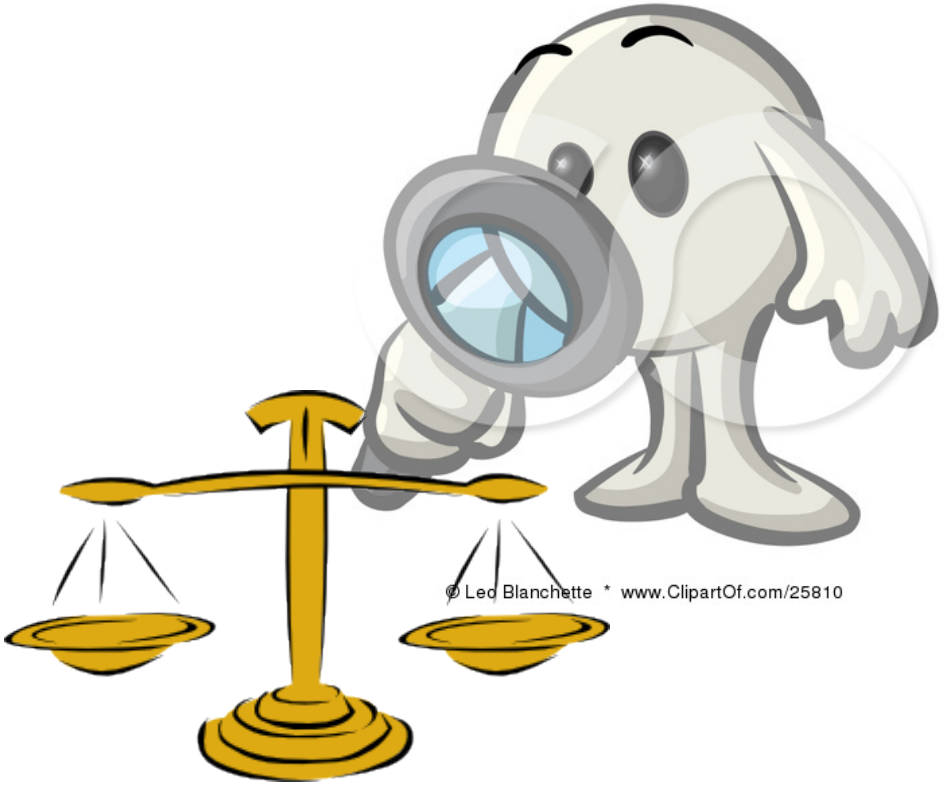
Balanced BST and AVL Trees



Carlos Moreno
cmoreno@uwaterloo.ca
EIT-4103

https://ece.uwaterloo.ca/~cmoreno/ece250

These slides, the course material, and course web site are based on work by Douglas W. Harder

Balanced BST and AVL Trees

Standard reminder to set phones to silent/vibrate mode, please!

Balanced BST and AVL Trees

- Last time on this topic:
  - Introduced AVL trees
  - Discussed some of its properties, emphasizing its height-balance attribute.
  - Looked into re-balancing techniques, necessary after insertions or removals.
  - Argued that both insertion and removal operations take $\Theta(\log n)$ time, since the re-balancing is done in constant time.

Balanced BST and AVL Trees

- During today's class:
  - We'll continue on the topic of AVL trees
    - Look more in detail on whether (or how) re-balancing can really be done in constant time.
    - Look into removals, and will verify that these, too, can be done in logarithmic time (though a little slower than insertions).
  - We'll discuss some details on how to use these Balanced BST, through an example / case-study of indexing a set of text documents.

Balanced BST and AVL Trees

- Last time, we claimed that:
  - These rotations take $\Theta(1)$, and the insertion takes $\Theta(h) = \Theta(\log n)$, so we're in good shape: insertion (including maintaining balance) takes $\Theta(\log n)$
- And I warned that there was a subtle detail that completely ruins that argument (the part of constant time for maintaining balance)

Balanced BST and AVL Trees

- Last time, we claimed that:
  - These rotations take Θ(1), and the insertion takes Θ(h) = Θ(log n), so we're in good shape: insertion (including maintaining balance) takes Θ(log n)
- And I warned that there was a subtle detail that completely ruins that argument (the part of constant time for maintaining balance)
- So ... Anyone?

Balanced BST and AVL Trees

- Last time, we claimed that:
  - These rotations take $\Theta(1)$, and the insertion takes $\Theta(h) = \Theta(\log n)$, so we're in good shape: insertion (including maintaining balance) takes $\Theta(\log n)$
- And I warned that there was a subtle detail that completely ruins that argument (the part of constant time for maintaining balance)
- So ... Anyone?
  - Hint: how do you know that you have to re-balance?

Balanced BST and AVL Trees

- We observe that this requires computing heights of *every* sub-tree!
  - A quick analysis tells us that this takes $\Theta(n^2)$ (right? *why?*)

Balanced BST and AVL Trees

- We observe that this requires computing heights of *every* sub-tree!
  - A quick analysis tells us that this takes $\Theta(n^2)$ (right? *why?*)
    - Computing the height for a tree takes linear time with respect to the size of the sub-tree, and there are $\Theta(n)$ sub-trees (arithmetic sum... again!)

