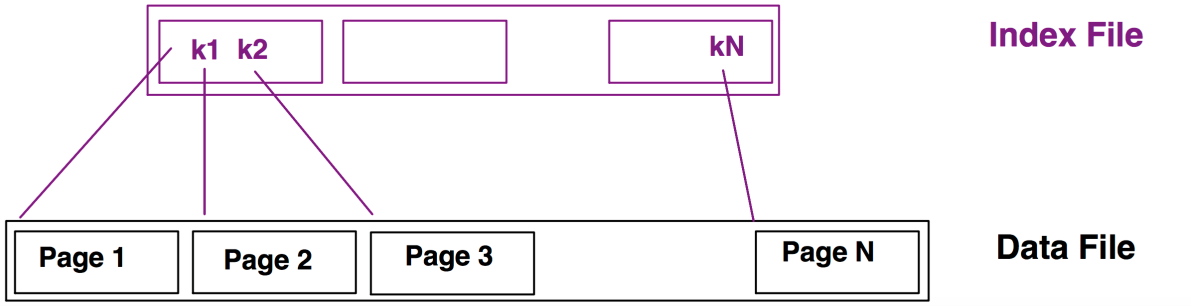
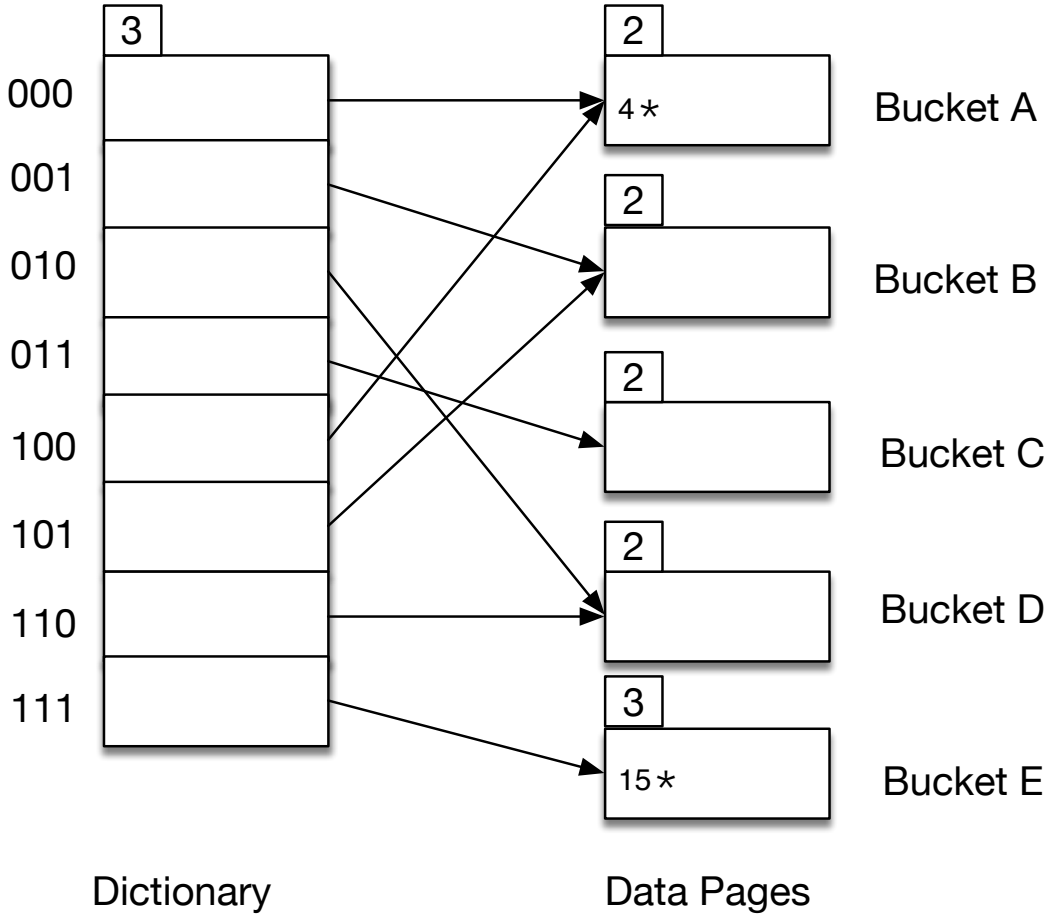
Question 1

