Exercise No. 7

Probability, Skip List, B-tree

Question 1

Let A[1,…,n] be an array of n distinct numbers. If i < j and A[i] > A[j] then the pair (i, j) is called an **inversion** of A. Suppose that the elements of A from a uniform random permutation of (1,2,..,n). What is the expected number of inversions in A?

Solution:

Let X_{ij} be an indicator random variable, for each $1 \le i,\ j \le n,\ \ i < j$.

$$X_{ij} = \begin{cases} 1, & if\ A[i] > A[j] \\ 0, & otherwise \end{cases}$$

Define $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$. X is a random variable counting the number of inversions in A. Hence, the expected number of inversions of A is: $E[X]$

$$E[X] = E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E\left[X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2}$$

We have $\Pr[X_{ij} = 1] = \frac{1}{2}$, because given to distinct random numbers from the {1,…n}, the probability that the first is bigger than the second is $\frac{1}{2}$ (Show the exact calculation).

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} = \binom{n}{2} \frac{1}{2} = \frac{n\,(n-1)}{4}$$

Skip List	
Definition	A skip list is a probabilistic data structure where elements are kept sorted by key. It allows quick search, insertions and deletions of elements with simple algorithms. It is basically a linked list with additional pointers such that intermediate nodes can be *skipped*. It uses a random number generator to make some decisions.

Skip Levels	• Doubly Linked lists $S_1..S_h$, each starts at $-\infty$ and ends at ∞ • Level S_1 - Doubly linked list containing all the elements in the set S • Level S_i is a subset of level S_{i-1} • Each element in Level i has the probability 1/2 to be in level i+1, thus if there are n elements in level S_1, the expected number of elements in level S_i is $n/2^{i-1}$. • The expected number of levels required is $\log_2 n$.
Time Complexity	• Search – O(logn) **expected** • Insert: search and then insert in O(log n) time – O(log n) **expected** • Delete search and then delete in O(log n) time – O(log n) **expected**
Memory Complexity	O(n) **expected**

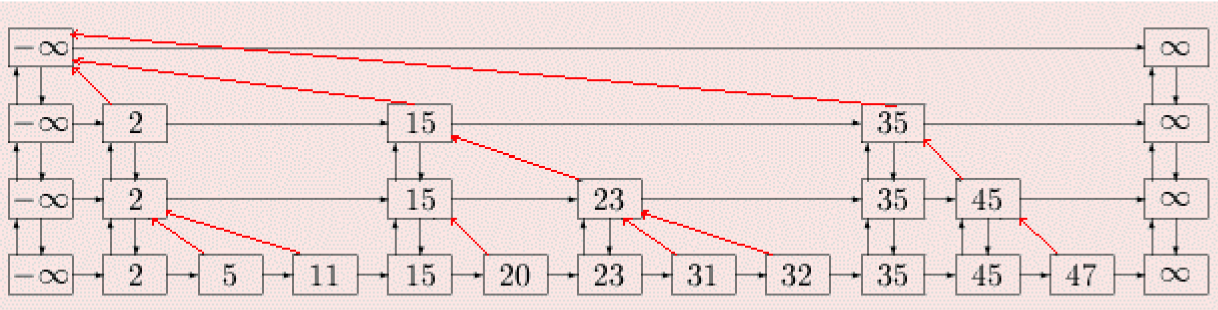
<u>Question 2</u>

Suggest a way to use a skip list to contain numbers in the range [*1..n*] (*n* is a not constant) and supports the following query:

- Given a pointer to element *x*, *x.key=i*, and $j < i$, find the element *y*, such that *y.key= j* in O(*log k*) expected time, where *k* is the distance between *x* and *y*.

Solution:

Skip list with the numbers [1, 2...n] in the lowest level S_1 and ($-\infty$, ∞) in the highest level S_h.
To each element, add a pointer up, to point at the parent in the level above.
Start with the given pointer to element x in level S_1, use the 'up' pointers in order to move to a higher level, until the current element's key is $\leq$ j and it's next element's key is > j.
From this point start to search the element j in a the smallest segment in the Skip List, that includes both elements x and y.
The additional pointer 'up', marked in red:



Complexity: The part of the skip list, where we perform the operations mentioned above is of an approximate size of k. It can be a bit bigger than k in case we found a node with *key<j*. Therefore, we'll assume that the list is of size *2k* in average (just in case). The part of skip list based on the *2k* is a tree with height *O(logk)*. So the search on the part of the skip list is like performing the operation on a skip list of size *O(k)*. Thus, the search time is *O(logk)* time.

