CLRS Answers

Karl Frederick Roldan

Contents

1 Getting Started

1.1 Exercises 2.1

1. Using Figure 2.2 as a model, illustrate the operation of `insertion-sort` on the array $A = \langle 31, 41, 59, 26, 41, 58\rangle$.

2. Rewrite the `insertion-sort` procedure to sort into nonincreasing instead of the decreasing order.

```
insertion-sort(A)
    for j=2 to A.length
        key = A[j]
        // Insert A[j] into the
        // sorted sequence A[1..j-1].
        i = j - 1
        while > 0 and A[i] < key
            A[i + 1] = A[i]
            i = i - 1
            A[i + 1] = key
```

3. Consider the **searching problem:**

   Input: A sequence of n numbers $A = \langle a_1, a_2, \ldots, a_n$ and a value v **Output:** An index i such that $v = A[i]$ or the special value `NIL` if v does not appear in A.

   Write the pseudocode for **linear search**, which scans through the sequence, looking for v. Using a loop invariant, prove that your algorithm is correct. Make sure that your loop invariant fulfills the necessary properties.

```
linear-search(A, v)
    for i = 1 to A.length
        if v == A[i]
            return i
    return NIL
```

Proof. The loop invariant of **linear search** is that the subarray $A[0..i-1]$ are all elements not equal to v. If $A[0] = v$, the loop terminates(more on this later) and the procedure returns 0. If $A[0] \neq v$, the loop invariant holds true trivially.
We maintain the loop invariant by checking if $A[j] = v$ for all $j < i$. Since all elements $A[0..i-1]$ or $A[0..j]$ are not equal to v, this preserves the loop invariant.
The loop terminates when either $A[i] = A[j] = v$ or when we reached the end of the loop but no element has been found to be equal to v. Since we're incrementing j in every iteration of the loop, we will come to a point where $j > A.length$ in which case we return `NIL`. Hence, we either found such a $j = v$ or not. Hence, the algorithm is correct. □

4. Consider the problem of adding two n-bit binary integers stored in two n-element arrays A and B. The sum of the two integers should be stored in binary form in an $n+1$-element array C. State the problem formally and write pseudocode for adding the two integers.

We consider a procedure called **add binary**
Input: Two n-element arrays A and B representing a binary string each. **Output:** The binary sum of size $n+1$ of A and B.

```
add-binary(A, B)
    // we assume A.length = B.length
    n = A.length
    carry = 0
    C = a new array of size n + 1
    for i = 1 to n + 2
        a = A[n - i]
        b = B[n - i]
        c = (a + b + carry) % 2
        carry = floor((a + b) / 2)
        C[n - i + 1] = c
    return C
```

1.2 Exercises 2.2

1. Express the function $n^3/1000 - 100n^2 - 100n + 3$ in terms of Θ-notation.

$n^3/1000 - 100n^2 - 100n + 3 = \Theta(n^3)$
Without rigorous proofs, we can determine that $n^3/1000 - 100n^2 - 100n + 3 \leq \Theta(n^3)$ which preserves the definition of Θ

2. Consider sorting n numbers stored in array A by first finding the smallest element of A and exchanging it with the element in $A[1]$. Then find the second smallest element of A, and exchange it with $A[2]$. Continue in this manner for the first $n-1$ elements of A. Write the pseudocode for this algorithm, which is known as **selection sort**. What loop invariant does this algorithm maintain? Why does it need to run for only the first $n-1$ elements rather than for all n elements? Give the best-case and worst-case running times of selection sort in Θ-notation.

