No supplementary material — No aids — Scantron — 3 hours
This question sheet has questions on pages 1–14. Please answer all questions on the provided Scantron sheet. Select only a single answer for each question. In case multiple answers are correct, select a single answer that best, or most precisely, answers the question.

1. The running time of the methods `get(i)` and `remove(i)` for an `ArrayList` are
   (a) $O(1)$ and $O(1)$, respectively
   (b) $O(1+\texttt{i})$ and $O(1+\texttt{i})$, respectively
   (c) $O(1)$ and $O(1+\texttt{i})$, respectively
   (d) $O(1+\texttt{i})$ and $O(1+\texttt{size()}-\texttt{i})$, respectively
   (e) $O(1)$ and $O(1+\texttt{size()}-\texttt{i})$, respectively

2. The running time of the methods `get(i)` and `remove(i)` for a `LinkedList`, as implemented in the Java Collections Framework, are
   (a) $O(1+\texttt{i})$ and $O(1+\texttt{i})$, respectively
   (b) $O(1)$ and $O(1+\texttt{size()}-\texttt{i})$, respectively
   (c) $O(1+\texttt{size()}-\texttt{i})$ and $O(1)$, respectively
   (d) $O(1+\min\{\texttt{i},\texttt{size()}-\texttt{i}\})$ and $O(1+\min\{\texttt{i},\texttt{size()}-\texttt{i}\})$, respectively
   (e) $O(1)$ and $O(1+\texttt{size()}-\texttt{i})$, respectively

3.
```
public static void insertAtFront(List<Integer> l, int n) {
  for (int i = 0; i < n; i++) {
    l.add(0, i);
  }
}
```
   The above method is
   (a) much faster when `l` is an `ArrayList`
   (b) much faster when `l` is a `LinkedList`
   (c) about the same speed independent of whether `l` is an `ArrayList` or a `LinkedList`

4. Recall that an `ArrayStack` stores `n` elements in a backing array `a` at locations `a[0]`,...,`a[n-1]`:
```
public class ArrayStack<T> extends AbstractList<T> {
  T[] a;
  int n;
  ...
}
```
   Also recall that, immediately after the backing array `a` is resized by `grow()` or `shrink` it has `a.length` = 2n.

   When adding an element, the `ArrayStack` grows the backing array `a` if it is full, i.e. if `a.length` = `n`.

   If are currently about to grow the backing array `a`, what can you say about the number of `add()` and `remove()` operations (as a function of the current value of `n`) since the last time the `ArrayStack` was resized?
   (a) At least n/2 `add()` operations have occurred since then
   (b) At least 2n/3 `add()` operations have occurred since then

(c) At least n/2 `remove()` operations have occurred since then

(d) At least 2n/3 `remove()` operations have occurred since then

(e) We can not bound either the number of `add()` nor `remove()` operations

5. Recall that we shrink the backing array a when $3n < $ `a.length`. If we are currently about to shrink the backing array a, what can you say about the number of `add()` and `remove()` operations since the last time the `ArrayStack` was resized?

   (a) At least n/2 `add()` operations have occurred since then

   (b) At least 2n/3 `add()` operations have occurred since then

   (c) At least n/2 `remove()` operations have occurred since then

   (d) At least 2n/3 `remove()` operations have occurred since then

   (e) We can not bound either the number of `add()` nor `remove()` operations

6. Recall that an `ArrayDeque` stores n elements at locations `a[j]`, `a[(j+1)%a.length]`,..., `a[(j+n-1)%a.length]`:

```
public class ArrayDeque<T> extends AbstractList<T> {
  T[] a;
  int j;
  int n;
  ...
}
```

   What is the amortized running time of the `add(i,x)` and `remove(i)` operations?