Question 3

Describe an algorithm *select(S,k)* to find the *k*-th sized element in a skip list S with *n* elements. You can add a field to each node of S. The average time of the algorithm should be O(*logn*).

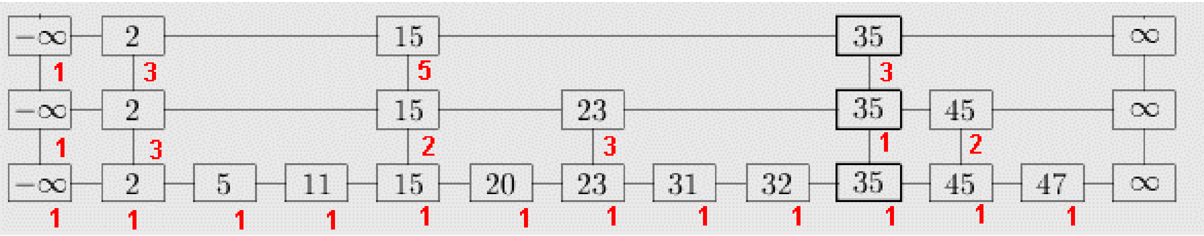
Solution:

Each node *p* in level S_i will have an additional field *dis(p)* - the number of nodes in level S1 from p to the next(p) in level S_i. Note that in order to get to the k-th element, we need to skip k elements starting from -∞.

```
Select(S, k)
  p ← leftmost and upmost node of S
  pos ←  0
  for i ← h (the height of S) downto 1
     while (pos + dis(p) ≤ k)
          pos ← pos + dis(p)
          p ← next(p)
      if (pos = k)
        return p //return the basis of p's tower
     else
        p ← below(p)
```

Complexity: O(*log n*) .

Example: When searching for the 7-th key (31) the search path is : -∞, next to 2, next to 15, down to 15, next to 23, down to 31.



Question 4

Write an algorithm that builds a skip list *S* from the given BST *T* with *n* elements, such that the worst query time in S will be *O(log n)*. *T* can be unbalanced. The time complexity of the algorithm should be *O(n)*.

Solution:

```
Build(T, S)
  S_1 ← inorder(T)
  int i ← 1
  while (i<log n)
    for j ← 1 to |S_i|
      if (j mod 2 = 0)
        S_{i+1}.add(S_i[j])
        S_{i+1}[|S_{i+1}|].setChildPtr(S_i[j])
    i ← i+1
```

Time Complexity: The inorder traversal is *O(n)*. The running time of the rest of the algorithm is linear in the number of elements in the skip list, that is *O(n)*. The worst query time in such a skip list is *O(log n)*. This question demonstrates how to construct a deterministic skip-list from an ordered set of *n* keys in *O(n)* time.

B-Trees

B-Tree Properties	1. Each node x has the following fields: ◦ n_x - the number of keys in X ◦ leaf(X) - true if x is a leaf and false otherwise 2. If X is an inner node with n_x keys $(K_1,\dots K_{nx})$ in ascending order, X has n_x+1 children $(C_1,\dots C_{nx+1})$ 3. If k_i is the i'th key in X, then all keys of C_i are smaller than k_i, and all keys of C_{i+1} are larger than k_i 4. All the leaves are in the same level 5. t = the minimal rank of a B-Tree Each node except the root, has at least t-1 keys and at most 2t-1 keys
Motivation	B-Trees are used when the data size is extremely big and can't be saved in the main memory but in a secondary memory (hard disk ...). Reading from a disk is relatively slow, but B-Trees insure that the number of disk accesses will be relatively small.
Theorem	If T is a B-Tree with $n \geq 1$ keys then $\Rightarrow$ $height(T) \leq \log_t\left(\frac{n+1}{2}\right)$
Insert(k) (Intuition)	Insert in leaf only. Go down from the root to the leaf into which the new key *k* will be inserted while splitting all full nodes on the path.
Delete(k) (Intuition)	Go down from the root to a node containing k while making manipulations on the tree to ensure that the current node X (except the root node) on the search path has at least t keys (the ancestor of X may have only t-1 keys after the manipulations on X). Thus, when the Delete function gets to a node containing k, the node will have at least t keys. **Delete (Node X, Key k)** 1. If node X has less than t keys (t-1 keys) a. if node X has a sibling Y with t keys: lend one key from it (key k goes to the father node of X and Y, replaces key k' such that X and Y are its child pointers and the key k' is added to X). b. node X has both siblings with only t-1 keys: merge X and one of its siblings Y, while adding the key k from the father node

	as a
	median key (X and Y are child pointers of k), into new node W,
	remove k and pointers to X and Y from the father node, add
	pointer to the newly created node W to the father node of X
	and Y.
	