1) If we do an equality search, a binary search can be conducted on either the data file or the index file, and the two behaviors have the same level of time complexity. However, the index file could be really smaller than its data file. For instance, in the case of the picture (borrow from our teacher's PPT), if we assume that the ordered file occupies *S* blocks and the index file occupies C blocks, because of C <= S, if we do a binary search on the index file, we can cost less than we do it on the data file.

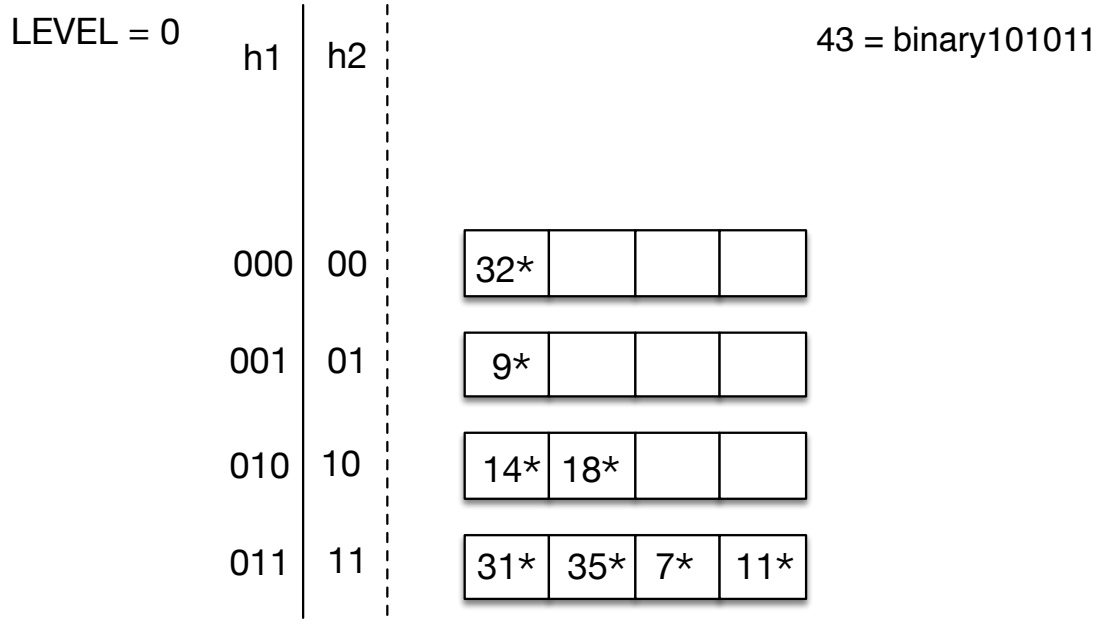


2) It is possible that deleting an entry reduces global depth by 2 in the Extendible Hashing. Here is the situation,

