node and y is the ending node. The distance between x and y is at most $2h$, and all the edges connecting the k nodes are visited twice, therefore it takes $O(k + h)$ time.

12.2-9

> Let T be a binary search tree whose keys are distinct, let x be a leaf node, and let y be its parent. Show that $y.\text{key}$ is either the smallest key in T larger than $x.\text{key}$ or the largest key in T smaller than $x.\text{key}$.

- If $x = y.\text{left}$, then calling successor on x will result in no iterations of the while loop, and so will return y.
- If $x = y.\text{right}$, the while loop for calling predecessor (see exercise 3) will be run no times, and so y will be returned.

12.3 Insertion and deletion

12.3-1

> Give a recursive version of the TREE-INSERT procedure.

```
RECURSIVE-TREE-INSERT(T, z)
    if T.root == NIL
        T.root = z
    else INSERT(NIL, T.root, z)
```

```
INSERT(p, x, z)
    if x == NIL
        z.p = p
        if z.key < p.key
            p.left = z
        else p.right = z
    else if z.key < x.key
        INSERT(x, x.left, z)
    else INSERT(x, x.right, z)
```

12.3-2

> Suppose that we construct a binary search tree by repeatedly inserting distinct values into the tree. Argue that the number of nodes examined in searching for a value in the tree is one plus the number of nodes examined when the value was first inserted into the tree.

Number of nodes examined while searching also includes the node which is searched for, which isn't the case when we inserted it.

12.3-3

> We can sort a given set of n numbers by first building a binary search tree containing these numbers (using TREE-INSERT repeatedly to insert the numbers one by one) and then printing the numbers by an inorder tree walk. What are the worst-case and best-case running times for this sorting algorithm?

- The worst-case is that the tree formed has height n because we were inserting them in already sorted order. This will result in a runtime of $\Theta(n^2)$.
- The best-case is that the tree formed is approximately balanced. This will mean that the height doesn't exceed $O(\lg n)$. Note that it can't have a smaller height, because a complete binary tree of height h only has $\Theta(2^h)$ elements. This will result in a rutime of $O(n \lg n)$. We showed $\Omega(n \lg n)$ in exercise 12.1-5.

12.3-4

> Is the operation of deletion "commutative" in the sense that deleting x and then y from a binary search tree leaves the same tree as deleting y and then x? Argue why it is or give a counterexample.

No, giving the following courterexample.

- Delete A first, then delete B:

```
  A       C       C
 / \     / \       \
B   D   B   D       D
   /
  C
```

- Delete B first, then delete A:

```
  A       A       D
 / \       \     /
B   D       D   C
   /       /
  C       C
```

12.3-5

> Suppose that instead of each node x keeping the attribute $x.p$, pointing to x's parent, it keeps $x.\text{succ}$, pointing to x's successor. Give pseudocode for SEARCH, INSERT, and DELETE on a binary search tree T using this representation. These procedures should operate in time $O(h)$, where h is the height of the tree T. (Hint: You may wish to implement a subroutine that returns the parent of a node.)

We don't need to change SEARCH.

We have to implement PARENT, which facilitates us a lot.

```
PARENT(T, x)
    if x == T.root
        return NIL
    y = TREE-MAXIMUM(x).succ
    if y == NIL
        y = T.root
    else
        if y.left == x
            return y
        y = y.left
    while y.right != x
        y = y.right
    return y
```

```
INSERT(T, z)
    y = NIL
    x = T.root
    pred = NIL
    while x != NIL
        y = x
        if z.key < x.key
            x = x.left
        else
            pred = x
            x = x.right
    if y == NIL
        T.root = z
        z.succ = NIL
    else if z.key < y.key
        y.left = z
        z.succ = y
        if pred != NIL
            pred.succ = z
    else
        y.right = z
        z.succ = y.succ
        y.succ = z
```

We modify TRANSPLANT a bit since we no longer have to keep the pointer of p.

```
TRANSPLANT(T, u, v)
    p = PARENT(T, u)
    if p == NIL
        T.root = v
    else if u == p.left
        p.left = v
    else
        p.right = v
```

Also, we have to implement TREE-PREDECESSOR, which helps us easily find the predecessor in line 2 of DELETE.

```
TREE-PREDECESSOR(T, x)
    if x.left != NIL
        return TREE-MAXIMUM(x.left)
    y = T.root
    pred = NIL
    while y != NIL
        if y.key == x.key
```

```
            break
        if y.key < x.key
            pred = y
            y = y.right
        else
            y = y.left
    return pred
```

```
DELETE(T, z)
    pred = TREE-PREDECESSOR(T, z)
    pred.succ = z.succ
    if z.left == NIL
        TRANSPLANT(T, z, z.right)
    else if z.right == NIL
        TRANSPLANT(T, z, z.left)
    else
        y = TREE-MIMIMUM(z.right)
        if PARENT(T, y) != z
            TRANSPLANT(T, y, y.right)
            y.right = z.right
        TRANSPLANT(T, z, y)
        y.left = z.left
```

Therefore, all these five algorithms are still $O(h)$ despite the increase in the hidden constant factor.

12.3-6

> When node z in TREE-DELETE has two children, we could choose node y as its predecessor rather than its successor. What other changes to TREE-DELETE would be necessary if we did so? Some have argued that a fair strategy, giving equal priority to predecessor and successor, yields better empirical performance. How might TREE-DELETE be changed to implement such a fair strategy?

Update line 5 so that y is set equal to TREE-MAXIMUM($z.\,\mathrm{left}$) and lines 6-12 so that every $y.\,\mathrm{left}$ and $z.\,\mathrm{left}$ is replaced with $y.\,\mathrm{right}$ and $z.\,\mathrm{right}$ and vice versa.

To implement the fair strategy, we could randomly decide each time TREE-DELETE is called whether or not to use the predecessor or successor.

12.4 Randomly built binary search trees

12.4-1

> Prove equation (12.3).

Consider all the possible positions of the largest element of the subset of $n+3$ of size 4. Suppose it were in position $i+4$ for some $i \le n-1$. Then, we have that there are $i+3$ positions from which we can select the remaining three elements of the subset. Since every subset with different largest element is different, we get the total by just adding them all up (inclusion exclusion principle).

12.4-2

> Describe a binary search tree on n nodes such that the average depth of a node in the tree is $\Theta(\lg n)$ but the height of the tree is $\omega(\lg n)$. Give an asymptotic upper bound on the height of an n-node binary search tree in which the average depth of a node is $\Theta(\lg n)$.

To keep the average depth low but maximize height, the desired tree will be a complete binary search tree, but with a chain of length $c(n)$ hanging down from one of the leaf nodes. Let $k = \lg(n - c(n))$ be the height of the complete binary search tree. Then the average height is approximately given by

$$\frac{1}{n}\left[\sum_{i=1}^{n-c(n)} \lg i + (k+1) + (k+2) + \cdots + (k + c(n))\right] \approx \lg(n - c(n)) + \frac{c(n)^2}{2n}.$$

The upper bound is given by the largest $c(n)$ such that $\lg(n - c(n)) + \frac{c(n)^2}{2n} = \Theta(\lg n)$ and $c(n) = \omega(\lg n)$. One function which works is $\sqrt{n}$.

12.4-3

> Show that the notion of a randomly chosen binary search tree on n keys, where each binary search tree of n keys is equally likely to be chosen, is different from the notion of a randomly built binary search tree given in this section. (Hint: List the possibilities when $n = 3$.)

Suppose we have the elements $\{1, 2, 3\}$. Then, if we construct a tree by a random ordering, then, we get trees which appear with probabilities some multiple of $\frac{1}{6}$. However, if we consider all the valid binary search trees on the key set of $\{1, 2, 3\}$. Then, we will have only five different possibilities. So, each will occur with probability $\frac{1}{5}$, which is a different probability distribution.

12.4-4

> Show that the function $f(x) = 2^x$ is convex.

The second derivative is $2^x \ln^2 2$ which is always positive, so the function is convex

12.4-5 ⋆

> Consider RANDOMIZED-QUICKSORT operating on a sequence of n distinct input numbers. Prove that for any constant $k > 0$, all but $O(1/n^k)$ of the $n!$ input permutations yield an $O(n \lg n)$ running time.

