Computer Sciences 367

Midterm Exam 2

66 points (15% of final grade)

Instructors: Debra Deppeler, Alexi Brooks

1. **Fill in these fields and their bubbles on the scantron form (use #2 pencil).**

   (a) LAST NAME - fill in your last (family) name starting at left column.

   (b) FIRST NAME - fill in first five letters of your first (given) name.

   (c) IDENTIFICATION NUMBER is your UW Student ID number.

   (d) Under ABC of SPECIAL CODES, write your lecture number as a three digit value 001, 002, or 003.

   (e) Under F of SPECIAL CODES, write the number 2 for Secondary and fill in the number (2) bubble.

2. **DOUBLE-CHECK THAT YOU HAVE FILLED IN THE BUBBLES FOR EACH COLUMN OF ABOVE FIELDS ON SCANTRON.**

3. **Read, agree to, and sign this ACADEMIC CONDUCT STATEMENT.**

I will keep my answers covered so that they may not be viewed by another student during the exam or prior to completion of their exam. I will not view or in any way use another's work or any unauthorized devices. I understand that I may not make any type of copy of any portion of this exam. I understand that being caught doing any of these or other actions that permit me or another student to submit work that is not our own will result in automatic failure of the course. All such penalties are reported to the Deans Office for all involved.

Sig. ______________________________

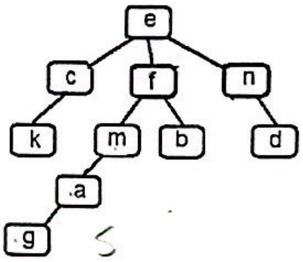
Parts	Number of Questions	Question Format	Possible Points
I	21	Simple Choice	21
II	15	Multiple Choice	45
	36	Total	66

Turn off and put away all electronic devices and wait for the proctor to signal the start of the exam.

Part I: Simple Choice (18 Questions 1 points each)

Select the word or phrase that makes the statement **true**. If there is no ____________ , then mark **A** if the statement is *true* or **B** if the statement is *false*. Be sure to mark the corresponding letter on the answer sheet (scantron). Unless otherwise specified, assume the ADTs, data structures, interfaces, and algorithms mentioned are those discussed in lecture and in the readings.

1. What is the height of the tree shown below?
   A. 3 B. 4 [✓ C.] 5 D. 6



5

2. AVL trees are binary search trees that **detect** out of balance by ____________ .
   [✓ A.] computing the balance factor of each node B. rotating the children of each node

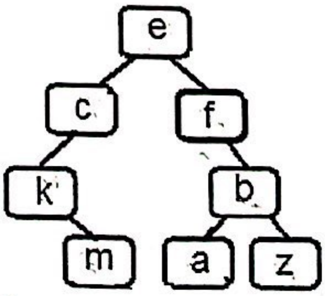
3. A binary search tree is *height-balanced* if the difference between the height of the left and right subtrees of each node is less than ____________ .
   A. 1 [✓ B.] 2

4. Program 3 could be used for merging *thesaurus* or *weather* data. What would need to be done to enable program 3 to handle joining data from files that stored different information such as various student record data?
   A. Write a `StudentRecord` class that *implements* the **abstract class Record**.
   [✓ B.] Write a `StudentRecord` class that *extends* the **abstract class Record**.

5. *Rotations* are used to maintain balance of ____________ .
   [✓ A.] AVL trees B. BTrees

6. **BTrees** are balanced search trees that have nodes with ____________ **keys**.
   A. 1, 2, or 3 [✓ B.] 2, 3, or 4

7. Which terms are correct descriptors of the tree shown?
   i full
   ii complete
   iii height-balanced
   [✓ iv] balanced
   ~~A. i and ii~~
   B. ii,iii, and iv
   C. iii and iv
   [✓ D.] iv only
   E. None of above.



$2^3 = 8$ $2^4 = 16$

$N = 8$ $h = 4$

$3 < \log_2(9) < 4$

8. **PriorityQueueADT** is a ____________ -oriented data type. [✓ A.] value B. position

9. *Binary search* is used to efficiently find items in an unsorted list. A. true [✓ B.] false

10. The **Big-O time complexity** of *recursive* solutions is typically ____________ the **Big-O time complexity** of *iterative* solutions.