	2. if k is in X and X is an internal node
	a. if the child node Y that precedes k in X has at least t keys:
	-k'=**Max**(Y) //find maximal key k' in subtree rooted in Y
	-Delete(Y, k')
	-replace k with k' in X
	b. symmetrically, if the child node Z that follows k in X has at least t keys
	-k'=Min(Z) //find minimal key k' in subtree rooted in Z
	-Delete(Z, k')
	-replace k with k' in X
	c. both Y and Z have t-1 keys
	-merge Y, k and Z into one node W
	-delete k from X and pointers to Y and Z and add instead
	pointer to W
	-Delete(W,k)
	3. else if k is in X and X is a leaf node
	a. delete k from X
	b. return
	4. else
	a. find child node Y_{i+1} of X such that $key_i(X)<k<key_{i+1}(X)$ // a pointer between $key_i(X)$ and $key_{i+1}(X)$ points to Y_{i+1}
	b. Delete (Y_{i+1}, k)

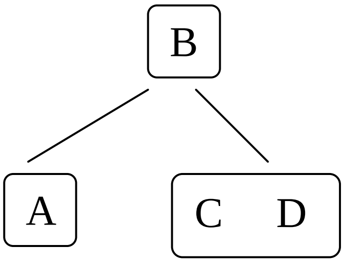
Question 5

Assume that t=2. Draw the B-tree that will be created after inserting the following elements (in this order) A,B,C,D,G,H,K,M,R,W,Z.

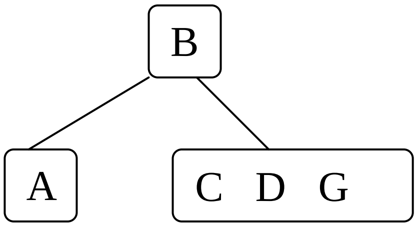
Solution:

After a node has 4 elements, it will be split (in here B becomes the root, and the rest of the elements are in its sons).

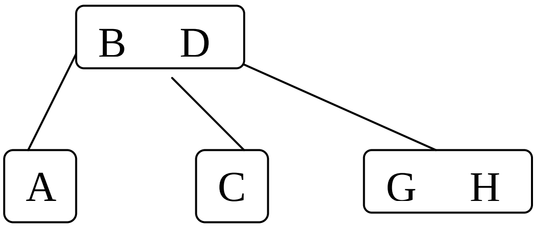
adding A, B, C, D:



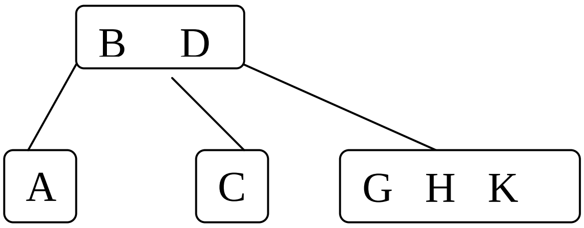
adding G:



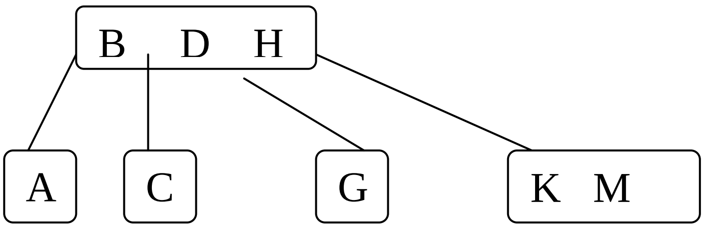
adding H (C D G H is split. D joins its father):