Let $A(n)$ denote the probability that when quicksorting a list of length n, some pivot is selected to not be in the middle $n^{1-k/2}$ of the numberes. This doesn't happen with probability $\frac{1}{n^{k/2}}$. Then, we have that the two subproblems are of size n_1, n_2 with $n_1 + n_2 = n - 1$, then

$$A(n) \le \frac{1}{n^{k/2}} + T(n_1) + T(n_2).$$

Since we bounded the depth by $O(1/\lg n)$ let $\{a_{i,j}\}_i$ be all the subproblem sizes left at depth j,

$$A(n) \le \frac{1}{n^{k/2}} \sum_j \sum_i \frac{1}{a}.$$

Problem 12-1 Binary search trees with equal keys

> Equal keys pose a problem for the implementation of binary search trees.
>
> **a.** What is the asymptotic performance of TREE-INSERT when used to insert n items with identical keys into an initially empty binary search tree?
>
> We propose to improve TREE-INSERT by testing before line 5 to determine whether $z.\,\mathrm{key} = x.\,\mathrm{key}$ and by testing before line 11 to determine whether $z.\,\mathrm{key} = y.\,\mathrm{key}$.
>
> If equality holds, we implement one of the following strategies. For each strategy, find the asymptotic performance of inserting n items with identical keys into an initially empty binary search tree. (The strategies are described for line 5, in which we compare the keys of z and x. Substitute y for x to arrive at the strategies for line 11.)
>
> **b.** Keep a boolean flag $x.\,b$ at node x, and set x to either $x.\,\mathrm{left}$ or $x.\,\mathrm{right}$ based on the value of $x.\,b$, which alternates between FALSE and TRUE each time we visit x while inserting a node with the same key as x.
>
> **c.** Keep a list of nodes with equal keys at x, and insert z into the list.
>
> **d.** Randomly set x to either $x.\,\mathrm{left}$ or $x.\,\mathrm{right}$. (Give the worst-case performance and informally derive the expected running time.)

a. Each insertion will add the element to the right of the rightmost leaf because the inequality on line 11 will always evaluate to false. This will result in the runtime being $\sum_{i=1}^{n} i \in \Theta(n^2)$.

b. This strategy will result in each of the two children subtrees having a difference in size at most one. This means that the height will be $\Theta(\lg n)$. So, the total runtime will be $\sum_{i=1}^{n} \lg n \in \Theta(n \lg n)$.

c. This will only take linear time since the tree itself will be height 0, and a single insertion into a list can be done in constant time.

d.

- **Worst-case:** every random choice is to the right (or all to the left) this will result in the same behavior as in the first part of this problem, $\Theta(n^2)$.
- **Expected running time:** notice that when randomly choosing, we will pick left roughly half the time, so, the tree will be roughly balanced, so, we have that the depth is roughly $\lg(n)$, $\Theta(n \lg n)$.

Problem 12-2 Radix trees

> Given two strings $a = a_0 a_1 \ldots a_p$ and $b = b_0 b_1 \ldots b_q$, where each a_i and each b_j is in some ordered set of characters, we say that string a is ***lexicographically less than*** string b if either
>
> 1. there exists an integer j, where $0 \le j \le \min(p, q)$, such that $a_i = b_i$ for all $i = 0, 1, \ldots j-1$ and $a_j < b_j$, or
> 2. $p < q$ and $a_i = b_i$ for all $i = 0, 1, \ldots, p$.
>
> For example, if a and b are bit strings, then $10100 < 10110$ by rule 1 (letting $j = 3$) and $10100 < 101000$ by rule 2. This ordering is similar to that used in English-language dictionaries.
>
> The ***radix tree*** data structure shown in Figure 12.5 stores the bit strings 1011, 10, 011, 100, and 0. When searching for a key $a = a_0 a_1 \ldots a_p$, we go left at a node of depth i if $a_i = 0$ and right if $a_i = 1$. Let S be a set of distinct bit strings whose lengths sum to n. Show how to use a radix tree to sort S lexicographically in $\Theta(n)$ time. For the example in Figure 12.5, the output of the sort should be the sequence $0, 011, 10, 100, 1011$.

(Removed)

Problem 12-3 Average node depth in a randomly built binary search tree

> In this problem, we prove that the average depth of a node in a randomly built binary search tree with n nodes is $O(\lg n)$. Although this result is weaker than that of Theorem 12.4, the technique we shall use reveals a surprising similarity between the building of a binary search tree and the execution of RANDOMIZED-QUICKSORT from Section 7.3.
>
> We define the ***total path length*** $P(T)$ of a binary tree T as the sum, over all nodes x in T, of the depth of node x, which we denote by $d(x, T)$.
>
> **a.** Argue that the average depth of a node in T is
>
> $$\frac{1}{n}\sum_{x \in T} d(x, T) = \frac{1}{n} P(T).$$
>
> Thus, we wish to show that the expected value of $P(T)$ is $O(n \lg n)$.
>
> **b.** Let T_L and T_R denote the left and right subtrees of tree T, respectively. Argue that if T has n nodes, then
>
> $$P(T) = P(T_L) + P(T_R) + n - 1.$$
>
> **c.** Let $P(n)$ denote the average total path length of a randomly built binary search tree with n nodes. Show that
>
> $$P(n) = \frac{1}{n}\sum_{i=0}^{n-1}(P(i) + P(n - i - 1) + n - 1).$$
>
> **d.** Show how to rewrite $P(n)$ as
>
> $$P(n) = \frac{2}{n}\sum_{k=1}^{n-1} P(k) + \Theta(n).$$
>
> **e.** Recalling the alternative analysis of the randomized version of quicksort given in Problem 7-3, conclude that $P(n) = O(n \lg n)$. At each recursive invocation of quicksort, we choose a random pivot element to partition the set of elements being sorted. Each node of a binary search tree partitions the set of elements that fall into the subtree rooted at that node.
>
> **f.** Describe an implementation of quicksort in which the comparisons to sort a set of elements are exactly the same as the comparisons to insert the elements into a binary search tree. (The order in which comparisons are made may differ, but the same comparisons must occur.)

(Removed)

Problem 12-4 Number of different binary trees

> Let b_n denote the number of different binary trees with n nodes. In this problem, you will find a formula for b_n, as well as an asymptotic estimate.
>
> **a.** Show that $b_0 = 1$ and that, for $n \ge 1$,
>
> $$b_n = \sum_{k=0}^{n-1} b_k b_{n-1-k}.$$
>
> **b.** Referring to Problem 4-4 for the definition of a generating function, let $B(x)$ be the generating function
>
> $$B(x) = \sum_{n=0}^{\infty} b_n x^n.$$
>
> Show that $B(x) = xB(x)^2 + 1$, and hence one way to express $B(x)$ in closed form is
>
> $$B(x) = \frac{1}{2x}(1 - \sqrt{1 - 4x}).$$
>
> The ***Taylor expansion*** of $f(x)$ around the point $x = a$ is given by
>
> $$f(x) = \sum_{k=0}^{\infty} \frac{f^{(k)}(a)}{k!}(x - a)^k,$$
>
> where $f^{(k)}(x)$ is the kth derivative of f evaluated at x.
>
> **c.** Show that
>
> $$b_n = \frac{1}{n+1}\binom{2n}{n}$$
>
> (the nth ***Catalan number***) by using the Taylor expansion of $\sqrt{1 - 4x}$ around $x = 0$. (If you wish, instead of using the Taylor expansion, you may use the generalization of the binomial expansion (C.4) to nonintegral exponents n, where for any real number n and for any integer k, we interpret $\binom{n}{k}$ to be $n(n-1)\cdots(n-k+1)/k!$ if $k \ge 0$, and 0 otherwise.)
>
> **d.** Show that
>
> $$b_n = \frac{4^n}{\sqrt{\pi} n^{3/2}}(1 + O(1/n)).$$

a. A root with two subtree.

b.