A. less than [✓ B.] greater than C. the same as

11. In a **heap**, all of the *leaves at depth d-1* are to the ____________ of the *leaves at depth d.*

A. left [✓ B.] right

12. To determine the complexity of a recursive solution, the recurrence equation has the form:

T(N) = growth rate fnctn of **non-recursive** part + # recursive calls * T(problem size of **non-recursive** call)

A. true [✓ B.] false

Not efficient enough

13. The **PriorityQueueADT** may be implemented using a **chain of nodes** as the internal data structure. [✓ A.] true B. false

14. The ____________ operation is **NOT** a **PriorityQueueADT** operation.

[✓ A.] insert(int pos, Comparable item) B. insert(Comparable item)

15. *Reheapify* restores the ____________ constraint of a **heap**. A. shape [✓ B.] order

16. *Execution tree trace* is used because it is a faster alternative to a *call stack trace*. A. true B. false

17. A **heap** is a ____________ data structure. A. linear [✓ B.] non-linear

18. In recursion, each *recursive case* must make progress towards ____________ case.

A. an explicit B. an implicit [✓ C.] a base D. a recursive

19. In Java, ____________ legal for *methodA* to call *methodB* and then for *methodB* to call *methodA*.

[✓ A.] it is B. it is not

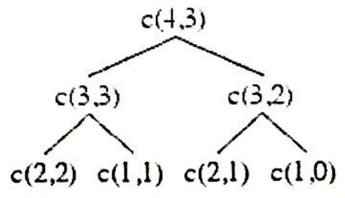
20. ____________ is present in the standard Java library as an *interface*.

A. Stack [✓ B.] Queue ~~C.~~ PriorityQueue

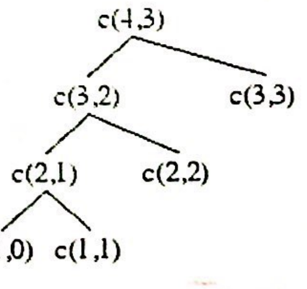
21. What is a correct execution tree trace for the following recursive method?

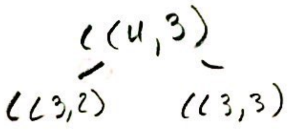
```
int c(int a, int b) {
    if ( b > a ) return 0;
    if ( b==0 || b==a )
        return 1;
    return c(a-1,b-1) +
        c(a-1,b);
}
```

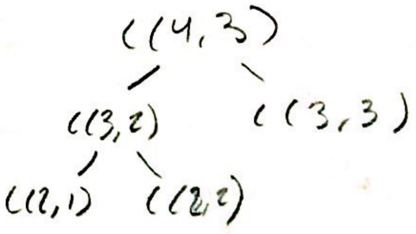
A.



[✓ B.]





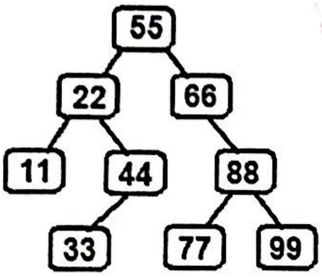


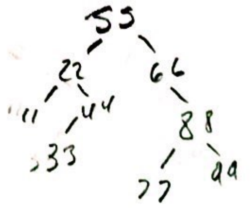
Part II: Multiple Choice (15 Questions 3 points each)

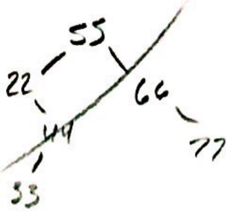
Choose the best answer of the available choices.

~~55,22,66,44,33,~~

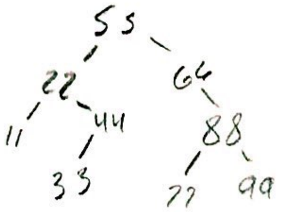
22. Which sequence of inserts produced the given binary search tree? Assume there were no deletes.