Balanced BST and AVL Trees

- We observe that this requires computing heights of *every* sub-tree!
  - A quick analysis tells us that this takes $\Theta(n^2)$ (right? *why?*)
    - Computing the height for a tree takes linear time with respect to the size of the sub-tree, and there are $\Theta(n)$ sub-trees (arithmetic sum... again!)
- Go figure — in this context, linear time would be *horrible* news; and we're getting quadratic time!

Balanced BST and AVL Trees

- Ok, so what do we do about it? (because let's make it very clear: we *want* constant time for the re-balancing, and we really won't let it go until we have it our way !!)

Balanced BST and AVL Trees

- Ok, so what do we do about it? (because let's make it very clear: we *want* constant time for the re-balancing, and we really won't let it go until we have it our way !!)
  - Hint: there is this common trick of trading space for computational efficiency (trade-off, really: we use more storage to save computations, or do more work on the computations to save storage space)

Balanced BST and AVL Trees

- How about we store, at each node, the height of the sub-tree at that node?

Balanced BST and AVL Trees

- How about we store, at each node, the height of the sub-tree at that node?
  - If the heights don't change, this definitely works... but unfortunately, heights do change, so we're out of luck

Balanced BST and AVL Trees

- How about we store, at each node, the height of the sub-tree at that node?
  - If the heights don't change, this definitely works... but unfortunately, heights do change, so we're out of luck — right?

Balanced BST and AVL Trees

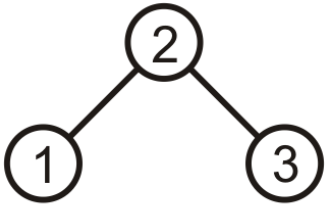
- How about we store, at each node, the height of the sub-tree at that node?
  - If the heights don't change, this definitely works... but unfortunately, heights do change, so we're out of luck — right?
  - No, not really: heights change with insertions or removals; but since only the heights of the nodes in the path from root are the ones that *can* change, then we can update all heights in logarithmic time (and the insertion or removal is already taking logarithmic time, so we're good).

Balanced BST and AVL Trees

- BTW ... what do I mean by “heights of the nodes in the path from root are the ones that *can* change”? (as opposed to “are the ones *that change*”)

Balanced BST and AVL Trees

- A simple example to illustrate: Removing 3 does not change the height at node 2; inserting 5 afterwards does not change the height at node 2 either:

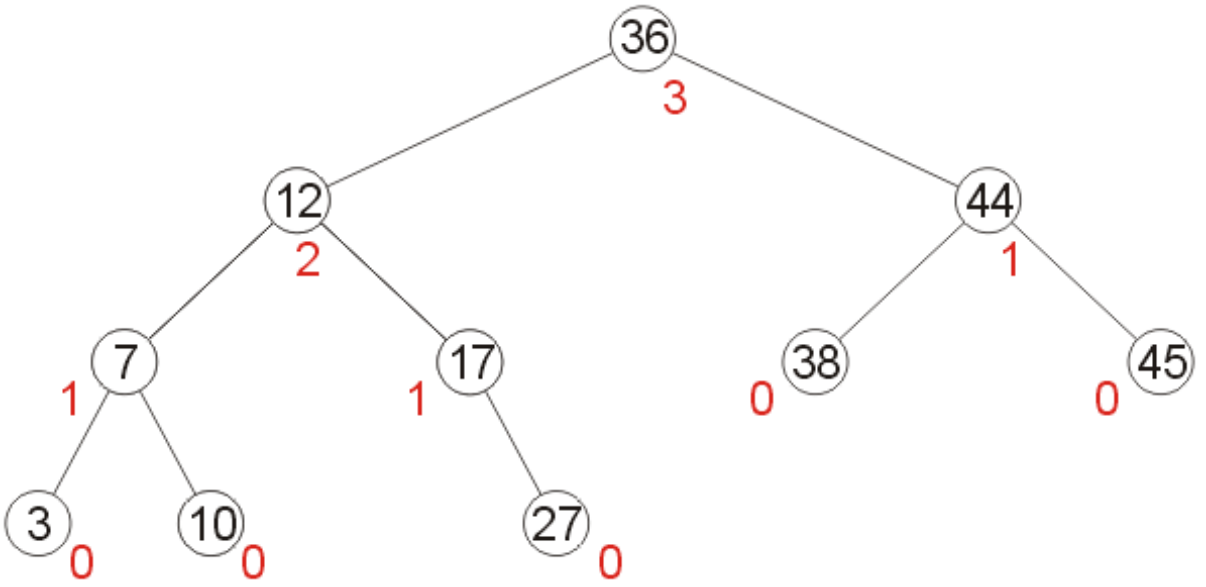


Balanced BST and AVL Trees

- Updating the height for each node takes constant time — just assign the higher of the current (stored) height and 1 + the new height for the sub-tree where the insertion or removal happened.
  - This is done for each node in the path from root, and there are $\Theta(\log n)$ of them.