   (a) $O(1 + \mathtt{i})$

   (b) $O(1 + |\mathtt{i} - \mathtt{n}/2|)$

   (c) $O(1 + \mathtt{n} - \mathtt{i})$

   (d) $O(1 + \min\{\mathtt{i}, \mathtt{n} - \mathtt{i}\})$

   (e) $O(1 + \min\{\mathtt{i} - \mathtt{n}, \mathtt{n} - \mathtt{i}\})$

7. Recall that a `DualArrayDeque` implements the `List` interface using two `ArrayStacks`:

```
public class DualArrayDeque<T> extends AbstractList<T> {
  ArrayStack<T> front;
  ArrayStack<T> back;
  ...
}
```

   In order to implement `get(i)` we need to get it from the `ArrayStack`, `front` or `back`. We can express this as

   (a) `front.get(i)`

   (b) `front.get(front.size()-i-1)`

   (c) `back.get(i-front.size())`

   (d) Either (b) or (c) depending on the value of `i` and `front.size()`

   (e) Either (a) or (c) depending on the value of `i` and `front.size()`

8. If a `RootishArrayStack` has 10 blocks (so `b.size()` = 10), then how many elements can it store?

(a) 90
(b) 110
(c) 45
(d) 55
(e) none of the above

9. In a `RootishArrayStack`, a call to `get(13)` will return

   (a) `blocks.get(0)[13]`
   (b) `blocks.get(13)[0]`
   (c) `blocks.get(4)[3]`
   (d) `blocks.get(3)[4]`
   (e) `blocks.get(5)[4]`

10. Recall the following implementation of a singly-linked list (`SSLList`)

```
protected class Node {
  T x;
  Node next;
}
public class SLList<T> extends AbstractList<T> {
  Node head;
  Node tail;
  int n;
  ...
}
```

Consider how to implement a `Queue` as an `SLList`. When we enqueue (`add(x)`) an element, where does it go? When we dequeue (`remove()`) an element, where does it come from?

   (a) We enqueue (`add(x)`) at the `head` and we dequeue (`remove()`) at the `tail`
   (b) We enqueue (`add(x)`) at the `tail` and we dequeue (`remove()`) at the `head`
   (c) We enqueue (`add(x)`) at the `head` and we dequeue (`remove()`) at the `head`
   (d) We enqueue (`add(x)`) at the `tail` and we dequeue (`remove()`) at the `tail`
   (e) None of the above

11. Consider how to implement a `Stack` as an `SLList`. When we push an element where does it go? When we pop an element where does it come from?

   (a) We push at the `head` and we pop at the `tail`
   (b) We push at the `tail` and we pop at the `head`
   (c) We push at the `head` and we pop at the `head`
   (d) We push at the `tail` and we pop at the `tail`
   (e) None of the above

12. Using the best method you can think of, how quickly can we find the ith node in an SLList?

   (a) in $O(1 + \mathtt{i})$ time
   (b) in $O(1 + \mathtt{n} - \mathtt{i})$ time

(c) in $O(1 + \mathtt{n} - \mathtt{i})$ time

(d) in $O(1 + \min\{\mathtt{i}, \mathtt{n} - \mathtt{i}\})$ time

(e) in $O(1 + \min\{\mathtt{i}, \mathtt{n} \cdot (\mathtt{n} - \mathtt{i} - 1)\})$ time

13. Recall the multiplicative hash function `hash(x) = (x.hashCode() * z) >>> w-d`, where `w` is the number of bits in an integer. How large is the table that is used with this hash function? (In other words, what is the *range* of this hash function?)

