**R-2.8** Illustrate the performance of the selection-sort algorithm on the following input sequence

(22, 15, 26, 44, 10, 3, 9, 13, 29, 25)

22 15 26 44 10 3 9 13 29 25

3 15 26 44 10 22 9 13 29 25

3 9 26 44 10 22 15 13 29 25

3 9 10 44 26 22 15 13 29 25

3 9 10 13 26 22 15 44 29 25

3 9 10 13 15 22 26 44 29 25

3 9 10 13 15 22 26 44 29 25

3 9 10 13 15 22 25 44 29 26

3 9 10 13 15 22 25 26 29 44

3 9 10 13 15 22 25 26 29 44

3 9 10 13 15 22 25 26 29 44

**R-2.9** insertion-sort

22 15 26 44 10 3 9 13 29 25

15 22 26 44 10 3 9 13 29 25

15 22 26 44 10 3 9 13 29 25

15 22 26 44 10 3 9 13 29 25

10 15 22 26 44 3 9 13 29 25

3 10 15 22 26 44 9 13 29 25

3 9 10 15 22 26 44 13 29 25

3 9 10 13 15 22 26 44 29 25

3 9 10 13 15 22 26 29 44 25

3 9 10 13 15 22 25 26 29 44

**R-2.10** Give an example of a worst-case sequence with n elements for insertion-sort runs in Ω(n2 ) time on such a sequence.

-*The worst-case sequence example is one in descending order such as 9 8 7 6 5 4 3 2 1 to be sorted to ascending order*

**R-2.13** Suppose a binary tree T is implemented using a vector S, as described in Section 2.3.4. If n items are stored in S in sorted order, starting with index 1, is the tree T a heap? Justify your answer.

-*Yes, it is a heap because: S[i] >= S[i/2] => key(v) >= key(parent(v)).*

**R-2-18** Draw an example of a heap whose keys are all the odd numbers from 1 to 49 (with no repeats), such that the insertion of an item with key 32 would cause upheap bubbling to proceed all the way up to a child of the root (replacing that child’s key with 32)

1,3,33; 5,7,35,37; 9,11,13,15,39,41,43,45

17,19,21,23,25,27,29,31,47,49

**C-2.32** Let T be a heap storing n keys. Give an efficient algorithm for reporting all the keys in T that are smaller than or equal to a given query key x (which is not necessarily in T). For example, given the heap on Figure 2.41 and query key x=7, the algorithm should report 4, 5, 6, 7. Note that the keys do not need to be reported in sorted order. Ideally, your algorithm should run in O(k) time, where k is the number of keys reported.

Algorithm **reportKey**(T, x)

Input: A heap T, and query value x

Output: A list of keys smaller than or equal to x

returnList <- new List

If ¬T.isEmpty()

reportKeyHelper(T, 1, x, returnList)

return returnList

Algorithm **reportKeyHelper**(T, i, x, returnList)

Input: A heap T, index of a node in the heap,

the query value x, and the return list to

contain reported values

if T[i] <= x returnList.add(T[i])

left <- i \* 2

if left < T.size()

leftList <- reportKeyHelper(T, left, x, returnList)

right <- i \* 2 + 1

if right < T.size()

rightList <- reportKeyHelper(T, right, x, returnList)

**Design an algorithm**, **isPermutation(A, B)** that takes two sequences A and B and determines whether or not they are permutations of each other, i.e., same elements but possibly occurring in a different order. Hint: A and B may contain duplicates. What is the worst-case time complexity of your algorithm? Justify your answer

Algorithm **isPermutation(A, B**)

Input: 2 sequences to check if they are permutations of each other

Output: true if they are permutations of each other, false otherwise

If A.size() ¬= B.size() return false

PQ <- new PriorityQueue with HeapSort implementation

sortedA <- PQ.sort(A)

sortedB <- PQ.sort(B)

elementsA <- sortedA.elements()

elementsB <- sortedB.elements()

while elementsA.hasNext()

elementA <- elementsA.nextObject()

elementB <- elementsB.nextObject()

if elementA ¬= elementB

return false return true

The worst-case complexity of this algorithm is O(n log n) because of the heap-sorting