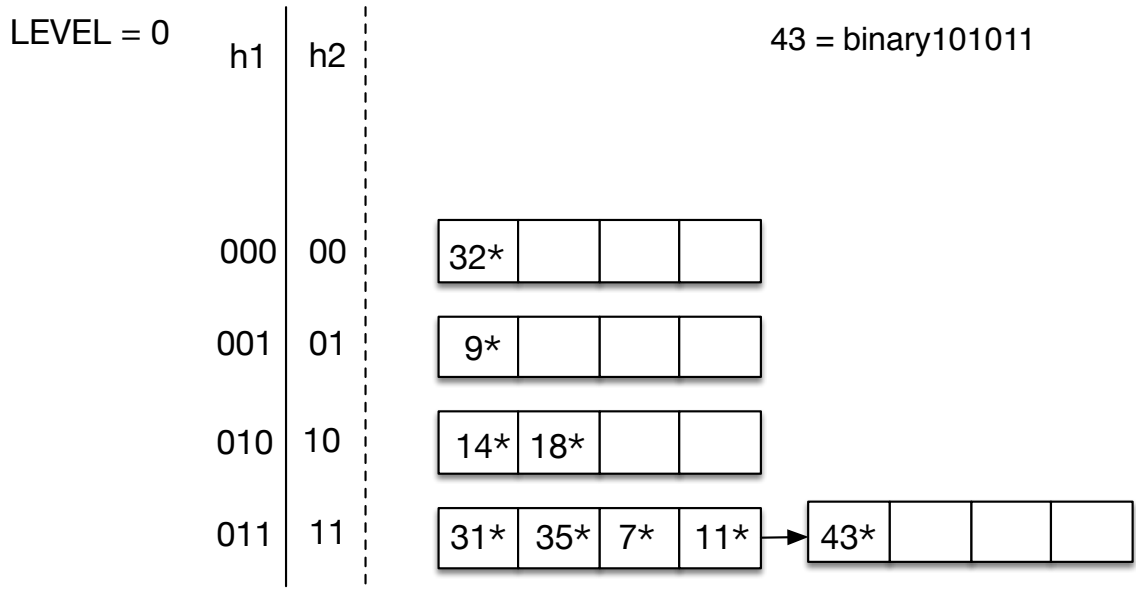


when we delete h(r) = 15, it will cause the bucket E empty. As we know that if removal of data entry makes bucket empty, can be merged with “split image”. Therefore, in this case, after we finish deleting 15, for the bucket B, C, D and E are empty, the dictionary can be merged and only two pages of dictionary can be left, hence the Global Depth could be reduced from 3 to 1.

3） This picture shows a situation that if next we insert record with key k such that h0(k) = 43, we need to put it in 4th page, and it will cause a overflow.



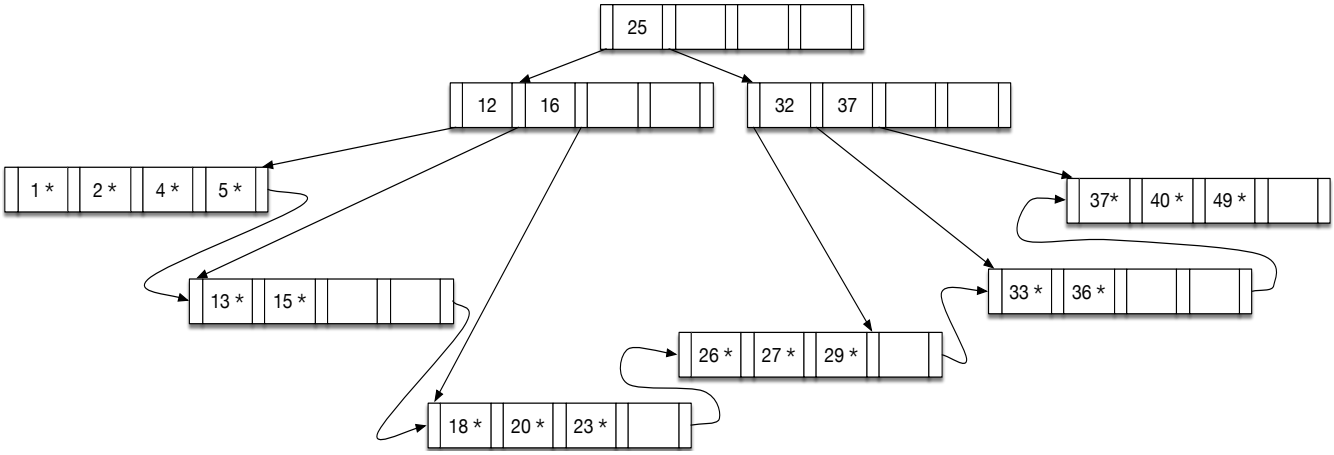
If the rule is “splitting whenever an overflow occurs performs”, it will cause a split and the primary pages will increase from 4 to 8. However, if the rule is “splitting only when the number of overflow pages exceeds a given threshold”, and as long as the threshold is more than 0, we can put the data entry in an overflow page like this:



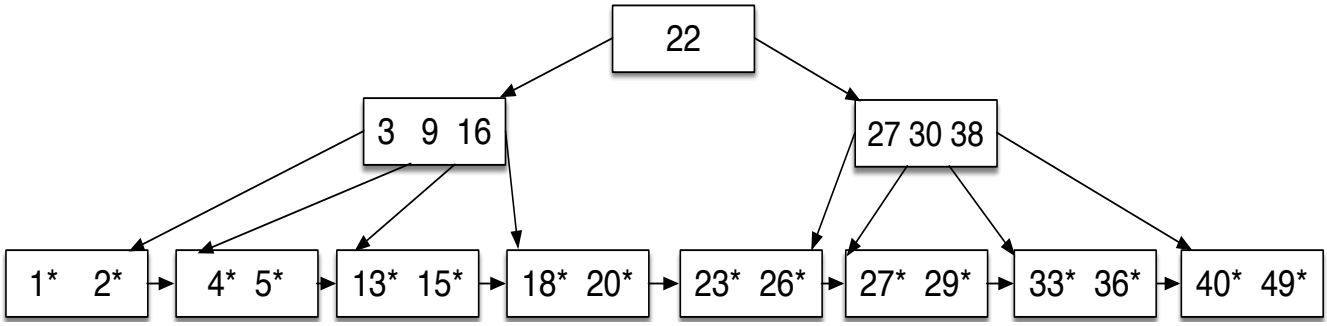
thus a split can be avoided and this is a scenario that "splitting whenever an overflow occurs performs worse, in terms of the number of total pages, than splitting only when the number of overflow pages exceeds a given threshold." in Linear Hashing.

Question 2

1) The picture shows the B+-tree after inserting a data entry with key 37 into the original tree.



2) The tree below is the valid B+-tree with the maximum number of tree nodes that contains the same data entries in the original tree, and the maximum number of tree nodes is 10.



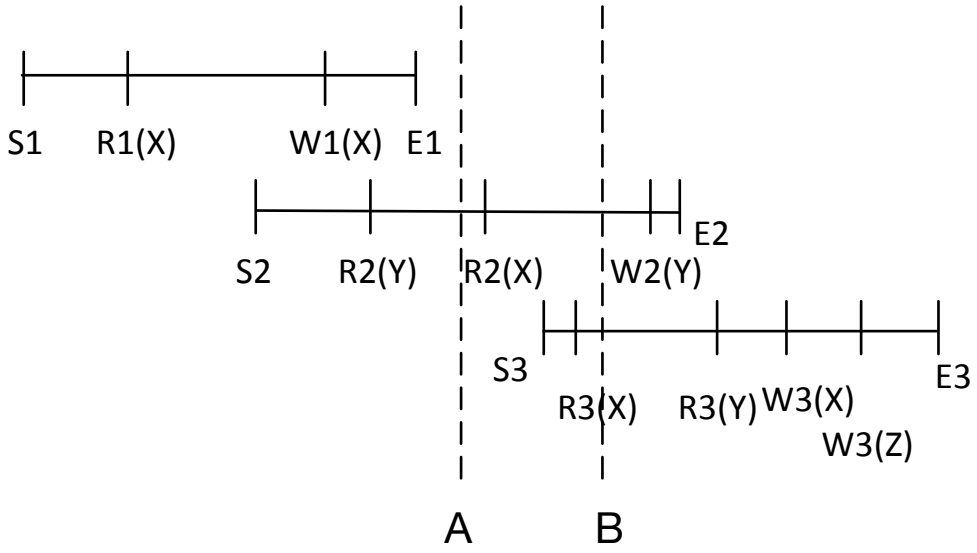
Question 3

Here is the preliminary analysis of the approaches:

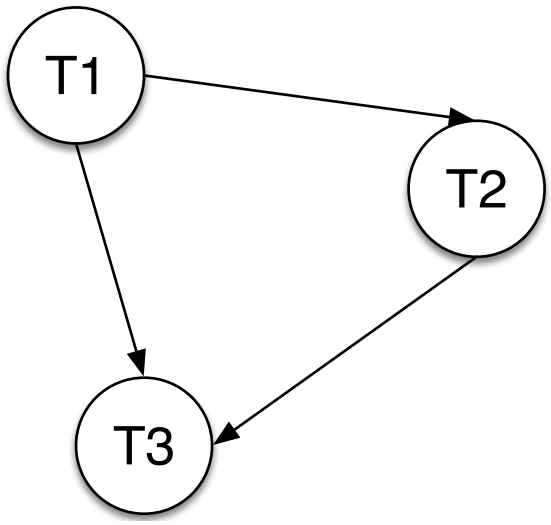
1) Since we have the options of using tree structured index and linear hashing, the approach of searching in the sorted directly seems have no chance to present in such a relation R containing 6,000,000.
2) Regarding range search, B+ tree index performs usually performs better than linear hashed index, what's more, although it is difficult to maintain, a clustered B+ tree performs better than unclustered B+ tree in terms of searching for large range of data. Furthermore, the 6. approach only on attribute R.b, but as there is no limited for R.b, this search could not help us.

To sum up, according to the consideration above, we exclude 1. 3. 5. 6., and we leave 2. & 4. both of which use a clustered B+ tree but are different between each other on attributes (R.a V.S. (R.a and R.b)). For R.a is a candidate key, and we use a clustered B+ tree index as well as the file is sorted, in this case, it is good enough for us to know the records which we need are the last 594,000 ((6,000,000 – 60,000) / 10) pages. First we can scan for the first record with R.a > 60,000, its approximately cost only 2 I/Os. Then we scan the leaf pages which are with the condition that R.a > 60,000 (594,000 pages). For the reason that we can get the a and b in the same time, 4th approach (Use a clustered B+ tree index on attributes (R.a,R.b)) is better than 2nd approach(Use a clustered B+ tree index on attribute R.a.), for the B+ tree index of 4th approach is on attribute R.a and R.b, which can help us get a and b at the same time. Therefore, 4th approach (Use a clustered B+ tree index on attributes (R.a,R.b).) seems the cheapest.