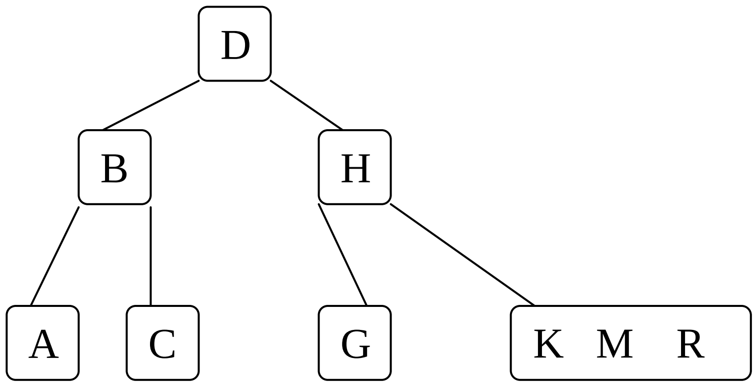
adding K:



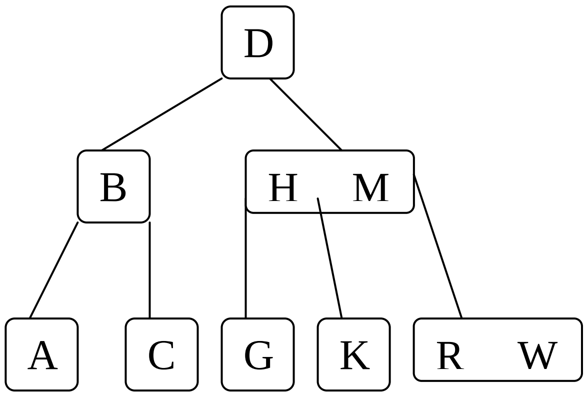
adding M:



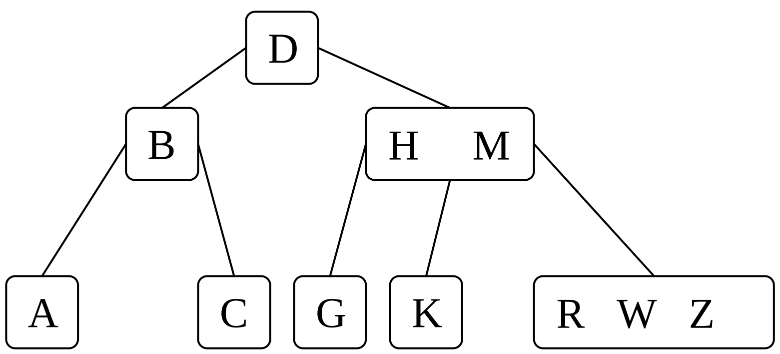
adding R (BDH is split. The tree is one level higher):



adding W (K M R W is split, M joins H):

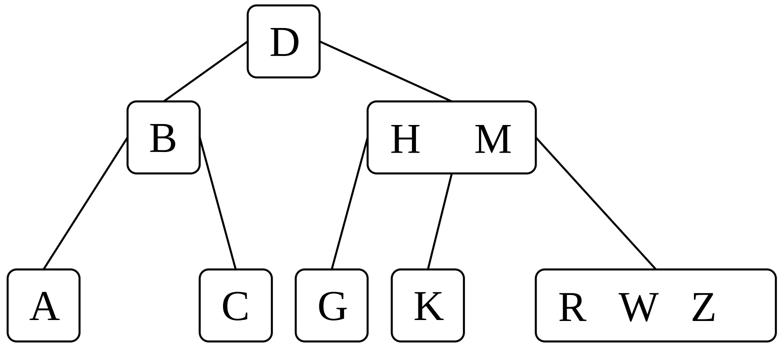


Adding Z.



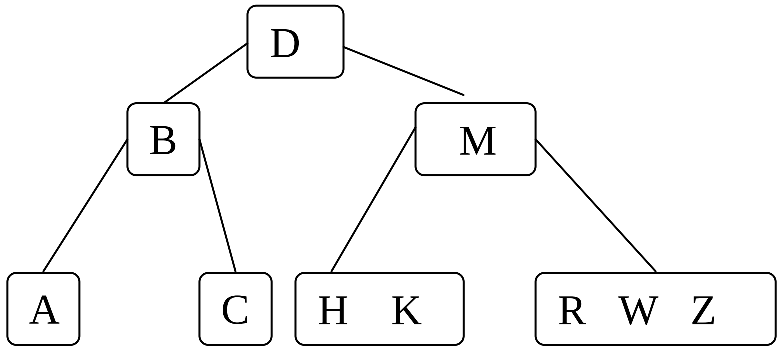
Question 6

Assume that t=2. Draw the tree that will result from deleting the element G, and then M.



