CIS 560, Fall 2013 **Name:______________________**
Exam 2 – 50 minutes

This test consists of four questions. The number of points for each question is shown below.

- Read all questions carefully before starting to answer them.
- Write all your answers on the space provided in the exam paper.
- The order of the questions is arbitrary, so the difficulty may vary from question to question. Don't get stuck by insisting on doing them in order.
- Show your work. Correct answers without justification will not receive full credit. However, also be concise. Excessively verbose answers may be penalized.
- Clearly state any simplifying assumptions you may make when answering a questions.
- **Be sure to write your name on the test paper.**

Question	1	2	3	4	total
Points	20	30	25	25	100
Your points					

Exercise I. [True/False Questions: 10 questions - you get 2 points for each correct answer; you lose 1 point for each incorrect answer; you get 0 points for questions that you don't answer]:

True/[✓ False]
For a set of integers, there exists a unique way to index them in a B+ tree.

[✓ True]/False
Transaction management consists of two main functional components: concurrency control and failure recovery.

[✓ True]/False
For logging, we prefer the UNDO+REDO scheme, so that we can be more flexible in when to write out dirty data to disk.

True/[✓ False]
You can avoid the phantom problem by using a concurrency control and recovery paradigm that prevents cascading rollback.

[✓ True]/False
Concurrency control by timestamps is superior to that by locks if most transactions are read-only.

True/[✓ False]
If possible, we should always use a bushy join tree instead of a left-deep join tree, because the former is more efficient than the latter.

True/[✓ False]
Hash indexes are efficient in answering range queries, such as "finding products with price higher than 100".

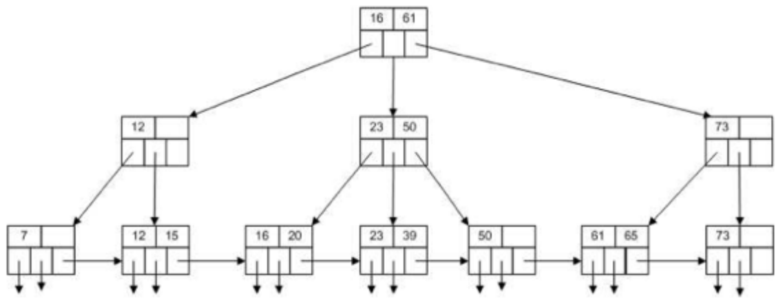
True/[✓ False]
Given a choice between materializing the result of a join and pipelining it to the next operation, in general, you should materialize it, if you can.

[✓ True]/False
For the same operation (e.g., join, sorting, grouping), two-pass algorithms generally require less memory buffer than their corresponding one-pass algorithms.

[✓ True]/False
"Pushing selections down" refers to the choice of processing selections "early" during query optimization, which is often advantageous because it means fewer tuples need to be manipulated in later steps.

Exercise II [B-tree indexes, 30 points]

Consider the B+ tree of degree d=1 (i.e., each index node can hold at least d = 1 key and at most 2d = 2 keys) shown below.



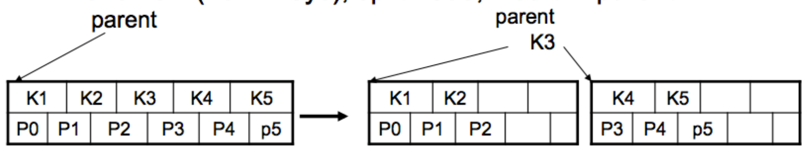
(i) [10 points] What is the largest number of records that can be inserted into this tree (more nodes may be created), while the height of the tree does not change. Justify your answer.

Maximum: The root can have up to 3 pointers to 3 nodes on level 2. Each of the nodes on level 2 can also have up to 3 pointers to leaf nodes. Each leaf node can have up to 2 pointers to records. Therefore, the maximum number or records that the tree can hold is 3*3*2 = 18 (nodes at the leaf level). Given that the tree contains 11 leaf nodes already, the largest number that can be inserted is 18-11 = 7.

(ii) [10 points] Show the tree that would result from inserting a data entry with key 19 in the tree above. [It's ok to redraw only the part of the tree where changes occur.]