Question 4

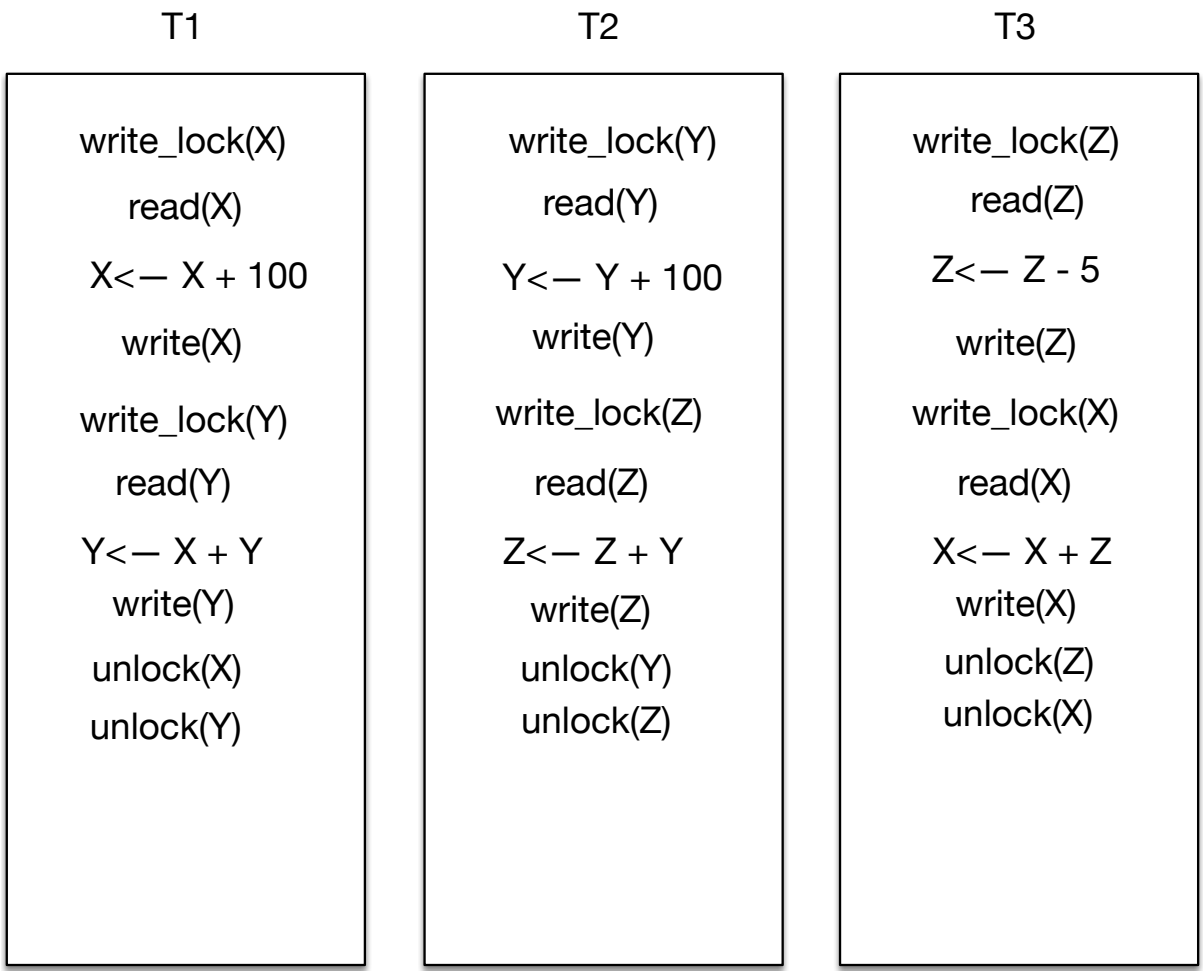


1) According the picture above, if the system crashes at B, the only transaction has been done is S1, so it will go to the Redo List, and we do the operations (i.e. R1(X), W1(X)) again, and S2 and S3 are not finished, hence they would go to the Undo list, and their reversing operations would be done for recovering the system.
2) If we made a checkpoint at A, and a crash occurs at B, we only need to redo the S2, for a start of checkpoint marker is written to the log, then the database updates in buffers are force-written, then and end of checkpoint marker is written to the log, there is no need to deal with S1.
3) The transaction schedule is conflict serializable, by drawing a precedence graph (also called conflict graph or serializability graph),



we can see that no directed cycle can be found. Therefore, the schedule is a conflict serializable one.

4) Using two-phase locking protocol can still cause a deadlock. Here is an example, the first graph shows three transactions, T1, T2 and T3:



all of which obey the two-phase locking protocol.

However, when the schedule below is operating:

T1	T2	T3
write_lock(X)		
read(X)		
X<— X + 100		
write(X)		
	write_lock(Y)	
	read(Y)	
	Y<— Y + 100	
	write(Y)	
		write_lock(Z)
		read(Z)
		Z<— Z - 5
		write(Z)
write_lock(Y)		
****waiting for Y****	write_lock(Z)	
****waiting for Y****	****waiting for Z****	write_lock(X)
****waiting for Y****	****waiting for Z****	****waiting for X****

we can see a deadlock occurs. Therefore, it is possible that a deadlock can be caused even all the transactions obey the two-phrase protocol.