We first have the `smallest` function which gets the smallest element in the array and returns its index. The function takes in a variable p which is the index where the algorithm will start searching and r which is the index of the array where the algorithm will end.

```
smallest(A, p, r)
    min = begin // smallest index
    for i=min to end
        if A[min] >= A[i]
            min = i
    return min
```

We then have the `selection-sort` function which does the actual sorting.

```
selection-sort(A)
    for i=1 to A.length - 1
        min = smallest(A, i, A.length)
        swap A[min] and A[i]
```

The loop invariant is the same as **insertion sort**. The subarray $A[1..i-1]$ during the loop are the sorted subarray while $A[i..length]$ are the unsorted subarray. The algorithm only needs to run for the first $n-1$ elements because having to run n elements means that we are swapping the last element with itself which is indempotent and has no effect since it is already sorted by itself.

3. Consider linear search again (see Exercise 2.1-3). How many elements of the input sequence need to be checked on the average, assuming that the element being searched for is equally liked to be any element in the array? How about in the worst case? What are the average-case and worst-case running times of linear search in Θ-notation? Justify your answers.

4. How can we modify almost any algorithm to have a good best-case running time?

We can assure a best-case scenario in an algorithm by making special cases. As in linear search, the best case would be $\Theta(1)$ if we check that the first element is equal to v where v is the value we are looking for.
Also, in sorting, we can use the indempotence property of an array by checking if the array is already sorted to begin with and eliminate sorting it again. This results to a best-case analysis of $\Theta(n)$.

1.3 Exercises 2.3

1. Using Figure 2.4 as a model, illustrate the operation of merge sort on the array

$$A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$$

.

2. Rewrite the merge procedure so it does not use sentinels, instead stopping once either array L or R has had all its elements copied back to A and then copying the remainder of the other array back into A.

3. Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 2, \\ 2T(\frac{n}{2}) + n & \text{if } n = 2^k, \text{ for } k > 1 \end{cases}$$

is $T(n) = n \lg n$.

Proof. The proof proceeds by induction. For the base case, suppose than $n = 2$ and from the recurrence, we get

$$2 = 2 \lg 2$$

and thus the base case is true.
Let's assume that the inductive hypothesis $T(i) = 2T(\frac{i}{2}) + n$ is true for some $n = i = 2^k$. Then by the principle of induction, it should be true when $n = i + 1 = 2^{k+1}$. To show,

$$\begin{aligned} T(i+1) &= 2T(\frac{2^{k+1}}{2}) + 2^{k+1} \\ &= 2T(2^k) + 2^{k+1}. \end{aligned}$$

From the induction hypothesis, we get

$$\begin{aligned} T(i+1) &= 2(i \lg i) + 2^{k+1} \\ &= 2(2^k \lg 2^k) + 2^{k+1} \\ &= k2^{k+1} + 2^{k+1} \\ &= 2^{k+1}(k+1) \end{aligned}$$

and since $\lg 2^n = n$,

$$T(i+1) = 2^{k+1} \lg kg + 1.$$

By the principle of induction, we have proven the statement. □

4. We can express insertion sort as a recursive procedure as follows. In order to sort $A[1..n]$, we recursively sort $A[1..n-1]$ and then insert $A[n]$ into the sorted array $A[1..n-1]$. Write the recurrence for the running time of this recursive version of insertion sort.

```
insertion-sort(A, j)
    if j > 1
        insertion-sort(A, j - 1)
        key = A[j]
        i = j - 1
        while i > 0 and A[i] > key
            A[i + 1] = A[i]
        i = i + 1
    A[i + 1] = key
```

From analyzing the pseudocode from above, we can easily get the recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ T(n-1) + n & \text{otherwise.} \end{cases}$$

5. Referring back to the searching problem(see Exercise 2.1-3), observe that if the sequence A is sorted, we can check the midpoint of the sequence against v and eliminate half of the sequence from further consideration. The **binary search** algorithm repeats this procedure, halving the size of the remaining portion of the sequence each time. Write pseudocode, either iterative or recursive for binary search. Argue that the worst-case running time of binary search is $\Theta(\lg n)$.