55,22,44,33,66,88,11
99, 77



case 1: No kids

Set to null

case 2: 1 kid

Switch w/ kid

~~A~~. 11,22,33,44,55,66,77,88,99

~~B~~. 55,22,66,44,33,77,11,88,99

[✓ C.] 55,22,44,33,66,88,11,99,77

~~D~~. 55,66,22,33,44,11,88,77,99

~~E~~. 99,88,77,66,55,44,33,22,11

Given four code fragments, labeled W, X, Y, and Z as shown:

Code fragment W (3) if either child is null

```
if (node.getLeft() == null)
    return node.getRight();
if (node.getRight() == null)
    return node.getLeft();
```

Code fragment X (1) Tests if current node is null

```
if (null == node)
    return null;
```

Code fragment Y (4)

```
//getMaxKey returns the largest key in a
    subtree
node.setKey(getMaxKey(node.getLeft()));
node.setLeft(delete(node.getLeft(),
    node.getKey()));
```

Code fragment Z (2)

```
if (node.getKey() > key)
    node.setLeft(delete(node.getLeft(),
        key));
if (node.getKey() < key)
    node.setRight(delete(node.getRight(),
        key));
```

23. Which ordering of the above code fragments completes the delete method of a binary search tree?

```
private static BSTNode delete(BSTNode node, int key) {
    CODE
    return node;
}
```

node

10

1

d(1)

d(10)

↳d(1)

~~A. X, W, Y, Z~~

[✓ B.] X, Z, W, Y

~~C. W, X, Y, Z~~

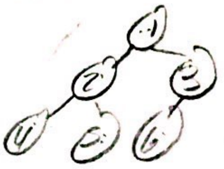
~~D~~. X, Z, Y, W

24. What is the *Big-O time complexity* for the binary search lookup on an efficiently implemented *binary search tree* that is *balanced*? Assume that array is sorted in *ascending* order.

$N = 6$
Find 5
3 times

[✓ A. $O(logN)$]
B. $O(N)$
C. $O(NlogN)$
D. $O(N^2)$
E. $O(N/2)$

$2 \le log_2 6 \le 3$



25. What is the Big-O *time complexity* of the *removeMax* operation of a **PriorityQueue** implemented using a *sorted array*? Assume: The number of items is stored and the values are in the array in ascending order.

A. $O(Nlog_2N)$ B. $O(N^2)$ C. $O(N)$ D. $O(log_2N)$ [✓ E. $O(1)$]

1	2	3	4	5	6

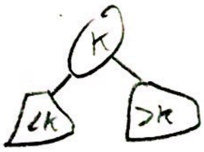
26. In Program 3, what would be the first line of the output file after *reducing* these three data files using `Reducer` and the *Thesaurus* record type?

apple : fruit, seed

```
data1.txt              data2.txt               data3.txt
----------------       ------------------      --------------------
banana:sliced          apple:seed,fruit        carrot:orange,green,veggie
carrot:fresh           banana:skin,fruit       pear:yellow,red,green,fruit
```

Recall: Entries in the input file with the same key value should be merged into a single record before they are written in the output file. For thesaurus data, this meant merging the different synonym lists (removing any duplicates), sorting them, and writing the resulting list on a single line.

[✓ A. apple:fruit,seed]
B. banana:sliced,skin,fruit
C. carrot:orange,green,veggie,fresh
D. apple:seed,fruit
E. banana:fruit,skin,sliced

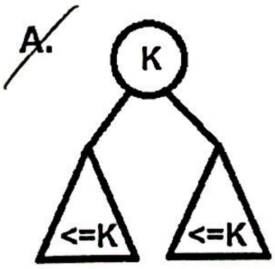


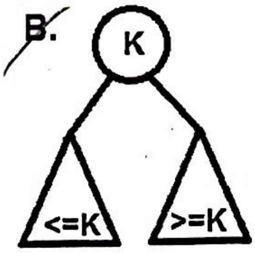
(parent*2)+1

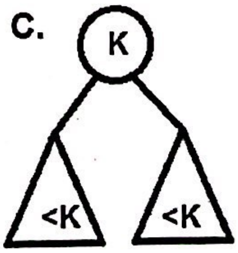
27. Which expression computes the index of the **right child** of the node storing the current data item in an array-based heap? Assume the heap's *root* node is at index **1**.

