CS3563: DBMS II
Assignment 3

Sagar Jain
CS17BTECH11034

April 25, 2020

Solutions

1. (a)

$$\begin{aligned} size &= (\#recording\ surfaces) * (\#tracks/surface) * (\#sectors/track) * (size\ of\ sector) \\ &= 8 * 8192 * 512 * 512\ B \\ &= 17179869184\ B \\ &= 16\ GB \end{aligned}$$

(b) The last given address is <**3, 4095, 127**>. With this information it can be assumed that all there might be some data on the addresses before and including the one given which we would not want to overwrite.
Each surface has 8192 tracks. Ideally, we would want to start storing a new file in a new track, unless the file is smaller than the size remaining in the track in which the read-write head is currently, this is so that we can decrease the totoal number of tracks over which the file is spread, this will reduce the total time needed to access the file as we reduce the total movement of the disk arm, therefore reducing the seek time.

Assuming that an 1 MB = 1024 KB and 1KB = 1024 B, the number of sectors needed would be,

$$fileSize\ /\ trackSize = (1000 * 1024 * 1024)/(512 * 512) = 4000\ tracks$$

So, if we begin at the address <**3, 4096, 0**>, the last sector of the file would be stored at <**3, 8095, 511**>.

(c) There are 4000 tracks which need to be read completely. So that is 4000 rotations, also the read-write would have to move 3999 times ($\because$ we are already on the last track). An assumption I am making here is that the file can be read backwards and assembled by the disk controller or other operating system utility. So the

total time required would be:

$$\begin{aligned} RPS &= RPM/60 \\ &= 7200/60 \\ &= 120 \\ Time\ to\ read\ one\ track &= 1/RPS \\ &= 1/120 \\ seekTime &= 5ms \\ Total\ time &= 3999 * 5/1000 + 4000/120 \\ &= 53.328 \end{aligned}$$

If the file can only be read from the beginning to the end we would have to seek to the beginning initially that would increase the total time by 5ms + 1/120s = 8.3ms (moving disk arm & rotating to the beginning of the track), in that case the total time would be **53.342s**.

(d) Assuming the file was read from beginning to end, the head would be on track 8095 after reading the file.

For FCFS, the order of the tracks visited and total distance would be as follows:

Track Number	Distance
8095	
200	7895
5000	4800
4200	800
5	4195
7200	7195
4000	3200
2200	1800
800	1400
6500	5700
Total Distance	36985

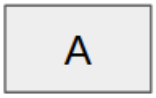
The total distance travelled by the disk arm would be 36985, in terms of number of cylinders when using the FCFS algorithm.

Using the elevator algorithm the following would be the order and the distance travelled.

Track Number	Distance
8095	
7200	895
6500	700
5000	1500
4200	800
4000	200
2200	1800
800	1400
200	600
5	195
Total Distance	8090

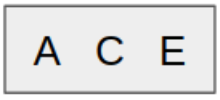
The total distance travelled by the disk arm would be 8090, in terms of number of cylinders when using the elevator algorithm.

