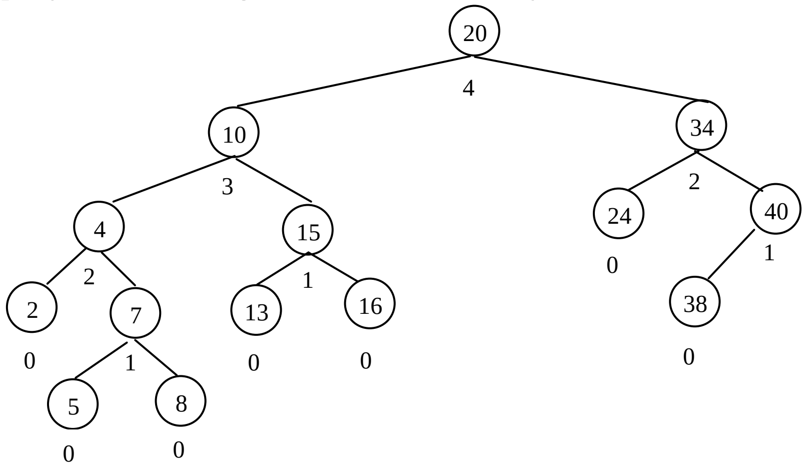
Quiz #5: AVL/N-ary/Digital Trees/More Heaps ICS-46 Fall 2016

Name ____________________

Recall the Academic Integrity statement that you signed. Write all answers clearly on these pages, ensuring your final answers are easily recognizable. The number of points for each problem is clearly marked, for a total of 25 points. I will post my solutions on the web on Friday, off the **Solutions** link, after class.

I have provided a project in which you can write and easily test your solutions (I've written a driver) to problems 2 and 3. **You must write all your answers here and turn in the project too.**

1. (12 pts) Examine the following tree, which has the BST/AVL order property. (a) Show that it has the AVL structure property: write the height of each node directly underneath the node.



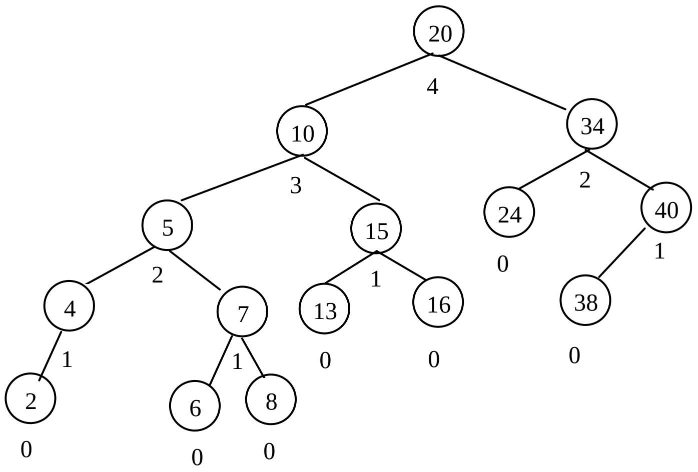
(b) Assume that we want to remove just **one leaf value** from the tree. List every **leaf** value that we could remove such that after its removal the tree is still an AVL tree without any rotations (don't actually remove any nodes or change the tree in any way; just list the nodes that can be removed).

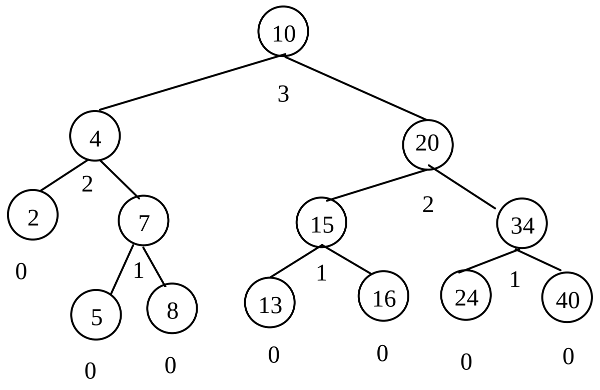
Values of leaves that can be easily removed: 5 8 13 16

(c) Assume we want to add one value to this tree. Circle the values can we add to the tree so it is still an AVL tree without any rotations: [✓ 1] 6 [✓ 19] [✓ 26] 39 [✓ 41]

(d) Insert the value **6** into the **tree shown above** and **show the entire resulting tree** after making the required rotation(s) to rebalance it (**include the heights of every node,** as you did above). **Do not show intermediate work; just show the final tree after rotation.** Show your answer on the left below.

(e) Remove the value **38** from the **original tree shown above** and … follow the same instructions as in part (d) and show your answer on the right below.





2. (4 pts) Given the class **NTN** below, fill in the function **maximum**, which is passed an **NTN** parameter, and returns the maximum **T** value in the N-ary tree it represents. Write the simplest code (mine uses both iteration and recursion): I also used two local variables. See the **q5helper**.

```
template<class T>
T maximum(const NTN<T>& root) {
  T max = root.value;{
  for (NTN<T> n : root.children) {
    T max_sub = maximum(n);
    if (max_sub > max)
       max = max_sub;
  }
  return max;
}
```

```
template<class T>
class NTN {
  public:
    T                 value;
    ics:ArraySet<NTN> children;
  ...
//A leaf node's children stores
//  a reference to an empty set
//See solution5.hpp for more details
```

3. (5 pts) Given the class **DTN** below, fill in the method **longestWord**, which is passed a **non-empty DTN** parameter, and returns the longest word **String** in its Digital Tree. Write the simplest code (mine uses both iteration and recursion): I used two local variables. See the text notes and **q5helper** for details about Digital Trees. All leaf nodes in the Digital Tree will represent words: e.g., in leaves **is_word** is **true**.

```
std::string longest_word (const DTN& root) {
  std::string longest = (root.is_word ? root.word_to_here : "");
  for (ics::pair<char,DTN>  c : root.children) {
    std::string check = longest_word(c.second);
    if (check.size() > longest.size())
      longest = check;
  }

  return longest;
}
```

```
class DTN {
  public:
    bool                    is_word;
    std::string              word_to_here;
    ics::ArrayMap<char,DTN> children;
}
//See box above: Note/A leaf …
//See solution5.hpp for more details
```

4. (4 pts) Suppose that we want to write a special **en_de_queue** method in a **max-heap** implementation of a priority queue. Its semantics are equivalent to first **enqueuing** its parameter to the heap and then **dequeuing** and returning the highest priority value from the resulting heap. In fact, we could implement **en_de_queue** trivially: by (a) calling the **enqueue** method first, which adds the value at the end of the array and then percolates it upward; and then (b) returning the result of calling the **dequeue** method second, which saves the value at the front of the array, then takes the value at the end of the array and moves it to the front, then percolates it downward, and finally returns the value saved from the front. It does two percolations.

But by thinking carefully about what will happen, we can write an **en_de_queue** method using a more efficient algorithm: still the same complexity class **O(Log N)**, but not as much work percolating values. State the better algorithm for **en_de_queue** in terms similar to how I stated the algorithm in the previous paragraph.

Faster algorithm description for the **en_de_queue** method (be specific/cover all cases); state how it is faster.

(a) If the heap is empty or the new value is ≥ the top, return the new value with no changes to the heap;

(b) otherwise, save the value at the front of the array, put the new value at the front of the array, percolate it down, and finally returned the saved value.

This approach at worst calls only percolate down (not percolate up and down) in (b) and maybe neither in (a).