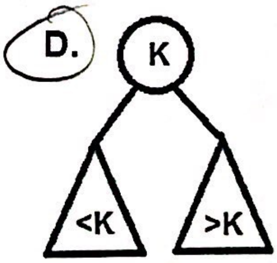
A. parent*2 [✓ B. parent*2+1] C. parent*2-1 D. parent/2 E. (parent-1)/2

28. Which diagram shows the *ordering constraint* for a **binary search tree**?

~~A.~~


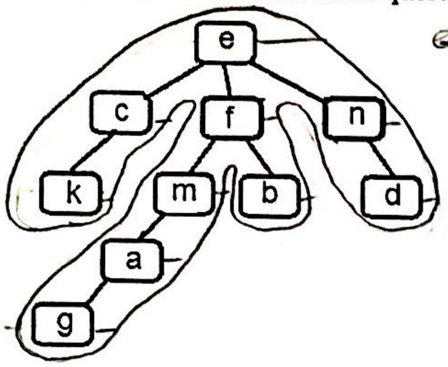
~~B.~~


C.


[✓ D.]


Scanned by CamScanner

Refer to this tree for the next three questions.



~~e,c,k,f,m,a,g,b,n,d~~

k, c, g, a, m, b, f, d, n, e

e, c, k, f, m, a, g, b, n, d

29. Which sequence of node labels identifies the **pre-order** traversal of the tree?

- A. e,c,f,n,k,m,b,d,a,g
- ~~B.~~ k,c,g,a,m,b,f,d,n,e
- [✓ C.] e,c,k,f,m,a,g,b,n,d
- ~~D.~~ g,a,k,m,b,d,c,f,n,e

30. Which sequence of node labels identifies the **level-order** traversal of the tree?

- [✓ A.] e,c,f,n,k,m,b,d,a,g
- ~~B.~~ k,c,g,a,m,b,f,d,n,e
- C. e,c,k,f,m,a,g,b,n,d
- ~~D.~~ g,a,k,m,b,d,c,f,n,e

ecfnkmbdag

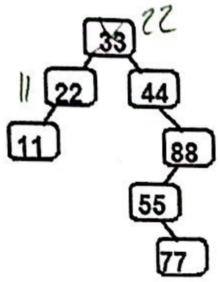
31. Which sequence of node labels identifies the **post-order** traversal of the tree?

- ~~A.~~ e,c,f,n,k,m,b,d,a,g
- [✓ B.] k,c,g,a,m,b,f,d,n,c
- ~~C.~~ e,c,k,f,m,a,g,b,n,d
- ~~D.~~ g,a,k,m,b,d,c,f,n,e

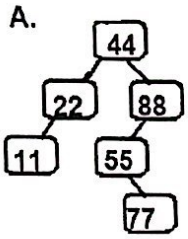
k, c, g, a, m, b, f, d, n, e

32. Which tree is the result of **delete(33)** on this binary search tree?

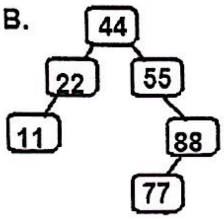
Assume the *inorder-predecessor* is chosen as replacement value for the **delete**.



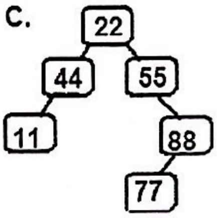
A.



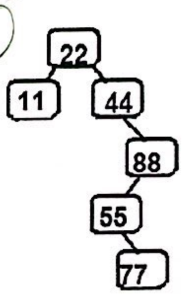
B.



C.

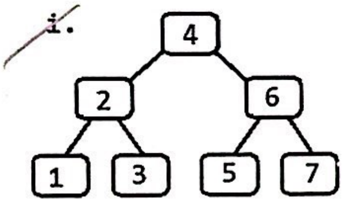


[✓ D.]