2. (a) **Insert A**

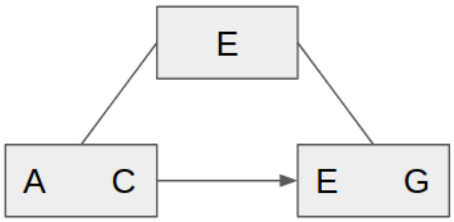


Insert C

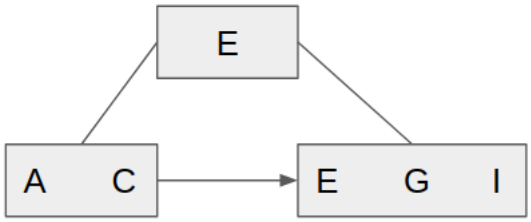
Insert E



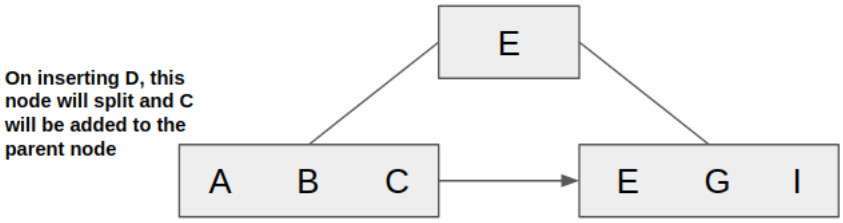
Insert G



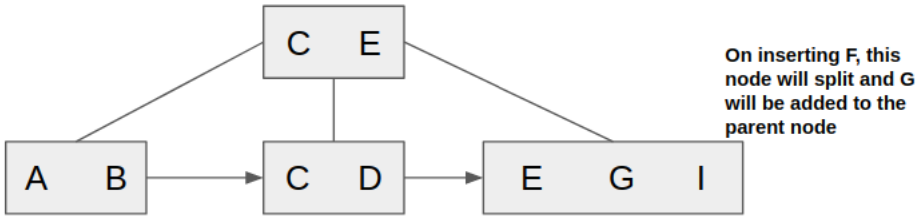
Insert I



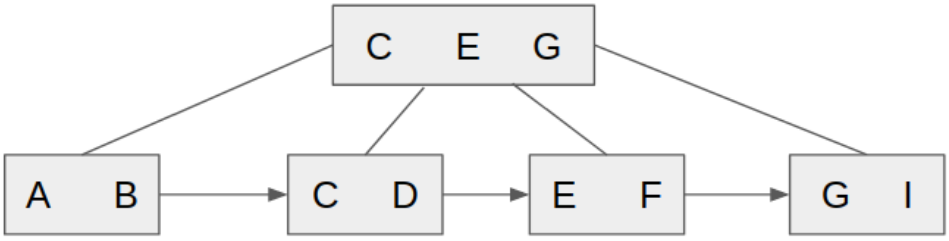
Insert B



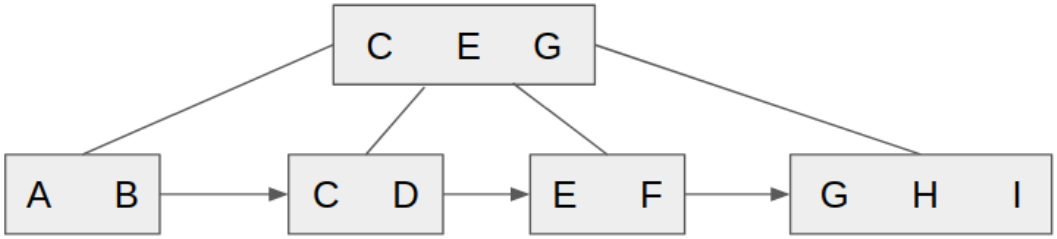
Insert D



Insert F

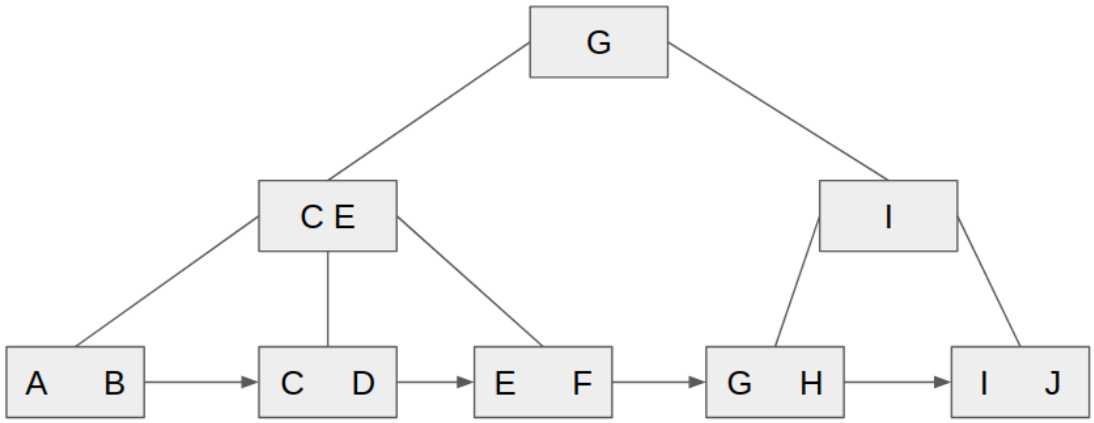


Insert H

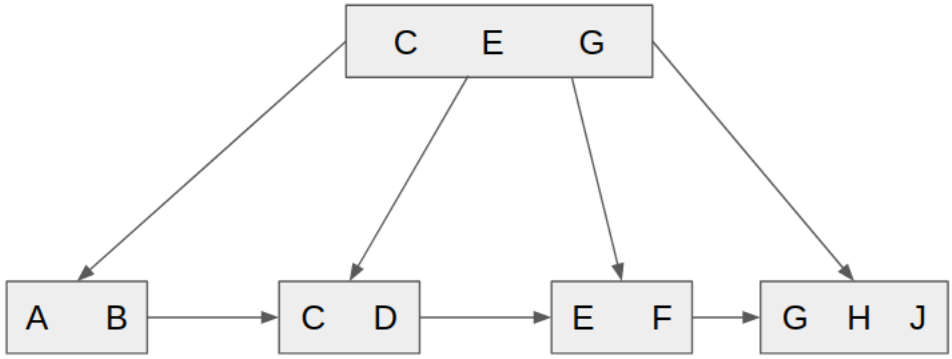


Insert J