    (a) $\{0, \ldots, 2^{\mathtt{d}}\}$

    (b) $\{0, \ldots, 2^{\mathtt{d}} - 1\}$

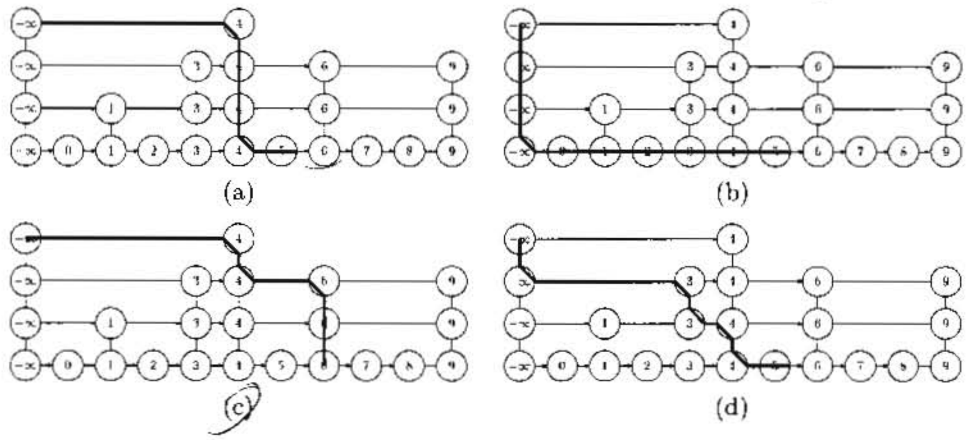
    (c) $\{0, \ldots, 2^{\mathtt{w}-\mathtt{d}}\}$

    (d) $\{0, \ldots, 2^{\mathtt{w}-\mathtt{d}} - 1\}$

    (e) $\{0, \ldots, 2^{\mathtt{w}} - 1\}$

Recall that a skiplist stores elements in a sequence of smaller and smaller lists $L_0, \ldots, L_k$. L_i is obtained from L_{i-1} by tossing a coin for each element in L_{i-1} and including the element in L_i if that coin comes up heads.

14. Which of the following pictures illustrates the search path for 6 in the skiplist?



15. Tossing a coin and counting how many times it comes up heads before the first tail is closely related to which of the following quantities in a skiplist?

    (a) The total size of the skiplist

    (b) The number of steps the search path takes at a particular level

    (c) The number of lists a particular element x takes part in

    (d) The total length of the search path

    (e) Both (b) and (c)

16. If the list L_0 contains n values, what is the expected number of elements in the list L_i?

    (a) $2i$

    (b) $i/2$

    (c) 2^i

    (d) $n/2^i$

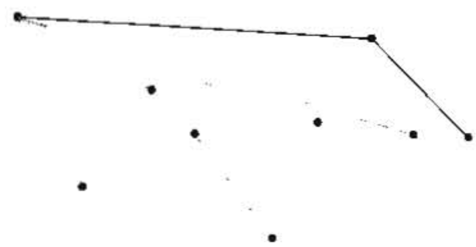
    (e) n^2

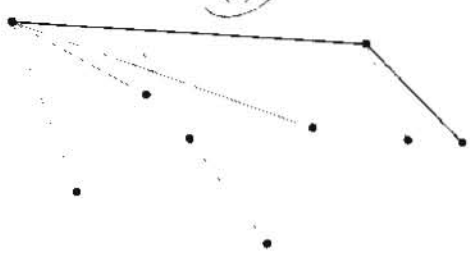
17. The expected length of a search path in a skiplist is at most $2\log n + 2$. This means the expected time to search in a skiplist is

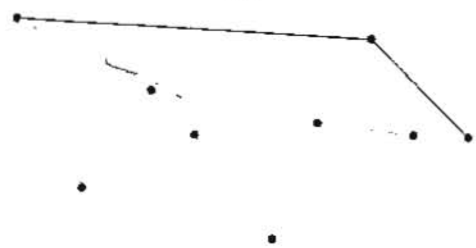
(a) $O(1)$
(b) $O(\log n)$
(c) $O((\log n)^2)$
(d) $O(n)$
(e) $O(2^n)$

18. Which of the following pictures best illustrates a trace of the Graham's Scan Algorithm for computing the upper hull?


