CS240E S26	Tutorial 4	June 5

Q1. BST successor. We are given a binary search tree on n nodes, storing n distinct keys. We can list all keys in increasing order using in-order traversal in time linear in n.

The operation *successor*(x) returns the in-order *successor* of x in the tree, which is the node z such that $z.key > x.key$ and no ther keys are stored in between (or $\varnothing$ if such z does not exist), by traversing the path between the two nodes:

Algorithm 1: *successor*(n)

```
1  if n.rightChild == ∅ then
2  |  while n.parent != ∅ do
3  |  |  last ← n;
4  |  |  n ← n.parent;
5  |  |  if last = n.leftChild then
6  |  |  |  return n
7  |  |  end
8  |  end
9  |  return ∅;
10 end
11 if n.rightChild != ∅ then
12 |  n ← n.rightChild;
13 |  while n.leftChild != ∅ do
14 |  |  n ← n.leftChild;
15 |  end
16 |  return n;
17 end
```

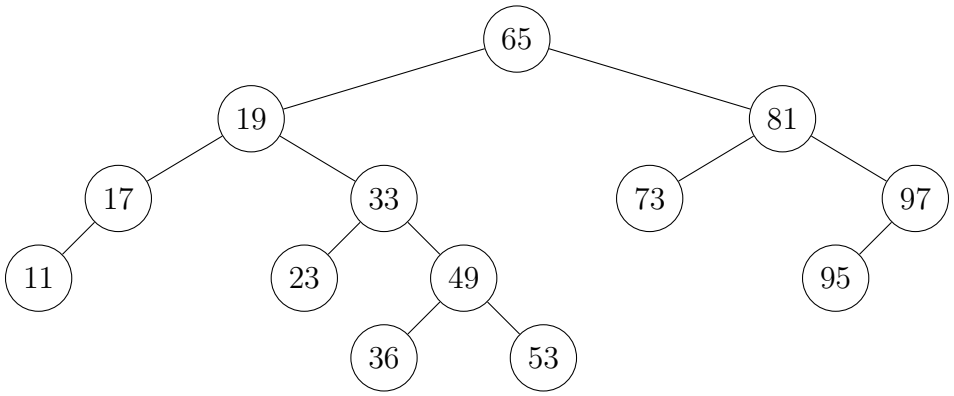
Show that the average-case runtime of successor is $\Theta(1)$. (Note that the binary search tree need not be balanced!)

Q2. (Balanced BST) Recall that a binary search tree is called *perfectly balanced* if for every node v we have

$$|v.\text{left.size} - v.\text{right.size}| \leq 1,$$

i.e., the size-difference between the left and right is as small as possible. Show that in any perfectly balanced binary search tree T, the leaves are only on the bottom two levels.

Q3. (AVL Rotations) Consider the AVL Tree shown below and perform `insert(61)`, then `delete(73)`.