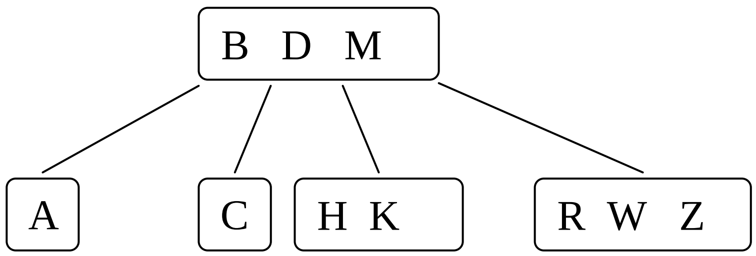
Solution:

Delete G: root node state is not relevant, node HM has 2 (=t) values, node G has t-1 keys, execute 1b: the leaf node will have GHK, delete G from the leaf node)



Deleting M :

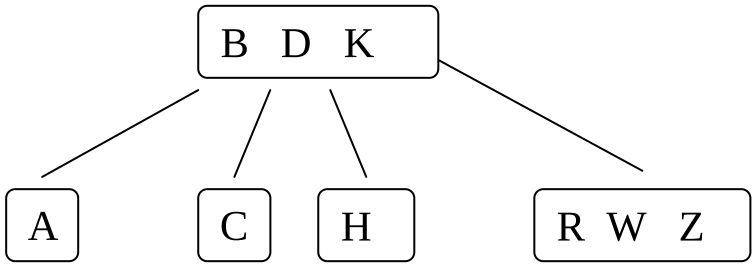
1. execute 1b:



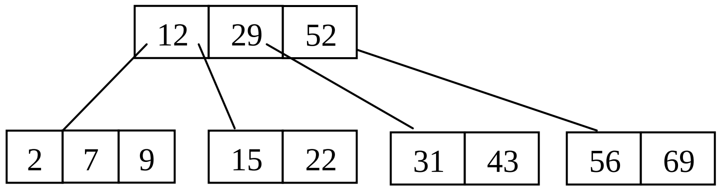
2. execute 2a:
    a. find a maximal key in the **left** subtree of M (this is K).

**We will always try first to ``take'' a key from the left sibling or a left subtree, and only if it's impossible, ``take'' the key from the right.*

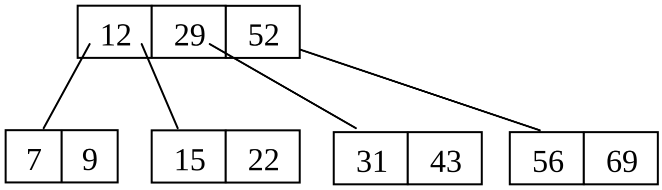
b. delete K from the leaf node HK (then number of keys in the root node is irrelevant and HK has t keys, thus we can delete it right away).
c. K replaces M.



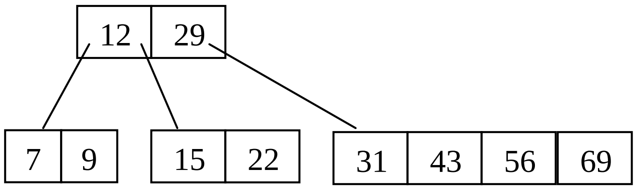
More examples of deletions:
t=3



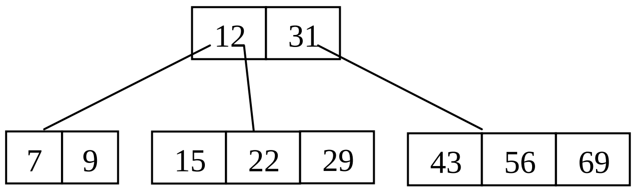
Delete 2:

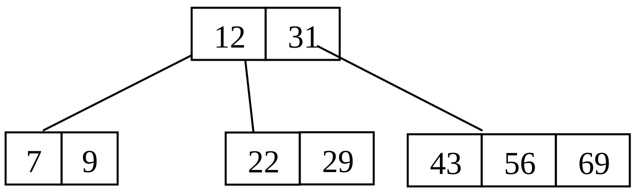


Delete 52 (2c):



Delete 15 (1a):





Question 7

Given two B-trees T_1 and T_2, both with parameter t=2. Each key in T_1 is smaller than each key in T_2. Suggest a way to efficiently merge T_1 and T_2 into a single B-tree T.

Solution:

First find $h(T_1)$ and $h(T_2)$ in $O(h(T_1)+h(T_2))$ time.
(*) Note that for t=2, the minimum keys limitation always holds.