**R-4.2** Give a pseudo-code description of the merge-sort algorithm. You can call the merge algorithm as a subroutine.

Algorithm **mergeSort(S, C)**

Input: Sequence S, comparator C

Output: Sequence S sorted according to C

If S.size() > 1 then

(S1, S2) <- partition(S, n/2)

mergeSort(S1, C)

mergeSort(S2, C)

S <- merge(S1, S2, C) return S

Algorithm **merge(A, B, C)**

Input: Sequences A and B with n/2 elements each, comparator C

Output: Sorted sequence of A v B

while ¬A.isEmpty() ^ ¬B.isEmpty() do

if C.isLessThan(B.first().element(), A.first().element()) then S.insertLast(B.remove(B.first()))

else S.insertLast(A.remove(A.first()))

while ¬!A.isEmpty() do S.insertLast(A.remove(A.first()))

while ¬B.isEmpty() do S.insertLast(B.remove(B.first()))

**R-4.5** Suppose we are given two n-element sorted sequences A and B that should not be viewed as sets (that is, A and B may contain duplicate entries). Give an O(n)-time pseudo-code algorithm for computing a sequence representing the set A ∪ B (with no duplicates).

Algorithm removeDuplicateAndUnion(A, B)

    Input:sequences A and B with n elements each

    Output:sorted sequence of A U B

    S <- empty sequence

    while !A.isEmpty() /\ !B.isEmpty() do

        if B.first().element() < A.first().element() then

            S.insertLast(B.remove(B.first()))

        else if B.first().element() > A.first().element() then

            S.insertLast(A.remove(A.first()))

        else

            S.insertLast(A.remove(A.first()))

            B.remove(B.first())

    return S

**R-4.9 Suppose** we modify the deterministic version of the quick-sort algorithm so that, instead of selecting the last element in an n-element sequence as the pivot, we choose the element at rank (index) ⎣n/2⎦, that is, an element in the middle of the sequence. What is the running time of this version of quick-sort on a sequence that is already sorted?

- because the pivot number is the middle and the sequence is sorted, it doesn’t take time for swapping, just takes time to compare the middle element with other n-1 elements, therefore the running time is O(n)

**C-4.10 Suppose** we are given an n-element sequence S such that each element in S represents a different vote in an election, where each vote is given as an integer representing the ID of the chosen candidate. Without making any assumptions about who is running or even how many candidates there are, design an O(n log n)-time algorithm to see who wins the election S represents, assuming the candidate with the most votes wins.

**Algorithm findElectionWinner(S)**

    Input: n-element sequence S where each element represents a different vote

    Output: ID of winning candidate

    mergeSort(S, C)

    winCandidateId <- S.first()

    maxVote <- 0

    prevId <- S.first()

    noOfVote <- 0

    while !S.isEmpty() do

        curId <- S.remove(S.first())

        if curId != prevId then

            if maxVote < noOfVote then

                   maxVote <- noOfVote

                   winCandidateId <- curId

    noOfVote <- 0

        else

                prevId <- curId

                noOfVote <- noOfVote + 1

     return winCandidateId

***Let L be a List of objects colored either red, green, or blue.*** Design an in-place algorithm **sortRBG(L)** that places all red objects in list L before the blue colored objects, and all the blue objects before the green objects. Thus the resulting List will have all the red objects followed by the blue objects, followed by the green objects. Hint: use the method swapElements to move the elements around in the List. To receive full credit, you must use positions for traversal, e.g., first, last, after, before, swapElements, etc. which is necessary to make it in-place.