We have the binary search algorithm:

```
binary-search(A, v, p, r)
    q = (p + r) / 2
    if r >= p
        if A[q] == v
            return q
        else if A[q] > v
            return binary-search(A, v, p, q)
        else
            return binary-search(A, v, q + 1, r)
    else
        return -1
```

We get the following recurrence:

$$T(n) = \begin{cases} 1 & \text{if } v = A[mid] \text{ or } v \notin A \\ T(\frac{n}{2}) + 1 & \text{if } v > A[mid] \text{ or } v < A[mid] \end{cases}$$

The worst case happens when the search wildcard v cannot be found in A. Binary search will divide through the sorted array recursively following the recurrence relation, where f is the **binary search** algorithm, v is the search wildcard, p is which index of A to begin searching, and r is which index of A do we end looking for v.

$$f(A, v, p, r) = \begin{cases} r & \text{if } A[mid] = v \\ f(A, v, p, mid) & \text{if } A[mid] > v \\ f(A, v, mid + 1, r) & \text{if } A[mid] < v \\ -1 & \text{if } p < r. \end{cases}$$

From this recurrence, we can conclude that when $p < r$, the algorithm will halt(this case is reachable through the other cases) and return -1 saying it has not found v. Since we halved the array of size n lgn times, it will do so until it satisfies the last case, which is our worst case scenario.

6. Observe that the **while** loop of lines 5-7 of the *insertion-sort* procedure in Section 2.1 uses a linear search to scan (backward) through the sorted subarray $A[1..j-1]$. Can we use a binary search (see Exercise 2.3-5) instead to improve the overall worst-case running time of insertion sort to $\Theta(n \lg n)$?

Yes. Since the subarray $A[1..j-1]$ is already sorted, we can use a binary search algorithm with the starting variables as $f(A, v, 1, j-1)$ where f is the binary search algorithm.

2 Growth of Functions

2.1 Exercises 3.1

1. Let $f(n)$ and $g(n)$ be asymptotically nonnegative functions. Using the based definition of Θ-notation, prove that $\max(f(n), g(n)) = \Theta(f(n) + g(n))$.

Suppose that $\max(f(n), g(n)) = \Theta(f(n) + g(n))$ then there exists positive constants c_1, c_2, and n_0 from the definition of the Θ-notation, such that for all $n \geq n_0$

$$0 \leq c_1(f(n) + g(n)) \leq \max(f(n), g(n)) \leq c_2(f(n) + g(n)).$$

We know that $\max(f(n), g(n)) \leq f(n) + g(n)$ so we can intuitively deduce that $c_2 = 1$.
Since $f(n) + g(n)$ and $\max(f(n), g(n))$ have the same most significant term, it is trivial to find a $0 < c_1 < 1$, such that $c_1(f(n) + g(n)) \leq \max(f(n), g(n))$ for all $n \geq n_0$.

2. Show that for any real constants a and b, where $b > 0$

$$(n + a)^b = \Theta(n^b).$$

3. Explain why the statement, "The running time of the algorithm A is at least $O(n^2)$," is meaningless.

O-notation is the upperbound, thus there will be no value of n for the function $f(n)$, where $n \geq n_0$, such that $f(n)$ will be greater than cn^2 where c is some constant. This is counter-intuitive to the definition of **upperbound**.

4. Is $2^{n+1} = O(2^n)$? Is $2^{2n} = O(2^n)$?

(a) $2^{n+1} = O(2^n)$.

Proof. If $2^{n+1} = O(2^n)$, then there exists a positive constant c and a nonnegative integer n_0, such that

$$0 \leq 2^{n+1} \leq c2^n, \text{ for all } n \geq n_0.$$

When $c = 2$, we have

$$0 \leq 2^{n+1} = 2 \cdot 2^n \quad = 2^{n+1}.$$

Since there exists a $c = 2$ and an $n_0 = 0$, then $2^{n+1} = O(2^n)$. □

(b) $2^{2n} \neq O(2^n)$.