Balanced BST and AVL Trees

- Example (storing the heights):



Balanced BST and AVL Trees

- Next, we'll take a look at removals...

Balanced BST and AVL Trees

- Removals are not that different from insertions:
  - After all, they can cause an imbalance, but only in the nodes in the path from root to the removed element — the imbalanced is fixed with the same rotations.

Balanced BST and AVL Trees

- Removals are not that different from insertions:
  - After all, they can cause an imbalance, but only in the nodes in the path from root to the removed element — the imbalanced is fixed with the same rotations.
  - However, the extra complication comes from the fact that the re-balancing of one sub-tree can cause an imbalance higher up!

Balanced BST and AVL Trees

- Removals are not that different from insertions:
  - After all, they can cause an imbalance, but only in the nodes in the path from root to the removed element — the imbalanced is fixed with the same rotations.
  - However, the extra complication comes from the fact that the re-balancing of one sub-tree can cause an imbalance higher up!
    - Is this intuitive / easy to visualize?

Balanced BST and AVL Trees

- Hopefully it is (easy to visualize) — we are reducing the height: if a removal caused an imbalance, it is because there was a difference of 1, and we removed the element from the upper branch of the sub-tree
  - Thus, when balancing, we reduce the height by 1, which can clearly cause an imbalance (if the sub-tree with root at the parent had already a difference of 1, with this one being the upper one, the removal would cause the difference to increase to 2 — i.e., causing an imbalance).

Balanced BST and AVL Trees

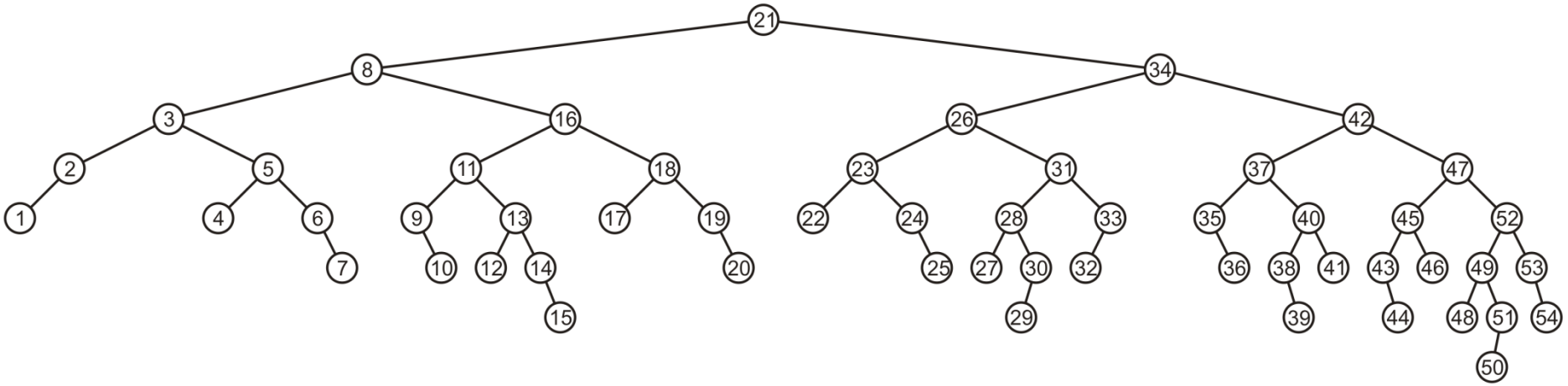
- The solution is quite obvious (right?):

Balanced BST and AVL Trees

- The solution is quite obvious (right?):
  - Continue checking all the way up to the root for imbalances after fixing each imbalance.
  - Let's look at an example...