(a)

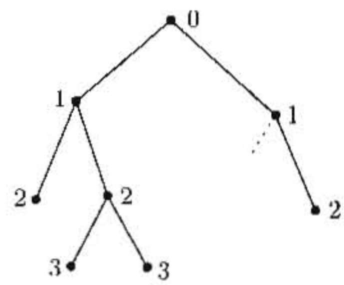

(b)


(c)


(d)

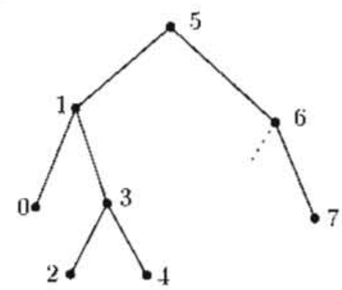
19. The following picture illustrates a numbering of the nodes of a binary tree

(a) By subtree size
(b) By the order nodes are processed during a preorder traversal
(c) By the order nodes are processed during a postorder traversal
(d) By the order nodes are processed during an inorder traversal
(e) By depth



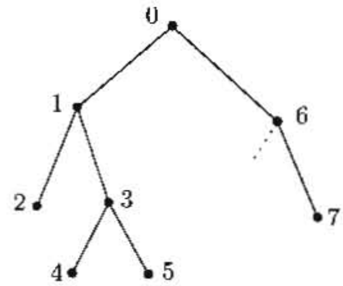
20. The following picture illustrates a numbering of the nodes of a binary tree

(a) By subtree size
(b) By the order nodes are processed during a preorder traversal
(c) By the order nodes are processed during a postorder traversal
(d) By the order nodes are processed during an inorder traversal
(e) By depth



21. The following picture illustrates a numbering of the nodes of a binary tree

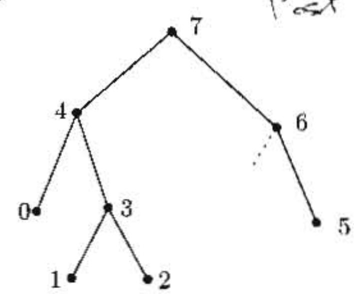
Pre order



(a) By subtree size
(b) By the order nodes are processed during a preorder traversal
(c) By the order nodes are processed during a postorder traversal
(d) By the order nodes are processed during an inorder traversal
(e) By depth

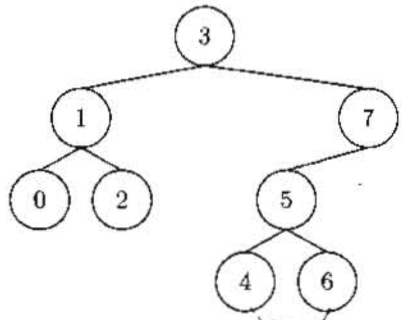
22. The following picture illustrates a numbering of the nodes of a binary tree

Post



(a) By subtree size
(b) By the order nodes are processed during a preorder traversal
(c) By the order nodes are processed during a postorder traversal
(d) By the order nodes are processed during an inorder traversal
(e) By depth

The next few questions are all asking about this binary search ~~tree~~:



23. The above binary search tree can be obtained by inserting the following sequence in order:

(a) $\langle 3, 7, 5, 2, 1, 4, 0, 6 \rangle$
(b) $\langle 3, 4, 5, 1, 2, 7, 0, 6 \rangle$
(c) $\langle 3, 7, 5, 1, 2, 4, 0, 6 \rangle$
(d) $\langle 3, 4, 5, 1, 2, 7, 0, 6 \rangle$
(e) $\langle 3, 7, 5, 0, 2, 4, 1, 6 \rangle$

24. In order to delete the root node (3), the standard deletion algorithm for binary search trees would

(a) Remove 3 and then merge the subtrees 1 and 7
(b) Delete 4 and store 4 at the root
(c) Delete 7 and store 4 at the root
(d) Delete 2 and store 4 at the root