$$\begin{aligned}
B(x)^2 &= (b_0 x^0 + b_1 x^1 + b_2 x^2 + \cdots)^2 \\
&= b_0^2 x^0 + (b_0 b_1 + b_1 b_0) x^1 + (b_0 b_2 + b_1 b_1 + b_2 b_0) x^2 + \cdots \\
&= \sum_{k=0}^{0} b_k b_{0-k} x^0 + \sum_{k=0}^{1} b_k b_{1-k} x^1 + \sum_{k=0}^{2} b_k b_{2-k} x^2 + \cdots
\end{aligned}$$

$$\begin{aligned}
xB(x)^2 + 1 &= 1 + \sum_{k=0}^{0} b_k b_{1-1-k} x^1 + \sum_{k=0}^{2} b_k b_{2-1-k} x^3 + \sum_{k=0}^{2} b_k b_{3-1-k} x^2 + \cdots \\
&= 1 + b_1 x^1 + b_2 x^2 + b_3 x^3 + \cdots \\
&= b_0 x^0 + b_1 x^1 + b_2 x^2 + b_3 x^3 + \cdots \\
&= \sum_{n=0}^{\infty} b_n x^n \\
&= B(x).
\end{aligned}$$

$$\begin{aligned}
xB(x)^2 + 1 &= x \cdot \frac{1}{4x^2}(1 + 1 - 4x - 2\sqrt{1 - 4x}) + 1 \\
&= \frac{1}{4x}(2 - 2\sqrt{1 - 4x}) - 1 + 1 \\
&= \frac{1}{2x}(1 - \sqrt{1 - 4x}) \\
&= B(x).
\end{aligned}$$

c. Let $f(x) = \sqrt{1 - 4x}$, the numerator of the derivative is

$$
\begin{aligned}
2 \cdot (1 \cdot 2) \cdot (3 \cdot 2) \cdot (5 \cdot 2) \cdots &= 2^k \cdot \prod_{i=0}^{k-2} (2k+1) \\
&= 2^k \cdot \frac{(2(k-1))!}{2^{k-1}(k-1)!} \\
&= \frac{2(2(k-1))!}{(k-1)!}.
\end{aligned}
$$

$$
f(x) = 1 - 2x - 2x^2 - 4x^3 - 10x^4 - 28x^5 - \cdots .
$$

The coefficient is $\frac{2(2(k-1))!}{k!(k-1)!}$.

$$
\begin{aligned}
B(x) &= \frac{1}{2x}(1 - f(x)) \\
&= 1 + x + 2x^2 + 5x^3 + 14x^4 + \cdots \\
&= \sum_{n=0}^{\infty} \frac{(2n)!}{(n+1)!n!} x \\
&= \sum_{n=0}^{\infty} \frac{1}{n+1} \frac{(2n)!}{n!n!} x \\
&= \sum_{n=0}^{\infty} \frac{1}{n+1} \binom{2n}{n} x.
\end{aligned}
$$

$$
b_n = \frac{1}{n+1}\binom{2n}{n}.
$$

d.

$$
\begin{aligned}
b_n &= \frac{1}{n+1}\frac{(2n)!}{n!n!} \\
&\approx \frac{1}{n+1}\frac{\sqrt{4\pi n}(2n/e)^{2n}}{2\pi n(n/e)^{2n}} \\
&= \frac{1}{n+1}\frac{4^n}{\sqrt{\pi n}} \\
&= (\frac{1}{n} + (\frac{1}{n+1} - \frac{1}{n}))\frac{4^n}{\sqrt{\pi n}} \\
&= (\frac{1}{n} - \frac{1}{n^2+n})\frac{4^n}{\sqrt{\pi n}} \\
&= \frac{1}{n}(1 - \frac{1}{n+1})\frac{4^n}{\sqrt{\pi n}} \\
&= \frac{4^n}{\sqrt{\pi n^{3/2}}}(1 + O(1/n)).
\end{aligned}
$$

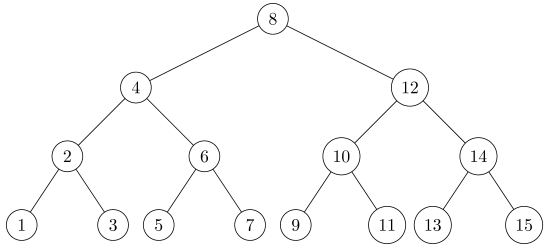
13 Red-Black Trees

13.1 Properties of red-black trees

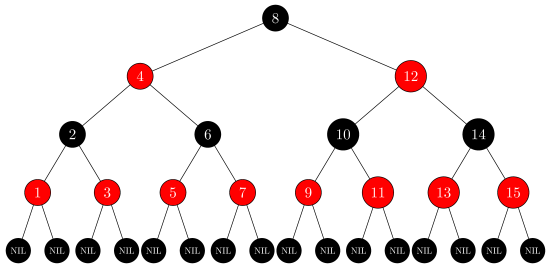
13.1-1

> In the style of Figure 13.1(a), draw the complete binary search tree of height 3 on the keys $\{1, 2, \ldots, 15\}$. Add the NIL leaves and color the nodes in three different ways such that the black-heights of the resulting red-black trees are 2, 3, and 4.

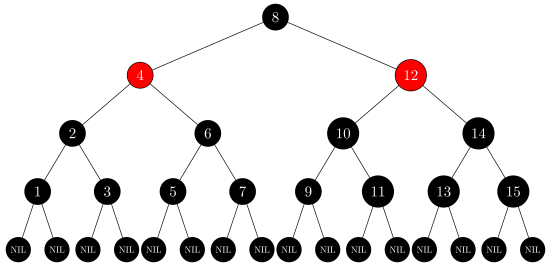
- Complete binary tree of height $= 3$:



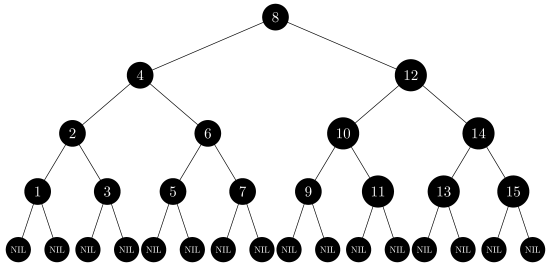
- Red-black tree of black-heights $= 2$:



- Red-black tree of black-heights $= 3$:



- Red-black tree of black-heights $= 4$:

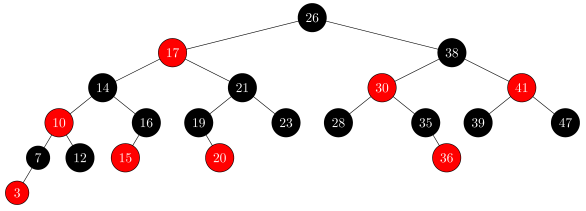


13.1-2

> Draw the red-black tree that results after TREE-INSERT is called on the tree in Figure 13.1 with key 36. If the inserted node is colored red, is the resulting tree a red-black tree? What if it is colored black?

- If the inserted node is colored red, the tree doesn't satisfy property 4 because 35 will be the parent of 36, which is also colored red.
- If the inserted node is colored black, the tree doesn't satisfy property 5 because there will be two paths from node 38 to $\mathrm{T.nil}$ which contain different numbers of black nodes.

We don't draw the *wrong* red-black tree; however, we draw the adjusted correct tree:



13.1-3

> Let us define a ***relaxed red-black tree*** as a binary search tree that satisfies red-black properties 1, 3, 4, and 5. In other words, the root may be either red or black. Consider a relaxed red-black tree T whose root is red. If we color the root of T black but make no other changes to T, is the resulting tree a red-black tree?

Yes, it is.

- Property 1 is trivially satisfied since only one node is changed and it is not changed to some mysterious third color.
- Property 3 is trivially satisfied since no new leaves are introduced.
- Property 4 is satisfied since there was no red node introduced, and root is in every path from the root to the leaves, but no others.
- Property 5 is satisfied since the only paths we will be changing the number of black nodes in are those coming from the root. All of these will increase by 1, and so will all be equal.

13.1-4

> Suppose that we "absorb" every red node in a red-black tree into its black parent, so that the children of the red node become children of the black parent. (Ignore what happens to the keys.) What are the possible degrees of a black node after all its red children are absorbed? What can you say about the depths of the leaves of the resulting tree?

The degree of a node in a rooted tree is the number of its children (see Section B.5.2). Given this definition, the possible degrees are 0, 2, 3 and 4, based on whether the black node had zero, one or two red children, each with either zero or two black children. The depths could shrink by at most a factor of $1/2$.

13.1-5

> Show that the longest simple path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from node x to a descendant leaf.

Suppose we have the longest simple path $(a_1, a_2, \ldots, a_s)$ and the shortest simple path $(b_1, b_2, \ldots, b_t)$. Then, by property 5 we know they have equal numbers of black nodes. By property 4, we know that neither contains a repeated red node. This tells us that at most $\left\lfloor \frac{s-1}{2} \right\rfloor$ of the nodes in the longest path are red. This means that at least $\left\lceil \frac{s+1}{2} \right\rceil$ are black, so, $t \geq \left\lceil \frac{s+1}{2} \right\rceil$. Therefore, if, by way of contradiction, we had that $s > t \cdot 2$, then $t \geq \left\lfloor \frac{s+1}{2} \right\rfloor \geq \left\lceil \frac{2t+2}{2} \right\rceil = t + 1$ results a contradiction.