Proof. We proceed by a proof by contradiction. Suppose that $2^{2n} = O(2^n)$, then for the purpose of contradiction, there exists a positive constant c and a nonnegative integer n_0, such that $2^{2n} = O(2^n)$ for all values of $n \geq n_0$.

$$2^{2n} = 2^n \cdot 2^n \leq c \cdot 2^n.$$

Dividing the inequality yields

$$2^n \leq c$$

which is a contradiction since c is a constant. Therefore, $2^{2n} \neq O(2^n)$. □

5. Prove Theorem 3.1.

Theorem (Theorem 3.1)**.** *For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.*

Proof. (a) Proving if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ then $f(n) = \Theta(g(n))$.

From the definition of Θ-notation, there must exist constants c_1, c_2, and a nonnegative integer n_0 such that $0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$, for all values $n \geq n_0$.

For the definition of the other asymptotic notations, we define $O(g(n))$ as there exists a positive constant c_O and $n_{\text{Big-O}}$ such that $0 \leq f(n) \leq c_O \cdot g(n)$ for all $n \geq n_{\text{Big-O}}$. And likewise, $\Omega(g(n))$ is defined as there exists a positive constant c_Ω and n_Ω such that $0 \leq c_\Omega \cdot g(n) \leq n_\Omega$, for all $n \geq n_\Omega$.

We construct $\Theta(g(n))$ from the definitions of O-notation and Ω-notation. We first set $c_1 = c_\Omega$ and $c_2 = c_O$. Then, $n_0 = \max(n_{\text{Big-O}}, n_\Omega)$.

We have proven the first part.

(b) Proving $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Suppose $f(n) = \Theta(g(n))$, then there exists constants c_1, c_2, and n_0 for all $n \geq n_0$, such that

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n).$$

Simply separating the inequalities, we get both $O(g(n))$ and $\Omega(g(n))$.

□

6. Prove that the running time of an algorithm is $\Theta(g(n))$ if and only if its worst-case running time is $O(g(n))$ and its best case running time is $\Omega(g(n))$.

7. Prove that $o(g(n)) \bigcap \omega(g(n))$ is the empty set.

Proof. We proceed with a proof by contradiction, From the limit definitions of the o-notation and ω-notation, we assume that $o(g(n)) \bigcap \omega(g(n)) \neq \emptyset$, hence

$$0 = \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

which is obviously a contradiction. Therefore, the intersection of $o(g(n))$ and $\omega(g(n))$ is the empty set. □

8. We can extend our notation to the case of two parameters n and m that can go to infinity independently at different rates. For a given function $g(n, m)$, we denote by $O(g(n, m))$ the set of functions.

(a) We define $\Theta(n, m)$ as the set

$$\Theta(n,m) = \left\{ f(n,m) \;\middle|\; \begin{array}{l} \text{There exists positive constants } c_1, c_2, n_0, m_0, \\ \text{such that } 0 \leq c_1 \cdot g(n,m) \leq f(n,m) \leq c_2 \cdot g(n,m) \\ \text{for } n \geq n_0 \text{ or } m \geq m_0\} \end{array} \right\}$$

(b) We define $\Omega(n,m)$ as the set

$$\Omega(n,m) = \left\{ f(n,m) \;\middle|\; \begin{array}{l} \text{There exists positive constants } c, n_0, m_0, \\ \text{such that } 0 \le c \cdot g(n,m) \le f(n,m) \\ \text{for } n \ge n_0 \text{ or } m \ge m_0 \} \end{array} \right\}$$