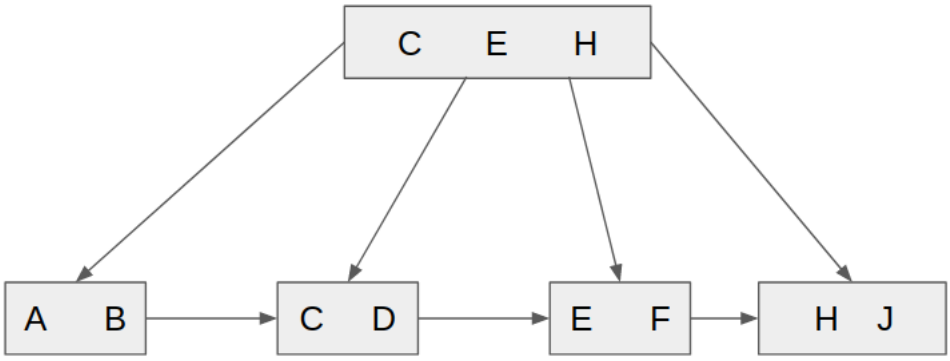
On inserting **J**, the node with <**G, H, I**> will split and **I** will be pushed to the parent node, but then the parent node will have four keys, so it will split too, this will push **G** again upwards to create a new parent node.



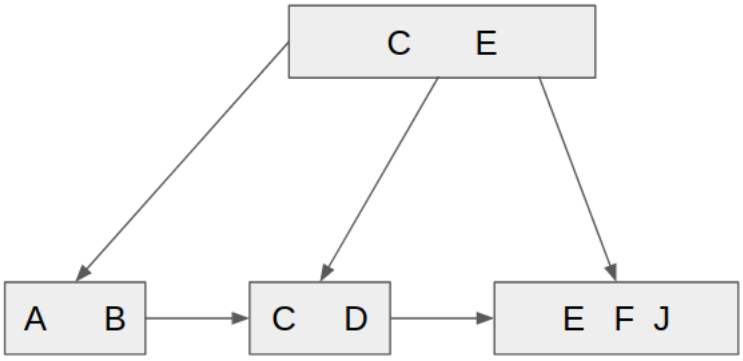
(b) **delete I**



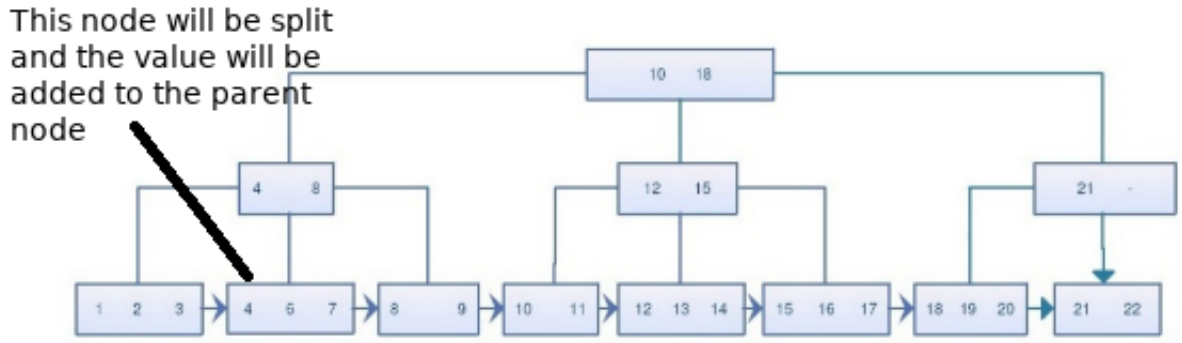
delete G



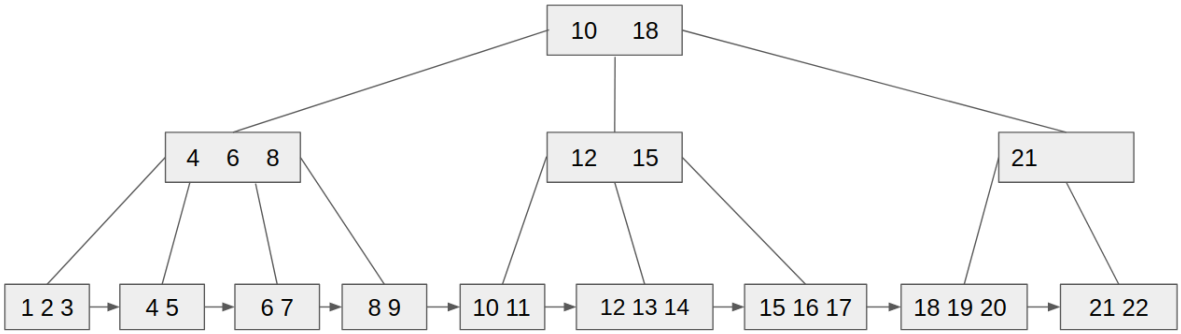
delete H



3.