**Algorithm sortRBG(L)**

size <- L.size()

curPos <- 0

curColor <- red

for i<-0 to size - 1 do

if L.elemAtRank(i).element() <> curColor then

if i = (size – 1) then

if curColor <> Green then

if curColor = red then

curColor = blue

else if curColor = blue then

curColor = green

else break i <- curPos

else L.swapElements(curPos, i)

curPos <- curPos + 1

else curPos <- curPos + 1

**R-3.8 Is** the multi way search tree of Figure 3.17a a (2,4) tree? Justify your answer.

-No, because the external nodes are in different depth R-3.10 A certain Professor Amongus claims that a (2,4) tree storing a set of items will always have the same structure, regardless of the order in which the items are inserted. Show that Professor Amongus is wrong. Inputs: 1 2 3 4 5 🡪 2; 1; 3,4,5

Inputs: 5 4 3 2 1 🡪 4; 2,3; 5

**C-4.11 Suppose** we are given an n-element sequence S such that each element in S represents a different vote in an election, where each vote is given as an integer representing the ID of the chosen candidate. Suppose we know who the candidates are and the number of candidates running is k < n. Describe an O(n log k)-time algorithm for determining who wins the election.

**Algorithm findWinner(S, C)**

B <- new Dictionary(BST)

cnt <- 0

for each id in C do

B.insertItem(id, cnt)

maxVote <- 0

winnerID <- 0

v <- 0

for i<-0 to S.size()-1 do

v <- S.elementAtRank(i) //return candidate ID at the sequence i

p <- B.findElement(v)

if p <> NO\_SUCH\_KEY then

cnt <- B.elem(p) + 1

B.insertElement(B.key(p), cnt)

if cnt > max then

max <- cnt

winnerID <- B.key(p)

return winnerID

**C-4-22 Let A and B** be two sequences of n integers each. Given an integer x, describe an O(n log n)-time algorithm for determining if there is an integer a in A and an integer b in B such that x = a + b.

**Algorithm checksum(A, B, x)**

Input: Sequence A and B, and sum x

Output: Whether A and B contain integers whose sum is equal to x

Dictionary <- Dictionary with AVL tree implementation iterator <- A.elements()

while iterator.hasNext() do

a <- iterator.nextObject()

dictionary. insertItem(a, a)

iterator <- B.elements()

while iterator.hasNext() do

b <- iterator.nextObject()

key <- x – b

a <- dictionary.findElement(key)

if a ¬= NO\_SUCH\_KEY

return true return false

**Algorithm push(e)**

  Input: Element e

  queue1.enqueue(e)

**Algorithm pop()**

  Output: Element at the top of the stack

  if queue1.isEmpty() and queue2.isEmpty() then

    throw EmptyStackException else

    for i <- 1 to queue1.size() - 1 do

      queue2.enqueue(queue1.dequeue())

    p <- queue1.dequeue()

    tmp <- queue1

    queue1 <- queue2

    queue2 <- tmp return p

    The running time of push method is O(1)

    The running time of pop method is O(n)

**R-2.1** Describe, using pseudo-code, implementations of the methods insertBefore(p, e), insertFirst(e), and insertLast(e) of the List ADT, assuming the list is implemented using a doubly-linked list.

**Algorithm insertBefore(p, e)**

Input: Position p, new element e

Output: New Position q

q <- new Position; q.element <- e

q.prev <- p.prev; q.next <- p

p.prev.next <- q; p.prev <- q; return q

**Algorithm insertFirst(e**)

Input: New element e

Output: New Position q

q <- new Position; q.element <- e

q.next <- header.next; q.prev <- header header.next.prev <- q; header.next <- q return q **Algorithm insertLast(e)**

Input: New element e

Output: New Position q;

q <- new Position; q.element <- e;

q.next <- trailer; q.prev <- trailer.prev;

trailer.prev.next <- q; trailer.prev <- q return q

**C-2.1 Describe**, in pseudo-code, a link-hopping method for finding the middle node of a doubly linked list with header and trailer sentinels, and an odd number of real nodes between them. (Note: This method can only use link-hopping; it cannot use a counter.) What is the running time of this method? **Algorithm findMiddleNode(L)**

Input: A doubly linked list L with an odd number of real nodes between header and trailer

Output: The middle node

p <- L.header; q <- L.trailer

while p ⌜= q do

p = first.next; q = last.prev

return p The running time of this algorithm is O(n/2)

**C-2.2** Describe, in pseudo-code, how to implement the queue ADT using two stacks. What is the running time of the enqueue() and dequeue() methods in this case?