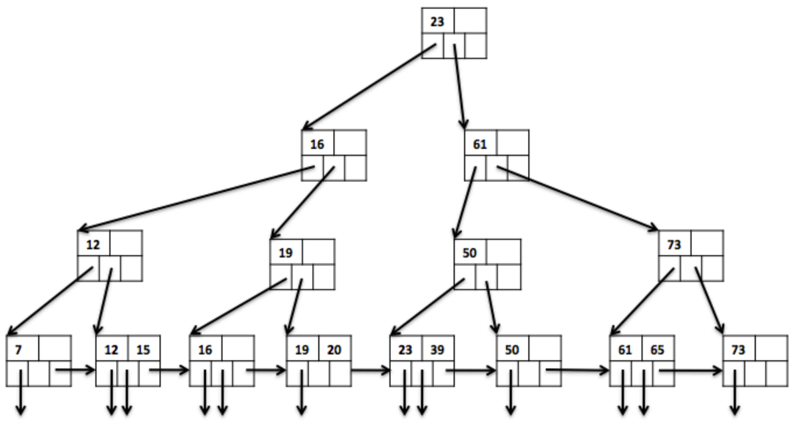
This is the pseudocode of the algorithm that we have used for inserting in a B+ tree:

Insert (K, P)
- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt
- If overflow (2d+1 keys), split node, insert in parent:



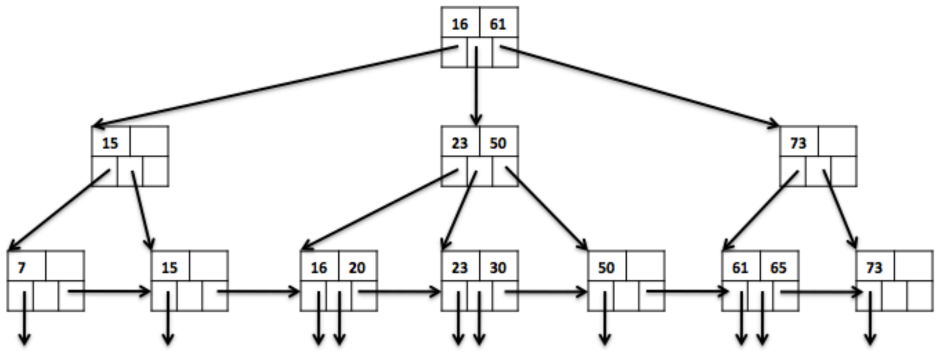
- If leaf, keep K3 too in right node
- When root splits, new root has 1 key only

Applying this algorithm to our problem, we get the following tree:



(iii)[10 points] Show the tree that would result by deleting a data entry with key 12 from the original tree. [It's ok to redraw only the part of the tree where changes occur.]

Note: this deletion is similar to the example "delete 30" that we discussed in class.



Exercise III [Transactions, 25 points]

1. [5 points] The following schedule is presented to a timestamp-based scheduler. Assume that the read and write timestamps of each element start at 0 (RT(X) = WT(X) = 0), and the commit bits for each element are set (C(X)=1). Explain what happens as the schedule executes: st1, r1(A), st2, w2(B), r2(A), w1(B).

T_1	T_2	A	B
		RT=0	RT=0
		WT=0	WT=0
st_1			
Assume $TS(T_1)=1$			
$r_1(A)$		RT=1	
	st_2		
	Assume $TS(T_2)=2$		
	$w_2(B)$		WT=2
	$r_2(A)$	RT=2	
$w_1(B)$			
$TS(T_1)<WT(B)$ i.e. later transaction has already written B. Wait until the other transaction commits or aborts.			

2. [5 points] For the following schedule:

```
r1(A), w1(B), r2(B), w2(C), r3(C), w3(A)
```

Answer the following questions:

i. [3 points] What is the precedence graph for the schedule?

$r_1(A)w_3(A)$ $T_1<T_3$

$w_1(B)r_2(B)$ $T_1<T_2$

$w_2(C)r_3(C)$ $T_2<T_3$

Precedence graph is : T1 -----> T2 -----> T3 (including a direct edge from T1 to T3)

[2 points] Is the schedule conflict-serializable? If so, what are the equivalent serial schedules?

Yes, schedule is conflict-serializable. Conflict-equivalent serial schedule is (T1,T2,T3).