13.1-6

> What is the largest possible number of internal nodes in a red-black tree with black-height k? What is the smallest possible number?

- The largest is a path with half black nodes and half red nodes, which has $2^{2k} - 1$ internal nodes.
- The smallest is a path with all black nodes, which has $2^k - 1$ internal nodes.

13.1-7

> Describe a red-black tree on n keys that realizes the largest possible ratio of red internal nodes to black internal nodes. What is this ratio? What tree has the smallest possible ratio, and what is the ratio?

- The largest ratio is 2, each black node has two red children.
- The smallest is 0.

13.2 Rotations

13.2-1

> Write pseudocode for RIGHT-ROTATE.

```
RIGHT-ROTATE(T, y)
    x = y.left
    y.left = x.right
    if x.right != T.nil
        x.right.p = y
    x.p = y.p
    if y.p == T.nil
        T.root = x
    else if y == y.p.right
        y.p.right = x
    else y.p.left = x
    x.right = y
    y.p = x
```

13.2-2

> Argue that in every n-node binary search tree, there are exactly $n - 1$ possible rotations.

Every node can rotate with its parent, only the root does not have a parent, therefore there are $n - 1$ possible rotations.

13.2-3

> Let a, b, and c be arbitrary nodes in subtrees α, β, and γ, respectively, in the left tree of Figure 13.2. How do the depths of a, b, and c change when a left rotation is performed on node x in the figure?

- a: increase by 1.
- b: unchanged.
- c: decrease by 1.

13.2-4

> Show that any arbitrary n-node binary search tree can be transformed into any other arbitrary n-node binary search tree using $O(n)$ rotations. (Hint: First show that at most $n - 1$ right rotations suffice to transform the tree into a right-going chain.)

Consider transforming an arbitrary n-node binary tree into a right-going chain as follows:

Let the root and all successive right children of the root be the elements of the chain initial chain. For any node x which is a left child of a node on the chain, a single right rotation on the parent of x will add that node to the chain and not remove any elements from the chain. Thus, we can convert any binary search tree to a right chain with at most $n - 1$ right rotations.

Let $r_1, r_2, \dots, r_k$ be the sequence of rotations required to convert some binary search tree T_1 into a right-going chain, and let $s_1, s_2, \dots, s_m$ be the sequence of rotations required to convert some other binary search tree T_2 to a right-going chain. Then $k < n$ and $m < n$, and we can convert T_1 to T_2 by performing the sequence $r_1, r_2, \dots, r_k, s'_m, s'_{m-1}, \dots, s'_1$ where s'_i is the opposite rotation of s_i. Since $k + m < 2n$, the number of rotations required is $O(n)$.

13.2-5 *

> We say that a binary search tree T_1 can be ***right-converted*** to binary search tree T_2 if it is possible to obtain T_2 from T_1 via a series of calls to RIGHT-ROTATE. Give an example of two trees T_1 and T_2 such that T_1 cannot be right-converted to T_2. Then, show that if a tree T_1 can be right-converted to T_2, it can be right-converted using $O(n^2)$ calls to RIGHT-ROTATE.

We can use $O(n)$ calls to rotate the node which is the root in T_2 to T_1's root, then use the same operation in the two subtrees. There are n nodes, therefore the upper bound is $O(n^2)$.

13.3 Insertion

13.3-1

> In line 16 of RB-INSERT, we set the color of the newly inserted node z to red. Observe that if we had chosen to set z's color to black, then property 4 of a red-black tree would not be violated. Why didn't we choose to set z's color to black?

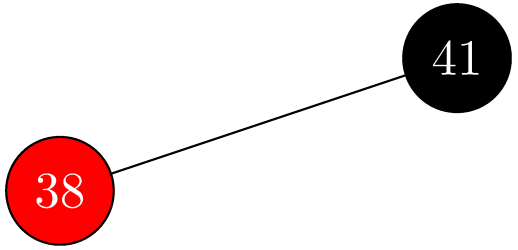
If we chose to set the color of z to black then we would be violating property 5 of being a red-black tree. Because any path from the root to a leaf under z would have one more black node than the paths to the other leaves.

13.3-2

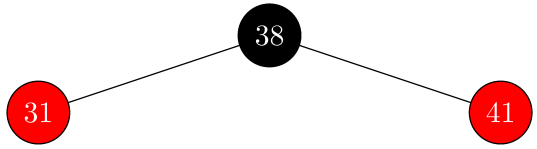
> Show the red-black trees that result after successively inserting the keys $41, 38, 31, 12, 19, 8$ into an initially empty red-black tree.

- insert 41:

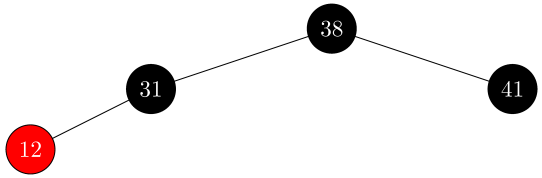
- insert 38:



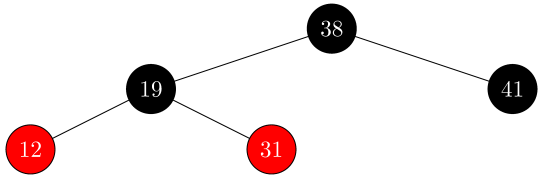
- insert 31:



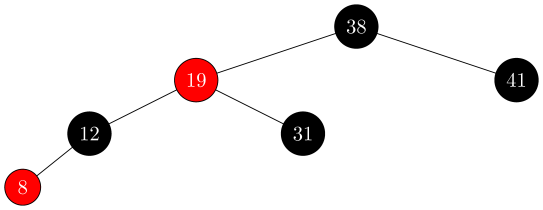
- insert 12:



- insert 19:



- insert 8:



13.3-3

> Suppose that the black-height of each of the subtrees $\alpha, \beta, \gamma, \delta, \epsilon$ in Figures 13.5 and 13.6 is k. Label each node in each figure with its black-height to verify that the indicated transformation preserves property 5.

(Removed)

13.3-4

> Professor Teach is concerned that RB-INSERT-FIXUP might set $T.nil.color$ to RED, in which case the test in line 1 would not cause the loop to terminate when z is the root. Show that the professor's concern is unfounded by arguing that RB-INSERT-FIXUP never sets $T.nil.color$ to RED.

First observe that RB-INSERT-FIXUP only modifies the child of a node if it is already RED, so we will never modify a child which is set to $T.nil$. We just need to check that the parent of the root is never set to RED.

Since the root and the parent of the root are automatically black, if z is at depth less than 2, the **while** loop will be broken. We only modify colors of nodes at most two levels above z, so the only case we need to worry about is when z is at depth 2. In this case we risk modifying the root to be RED, but this is handled in line 16. When z is updated, it will be either the root or the child of the root. Either way, the root and the parent of the root are still BLACK, so the **while** condition is violated, making it impossibly to modify $T.nil$ to be RED.

13.3-5

> Consider a red-black tree formed by inserting n nodes with RB-INSERT. Argue that if $n > 1$, the tree has at least one red node.

- **Case 1:** z and $z.p.p$ are RED, if the loop terminates, then z could not be the root, thus z is RED after the fix up.
- **Case 2:** z and $z.p$ are RED, and after the rotation $z.p$ could not be the root, thus $z.p$ is RED after the fix up.
- **Case 3:** z is RED and z could not be the root, thus z is RED after the fix up.

Therefore, there is always at least one red node.

13.3-6

> Suggest how to implement RB-INSERT efficiently if the representation for red-black trees includes no storage for parent pointers.

Use stack to record the path to the inserted node, then parent is the top element in the stack.

- **Case 1:** we pop $z.p$ and $z.p.p$.
- **Case 2:** we pop $z.p$ and $z.p.p$, then push $z.p.p$ and z.
- **Case 3:** we pop $z.p$, $z.p.p$ and $z.p.p.p$, then push $z.p$.

13.4 Deletion

13.4-1

> Argue that after executing RB-DELETE-FIXUP, the root of the tree must be black.

- **Case 1:** transform to 2, 3, 4.
- **Case 2:** if terminates, the root of the subtree (the new x) is set to black.
- **Case 3:** transform to 4.
- **Case 4:** the root (the new x) is set to black.

13.4-2

> Argue that if in RB-DELETE both x and $x.p$ are red, then property 4 is restored by the call to RB-DELETE-FIXUP(T, x).