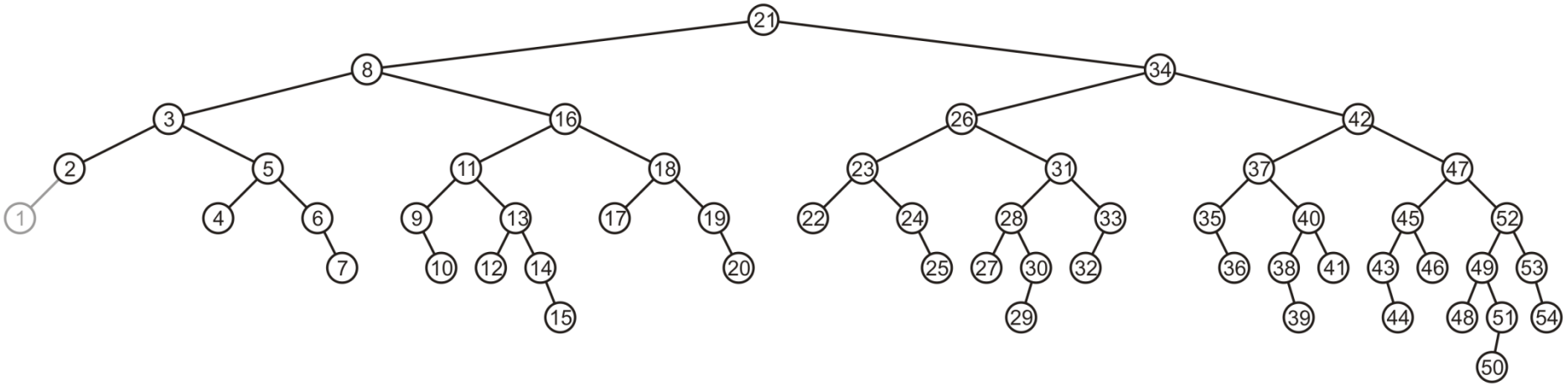
Balanced BST and AVL Trees

- Consider the following AVL tree:



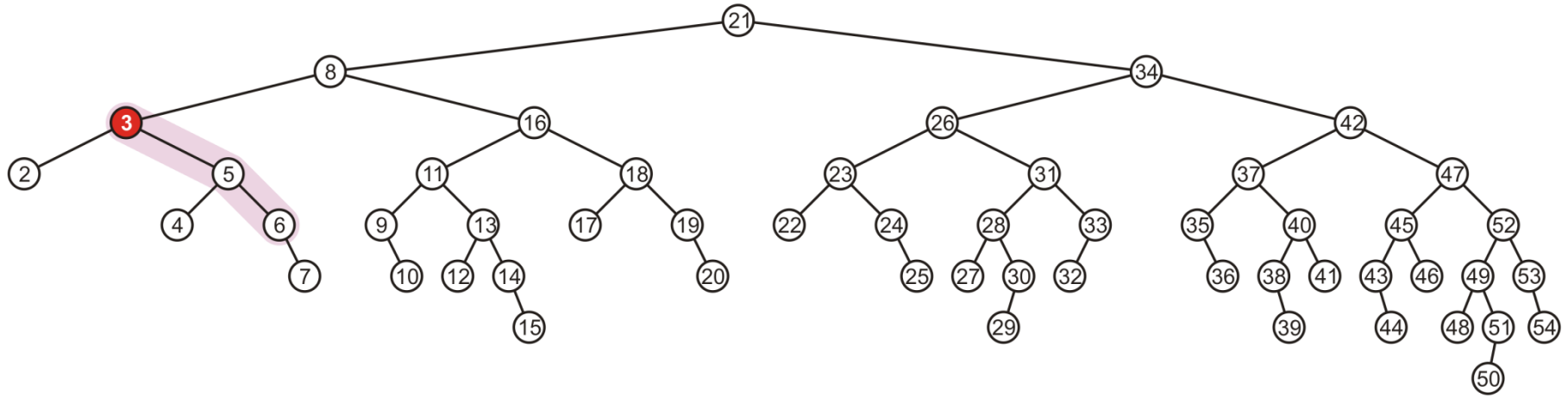
Balanced BST and AVL Trees

- Say that we remove the node 1:



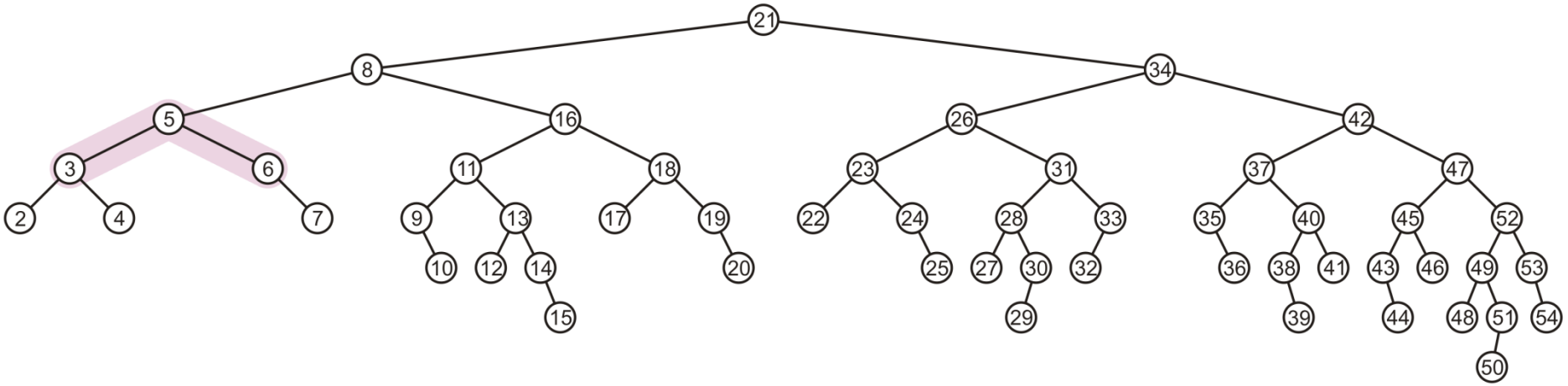
Balanced BST and AVL Trees

- So we do — causing an imbalance at 3 (its grandparent):