(e) Delete 0 and store 7 at the root

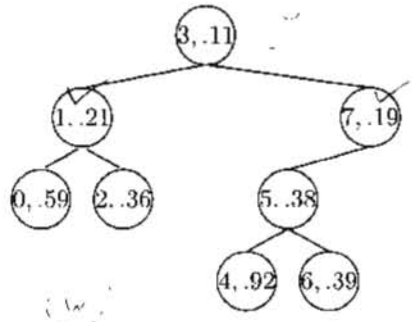
25. If we insert the values 4.5 and 5.5 into this tree, the newly created nodes would become

    (a) The right child of 4 and the right child of 4.5, respectively
    (b) The right child of 4 and the left child of 6, respectively
    (c) The right child of 4 and the left child of 6, respectively
    (d) The left child of 6 and the right child of 6, respectively
    (e) The left child of 6 and the left child of 5.5, respectively

26. Suppose the above tree represents a quicksort recursion tree. Then, this means that recursive invocations of quicksort have been called to sort the sets

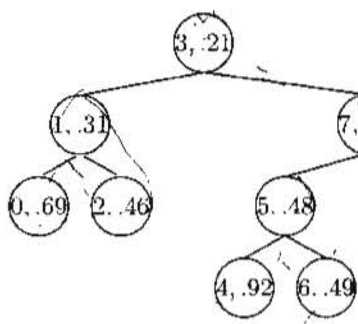
    (a) $\{3\}, \{1, 7\}, \{0, 2, 5\}, \{4, 6\}$
    (b) $\{0, \ldots, 7\}, \{0, \ldots, 2\}, \{4, \ldots, 7\}, \{4, \ldots, 6\}$
    (c) $\{0, \ldots, 7\}, \{0, \ldots, 3\}, \{4, \ldots, 7\}, \{4, 5\}$
    (d) $\{0, 1, 2\}, \{3\}, \{7\}, \{5, 4, 6\}$
    (e) $\{3\}, \{1\}, \{7\}, \{5\}$

27. The following pictures shows a binary search tree where each node is also assigned a priority. Does this picture show a valid treap (i.e., that satisfies both the binary search tree and heap properties)?



    (a) Yes
    (b) No
    (c) Not enough information to decide

The next two questions refer to the following treap:



28. If we insert the value 5.5 with the priority .35, then

    (a) 5.5 will become a right child of 3
    (b) 5.5 will become a left child of 7
    (c) 5.5 will become a right child of 5
    (d) 5.5 will become a left child of 6

(e) None of the above

29. If we remove the value 3

(a) 1 will become the root

(b) 7 will become the root

(c) 4 will become the root

(d) 2 will become the root

(e) None of the above

30. When we analyze the cost of deletion in a treap of size n, we can relate this cost to

(a) The cost of insertion in a treap of size n

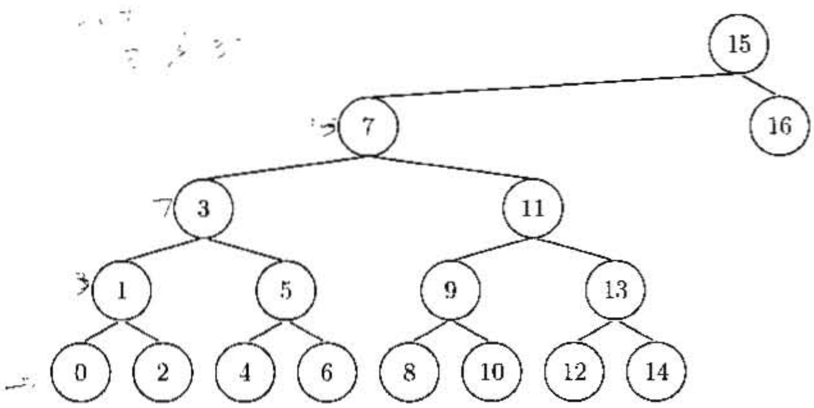
(b) The cost of insertion in a treap of size $n-1$

(c) The depth of a node in heap

(d) The depth of a node in a random binary search tree

(e) The depth of a node in a random 2red-4black tree-heap with sprinkles

31. Recall that the definition of a scapegoat node is a node `u.parent` such that `size(u)` > (2/3)`size(u.parent)`. Is the following tree a valid scapegoat tree ($n = 17, q = 17$)?