3 Heapsort

3.1 Exercises 6.1

1. What are the minimum and maximum numbers of elements in a heap of height h?

A binary heap is a nearly-complete binary tree. Therefore, the maximum number of elements a binary heap of height h has is when it is a **complete binary tree** where the number of nodes can be determined by

$$n = 2^{h+1} - 1.$$

Likewise, the minimum number of elements in a binary heap of height h is when there is only one element in the last depth. Therefore, the number of nodes in a minimum binary heap is

$$n = 2^h.$$

2. Show that an n-element heap has height $\lfloor \lg n \rfloor$.

Proof. We note that a complete binary tree has $2^{h+1} - 1$ elements where h is the height and that 2^h is the minimum number of elements for a given tree of that height. Therefore, we have

$$2^h \le n \le 2^{h+1} - 1 < 2^{h+1}$$

where 2^{h+1} is the minimum number of elements in the tree of height $h+1$. Thus,

$$h \le \lg n < h + 1.$$

□

3. Show that in any subtree of a max-heap, the root of the subtree contains the largest value occuring anywhere in that subtree.

Suppose that there is a subtree where the root is not the largest element, then it obviously violates the max-heap property that states that any parent element should be larger or equal to their children. This follows recursively.

4. Where in a max-heap might the smallest element reside, assuming that all elements are distinct

The smallest element will reside in the last level of the tree. If the smallest element is not a leaf node, then it violates the max-heap property since a parent should always be larger than their children(if all elements are distinct.)

5. Is an array that is in sorted order a min-heap?

An array that is in sorted order maintains the min-heap property because all parents are less than or equal to their children. Therefore, a sorted array(in an increasing order) is a min-heap.

6. Is the array with values $\langle 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 \rangle$ a max-heap?

No. The parent of 7 is 6 and this violates the max-heap property.

7. Show that, with the array representation for storing an n-element heap, the leaves are the nodes indexed by $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \ldots, n$.

4 Elementary Data Structures

4.1 Exercises 10.1

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

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.

For any n-sized array A, we can implement two stacks S_1 and S_2 where $S_1.top = 0$ which denotes that there are no elements and $S_2.top = n + 1$ which also denotes that there are no elements in S_2.
When pushing in S_1 should increment $S_1.top$ and pushing in S_2 should decrement $S_2.top$. Hence, both stacks should not `PUSH` further when $S_1.top = S_2.top$ since that denotes that both stacks are full.

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

4. Rewrite `ENQUEUE` and `DEQUEUE` to detect underflow and underflow of a queue.

For the `ENQUEUE` procedure, the queue will *overflow* when $Q.head = Q.tail + 1$ and we attempt to `ENQUEUE` in some queue Q.

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

Likewise, for the `DEQUEUE` procedure, the queue will *underflow* when we call `DEQUEUE` when $Q.tail = Q.head$ which denotes that the queue Q is empty.

```
DEQUEUE(Q)
    if Q.head == Q.tail
```

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

4.2 Exercises 10.2

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

   (a) Suppose we have a new node x, we can implement `INSERT` in a singly-linked list in $O(1)$ time by simply assigning $x.next$ to $L.head$ and assigning $L.head$ to x.

   (b) However, we can't implement `DELETE` in constant time in a singly-linked list because we don't know the previous node p where $p.next = x$. So to implement delete, we have to traverse the list and check if $p.next = x$ and if it is, assign $p.next$ to $x.next$ and that deletes x. The worst case scenario is $O(n)$ time.

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

```
PUSH(L, x)
    // Assuming LIST-INSERT inserts to
    // the head of the List L
    LIST-INSERT(L, x)

POP(L)
    x = L.head.value
    LIST-DELETE(L, L.head)
    return x
```

3. Implement a queue using a singly-linked list L. The operations `ENQUEUE` and `DEQUEUE` should still take $O(1)$ time.

```
ENQUEUE(L, x)
    // Assuming LIST-INSERT inserts to
    // the head of the List L
    LIST-INSERT(L, x)

DEQUEUE(L)
    x = L.tail.value
    LIST-DELETE(L, L.tail)
    return x
```

4.3 Exercises 10.4

1. Draw the binary tree rooted at index 6 that is represented by the following attributes

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

Consider the following procedure where T is an n-node binary tree and d is the current depth of the recursion which is initialized at 0. We call the procedure on a tree T by `PRINT-TREE(T, 0)`.

```
PRINT–TREE(T)
    print(T.key)
    if T.left != NIL
        PRINT–TREE(T.left)
    if T.right != NIL
        PRINT–TREE(T.right)
```

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.

