Data Structures II

Homework 3

Question 1

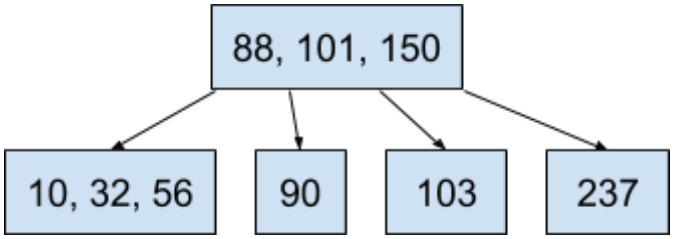
For a B-tree of order 3:

A. Each non root internal node can have between 2 and 3 children.
B. Each non root internal node can have between 1 and 2 keys.
C. It can have between $\Sigma_{i=0}^{h}(2^i) = 2^{h+1}-1$ and $\Sigma_{i=0}^{h}(3^i) = (3^{h+1}-1)/2$.

For a B-tree of order 20:

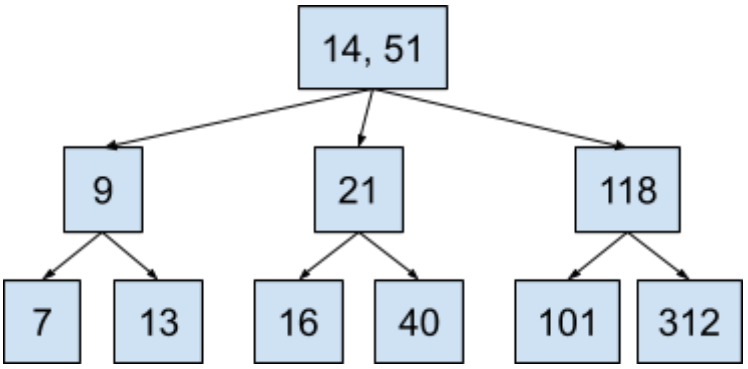
A. Each non root internal node can have between 10 and 20 children.
B. Each non root internal node can have between 9 and 19 keys.
C. It can have between $\Sigma_{i=0}^{h}(10^i) - 8 = (10^{h+1}-1)/9 - 8$ and $\Sigma_{i=0}^{h}(20^i) = (20^{h+1}-1)/19$

Question 2

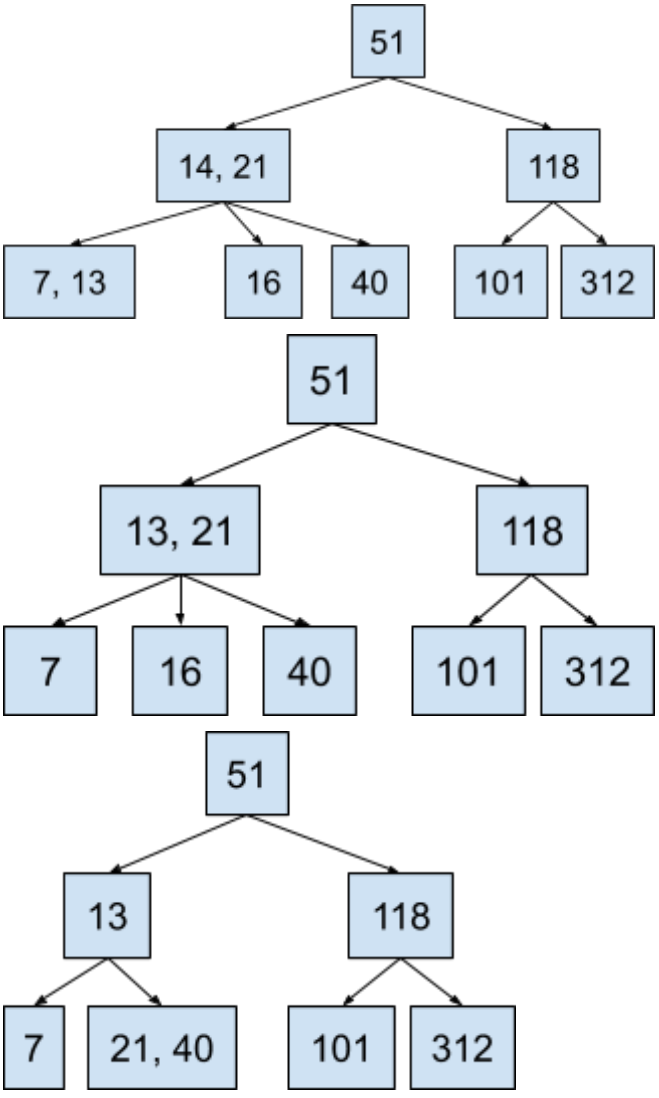


Question 3

A.



B.



Question 4

A. If the tree has the maximum number of keys in all the nodes from the root down to the leaf in which the new element would be inserted, then the leaf node, and every node above it would have to be split, each giving one node to its parent. Eventually the root would have one key too many, and it would have to be split into two nodes, making the middle node a new root. This would add a new level to the tree, increasing the height by one.

   Otherwise insertion will have no effect on the height of the tree.

B. If the leaf node where the new element is to be inserted is not full, then it would be added to that leaf, having no effect on the number of leaves.

   If the leaf node is full, then it will be split into two, causing the number of leaves to increase by one.

C. If the leaf node is not full, then the new element would be inserted into that leaf, so there would be no effect on the number of internal nodes.

   If the leaf node is full, it will be split, sending its median key to its parent. Each parent has the possibility to split and send an element up to its parent, all the way until the root. Therefore a single insertion has the possibility to increase the number of internal nodes by up to *h*, where *h* is the height of the tree.

Question 5

```
pair<node*, int> find_nth_element(int n) {
	if (leaf()) {
		if (num_keys < n) {
			return make_pair(NULL, num_keys);
		}
		return make_pair(elements[n - 1], 0);
	}
	int elem_index = 0, elements = 0;
	for (auto i = children.begin(); i != children.end(); ++i, ++elem_index) {
		pair<node*, int> p = (*i)->find_nth_element(n - elements);
		if (p.first) {
			return p;
		}
		elements += p.second + 1;
		if (elements == n && elem_index < num_keys) {
			return make_pair(elements[elem_index], 0);
		}
	}
	return make_pair(NULL, elements - 1);
}
```

The complexity of this algorithm is θ (lg(n) + k) where n is the number of nodes in the tree, and the k^{th} (in this case 4th) element is the one to be found. The function will call itself recursively until the smallest element is found. Since this element is guaranteed to be in a leaf, we have at least θ (lg(n)) to get to the leaf. From there, the algorithm will begin scanning all of its successors until k nodes have been found. This step is θ (k). In total we have θ (lg(n) + k).