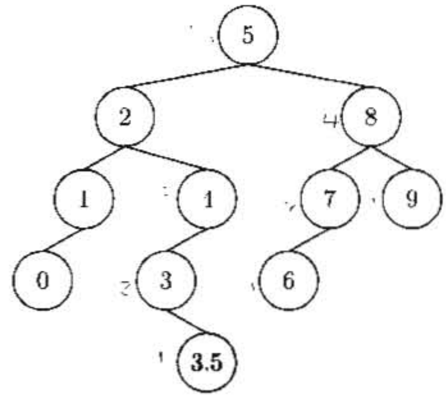
(a) Yes

(b) No

(c) Not enough information to decide

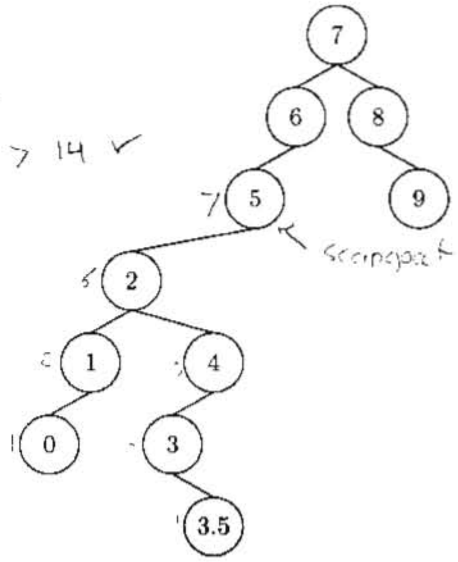
32. In the following scapegoat tree, we have just inserted the value 3.5

(a) 3 is a scapegoat
(b) 4 is a scapegoat
(c) 5 is a scapegoat
(d) 2 is a scapegoat
(e) None of the above

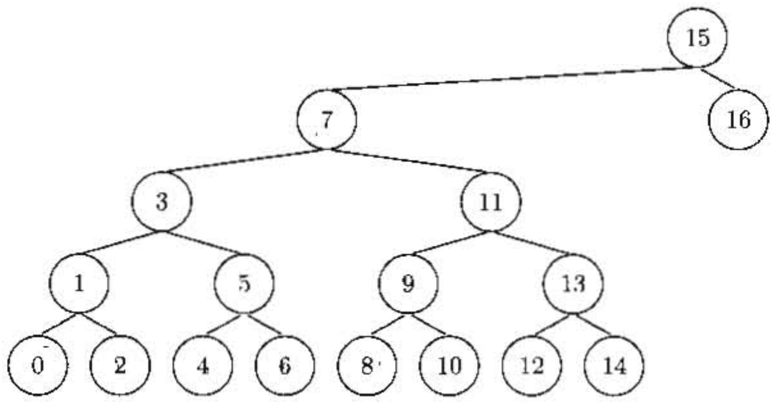


33. In the following scapegoat tree, we have just inserted the value 3.5

(a) 3 is a scapegoat
(b) 4 is a scapegoat
(c) 5 is a scapegoat
(d) 2 is a scapegoat
(e) None of the above



34. Suppose that the following tree is a scapegoat tree.



How many insertions/deletions have been performed since the last time the root node was rebuilt?