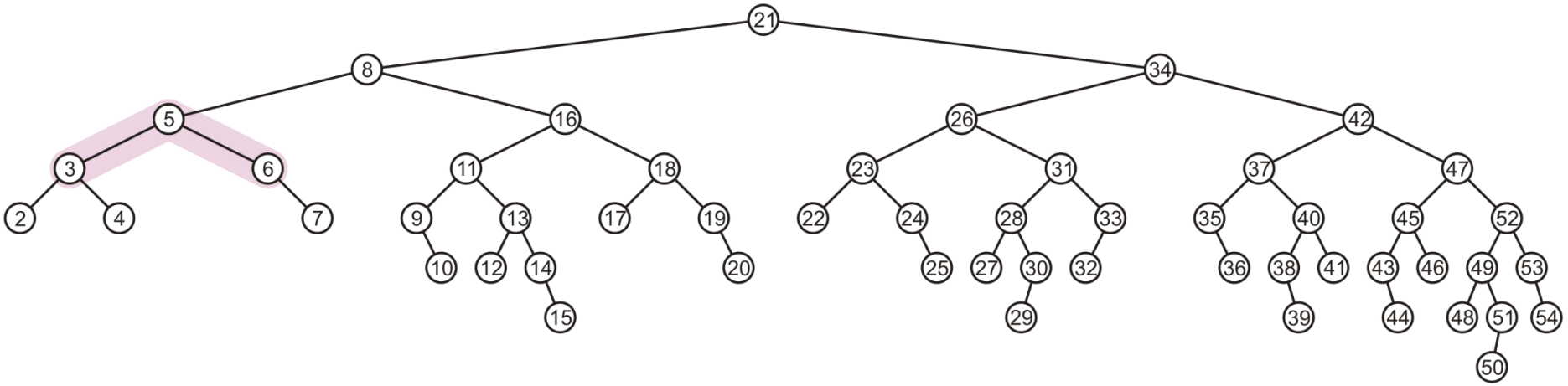
Balanced BST and AVL Trees

- So we fix it (with a single rotation, since it was a right-right imbalance):



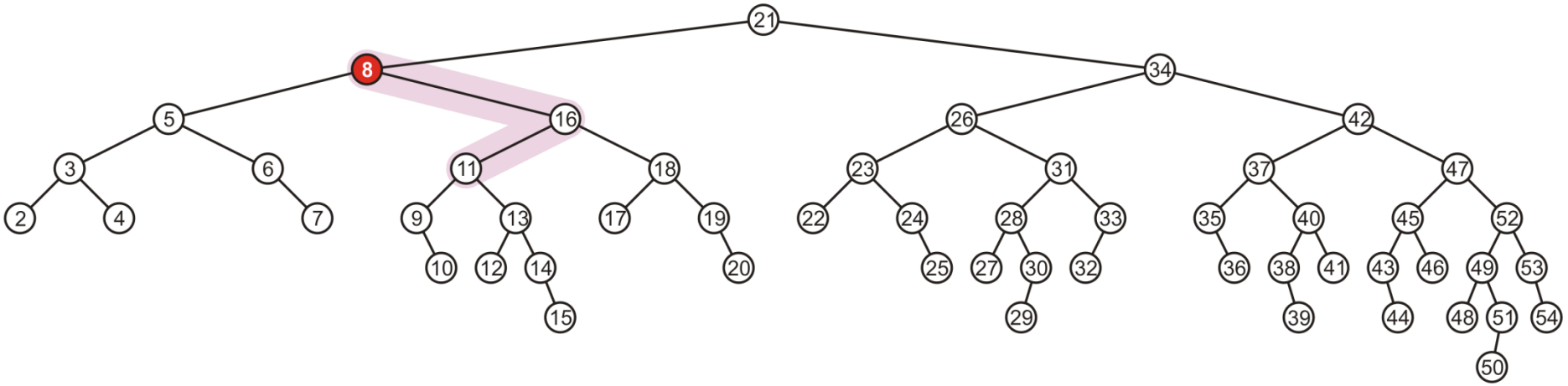
Balanced BST and AVL Trees

- Now we have balance at node 5, but the maneuver caused an imbalance at node 8.
  - So, we work on that one next.