3. [9 points] After a system's crash, the redo-log using nonquiescent checkpointing contains the following data.

`<START S>`
`<S, A, 60>`
`<COMMIT S>`
`<START T>`
`<T, A, 10>`
`<START U>`
`<U, B, 20>`
`<START CKPT ???>`
`<T, C, 30>`
`<START V>`
`<U, D, 40>`
`<V, F, 70>`
`<END CKPT>`
`<COMMIT U>`
`<T, E, 50>`
`<COMMIT T>`
`<V, B, 80>`
`<COMMIT V>`
`CRASH !!!`

i. [2 points] What are the transactions specified in the <START CKPT ???> record?

T, U – as they are both still active at the time that CKPT is written.

ii. [2 points] Indicate and explain what fragment of the log the recovery manager needs to read.

`From <START T> to the end.`

iii. [5 points] Assuming that the <START CKPT ???> record is correctly stored in the log, show which elements are recovered by the redo recovery manager and compute their values after recovery.

Redo actions of transactions T,U,V, specifically.

`<T, A, 10>, <U, B, 20>, <T, C, 30>, <U, D, 40>, <V, F, 70>,`

```
<T, E, 50>, <V, B, 80>
```

Thus, the values of the elements after recovery are A=10, B=80, C=30, D=40, F=70, E = 50.

Exercise IV [Query optimization, 25 points]

(1) [5 points] Consider a two-pass sort-merge algorithm to sort a relation R with 25 pages. What is the cost of the algorithm and how many blocks of memory are needed?

We need to find the smallest integer M satisfying B <=M(M-1), which means the memory requirement is M>=6. The I/O cost is 3*B = 75.

(2) [10 points] Consider joining two relations R(x, y) and S(x, z) on their common attribute x. The size of relation R is 1000 blocks and the size of relation S is 500 blocks. Attribute x of relation R has 50 different values and the values are evenly distributed in R. Attribute x of relation S also has the same 50 different values and the values are evenly distributed in S. Furthermore, suppose that the memory buffer has 101 blocks. Can we use a two-pass partitioned join algorithm to join the relations R and S? If yes, explain how the algorithm works and compute the cost of the join. If not, explain how you came to this conclusion.

Yes, we can use a two-pass partitioned join algorithm to join the relations R and S.

First, we split R into partitions:

- Allocate one page for the input buffer. Allocate 100 pages for the output buffers: one page per partition.
- Partition R: Read in R one page at the time. Hash into 100 buckets. As the pages of the different buckets fill-up, write them to disk. Once we process all of R, write remaining incomplete pages to disk. At the end of this step, we have 100 partitions of R on disk. Assuming uniform data distribution, each partition comprises 10 pages.
- Partition S: Split S into 100 partitions the same way we split R (must use same hash function). At the end of this step, we have 100 partitions of S on disk. Assuming uniform data distribution, each partition comprises 5 pages.
- Perform the join: For each pair of partitions that match
    - Allocate one page for input buffer
    - Allocate one page for the output buffer.
    - Read one 10-page partition of R into memory. Create a hash table for it using a different hash function than above.
    - Read corresponding S partition into memory one page at the time. Probe the hash table and output matches.

Cost is 3B(R)+3B(S) = 4500

(3) [10 points] Consider the join $R \bowtie_{R.a=S.b} S$, given the following information about the relations to be joined. As usual, the cost is given by the number of disk I/Os (the cost of writing the result is ignored).

- Relation *R* contains 200,000 tuples and has 20 tuples per block
- Relation *S* contains 4,000 tuples and also has 20 tuples per block
- Both relations R and S are clustered
- 1002 memory pages are available

What is the cost of joining *R* and *S* using a block-nested loops join? What is the number of buffer memory required for this cost to remain unchanged?

Assuming R to be the outer relation, the cost is B(R) + B(R)*B(S)/(M-2) = 10,000 + 200*10,000/1000 = 12,000.

We need 1002 blocks of memory to guarantee this cost.

If you assumed S to be the outer relation, the cost is B(S) + B(R) = 200 + 10,000 = 10,200 (as S fits in the memory). We need at least 202 blocks of memory to ensure the cost.

Both solutions received full credit.