(a) at least 1000

(b) at least 14 m log n

(c) at least 16

(d) at most 14

(e) at most 1000

35. Consider a complete binary heap stored in an array using the Eytzinger Method. The formula (in Java) for the parent of a node stored at location `i` in the array is

(a) `2*(i+1)-1`

(b) `2*(i+1)`

(c) `i/2`

(d) `(i-1)/2`

(e) `(i+1)/2`

36. The formula (in Java) for the left child of a node stored at location `i` in the array is

(a) `2*(i+1)-1`

(b) `2*(i+1)`

(c) `2*i`

(d) `(i-1)/2`

(e) `(i+1)/2`

37. The formula (in Java) for the right child of a node stored at location `i` in the array is

(a) `2*(i+1)-1`

(b) `2*(i+1)`

(c) `2*i`

(d) `(i-1)/2`

(e) `(i+1)/2`

38. When implementing a complete binary heap using the Eytzinger method. the DeleteMin operation replaces the root with the value

(a) `a[0]`

(b) `a[1]`

(c) `a[a.length-1]`

(d) `a[n-1]`

(e) None of the above

39. This picture represents a binary heap represented using the Eytzinger Method:

0.1	0.5	0.3	0.9	1.1	0.8	0.4	2.1	1.3	9.2	5.1	1.9			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

If we insert the priority 0.7, it will get stored at index

(a) 0

(b) 2

(c) 3

(d) 8

(e) 11

40. This picture represents a binary heap represented using the Eytzinger Method:

0.1	0.5	0.3	0.9	1.1	0.8	0.4	2.1	1.3	9.2	5.1	1.9			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

If we call deleteMin(), the value 1.9 will get stored at index

(a) 0

(b) 2

(c) 6

(d) 8

(e) 10

41. The HeapSort algorithm is sometimes preferable to MergeSort because

(a) it builds a heap and then extracts the elements one at a time

(b) it works entirely in-place and doesn't need to allocate extra arrays

(c) it runs in $O(n \log n)$ worst-case time

(d) if n is a power of 2 then it does no more than $n \log_2 n$ comparisons

42. The MergeSort algorithm is sometimes preferable to HeapSort because

(a) it builds a heap and then extracts the elements one at a time

(b) it works entirely in-place and doesn't need to allocate extra arrays

(c) it runs in $O(n \log n)$ worst-case time

(d) if n is a power of 2 then it does no more than $n \log_2 n$ comparisons

43. The Quicksort algorithm is sometimes preferable to MergeSort because

(a) it builds a heap and then extracts the elements one at a time
(b) it works entirely in-place and doesn't need to allocate extra arrays
(c) it runs in $O(n \log n)$ worst-case time
(d) if n is a power of 2 then it does no more than $n \log_2 n$ comparisons

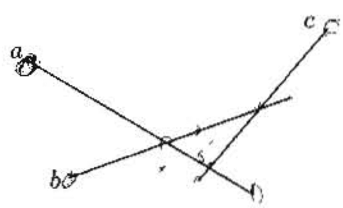
44. The HeapSort algorithm is sometimes preferable to Quicksort because

(a) it builds a heap and then extracts the elements one at a time
(b) it works entirely in-place and doesn't need to allocate extra arrays
(c) it runs in $O(n \log n)$ worst-case time
(d) if n is a power of 2 then it does no more than $n \log_2 n$ comparisons

45. The HeapSort algorithm does at most $2n \log n + 5n$ comparisons. This means that the number of comparisons done by HeapSort is in

(a) $O(\log n)$
(b) $O(n)$
(c) $O(n \log n)$
(d) $O(n^2)$
(e) $O(n^2 \log n)$

The next few questions are about the Bentley-Ottman plane sweep algorithm applied to this set of 3 lines:



46. The intersection event for a and b is added to the event queue when

(a) processing an left-endpoint event for a
(b) processing an left-endpoint event for b
(c) processing an left-endpoint event for c
(d) processing a crossing event for the pair (a, b)
(e) processing a crossing event for the pair (a, c)