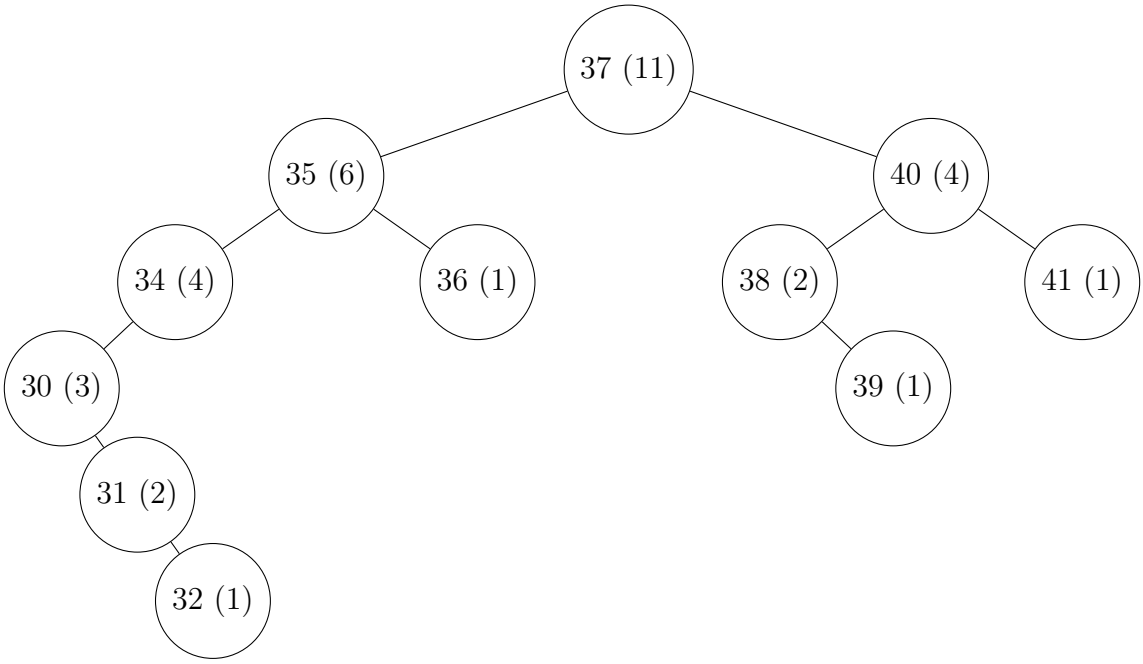


Q4. 2-AVL Trees

Define a 2-AVL tree to be a binary search tree where for every node, the difference of heights of its left and right subtree is at most 2. Prove that a 2-AVL tree has height at most $3 \log n$ where n is the number of nodes in the tree.

Q5. Scapegoat Tree Reconstruction.

Insert the key 33 in the following scapegoat (2/3)-tree. The numbers in the brackets are the number of nodes in that subtree.



Q6. (vEB tree) This question is based on CLRS, and past tutorial materials prepared by other ISAs. We may not have time to cover this question during tutorial 4. This may be delayed until tutorial 5.

A *van Emde Boas* (vEB) tree is a data-structure that supports operations search, insert, delete, successor operations (and their immediate derivatives) in worst-case $O(\log \log u)$ under the assumption that all keys come from the set $\{0, 1, \ldots, u-1\}$, which we call the *universe*.

In this question, we motivate and develop the vEB tree.

(a) For the following data structures with a fixed universe size, what is the time complexity of each of the `search`, `insert`, `delete`, and `successor` operations? You can assume $u = 2^{2k}$ for some appropriate k.

(i) An array of size u, each slot representing whether that integer is in the set.

(ii) A complete binary tree.

(iii) A tree of height 2, where each node has $\sqrt{u}$ children. Each non-leaf node contains a summary of its children - namely, if any single value is a 1.

(b) Consider (iii) from part (a). We note that the summary itself is a $\sqrt{u}$ bit array, and so is each of the children. Suppose $u = 2^{2^k}$ for convenience. What if we recursively apply the structure where each node of size n has $\sqrt{n}$ children each of size $\sqrt{n}$? (CLRS calls this a proto-vEB-tree.)

(i) How tall is the tree?

(ii) Consider the successor algorithm on the following page. Provide a recursive formula, and analyze its runtime. Note that high(x) gets the upper half of x's bits, while low(x) gets the lower half. We use $V[i]$ to indicate the ith cluster of V. Assume minimum takes $O(\log u)$ time - with the current structure, we can't do better.

Note that when returning (high(x), offset), we are returning the integer with high(x) as high bits and offset as low bits.

(iii) What about implementing the other operations, such as insertion? Deletion? Are there any issues?

(c) Suppose we also keep track of the minimum element and the maximum element. How can you improve on the operations, such as the example successor() operation, to get the desired $O(\log \log u)$ time complexity?

(d) What if we relax the restriction to $u = 2^k$? How do we still achieve the same height / time complexity?

Algorithm 2: *successor*(V, x)

```
1  if V.u == 2 then
2  |  if x == 0 and V[1] == 1 then
3  |  |  return 1;
4  |  end
5  |  return ∅;
6  end
7  offset ← successor(V[high(x)], low(x));
8  if offset ≠ ∅ then
9  |  return high(x), offset;
10 end
11 successor-cluster ← successor(V.summary, high(x));
12 if successor-cluster == ∅ then
13 |  return ∅;
14 end
   // assume minimum takes O(log u) time.
15 offset = minimum(V[successor-cluster]);
16 return successor-cluster, offset;
```

References

[CLRS] Cormen T., Leiserson C., Rivert R., Stein C. *Introduction to Algorithms.* Third edition.