1) [39] TRUE/FALSE **(mark each as T or F)**

F (T) Linear (i.e., "progressive overflow") with a separate collision area is the most time-efficient collision resolution algorithm

F A BST index gives faster random access (e.g., for a query) than a hash table index - to access MainData file records

T A random access file structure (like Asg1's MainData file) must use fixed-length record locations (vs. variable-length)

F For a given BST with specific values, you can "reverse-engineer" the tree to determine the record-order of the raw data file

T There can be > 1 program in a VisualStudio/C# or NetBean/Java project

If an app needed random access for WMU students using WIN (i.e., ID#) as the key

T (F) A hash FILE is a good choice F (T) A direct address FILE is a good choice

If an app needed random access for WMU students using email address as the key

T (T) A hash FILE is a good choice F (F) A direct address FILE is a good choice

If MainData was a hash FILE (hashed on NAME), you could do key-sequential access of that file by ID if you had an ID INDEX which was structured as a(n) . . . *[each of the following 5 is a separate T/F question]*

F hash table T BST T direct address table T ordered linked list (T) serial table

F (T) 1 I/O takes about 50 times as long as 1 internal instruction

F (T) Random access files must be binary files

T The order of the input data affects a BST's search time

F Binary files must be random access files

F A given GROUP of values always results in the same exact BST

T You can do key-sequential access of a BST

T A BST can be used for a lookup table in memory

T A given input file always gives the same exact BST

F External indexes have faster search time than internal indexes

T Short fat trees are better than tall skinny trees

F (T) An internal index is always better than an external index

T A BST is "logically" in key-sequence

T Direct address files are a type of hash file without collisions

T Post-order traversal of a BST isn't really of any use

F (T) Direct address tables have slower query time than hash tables

F A BST would make a good external index

T A C#/Java program can run another C#/Java program

F You can do key-sequential access on a hash table

T (F) You can use direct address for a lookup table in memory

T BST nodes could be stored in an array of nodes

F InOrder traversal of a hash table gives key-sequential access

F (T) BST's grow taller on the root end (vs. the leaf end)

T If chaining's used for a hash table, the links are subscript values

T The DRPs in the Hash table index are RRNs

F (T) A BST is physically in key-sequence in the array

2) [3] For some BST of height 4, could the root possibly contain the 5th smallest value? (Yes or No) yes

What's the height of the shortest BST you could get with 13 KV's? 4 (a number)

What's the height of the tallest BST you could get with 13 KV's? 13 (a number)

3) [9] What's the ORDER of COMPLEXITY (for time, not space) for a SINGLE operation on the specified DATA structure?

	BST	Hash Table	Ordered List
INSERT →	$\theta(\log_2 n)$	$\theta(1)$	$\theta(n)$
DELETE →	$\theta(\log_2 n)$	$\theta(1)$	$\theta(n)$
QUERY →	$\theta(\log_2 n)$	$\theta(1)$	$\theta(n)$ O(log2n)

4) [4] What's the ORDER of COMPLEXITY for 1 QUERY transaction for each of these kinds of FILE structures?

DIRECT ADDRESS $\theta(1)$ KEY-SEQUENTIAL $\theta(n)$ O(log2n)

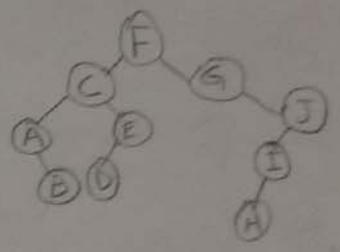
SERIAL $\theta(n)$ HASH $\theta(1)$

[1] When determining the above order of complexities, what's the important OPERATION that's used for determining the estimate? (that is, what's the "measurement unit") Time - unit

Big O notation

5) Grow a BST from this data (in the order shown): F G J C E A I H D B

[5] Show it "conceptually" (as **a tree** "picture")



[7] Show it "physically" (as it's actually stored in memory) ~~(like asgn 2 workshee)~~ **INCLUDE ALL PARTS**

[3] Average search path (successful): 3/10 3.1

[3] PreOrder traversal: F C A B E D G J I H

[Sub]	[LPtr]	[Key]	[RPtR]
0	3	F	1
1	-1	G	2
2	6	J	-1
3	5	C	4
4	8	E	-1
5	-1	A	9
6	7	I	-1
7	-1	H	-1
8	-1	D	-1
9	-1	B	-1

$$\frac{1+2(2)+3(3)+4(3)+5}{10}$$

$$\frac{1+4+9+12+5}{10}$$

$$= \frac{14+17}{10}$$

$$= \frac{31}{10}$$

$$= 3.1$$

6) [2] What's the offset calculation for random access for an RRN if the file's headerRecord had 3 int's & RRNs start with 1, not 0.

-1 [(RRN-1) * dataSize] + 12 headerRecordSize = 12

[1] What statement is used in C# or Java to move the file-position-ptr to the correct offset location? seek(offset)

7) [5] The best performing hash FILE structure uses fixed home location with separate chaining for the collision resolution algorithm, because with that (and a reasonably good hash function), one can get about (on the average)

8(1.5) (a number) time-unit (the measurement unit) as the time-performance for a single query, and still have about 95 % for the packing density.

8) [7+7] Create two hash **TABLES** using this input data in this order . [The following are the keys].

18, 27, 48, 44, 98, 88, 37, 00, 87

Both tables use DivisionRemainder for the HF. The 2 different CR Algorithms are specified below. Use 10 as MaxN (i.e., **MAX_N_LOC** or **MAX_N_HOME_LOC,** as appropriate) . Show the 2 tables – with labels at the top of & subscripts on the left.

This table uses CHAINING with SEPARATE area

MAX_N_LOC = 10

	[KEY]	[LINK]	[SUB]
	0	-1	0
		-1	1
		-1	2
		-1	3
	44	-1	4
		-1	5
		-1	6
	27	14	7
	18	12	8
		-1	9
	48	-1	10
	98	10	11
	88	11	12
	37	-1	13
	87	13	14

$\frac{1+1+3+4}{1+1+1+1} = \frac{9}{4}$

[2+2] Average Search Path for this table (for successful searches): 9/4

This table uses LINEAR with EMBEDDED

-4

MAX_N_LOC = 10

[KEY]	[SUB]
0	0
	1
	2
	3
44	4
	5
	6
27	7
18	8
	9
48	10
98	11
88	12
37	13
87	14

Average Search Path for this table (for successful searches): 14