Consider the following procedure where T is an n-node binary tree and S is a stack that has the `PUSH` and `POP` procedures defined. We let x be a pointer pointing to the current node in T. The variable T also doubles as a pointer to the root of T.

```
PRINT–TREE(T)
    x = T
    loop
        if x != NIL
            S.push(x)
            x = x.left
        else
            p = x.popped()
            print(p.key)
            x = x.right

        if S.empty() and x == NIL
            break loop
```

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–TREE(T)
    print(T.key)
    if not IS–LEAF(T)
        PRINT–TREE(T.child)
    if HAS–RIGHT–SIBLING(T)
        PRINT–TREE(T.sibling)
```

5 Binary Search Trees

5.1 Exercises 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.

2. What is the difference between the binary-search tree property and the min heap property? 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 min-heap property supposes that the root node is less than or equal to any of its child elements, whereas the binary search tree maintains that the left child is less than or equal to the parent while the right child is greater than or equal to the parent.
   Due to this, in the mean heap property, the next smallest element could either be the left child or the right child and there is no way of knowing that in linear time.

3. Give a nonrecursive algorithm that performs an inorder tree walk

   The algorithm for inorder tree walk is the same as my answer for `Exercise 10.4-3`.
   Since we visit each node, the algorithm runs at $\Theta(n)$ time.

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
```

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.

   For any n-sized arbitrary list, inserting an element x takes $\Omega(\lg n)$ time. Since we repeat that for all elements in the list, it then takes $\Omega(n \lg n)$. However, there does not exist a comparison-based sorting algorithm that runs at $o(n \lg n)$ time, therefore it is not possible to build a binary search tree from an unsorted n-element list in $\Omega(n \lg n)$ time.

5.2 Exercises 12.2

1. Suppose that we have the 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) $935, 278, 347, 621, 299, 392, 358, 363$.

We couldn't have come across c and e as they are impossible as they do not follow the binary search tree property.

2. Write the recursive versions of `Tree-Minimum` and `Tree-Maximum`.

```
Tree-Minimum(x)
    if x.left == NIL
        return x
    else
        return Tree-Minimum(x.left)

Tree-Maximum(x)
    if x.right == NIL
        return x
    else
        return Tree-Maximum(x.right)
```

3. Write the `Tree-Predecessor` procedure.

```
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
```

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 $a \leq b \leq c$. Give a smallest possible counterexample to the professor's claim.

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.

For a given node x, we have the predecessor and successor procedures, We go to the second case where the left and right child, respectively, are nonempty. If its successor has a left child, then by contradiction of the definition of the `Tree-Minimum` procedure, $successor(x)$ isn't the successor of x which is obviously a constradiction. Likewise, if the predecessor has a right child, then by contradiction, $predecessor(x)$ isn't the predecessor of x.

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.

Proof. Suppose for the purpose of contradiction that y is not an ancestor of x, then we have some node z that is a common ancestor of y and x. By the binary search tree property, it must be that $x < z < y$, so y cannot be the successor of x if y is not an ancestor.
To prove that y is the lowest ancestor of x whose left child is also an ancestor of x, for any ancestor of x, it must be that the right child of every ancestor not equal to y is also an ancestor of x. This implies that for all ancestors $a \in A$ of x, it must be $a < x$. But since y is an ancestor of x whose left child is is A, by the binary search property, $y > x$ and is the lowest ancestor whose left child is an ancestor of x. □

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` abd then making $n-1$ calls to the `Tree-Successor`. Prove that this algorithm runs on $\Theta(n)$ time.