**Algorithm enqueue(e)**

Input: Element e stack1.push(e)

**Algorithm dequeue()**

Output: Element at the front of the queue

if stack1.isEmpty() and stack2.isEmpty() then

throw EmptyQueueException

else if stack2.isEmpty() then

while ⌜stack1.isEmpty() do stack2.push(stack1.pop())

p <- stack2.pop() return p

The running time for enqueue algorithm is O(1)

The running time for dequeue algorithm is O(n)

**C-2.3** Describe how to implement the stack ADT using two queues. What is the running time of the push() and pop() methods in this case?

**Algorithm push(e)**

Input: Element e queue1.enqueue(e)

**Algorithm pop()**

Output: Element at the top of the stack

if queue1.isEmpty() and queue2.isEmpty() then

throw EmptyStackException

else for i <- 1 to queue1.size() - 1 do queue2.enqueue(queue1.dequeue())

p <- queue1.dequeue() tmp <- queue1

queue1 <- queue2 queue2 <- tmp return p

The running time of push method is O(1)

The running time of pop method is O(n)

**C-2-4** Describe a recursive algorithm for enumerating all permutations of the numbers {1, 2,…, n}. What is the running time of your method?

**Algorithm swap(s, i, j)**

Input: Sequence s, indexes i and j to swap data in s Output: Sequence with data swapped

temp <- s[i] s[i] <- s[j] s[j] <- temp

**Algorithm permute(s, i)**

Input: Sequence s, and index i of a position in s whose value is 0 initially

Output: A sequence of permutations of s permutations <- new Sequence

if i < s.size() – 1 then

for j from i to s.size() do

if j ⌜= I then swap(s, i, j)

result <- permute(s, i + 1)

for all permutation in result do permutations.insertLast(permutation)

if j ⌜= I then swap(s, i, j)

else permutations.insertLast(s)

return permutations

**C-2-5** Describe the structure and pseudo-code for an array-based implementation of the vector ADT that achieves O(1) time for insertions and removals at rank 0, as well as insertions and removals at the end of the vector. Your implementation should also provide for a constant-time elemAtRank method. **Algorithm elemAtRank(r)**

Input: Rank r Output: Element at rank r

return V[(f + r) mod N]

**Algorithm insertAtRank(r, o)**

Input: Rank r, and object o to be inserted at rank r

If size() = N – 1 then throw FullVectorException

start <- (f + r) mod N

if start = f do // insert at rank 0

f <- (f – 1) mod N V[f] <- o

else if start = l then // insert at the end

V[l] <- o l <- (l + 1) mod N

else then // insert somewhere between f and l

i <- l while i ⌜= start do

V[i] <- V[I - 1]

i <- i – 1

if i < 0 then i <- i + N

V[start] <- o

l <- (l + 1) mod N

**Algorithm removeAtRank(r)**

Input: Rank r Output: Removed element at rank r

If isEmpty() then throw EmptyVectorException

start <- (f + r) mod N o <- V[start]

if start = f then // remove at rank 0

f <- (f + 1) mod N

else if start = l – 1 then // remove at the end

l <- (l – 1) mod N

else then // remove somewhere between f and l

i <- start

end <- l – 1

if end < 0 then end <- end + N

while i ⌜= end do next <- (i +1) mod N

V[i] <- V[next]

i <- next return o

**R-1.6** 1/n < loglogn < n1/2 < nlogn < 5n < 4nn1/2 < 4logn < 2nlog2n < n2logn < n3 < 2n < 4n

**C-4.13** Suppose we are given two sequences A and B of n elements, possibly containing duplicates, on which a total order relation is defined.  Describe an efficient algorithm for determining if A and B contain the same set of elements (possibly in different orders).  What is the running time of this method?