After inserting data entry with key 5 into the tree it would like the following:



4. (a) Index records have the search key value and the assciated pointer, so the size of an index records would the sum of the sizes of the two, i.e. $45 + 15 = 60$ *bytes*. The number of pointers in a single node would be the same as the number of index records that can be stored in a single disk page. We have 30,000 records and the size of a disk page is 1500 bytes. So, the number of index records in a single node is 1500/60=25, but leaf nodes need extra space for next node pointer so we can have 24 records per node. Since the nodes at each level are filled as much as possible, the number of levels in the B+ tree is $ceil(log(30000/24)) = 4$

   (b) Level 1: 1 node (root)
   Level 2: 3 nodes
   Level 3: 53 nodes
   Level 4: 1250 nodes

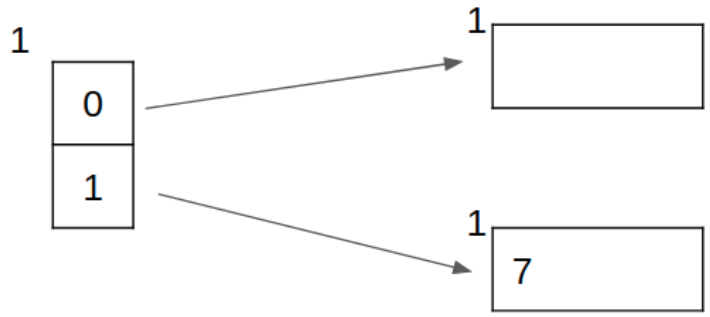
   (c) If the size of key is reduced to 10 bytes then we could have 59 index records in each node. So the number of levels in the B+ tree would be $ceil(log(30000/59)) = 3$.

   (d) If each page in only 70% full then the number of index records in each node would be 17 since $17 * 60 + 10 < 70 * 1500/100 < 18 * 60 + 10$. So the number of levels in the B+ tree would be $log(30000/17) = 4$

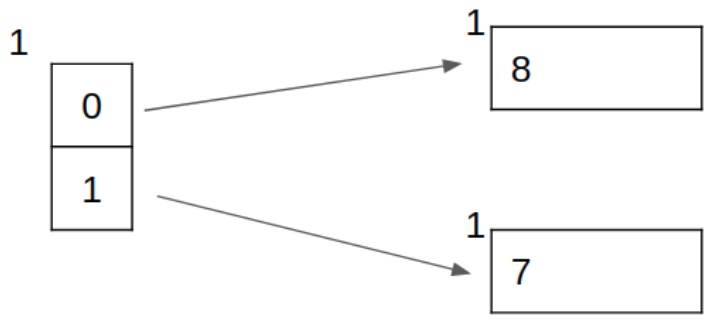
5. The algorithm used for extendible hashing is the one found here.

We start of with the global depth equal to 1. We have two buckets initially and both have local depth equal to 1.
We use the LSBs of the hashed value to compute the bucket into which the value has to be stored.
The following images show in a step wise manner the insertion of elements into the hashtable.