Proof. Since the algorithm visits every node, then it runs at $\Omega(n)$ time. We now only have to prove that the algorithm runs at $T(n) = O(n)$ time.
For any given tree, height h is less than or equal to the number of nodes n. We only call `Tree-Minimum` once and it takes $O(h)$ time to get the minimum. Likewise, the `Tree-Successor` procedure also takes $O(h)$ time to finish and calls it n times. Thus, we have the recurrence

$$T(n) = O(h) + (n-1)O(h).$$

By the recurrence expansion method, we can get $T(n) = O(n)$. Thus, $T(n) = \Theta(n)$. □

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.

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 $y.key$ is either the smallest key in T larger than $x.key$ or be the largest key in T smaller than $x.key$.

Proof. We have two cases:

- If $y.key > x.key$, then x must be a left child of y. From the second case of the `Tree-Successor` procedure, $y.right \neq x$, and therefore, y is the successor of x and is the smallest key that is larger than $x.key$.
- If $y.key < x.key$, then x must be the right child of y. From the `Tree-Predecessor` procedure defined in *Exercise 12.2-3*, $y.left \neq x$, hence, y is the predecessor of x which makes it the largest key smaller than $x.key$.

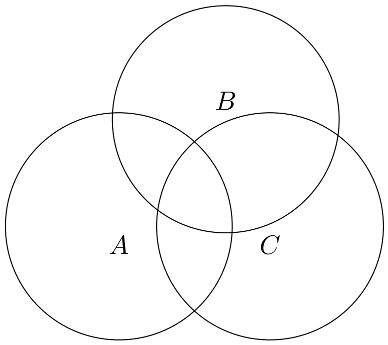
□

A Summations

B Sets, etc

B.1 Sets

1. Draw Venn diagrams that illustrate the first of the distributive laws.



2. Prove the generalization of DeMorgan's Laws to any finite collection of sets:

   - $\overline{A_1 \bigcap A_2 \bigcap \cdots \bigcap A_n} = \overline{A_1} \bigcup \overline{A_2} \bigcup \cdots \bigcup \overline{A_n}$
   - $\overline{A_1 \bigcap A_2 \bigcap \cdots \bigcap A_n} = \overline{A_1} \bigcup \overline{A_2} \bigcup \cdots \bigcup \overline{A_n}$

Proof. The proof proceeds by induction.
We first prove the first statement. $\overline{A_1 \bigcap A_2 \bigcap \cdots \bigcap A_n} = \overline{A_1} \bigcup \overline{A_2} \bigcup \cdots \bigcup \overline{A_n}$. For the base case, we have $n = 2$ which is the DeMorgan's law $\overline{A_1 \bigcap A_2} = \overline{A_1} \bigcup \overline{A_2}$.
Suppose the proposition is true for $n = k$ for some k, $\overline{A_1 \bigcap A_2 \bigcap \cdots \bigcap A_k} = \overline{A_1} \bigcup \overline{A_2} \bigcup \cdots \bigcup \overline{A_k}$ and let $B = \overline{A_1 \bigcap A_2 \bigcap \cdots \bigcap A_k}$. Then, by the inductive hypothesis, this must also be true for $n = k + 1$ for some k. Hence,

$$\overline{A_1 \bigcap A_2 \bigcap \cdots \bigcap A_k \bigcap A_{k+1}} = \overline{B \bigcap A_{k+1}}$$

By DeMorgan's Laws, we have

$$\overline{B \bigcap A_{k+1}} = \overline{B} \bigcup \overline{A_{k+1}}$$

and by definition

$$\overline{B} \bigcup \overline{A_{k+1}} = \overline{A_1} \bigcup \overline{A_2} \bigcup \cdots \bigcup \overline{A_k} \bigcup \overline{A_k + 1}.$$

This proves the first statement.
To prove the second statement, we use the same logic where the base case is also $n = 2$. Thus, we have proved the theorem.