**Agorithm isSameElements(A, B)**

if A.size() <> B.size() then return false

\_inPlaceQuickSort(A, 0, A.size())

\_inPlaceQuickSort(B, 0, B.size())

for i<-0 to A.size() do

p <- A.elementAtRank(i)

q <- B.elementAtRank(i)

if p.element() <> q.element() then

return false return true

\_**inPlaceQuickSort(S, l, h)**

if l < h then

k <- \_inPlacePartition(S, l, h)

\_inPlaceQuickSort(S, l, k-1)

\_inPlaceQuickSort(S, k+1, h)

**\_inPlacePartition(S, l, h)**

p <- randomInt(l, h)

S.swapElement(S.atRank(l), S.atRank(p))

j <- l+1

k <- h

while j < k do

while k > j /\ S.elementAtRank(k) >= S.elementAtRank(l) do

k <- k - 1

while j < k /\ S.elementAtRank(j) <= S.elementAtRank(l) do

j <- j + 1

if j < k then

S.swapElement(S.elementAtRank(j), S.elementAtRank(k))

S.swapElement(S.elementAtRank(k), S.elementAtRank(l) return k

**C-4-22** Let A and B be two sequences of n integers each.  Given an integer x, describe an O(n log n)-time pseudo code algorithm for determining if there is an integer a in A and an integer b in B such that x = a+b.

**Algorithm findPair(A, B, x) nlogn**

Input: n-element sequence A and B include n integers

Ouput: true if existing a pair a & b so that a + b = x

B <- new Dictionary(HT)

for each n in B do

B.insertItem(n, n)

for each a in A.elements() do

b <- B.findElement(x - a)

if b <> NO\_SUCH\_KEY then return true} return false

**C-4.16** Given a sequence S of n comparable elements, describe an efficient method for determining whether there are two equal elements in S. What is the running time of your method?

**Algorithm isExistTwoEqualElement(S)**

    Input: Sequence S with n elements

    Output: true or false if two equal elements exis in the sequence, otherwise false

    D <- new Dictionary(HashTable)

    for each x of S do

        cnt <-D.findElement(x)

        if cnt != NO\_SUCH\_KEY then

            D.insertItem(x,0)

        else

            return true return false

**C-4-19** Let S be a sequence of n elements on which a total order relation is defined. An inversion in S is a pair of elements x and y such that x appears before y in S but x > y. Describe an algorithm running in O(n log n) time for determining the number of inversions in S.  Hint: try to modify the merge-sort algorithm to solve this problem.

**Algorithm countInversion(S, C)**

    Input : sequence S with total order n elements, comparator C

    Output: number of Inversion

    if S.size() > 1 then

        (S1, S2)<-partition(S, n/2)

        countInversion(S1, C)

        countInversion(S2, C)

        (S, cnt) <- \_changeMerge(S1, S2, C)

        count <- count + cnt

    return count

\_ changeMerge(A, B, C)

    Input: sequences A and B with n/2 elements each, comparator C

    Output: count of number of inversion

count<-0

    S <- empty sequence

    while !A.isEmpty() ^ !B.isEmpty() do

        if C.isLessThan( B.first().element(), A.first().element() ) then

            S.insertLast(B.remove(B.first()))

            count <- count + 1

        else

            S.insertLast(A.remove(A.first()))

        while !A.isEmpty() do

            S.insertLast(A.remove(A.first()))

        while !B.isEmpty() do

            S.insertLast(B.remove(B.first()))

    return S, count

**C-4.25** Bob has a set A of n nuts and a set B of n bolts, such that each nut in A has a unique matching bolt in B. Unfortunately, the nuts in A all look the same, and the bolts in B all look the same as well. The only kind of comparison that Bob can make is to take a nut-bolt pair (a,b), such that a is from A and b is from B, and test it to see if the threads are larger, smaller or a perfect match with the threads of b.  Describe an efficient  algorithm for Bob to match up all of his nuts and bolts. What is the running time of this algorithm, in terms of nut-bolt tests that Bob must make?

**Algorithm nutsBoltsMatchup(A, B)**

    Input : Sequence A of nuts, sequence B of bolts

    Output : Matched set of nuts and bolts

    T <- insertIntoRedBlackTree(B)

    PQ <- new Priority Queue Array

    for each x of A do

        PQ.insert(x, PQ.remove(x))

    return PQ