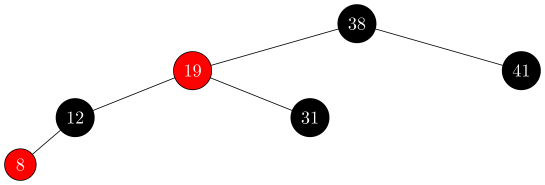
Suppose that both x and $x.p$ are red in RB-DELETE. This can only happen in the else-case of line 9. Since we are deleting from a red-black tree, the other child of y.p which becomes x's sibling in the call to

RB-TRANSPLANT on line 14 must be black, so x is the only child of $x.p$ which is red. The while-loop condition of RB-DELETE-FIXUP(T, x) is immediately violated so we simply set $x.color = \text{black}$, restoring property 4.

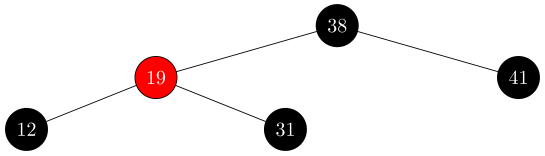
13.4-3

> In Exercise 13.3-2, you found the red-black tree that results from successively inserting the keys $41, 38, 31, 12, 19, 8$ into an initially empty tree. Now show the red-black trees that result from the successive deletion of the keys in the order $8, 12, 19, 31, 38, 41$.

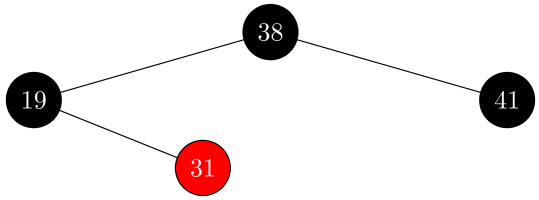
- initial:



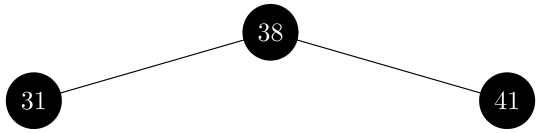
- delete 8:



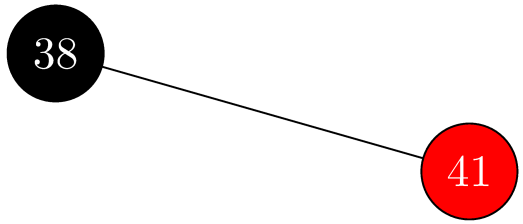
- delete 12:



- delete 19:



- delete 31:



- delete 38:

- delete 41:

13.4-4

> In which lines of the code for RB-DELETE-FIXUP might we examine or modify the sentinel $T.nil$?

When the node y in RB-DELETE has no children, the node $x = T.nil$, so we'll examine the line 2 of RB-DELETE-FIXUP.

When the root node is deleted, $x = T.nil$ and the root at this time is x, so the line 23 of RB-DELETE-FIXUP will draw x to black.

13.4-5

> In each of the cases of Figure 13.7, give the count of black nodes from the root of the subtree shown to each of the subtrees $\alpha, \beta, \ldots, \zeta$, and verify that each count remains the same after the transformation. When a node has a color attribute c or c', use the notation $\text{count}(c)$ or $\text{count}(c')$ symbolically in your count.

Our count will include the root (if it is black).

- **Case 1:** For each subtree, it is 2 both before and after.
- **Case 2:**
  - For α and β, it is $1 + \text{count}(c)$ in both cases.
  - For the rest of the subtrees, it is from $2 + \text{count}(c)$ to $1 + \text{count}(c)$.

  This decrease in the count for the other subtreese is handled by then having x represent an additional black.
- **Case 3:**
  - For ϵ and ζ, it is $2 + \text{count}(c)$ both before and after.
  - For all the other subtrees, it is $1 + \text{count}(c)$ both before and after.
- **Case 4:**
  - For α and β, it is from $1 + \text{count}(c)$ to $2 + \text{count}(c)$.
  - For γ and δ, it is $1 + \text{count}(c) + \text{count}(c')$ both before and after.
  - For ϵ and ζ, it is $1 + \text{count}(c)$ both before and after.

This increase in the count for α and β is because x before indicated an extra black.

13.4-6

> Professors Skelton and Baron are concerned that at the start of case 1 of RB-DELETE-FIXUP, the node $x.p$ might not be black. If the professors are correct, then lines 5–6 are wrong. Show that $x.p$ must be black at the start of case 1, so that the professors have nothing to worry about.

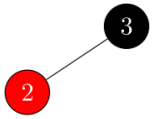
At the start of case 1 we have set w to be the sibling of x. We check on line 4 that $w.color == red$, which means that the parent of x and w cannot be red. Otherwise property 4 is violated. Thus, their concerns are unfounded.

13.4-7

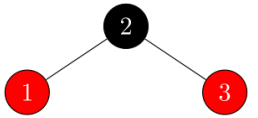
> Suppose that a node x is inserted into a red-black tree with RB-INSERT and then is immediately deleted with RB-DELETE. Is the resulting red-black tree the same as the initial red-black tree? Justify your answer.

No, the red-black tree will not necessarily be the same.

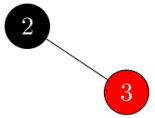
- Example 1:
  - initial:



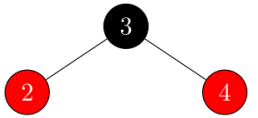
  - insert 1:



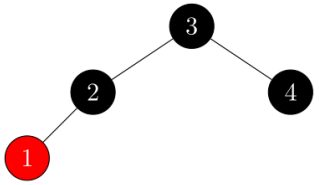
  - delete 1:



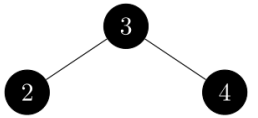
- Example 2:
  - initial:



  - insert 1:



  - delete 1:



Problem 13-1 Persistent dynamic sets

> During the course of an algorithm, we sometimes find that we need to maintain past versions of a dynamic set as it is updated. We call such a set ***persistent***. One way to implement a persistent set is to copy the entire set whenever it is modified, but this approach can slow down a program and also consume much space. Sometimes, we can do much better.
>
> Consider a persistent set S with the operations INSERT, DELETE, and SEARCH, which we implement using binary search trees as shown in Figure 13.8(a). We maintain a separate root for every version of the set. In order to insert the key 5 into the set, we create a new node with key 5. This node becomes the left child of a new node with key 7, since we cannot modify the existing node with key 7. Similarly, the new node with key 7 becomes the left child of a new node with key 8 whose right child is the existing node with key 10. The new node with key 8 becomes, in turn, the right child of a new root r' with key 4 whose left child is the existing node with key 3. We thus copy only part of the tree and share some of the nodes with the original tree, as shown in Figure 13.8(b).
>
> Assume that each tree node has the attributes key, $left$, and $right$ but no parent. (See also Exercise 13.3-6.)
>
> **a.** For a general persistent binary search tree, identify the nodes that we need to change to insert a key k or delete a node y.

b. Write a procedure PERSISTENT-TREE-INSERT that, given a persistent tree T and a key k to insert, returns a new persistent tree T' that is the result of inserting k into T.

c. If the height of the persistent binary search tree T is h, what are the time and space requirements of your implementation of PERSISTENT-TREE-INSERT? (The space requirement is proportional to the number of new nodes allocated.)

d. Suppose that we had included the parent attribute in each node. In this case, PERSISTENT-TREE-INSERT would need to perform additional copying. Prove that PERSISTENT-TREE-INSERT would then require $\Omega(n)$ time and space, where n is the number of nodes in the tree.

e. Show how to use red-black trees to guarantee that the worst-case running time and space are $O(\lg n)$ per insertion or deletion.

(Removed)

Problem 13-2 Join operation on red-black trees

The ***join*** operation takes two dynamic sets S_1 and S_2 and an element x such that for any $x_1 \in S_1$ and $x_2 \in S_2$, we have $x_1.key \le x.key \le x_2.key$. It returns a set $S = S_1 \cup \{x\} \cup S_2$. In this problem, we investigate how to implement the join operation on red-black trees.

a. Given a red-black tree T, let us store its black-height as the new attribute $T.bh$. Argue that RB-INSERT and RB-DELETE can maintain the bh attribute without requiring extra storage in the nodes of the tree and without increasing the asymptotic running times. Show that while descending through T, we can determine the black-height of each node we visit in $O(1)$ time per node visited.