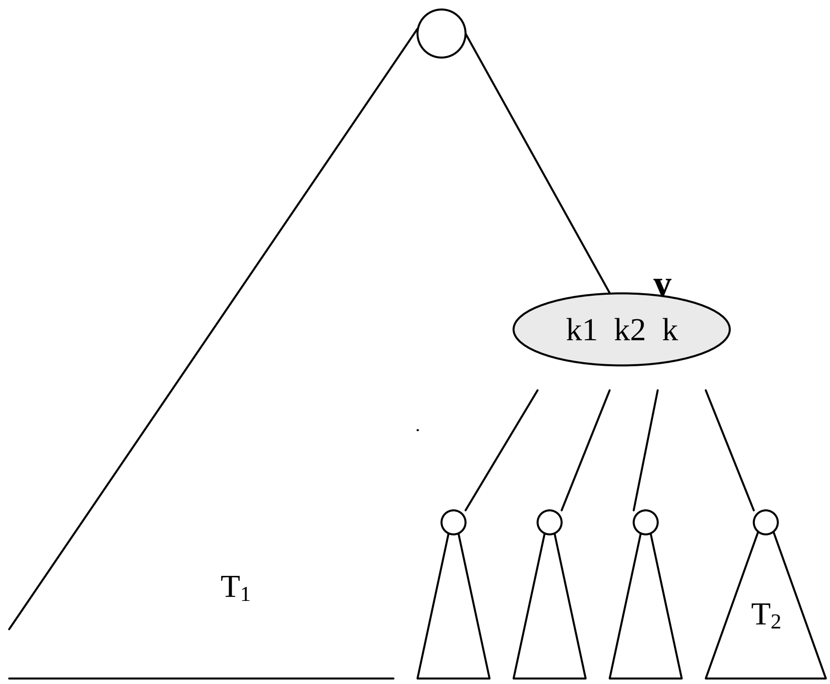
Case a: h(T1)=h(T2):

a. $k \leftarrow \frac{\text{maxKey}(T_1) + \text{minKey}(T_2)}{2}$
b. create a new root with the key k (with null record pointer)
c. T_1 will be the left sub-tree of k and T_2 will be the right sub-tree.
d. Btree-Delete (k)

Case b: $h(T_1) \neq h(T_2)$:
Without loss of generality assume $h(T_1) > h(T_2)$.

1. $k \leftarrow \frac{\text{maxKey}(T_1) + \text{minKey}(T_2)}{2}$
2. Like in Btree-Insert, go down on the rightmost path in T_1 (k is larger than all the keys in T_1), while splitting all full nodes. Stop at the node y, such that $h(y) = h(T_2)+1$. y is not full, i.e., has 2 keys at most.
3. Add k as the largest key in y
4. T_2 will be the right sub-tree of k in the node y.
5. Btree-Delete(k)

Running time: $O(h(T_1)+h(T_2))$



<u>Question 8</u>

B-Tree T has (10^5+1) keys. The maximal number of keys in a node is 17. How many disk accesses are required in the worst case while looking for a certain key?

Solution:

The number of disk accesses = the number of levels in T
The worst case is when the number of levels is maximal $\Rightarrow$
number of nodes is maximal $\Rightarrow$ number of keys in each node is minimal

The maximum number of keys in a node is 17 $\Rightarrow t = 9 \Rightarrow$ the minimum number of keys in each node other than the root is 8. Each node other than the root has at least 9 children.
In the worst case the root has only one key, thus 2 children.

Level 0: 1 node
Level 1: 2 nodes
Level 2: 2*9 nodes
Level 3: 2*9*9 nodes
...

Level i: $2*9^{i-1}$ nodes

The number of nodes: $1 + 2(9^0 + 9^1 + ... + 9^{h-1})$
The root has one key, every other node has 8 keys, therefore the number of keys is:

$1 + 8*2(9^0 + 9^1 + ... + 9^{h-1}) = 10^5 +1$
$2(9^h - 1) =10^5$
$9^h = 50001$
$h \approx 4.924 \approx 5$

$h = 5 \Rightarrow$ number of levels is 6 (we started to counting from level 0). .

You can use the theorem studied in class:
$h \le \log_t((n+1)/2)$ and get: $h \le \log_9((10^5+1)/2) \approx 4.92 \Rightarrow h =5$ (where tree height is the largest depth of a node in T) $\Rightarrow$ number of levels =6