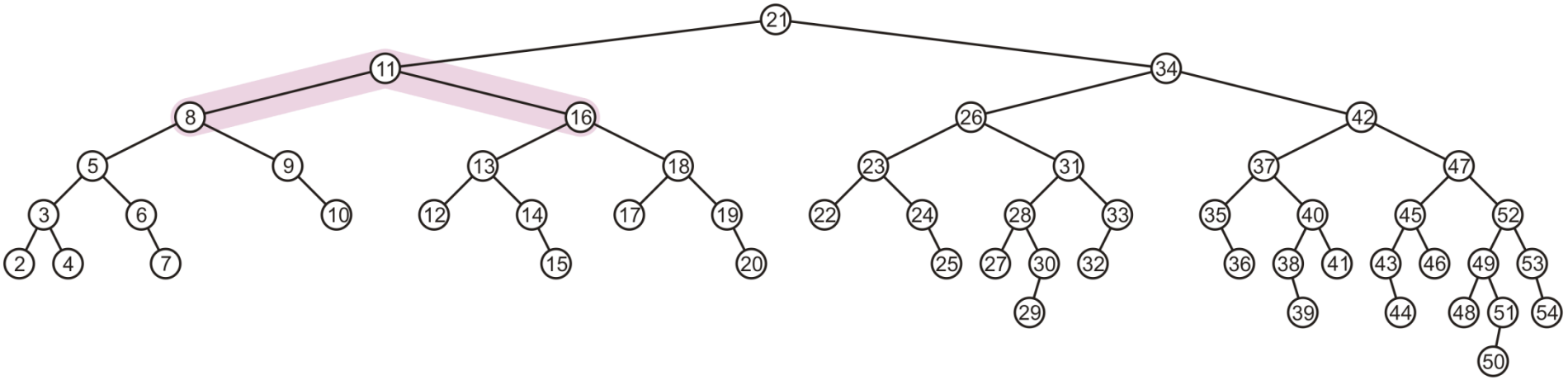
Balanced BST and AVL Trees

- Looking at the imbalance at node 8, we have a "zig-zag" type (right-left)



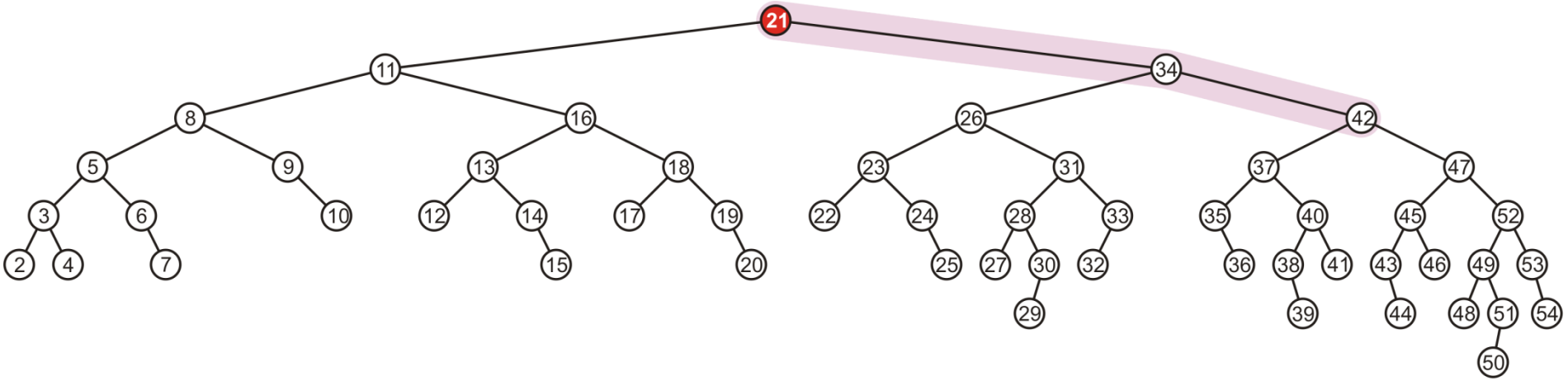
Balanced BST and AVL Trees

- So we fix it with a double rotation.
  - But this one causes now an imbalance at node 21 (the root node).



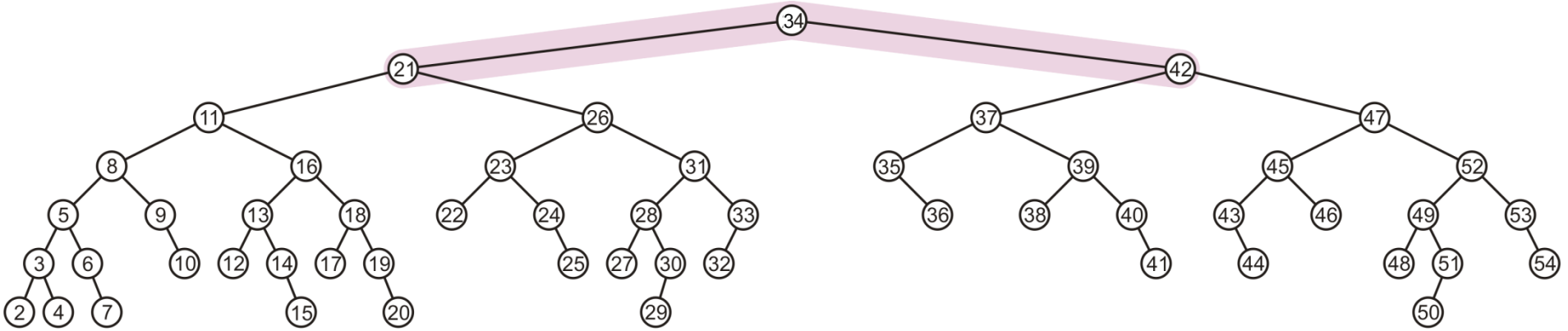
Balanced BST and AVL Trees

- So, we address our imbalance at the root:



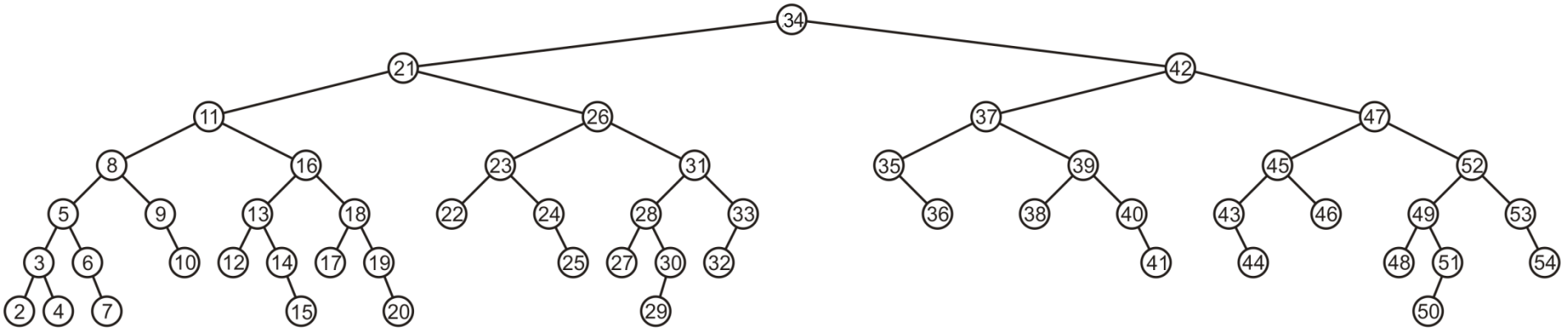
Balanced BST and AVL Trees

- Which is fixed with a single rotation:



Balanced BST and AVL Trees

- At this point, we're done — we have a valid (balanced) AVL tree:



Balanced BST and AVL Trees

- Next ... How about a small dose of *Reality*? :-)

Balanced BST and AVL Trees

- That is — how do we use these AVL (or any form of Balanced Binary Search Tree) for real-life applications?
  - These trees with just numbers are good for understanding how things work — but they won't get us too far in terms of real-life applications.

Balanced BST and AVL Trees

- So, as a case-study, a product that I'm working on...
- Meet YAUSE (I'm planning to implement this for real, and promote it and market it and become a, what, I guess we should say a trillionaire? Billionaires seem to be so common nowadays that there's no longer any cachet in it :-))

Balanced BST and AVL Trees

- My tag line (the “subtitle” or advertising slogan) for YAUSE will be:

Balanced BST and AVL Trees

- My tag line (the “subtitle” or advertising slogan) for YAUSE will be:
  - *Because all your searches deserve logarithmic time!*

Balanced BST and AVL Trees

- My tag line (the “subtitle” or advertising slogan) for YAUSE will be:
  - *Because all your searches deserve logarithmic time!*
  - Originally, I had phrased it as “deserve $\Theta(\log n)$ time” and with each word in a different color — but my marketing guy convinced me to switch.

Balanced BST and AVL Trees

- My tag line (the "subtitle" or advertising slogan) for YAUSE will be:
  - *Because all your searches deserve logarithmic time!*
  - Originally, I had phrased it as "deserve $\Theta(\log n)$ time" and with each word in a different color — but my marketing guy convinced me to switch.
  - I know this will sound hard to believe, but he still thinks that people will not understand the tag line; that it does not speak to the masses, etc. *sigh*

Balanced BST and AVL Trees

- My tag line (the "subtitle" or advertising slogan) for YAUSE will be:
  - *Because all your searches deserve logarithmic time!*
  - Originally, I had phrased it as "deserve $\Theta(\log n)$ time" and with each word in a different color — but my marketing guy convinced me to switch.
  - I know this will sound hard to believe, but he still thinks that people will not understand the tag line; that it does not speak to the masses, etc. *sigh*
    - Hopefully, I have here some 100+ people that will disagree with him? :-)

Balanced BST and AVL Trees

- BTW, in case you're wondering... YAUSE stands for *Yet Another Useless Search Engine*.

Balanced BST and AVL Trees

- BTW, in case you're wondering... YAUSE stands for *Yet Another Useless Search Engine*.

  For this part, my marketing guy did indeed convince me that it was better to remove this from the logo!

Balanced BST and AVL Trees

- BTW, in case you're wondering... YAUSE stands for *Yet Another Useless Search Engine*.