We wish to implement the operation RB-JOIN(T_1, x, T_2), which destroys T_1 and T_2 and returns a red-black tree $T = T_1 \cup \{x\} \cup T_2$. Let n be the total number of nodes in T_1 and T_2.

b. Assume that $T_1.bh \ge T_2.bh$. Describe an $O(\lg n)$-time algorithm that finds a black node y in T_1 with the largest key from among those nodes whose black-height is $T_2.bh$.

c. Let T_y be the subtree rooted at y. Describe how $T_y \cup \{x\} \cup T_2$ can replace T_y in $O(1)$ time without destroying the binary-search-tree property.

d. What color should we make x so that red-black properties 1, 3, and 5 are maintained? Describe how to enforce properties 2 and 4 in $O(\lg n)$ time.

e. Argue that no generality is lost by making the assumption in part (b). Describe the symmetric situation that arises when $T_1.bh \le T_2.bh$.

f. Argue that the running time of RB-JOIN is $O(\lg n)$.

a.

- Initialize: $bh = 0$.
- RB-INSERT: if in the last step the root is red, we increase bh by 1.
- RB-DELETE: if x is root, we decrease bh by 1.
- Each node: in the simple path, decrease bh by 1 each time we find a black node.

b. Move to the right child if the node has a right child, otherwise move to the left child. If the node is black, we decease bh by 1. Repeat the step until $bh = T_2.bh$.

c. The time complexity is $O(1)$.

```
RB-JOIN'(T[y], x, T[2])
    TRANSPLANT(T[y], x)
    x.left = T[y]
    x.right = T[2]
    T[y].parent = x
    T[2].parent = x
```

d. Red. Call INSERT-FIXUP(T[1], x).

The time complexity is $O(\lg n)$.

e. Same, if $T_1.bh \le T_2.bh$, then we can use the above algorithm symmetrically.

f. $O(1) + O(\lg n) = O(\lg n)$.

Problem 13-3 AVL trees

An ***AVL tree*** is a binary search tree that is ***height balanced***: for each node x, the heights of the left and right subtrees of x differ by at most 1. To implement an AVL tree, we maintain an extra attribute in each node: $x.h$ is the height of node x. As for any other binary search tree T, we assume that $T.root$ points to the root node.

a. Prove that an AVL tree with n nodes has height $O(\lg n)$. (Hint: Prove that an AVL tree of height h has at least F_h nodes, where F_h is the hth Fibonacci number.)

b. To insert into an AVL tree, we first place a node into the appropriate place in binary search tree order. Afterward, the tree might no longer be height balanced. Specifically, the heights of the left and right children of some node might differ by 2. Describe a procedure BALANCE(x), which takes a subtree rooted at x whose left and right children are height balanced and have heights that differ by at most 2, i.e., $|x.right.h - x.left.h| \le 2$, and alters the subtree rooted at x to be height balanced. (Hint: Use rotations.)

c. Using part (b), describe a recursive procedure AVL-INSERT(x, z) that takes a node x within an AVL tree and a newly created node z (whose key has already been filled in), and adds z to the subtree rooted at x, maintaining the property that x is the root of an AVL tree. As in TREE-INSERT from Section 12.3, assume that $z.key$ has already been filled in and that $z.left = NIL$ and $z.right = NIL$; also assume that $z.h = 0$. Thus, to insert the node z into the AVL tree T, we call AVL-INSERT$(T.root, z)$.

d. Show that AVL-INSERT, run on an n-node AVL tree, takes $O(\lg n)$ time and performs $O(1)$ rotations.

a. Let $T(h)$ denote the minimum size of an AVL tree of height h. Since it is height h, it must have the max of it's children's heights is equal to $h - 1$. Since we are trying to get as few nodes total as possible, suppose that the other child has as small of a height as is allowed. Because of the restriction of AVL trees, we have that the smaller child must be at least one less than the larger one, so, we have that

$$T(h) \ge T(h-1) + T(h-2) + 1,$$

where the $+1$ is coming from counting the root node.

We can get inequality in the opposite direction by simply taking a tree that achieves the minimum number of number of nodes on height $h - 1$ and on $h - 2$ and join them together under another node.

So, we have that

$$T(h) = T(h-1) + T(h-2) + 1, \text{ where } T(0) = 0, T(1) = 1.$$

This is both the same recurrence and initial conditions as the Fibonacci numbers. So, recalling equation (3.25), we have that

$$T(h) = \left\lfloor \frac{\phi^h}{\sqrt{5}} + \frac{1}{2} \right\rfloor \le n.$$

Rearranging for h, we have

$$\begin{aligned} \frac{\phi^h}{\sqrt{5}} - \frac{1}{2} &\le n \\ \phi^h &\le \sqrt{5}(n + \frac{1}{2}) \\ h &\le \frac{\lg \sqrt{5} + \lg(n + \frac{1}{2})}{\lg \phi} \in O(\lg n). \end{aligned}$$

b. Let UNBAL(x) denote $x.left.h - x.right.h$. Then, the algorithm BALANCE does what is desired. Note that because we are only rotating a single element at a time, the value of UNBAL(x) can only change by at most 2 in each step.

Also, it must eventually start to change as the tree that was shorter becomes saturated with elements. We also fix any breaking of the AVL property that rotating may of caused by our recursive calls to the children.

```
BALANCE(x)
    while |UNBAL(x)| > 1
        if UNBAL(x) > 0
            RIGHT-ROTATE(T, x)
        else
            LEFT-ROTATE(T, x)
            BALANCE(x.left)
            BALANCE(x.right)
```

c. For the given algorithm AVL-INSERT(x, z), it correctly maintains the fact that it is a BST by the way we search for the correct spot to insert z. Also, we can see that it maintains the property of being AVL, because after inserting the element, it checks all of the parents for the AVL property, since those are the only places it could of broken. It then fixes it and also updates the height attribute for any of the nodes for which it may of changed.

d. Both **for** loops only run for $O(h) = O(\lg(n))$ iterations. Also, only a single rotation will occur in the second while loop because when we do it, we will be decreasing the height of the subtree rooted there, which means that it's back down to what it was before, so all of it's ancestors will have unchanged heights, so, no further balancing will be required.

```
AVL-INSERT(x, z)
    w = x
    while w != NIL
        y = w
        if z.key > y.key
            w = w.right
        else w = w.left
    if z.key > y.key
        y.right = z
            if y.left = NIL
                y.h = 1
    else
        y.left = z
        if y.right = NIL
            y.h = 1
    while y != x
        y.h = 1 + max(y.left.h, y.right.h)
        if y.left.h > y.right.h + 1
            RIGHT-ROTATE(T, y)
        if y.right.h > y.left.h + 1
            LEFT-ROTATE(T, y)
            y = y.p
```

Problem 13-4 Treaps

If we insert a set of n items into a binary search tree, the resulting tree may be horribly unbalanced, leading to long search times. As we saw in Section 12.4, however, randomly built binary search trees tend to be balanced. Therefore, one strategy that, on average, builds a balanced tree for a fixed set of items would be to randomly permute the items and then insert them in that order into the tree.

What if we do not have all the items at once? If we receive the items one at a time, can we still randomly build a binary search tree out of them?

We will examine a data structure that answers this question in the affirmative. A ***treap*** is a binary search tree with a modified way of ordering the nodes. Figure 13.9 shows an example. As usual, each node x in the tree has a key value $x.key$. In addition, we assign $x.priority$, which is a random number chosen independently for each node. We assume that all priorities are distinct and also that all keys are distinct. The nodes of the treap are ordered so that the keys obey the binary-search-tree property and the priorities obey the min-heap order property:

- If v is a left child of u, then $v.key < u.key$.
- If v is a right child of u, then $v.key > u.key$.
- If v is a child of u, then $v.priority > u.priority$.

(This combination of properties is why the tree is called a "treap": it has features of both a binary search tree and a heap.)

It helps to think of treaps in the following way. Suppose that we insert nodes $x_1, x_2, \ldots, x_n$, with associated keys, into a treap. Then the resulting treap is the tree that would have been formed if the nodes had been inserted into a normal binary search tree in the order given by their (randomly chosen) priorities, i.e., $x_i.priority < x_j.priority$ means that we had inserted x_i before x_j.

a. Show that given a set of nodes $x_1, x_2, \ldots, x_n$, with associated keys and priorities, all distinct, the treap associated with these nodes is unique.

b. Show that the expected height of a treap is $\Theta(\lg n)$, and hence the expected time to search for a value in the treap is $\Theta(\lg n)$.