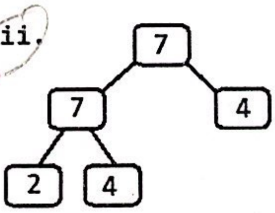


33. Which figures are max heaps?

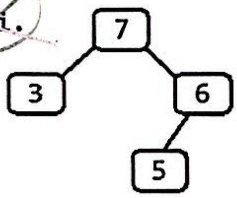
i.



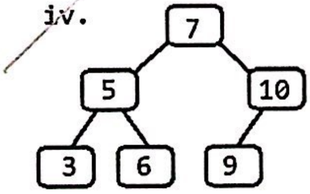
[✓ ii.]



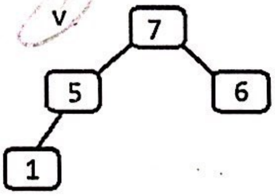
~~iii.~~



iv.



[✓ v]



~~A. i, ii, iv, and v~~

B. ii only

[✓ C.] ii and v

~~D. iii and v~~

~~E. i and iv~~

34. Which code replacement completes this recursive method so that it performs a binary search of a sorted array for the item x?

```
// pre-condition: a and b are valid indexes of the array
boolean search(T[] array, int a, int b, T x) {
    if ( a > b ) not in list         front    end
       return false;

    int c = (a+b)/2; center
    T d = array[c]; temp variable

    if ( x.equals(d) ) test if this is searched item
       return true;

    if ( x.compareTo(d) COND )        x>d        >0
       return search(array,c+1,b,x) ;

    return EXPR; search(array, a, (-1, x)
}
```

A. COND: < 0 EXPR: search(array,a,c-1,x)

[✓ B. COND: > 0 EXPR: search(array,a,c-1,x)]

~~C. COND: < 0 EXPR: search(array,a+1,c,x)~~

~~D. COND: > 0 EXPR: search(array,a+1,c,x)~~

~~E. COND: < 0 EXPR: search(array,a+1,c-1,x)~~

4

1
—
0

0	1	2	3
a	b	c	d
d	c	b	a

35. Which code completion reverses the characters in a char array?

```
static void reverse(char [] a) { reverse(0,a); }
static void reverse(int i, char[]a) {
    if ( i < a.length/2 ) {
        CODE ; reverse (i+1, a)
        char t = a[i];
        a[i] = a[EXPR]; a.length-1-i
        a[EXPR] = t; a.length-1-i
    }
}
```

Note: **EXPR** is the same expression used in two places in the method.

~~A. CODE: reverse(i-1,a) EXPR: a.length-i~~

B. CODE: reverse(i+1,a) EXPR: a.length-i

~~C. CODE: reverse(i-1,a) EXPR: a.length-1-i~~

[✓ D. CODE: reverse(i+1,a) EXPR: a.length-1-i]

~~E. CODE: reverse(i,a) EXPR: a.length~~

r(0)
br(1)←
br(2)

Scanned by CamScanner

[redacted]

36. Which code replacement ensures that weather records will be ordered first by *station ID* and then by *date*? Assume the following:

- *compareTo* method is in class WeatherRecord
- the desired order for the WeatherRecord type is ascending
- station and date are both java.util.Comparable
- station and date are each data members of the WeatherRecord type

```
int compareTo(WeatherRecord wr) {
    if ( COND ) station.compareTo(wr.station)==0
        return date.compareTo(wr.date);
    return EXPR ; station.compareTo(wr.station)
}
```

A. COND: (station+date).compareTo(wr.station+wr.date) EXPR: station.compareTo(wr.station)

[✓ B.] COND: station.equals(wr.station) EXPR: station.compareTo(wr.station)

C. COND: (date+station).equals(wr.date+wr.station) EXPR: wr.station.compareTo(station)

D. COND: station.equals(wr.station) EXPR: wr.station.compareTo(station)

station.equals(wr.station

station.compareTo(wr.station)