  For this part, my marketing guy did indeed convince me that it was better to remove this from the logo! My only guess is that maybe he didn't like the over-used theme of each word in a different color — don't know what else could he had objected to!

Balanced BST and AVL Trees

- Anyway ... Back to the serious stuff that we need to do (or I would never become a trillionaire!)

Balanced BST and AVL Trees

- The idea being to create an *index* of search keywords for a bunch of documents — literally, like the index at the end of a book that makes it possible for us to find out what page to go if we need info on some particular word.
  - The indexed words appear alphabetically (in *lexicographical* order), allowing us to do the equivalent of a binary search, and it works as an associative container — it associates keywords with a list of page numbers.

Balanced BST and AVL Trees

- The idea being to create an *index* of search keywords for a bunch of documents — literally, like the index at the end of a book that makes it possible for us to find out what page to go if we need info on some particular word.
  - The indexed words appear alphabetically (in *lexicographical* order), allowing us to do the equivalent of a binary search
  - This index works as an associative container — it associates keywords with a list of page numbers.

Balanced BST and AVL Trees

- We want something like this — but dynamic, so that we can add and remove documents from the indexed set!

Balanced BST and AVL Trees

- So, we want something like a tree where the elements are not just numbers, but are objects that encapsulate {key, value} pairs (recall that this was the standard terminology for associative containers).
  - Of course, it better be a Balanced Binary Search Tree, if we want it to work efficiently.

Balanced BST and AVL Trees

```
class Index_pair
{
    string d_keyword;
    std::list<string> d_documents;
      // filenames of the matching documents  (could be
      // list of URLs, if this was a web search engine),
      // or list of pointers, if all is in memory

public:
    Node (const string & keyword, const string & doc);

    string keyword() const;
    void add_document (const string & filename);

    void print_documents() const;
};
```

Balanced BST and AVL Trees

```
class Index_pair
{
    string d_keyword;
    std::list<string> d_documents;
      // filenames of the matching documents  (could be
      // list of URLs, if this was a web search engine),
      // or list of pointers, if all is in memory

public:
    Node (const string & keyword, const string & doc);

    string keyword() const;
    void add_document (const string & filename);

    void print_documents() const;
};

    // then, at some point, declare the tree:
AVL_tree<Index_pair> index;
```

Balanced BST and AVL Trees

- One detail: the tree requires linearly ordered values, and it assumes the less-than relation through the use of < in its implementation.
  - The first part is not a problem (keywords are linearly ordered — lexicographical order!)

Balanced BST and AVL Trees

- One detail: the tree requires linearly ordered values, and it assumes the less-than relation through the use of < in its implementation.
  - The first part is not a problem (keywords are linearly ordered — lexicographical order!)
  - But the less-than operator works well with integers (like in our “silly” visual examples)
  - What about less-than being used with two Index_pair objects?
    - It should do a lexicographical comparison of the keywords.... but, how?

Balanced BST and AVL Trees

- In C++, we would definitely want to make use of operator overloading for this situation:
  - We define functions that do comparisons, and name them operator__ , so that when we use the given operator, the compiler will know that it should invoke that function to get the work done.

Balanced BST and AVL Trees

- In this case, we want something like this:

```
bool operator< (const Index_pair & p1, const Index_pair & p2)
{
    return p1.keyword() < p2.keyword();
}

bool operator== (const Index_pair & p1, const Index_pair & p2)
{
    return p1.keyword() == p2.keyword();
}

bool operator<= (const Index_pair & p1, const Index_pair & p2)
{
    return p1.keyword() <= p2.keyword();
}

// etc.
```

Balanced BST and AVL Trees

- With this, we're pretty much done:
  - Indexing a document involves reading its content, isolating words, and locating the node for that word (or insert it, if it's not present) and append the document to the list of documents that contain this keyword!
  - Code could be something like this:

    (you may need to read up a little bit on file IO in C++, as well as possibly strings and string manipulation)

Balanced BST and AVL Trees

```
void index_document (const string & filename)
{
    ifstream document (filename.c_str());

    string keyword;
    while (document >> keyword)
    {
        if (valid_word (keyword))  // some validation function
        {
            Node<Index_pair> * pair = index.find (keyword);
            if (pair != NULL)
            {
                pair->retrieve().add_document (filename);
            }
            else
            {
                index.insert (Index_pair(keyword, filename));
            }
        }
    }
}
```

Balanced BST and AVL Trees

- Searching would simply be something like:

```
void print_search_results (const string & keyword)
{
    Node<Index_pair> * pair = index.find (keyword);
    if (pair != NULL)
    {
        pair->retrieve().print_documents();
    }
    else
    {
        cout << “Your search for ” << keyword << “ did not ”
                “match any documents” << endl;
    }
}
```

Summary

- During today's class:
  - We conclude the topic of AVL trees
    - Discussed re-balancing efficiency.
    - Discussed removals and saw how to handle the re-balancing in this case.
  - Discussed some details on how to use/implement these Balanced BST through an example / case-study of indexing a set of text documents.