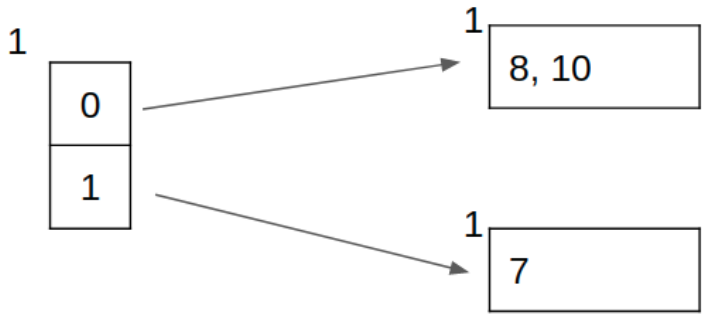
Insert 7



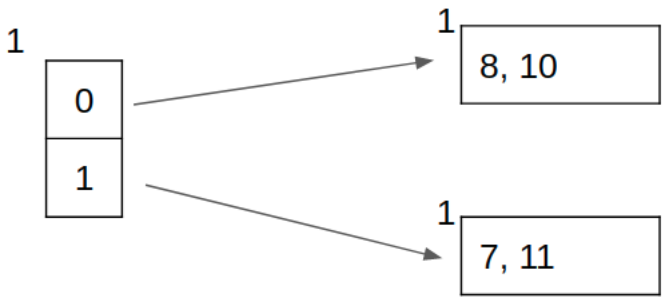
Insert 8



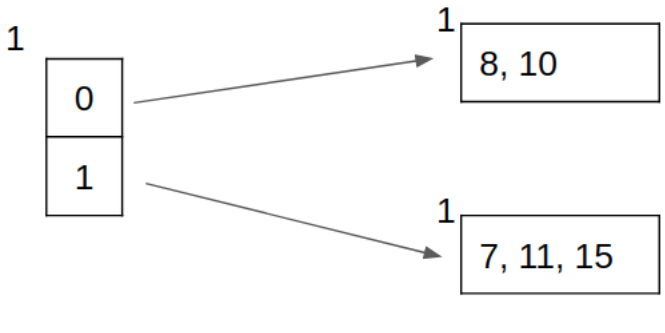
Insert 10

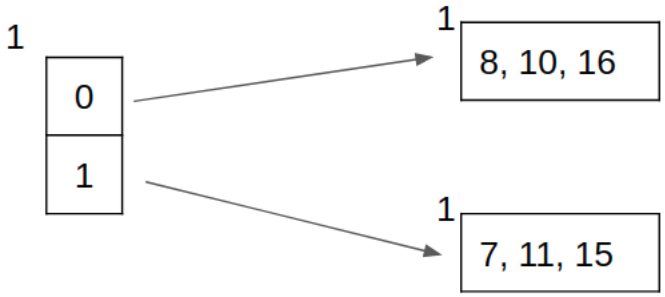


Insert 11

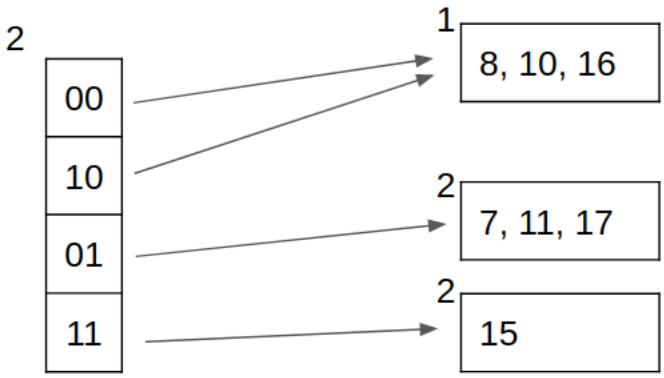


Insert 15, 16

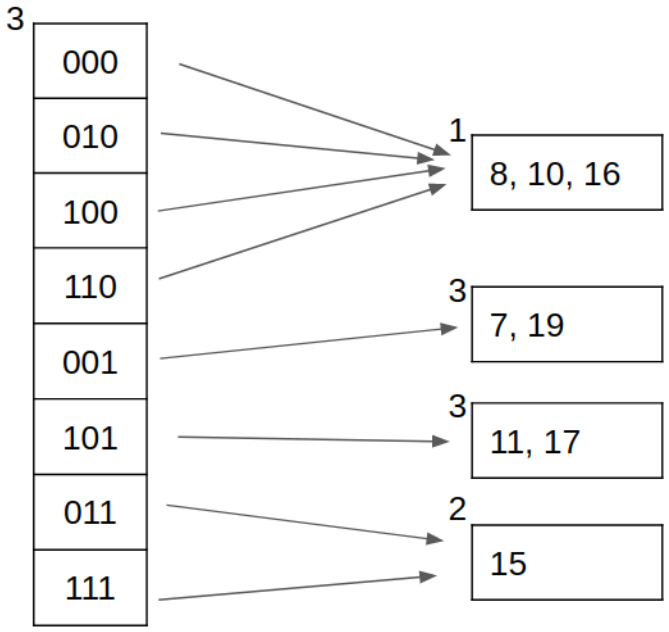




On **inserting 17**, there will be a bucket split in the second bucket. Global depth will become four. Now we will have, two with depth two and one with depth one.



On **inserting 19** once again there will be a bucket split in the second bucket.



On **inserting 33, 53** there will be no split, but on **inserting 58**, we will have to split the first bucket. Then all the values will have to be rehashed and we would consider the last two bits of the values to decide which bucket to put them in.