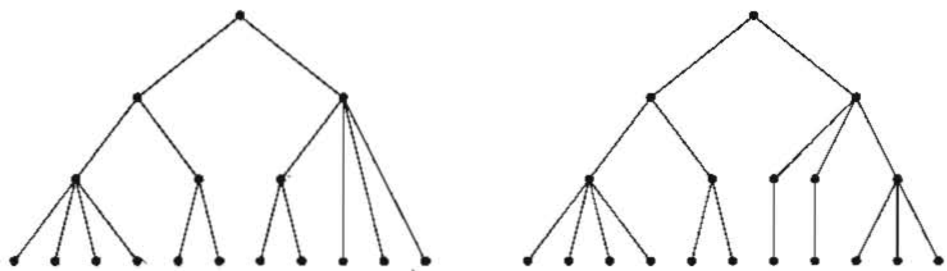
47. The intersection event for a and c is added to the event queue when

(a) processing an left-endpoint event for a
(b) processing an left-endpoint event for b
(c) processing an left-endpoint event for c
(d) processing a crossing event for the pair (a, b)
(e) processing a crossing event for the pair (a, c)

48. The intersection event for b and c is added to the event queue when

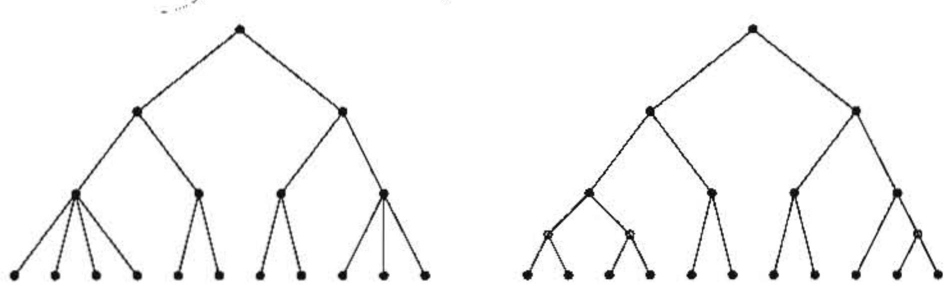
(a) processing an left-endpoint event for a
(b) processing an left-endpoint event for b
(c) processing an left-endpoint event for c
(d) processing a crossing event for the pair (a, b)
(e) processing a crossing event for the pair (a, c)

49. Yes or No: The following two trees are valid 2-4 trees



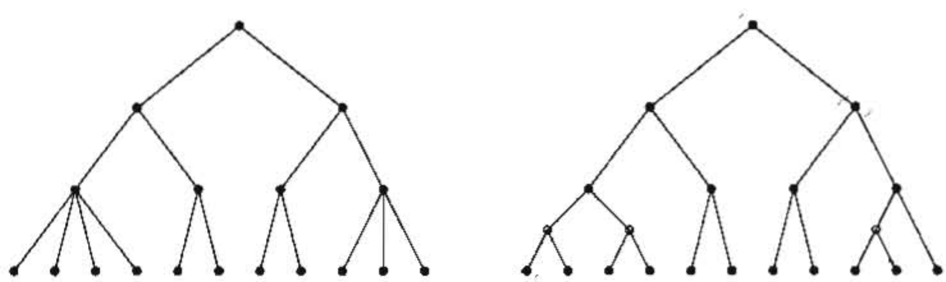
(a) Yes and yes
(b) Yes and no
(c) No and yes
(d) No and no

50. Yes or No: The following picture shows a 2-4 tree and a red-black that represents that 2-4 tree (red nodes are drawn as circles, black nodes as disks)



(a) Yes
(b) No
(c) Not enough information to decide

51. Yes or No: The following picture shows a 2-4 tree and a red-black that represents that 2-4 tree (red nodes are drawn as circles, black nodes as disks)



(a) Yes

(b) No

(c) Not enough information to decide

52. The following is a picture of

(a) Jon Bentley

(b) Ron Graham

(c) Avrim Melkman

(d) Thomas Ottman

(e) Godfried Toussaint