Let us see how to insert a new node into an existing treap. The first thing we do is assign to the new node a random priority. Then we call the insertion algorithm, which we call TREAP-INSERT, whose operation is illustrated in Figure 13.10.

c. Explain how TREAP-INSERT works. Explain the idea in English and give pseudocode. (Hint: Execute the usual binary-search-tree insertion procedure and then perform rotations to restore the min-heap order property.)

d. Show that the expected running time of TREAP-INSERT is $\Theta(\lg n)$.

TREAP-INSERT performs a search and then a sequence of rotations. Although these two operations have the same expected running time, they have different costs in practice. A search reads information from the treap without modifying it. In contrast, a rotation changes parent and child pointers within the treap. On most computers, read operations are much faster than write operations. Thus we would like TREAP-INSERT to perform few rotations. We will show that the expected number of rotations performed is bounded by a constant.

In order to do so, we will need some definitions, which Figure 13.11 depicts. The ***left spine*** of a binary search tree T is the simple path from the root to the node with the smallest key. In other words, the left spine is the simple path from the root that consists of only left edges. Symmetrically, the ***right spine*** of T is the simple path from the root consisting of only right edges. The ***length*** of a spine is the number of nodes it contains.

e. Consider the treap T immediately after TREAP-INSERT has inserted node x. Let C be the length of the right spine of the left subtree of x. Let D be the length of the left spine of the right subtree of x. Prove that the total number of rotations that were performed during the insertion of x is equal to $C + D$.

We will now calculate the expected values of C and D. Without loss of generality, we assume that the keys are $1, 2, \ldots, n$ since we are comparing them only to one another.

For nodes x and y in treap T, where $y \ne x$, let $k = x.key$ and $i = y.key$. We define indicator random variables

$$X_{ik} = I\{y \text{ is in the right spine of the left subtree of } x\}.$$

f. Show that $X_{ik} = 1$ if and only if $y.priority > x.priority$, $y.key < x.key$, and, for every z such that $y.key < z.key < x.key$, we have $y.priority < z.priority$.

g. Show that

$$\begin{aligned} \Pr\{X_{ik} = 1\} &= \frac{(k-i-1)!}{(k-i+1)!} \\ &= \frac{1}{(k-i+1)(k-i)}. \end{aligned}$$

h. Show that

$$\begin{aligned} E[C] &= \sum_{j=1}^{k-1} \frac{1}{j(j+1)} \\ &= 1 - \frac{1}{k}. \end{aligned}$$

i. Use a symmetry argument to show that

$$E[D] = 1 - \frac{1}{n-k+1}.$$

j. Conclude that the expected number of rotations performed when inserting a node into a treap is less than 2.

a. The root is the node with smallest priority, the root divides the sets into two subsets based on the key. In each subset, the node with smallest priority is selected as the root, thus we can uniquely determine a treap

with a specific input.

b. For the priority of all nodes, each permutation corresponds to exactly one treap, that is, all nodes forms a BST in priority, since the priority of all nodes is spontaneous, treap is, essentially, randomly built binary search tress. Therefore, the expected height of a treap is $\Theta(\lg n)$.

c. First insert a node as usual using the binary-search-tree insertion procedure. Then perform left and right rotations until the parent of the inserted node no longer has larger priority.

d. Rotation is $\Theta(1)$, at most h rotations, therefore the expected running time is $\Theta(\lg n)$.

e. Left rotation increase C by 1, right rotation increase D by 1.

f. The first two are obvious.

The min-heap property will not hold if $y.\text{priority} > z.\text{priority}$.

g.

$$\Pr\{X_{ik} = 1\} = \frac{(k-i-1)!}{(k-i+1)!} = \frac{1}{(k-i+1)(k-i)}.$$

h.

$$\begin{aligned} E[C] &= \sum_{j=1}^{k-1} \frac{1}{(k-i+1)(k-i)} \\ &= \sum_{j=1}^{k-1} (\frac{1}{k-i} - \frac{1}{k-i+1}) \\ &= 1 - \frac{1}{k}. \end{aligned}$$

i.

$$\begin{aligned} E[D] &= \sum_{j=1}^{n-k} \frac{1}{(k-i+1)(k-i)} \\ &= 1 - \frac{1}{n-k+1}. \end{aligned}$$

j. By part (e), the number of rotations is $C + D$. By linearity of expectation, $E[C + D] = 2 - \frac{1}{k} - \frac{1}{n-k+1} \leq 2$ for any choice of k.

14 Augmenting Data Structures

14.1 Dynamic order statistics

14.1-1

> Show how $\text{OS-SELECT}(T.\text{root}, 10)$ operates on the red-black tree T of Figure 14.1.

- $26 : r = 13, i = 10$, go left.
- $17 : r = 8, i = 10$, go right.
- $21 : r = 3, i = 2$, go left.
- $19 : r = 1, i = 2$, go right.
- $20 : r = 1, i = 1$, choose 20.

14.1-2

> Show how $\text{OS-RANK}(T, x)$ operates on the red-black tree T of Figure 14.1 and the node x with $x.\text{key} = 35$.

- $35 : r = 1$.
- $38 : r = 1$.
- $30 : r = r + 2 = 3$.
- $41 : r = 3$.
- $26 : r = r + 13 = 16$.

14.1-3

> Write a nonrecursive version of OS-SELECT.

```
OS-SELECT(x, i)
    r = x.left.size + 1
    while r != i
        if i < r
            x = x.left
        else x = x.right
            i = i - r
        r = x.left.size + 1
    return x
```

14.1-4

> Write a recursive procedure $\text{OS-KEY-RANK}(T, k)$ that takes as input an order-statistic tree T and a key k and returns the rank of k in the dynamic set represented by T. Assume that the keys of T are distinct.

```
OS-KEY-RANK(T, k)
    if k == T.root.key
        return T.root.left.size + 1
    else if T.root.key > k
        return OS-KEY-RANK(T.left, k)
    else return T.root.left.size + 1 + OS-KEY-RANK(T.right, k)
```

14.1-5

> Given an element x in an n-node order-statistic tree and a natural number i, how can we determine the ith successor of x in the linear order of the tree in $O(\lg n)$ time?

The desired result is $\text{OS-SELECT}(T, \text{OS-RANK}(T, x) + i)$. This has runtime $O(h)$, which by the properties of red black trees, is $O(\lg n)$.

14.1-6

> Observe that whenever we reference the size attribute of a node in either OS-SELECT or OS-RANK, we use it only to compute a rank. Accordingly, suppose we store in each node its rank in the subtree of which it is the root. Show how to maintain this information during insertion and deletion. (Remember that these two operations can cause rotations.)

First perform the usual BST insertion procedure on z, the node to be inserted. Then add 1 to the rank of every node on the path from the root to z such that z is in the left subtree of that node. Since the added node is a leaf, it will have no subtrees so its rank will always be 1.

When a left rotation is performed on x, its rank within its subtree will remain the same. The rank of $x.\text{right}$ will be increased by the rank of x, plus one. If we perform a right rotation on a node y, its rank will decrement by $y.\text{left}.\text{rank} + 1$. The rank of $y.\text{left}$ will remain unchanged.

For deletion of z, decrement the rank of every node on the path from z to the root such that z is in the left subtree of that node. For any rotations, use the same rules as before.

14.1-7

> Show how to use an order-statistic tree to count the number of inversions (see Problem 2-4) in an array of size n in time $O(n \lg n)$.

The runtime to build a red-black tree is $O(n \lg n)$, so we need to calculate inversions while building trees.

Every time INSERT, we can use OS-RANK to calculate the rank of the node, thus calculating inversions.

14.1-8 ⋆

> Consider n chords on a circle, each defined by its endpoints. Describe an $O(n \lg n)$-time algorithm to determine the number of pairs of chords that intersect inside the circle. (For example, if the n chords are all diameters that meet at the center, then the correct answer is $\binom{n}{2}$.) Assume that no two chords share an endpoint.

Sort the vertices in clock-wise order, and assign a unique value to each vertex. For each chord its two vertices are u_i, v_i and $u_i < v_i$. Add the vertices one by one in clock-wise order, if we meet a u_i, we add it to the order-statistic tree, if we meet a v_i, we calculate how many nodes are larger than u_i (which is the number of intersects with chord i), and remove u_i.

14.2 How to augment a data structure

14.2-1

> Show, by adding pointers to the nodes, how to support each of the dynamic-set queries MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR in $O(1)$worstcase time on an augmented order-statistic tree. The asymptotic performance of other operations on order-statistic trees should not be affected.

- **MINIMUM:** A pointer points to the minimum node, if the node is being deleted, move the pointer to its successor.
- **MAXIMUM:** Similar to MINIMUM.
- **SUCCESSOR:** Every node records its successor, the insertion and deletion is similar to that in linked list.
- **PREDECESSOR:** Similar to MAXIMUM.

14.2-2

> Can we maintain the black-heights of nodes in a red-black tree as attributes in the nodes of the tree without affecting the asymptotic performance of any of the redblack tree operations? Show how, or argue why not. How about maintaining the depths of nodes?

Since the black height of a node depends only on the black height and color of its children, Theorem 14.1 implies that we can maintain the attribute without affecting the asymptotic performance of the other red-black tree operations. The same is not true for maintaining the depths of nodes. If we delete the root of a tree we could potentially have to update the depths of $O(n)$ nodes, making the DELETE operation asymptotically slower than before.

14.2-3 ⋆

> Let $\otimes$ be an associative binary operator, and let a be an attribute maintained in each node of a red-black tree. Suppose that we want to include in each node x an additional attribute f such that $x.f = x_1.a \otimes x_2.a \otimes \cdots \otimes x_m.a$, where $x_1, x_2, \ldots, x_m$ is the inorder listing of nodes in the subtree rooted at x. Show how to update the f attributes in $O(1)$ time after a rotation. Modify your argument slightly to apply it to the size attributes in order-statistic trees.

$x.f = x.\text{left}.f \otimes x.a \otimes x.\text{right}.f$.

14.2-4 ⋆

> We wish to augment red-black trees with an operation $\text{RB-ENUMERATE}(x, a, b)$ that outputs all the keys k such that $a \leq k \leq b$ in a red-black tree rooted at x. Describe how to implement RB-ENUMERATE in $\Theta(m + \lg n)$ time, where m is the number of keys that are output and n is the number of internal nodes in the tree. (Hint: You do not need to add new attributes to the red-black tree.)

- $\Theta(\lg n)$: Find the smallest key that larger than or equal to a.
- $\Theta(m)$: Based on Exercise 14.2-1, find the m successor.

14.3 Interval trees

14.3-1

> Write pseudocode for LEFT-ROTATE that operates on nodes in an interval tree and updates the $\max$ attributes in $O(1)$ time.

Add 2 lines in LEFT-ROTATE in 13.2

```
    y.max = x.max
    x.max = max(x.high, x.left.max, x.right.max)
```

14.3-2

> Rewrite the code for INTERVAL-SEARCH so that it works properly when all intervals are open.

```
INTERVAL-SEARCH(T, i)
    x = T.root
    while x != T.nil and i does not overlap x.int
        if x.left != T.nil and x.left.max > i.low
            x = x.left
        else x = x.right
    return x
```

14.3-3

> Describe an efficient algorithm that, given an interval i, returns an interval overlapping i that has the minimum low endpoint, or $T.\text{nil}$ if no such interval exists.

Consider the usual interval search given, but, instead of breaking out of the loop as soon as we have an overlap, we just keep track of the overlap that has the minimum low endpoint, and continue the loop. After the loop terminates, we return the overlap stored.

14.3-4

> Given an interval tree T and an interval i, describe how to list all intervals in T that overlap i in $O(\min(n, k \lg n))$ time, where k is the number of intervals in the output list. (Hint: One simple method makes several queries, modifying the tree between queries. A slightly more complicated method does not modify the tree.)

```
INTERVALS-SEARCH(T, x, i)
    let list be an empty array
    if i overlaps x.int
        list.APPEND(x)
    if x.left != T.nil and x.left.max > i.low
        list = list.APPEND(INTERVALS-SEARCH(T, x.left, i))
    if x.right != T.nil and x.int.low ≤ i.high and x.right.max ≥ i.low
        list = list.APPEND(INTERVALS-SEARCH(T, x.right, i))
    return list
```

14.3-5

> Suppose modifications to the interval-tree procedures to support the new operation $\text{INTERVAL-SEARCH-EXACTLY}(T, i)$, where T is an interval tree and i is an interval. The operation should return a pointer to a node x in T such that $x.\text{int}.\text{low} = i.\text{low}$ and $x.\text{int}.\text{high} = i.\text{high}$, or $T.\text{nil}$ if T contains no such node. All operations, including $\text{INTERVAL-SEARCH-EXACTLY}$, should run in $O(\lg n)$ time on an n-node interval tree.

Search for nodes which has exactly the same low value.

```
INTERVAL-SEARCH-EXACTLY(T, i)
    x = T.root
    while x != T.nil and i not exactly overlap x
        if i.high > x.max
            x = T.nil
        else if i.low < x.low
            x = x.left
        else if i.low > x.low
            x = x.right
        else x = T.nil
    return x
```

14.3-6

> Show how to maintain a dynamic set Q of numbers that supports the operation MIN-GAP, which gives the magnitude of the difference of the two closest numbers in Q. For example, if $Q = \{1, 5, 9, 15, 18, 22\}$, then MIN-GAP($Q$) returns $18 - 15 = 3$, since 15 and 18 are the two closest numbers in Q. Make the operations INSERT, DELETE, SEARCH, and MIN-GAP as efficient as possible, and analyze their running times.

Store the elements in a red-black tree, where the key value is the value of each number itself. The auxiliary attribute stored at a node x will be the min gap between elements in the subtree rooted at x, the maximum value contained in subtree rooted at x, and the minimum value contained in the subtree rooted at x. The min gap at a leaf will be ∞. Since we can determine the attributes of a node x using only the information about the key at x, and the attributes in $x.left$ and $x.right$, Theorem 14.1 implies that we can maintain the values in all nodes of the tree during insertion and deletion without asymptotically affecting their $O(\lg n)$ performance. For MIN-GAP, just check the min gap at the root, in constant time.

14.3-7 *

> VLSI databases commonly represent an integrated circuit as a list of rectangles. Assume that each rectangle is rectilinearly oriented (sides parallel to the x- and y-axes), so that we represent a rectangle by its minimum and maximum x and y-coordinates. Give an $O(n \lg n)$-time algorithm to decide whether or not a set of n rectangles so represented contains two rectangles that overlap. Your algorithm need not report all intersecting pairs, but it must report that an overlap exists if one rectangle entirely covers another, even if the boundary lines do not intersect. (Hint: Move a "sweep" line across the set of rectangles.)

Let L be the set of left coordinates of rectangles. Let R be the set of right coordinates of rectangles. Sort both of these sets in $O(n \lg n)$ time. Then, we will have a pointer to L and a pointer to R. If the pointer to L is smaller, call interval search on T for the up-down interval corresponding to this left hand side. If it contains something that intersects the up-down bounds of this rectangle, there is an intersection, so stop.

Otherwise add this interval to T and increment the pointer to L. If R is the smaller one, remove the up-down interval that that right hand side corresponds to and increment the pointer to R. Since all the interval tree operations used run in time $O(\lg n)$ and we only call them at most $3n$ times, we have that the runtime is $O(n \lg n)$.

Problem 14-1 Point of maximum overlap

> Suppose that we wish to keep track of a ***point of maximum overlap*** in a set of intervals—a point with the largest number of intervals in the set that overlap it.
>
> **a.** Show that there will always be a point of maximum overlap that is an endpoint of one of the segments.

> **b.** Design a data structure that efficiently supports the operations INTERVAL-INSERT, INTERVAL-DELETE, and FIND-POM, which returns a point of maximum overlap. (Hint: Keep a red-black tree of all the endpoints. Associate a value of $+1$ with each left endpoint, and associate a value of -1 with each right endpoint. Augment each node of the tree with some extra information to maintain the point of maximum overlap.)

(Removed)

Problem 14-2 Josephus permutation

> We define the ***Josephus problem*** as follows. Suppose that n people form a circle and that we are given a positive integer $m \leq n$. Beginning with a designated first person, we proceed around the circle, removing every mth person. After each person is removed, counting continues around the circle that remains. This process continues until we have removed all n people. The order in which the people are removed from the circle defines the (n, m)***-Josephus permutation*** of the integers $1, 2, \ldots, n$. For example, the $(7, 3)$-Josephus permutation is $\langle 3, 6, 2, 7, 5, 1, 4 \rangle$.
>
> **a.** Suppose that m is a constant. Describe an $O(n)$-time algorithm that, given an integer n, outputs the (n, m)-Josephus permutation.
>
> **b.** Suppose that m is not a constant. Describe an $O(n \lg n)$-time algorithm that, given integers n and m, outputs the (n, m)-Josephus permutation.

(Removed)