Binary Search Trees

Carlos Moreno
cmoreno @ uwaterloo.ca
EIT-4103

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

Binary Search Trees

- Previously, on ECE-250 ...
  - We discussed trees (the general type) and their implementations.
    - We looked at traversals — pre-order and post-order.
  - We saw binary trees, a specific type of tree, and its implementation details.
    - Not the same implementation as for general trees!
  - An important detail was that trees are used to store hierarchically ordered data.

Binary Search Trees

- For today, we have a slight plot twist!
  - We'll look at binary search trees and some related concepts
    - In particular, we'll look at *in-order* traversal.
  - We'll look at some of the operations on them and how to implement them.

Binary Search Trees

- For today, we have a slight plot twist!
  - We'll look at binary search trees and some related concepts
    - In particular, we'll look at *in-order* traversal.
  - We'll look at some of the operations on them and how to implement them.
  - The plot twist being that these types of trees store linearly (totally) ordered data!

Binary Search Trees

- For a brief context/rationale, let's recall binary search:
  - If we have elements stored in ascending order in an array, then we check the middle element:

Binary Search Trees

- For a brief context/rationale, let's recall binary search:
  - If we have elements stored in ascending order in an array, then we check the middle element:
  - Under the "standard" assumption that we write the elements from left to right, we have that for each position:
    - Every element at the *left* of that position is less than the one at that position.
    - Every element at the right of that position is greater than the element at that position.

Binary Search Trees

- For a brief context/rationale, let's recall binary search:
  - That's precisely what allows us to efficiently search for values — check at the middle, and if the value we're searching is less than what we have there, then take the left chunk; greater than, we take the right chunk; and of course, if equal, then we're done.

Binary Search Trees

- For a brief context/rationale, let's recall binary search:
  - Hmmm ... left ... right ... I wonder what that sounds like .....

Binary Search Trees

- Coming back to binary search in an array...
  - What if we're trying to keep a sequence of ordered values? That is, what if we need to do insertions and removals?
    - Searching would still be efficient (logarithmic time, since binary search is feasible), but insertions and removals are very inefficient (linear time).

Binary Search Trees

- Coming back to binary search in an array...
  - What if we're trying to keep a sequence of ordered values? That is, what if we need to do insertions and removals?
    - Searching would still be efficient (logarithmic time, since binary search is feasible), but insertions and removals are very inefficient (linear time).
  - We won't even ask «*what if we use a linked list?*»
    - Been there, chosen NOT to do that — can't do binary search in a linked list.

Binary Search Trees

- So....
  - We tried arrays — no good
  - We tried linked lists — sorry, no good either
- The only two things we haven't tried (from the data structures we've seen) are hash tables and binary trees, right?
  - Hash tables wouldn't even come close to a viable option — *why?*

Binary Search Trees

- So.... we try binary trees!
  - After all, we have left and right sub-trees — what if we add the constraint that every value in the left sub-tree is less than the value at the given node, and every value in the right sub-tree is greater than the given node?
    - Let's say that we're making the implicit assumption of no duplicate data (i.e., no two values are the same)

Binary Search Trees

- Ok, question — can we enforce that constraint? (say, when inserting values?)

Binary Search Trees

- Ok, question — can we enforce that constraint? (say, when inserting values?)
  - So, that's not too bad — if we think in terms of a recursive function or method: if the value to be inserted is less than the value at the root, just pass the value to the left subtree to be inserted there, and if greater than, pass it to the right subtree.
  - Obviously, the base case is reached at leaf nodes (in which case, a new node is created/allocated and we're done)

Binary Search Trees

- But if we can enforce that constraint with insertions, then we're done — right? (*why?*)

Binary Search Trees

- Summarizing — this is essentially the definition of a binary search tree. Formally written: A (non-empty) binary search tree is a binary tree where:
  - The left sub-tree (if any) is a binary search tree and all elements are less than the root element; and
  - The right sub-tree (if any) is a binary search tree and all elements are greater than the root element.

Binary Search Trees

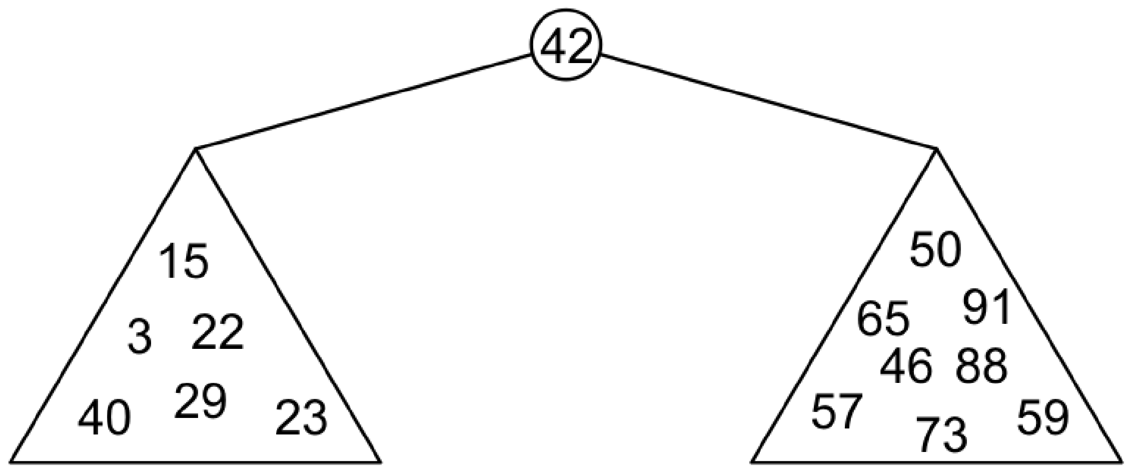
- Normally, it would be a shocking surprise seeing that we want to store linearly ordered data in a tree... But I already spoiled that surprise, right? So, no plot twist at this point !
- The interesting aspect is that it's not the hierarchical nature what's of interest now, but the “geometry” and the properties that derive from the tree structure that are useful for this purpose!

Binary Search Trees

- And yes, as soon as we store the values in a binary tree, we could accuse them of being hierarchical!
  - True enough — they follow the hierarchy imposed by the storage in the tree. But this is an artificially imposed hierarchy — it's part of the storage structure, and not part of the data.

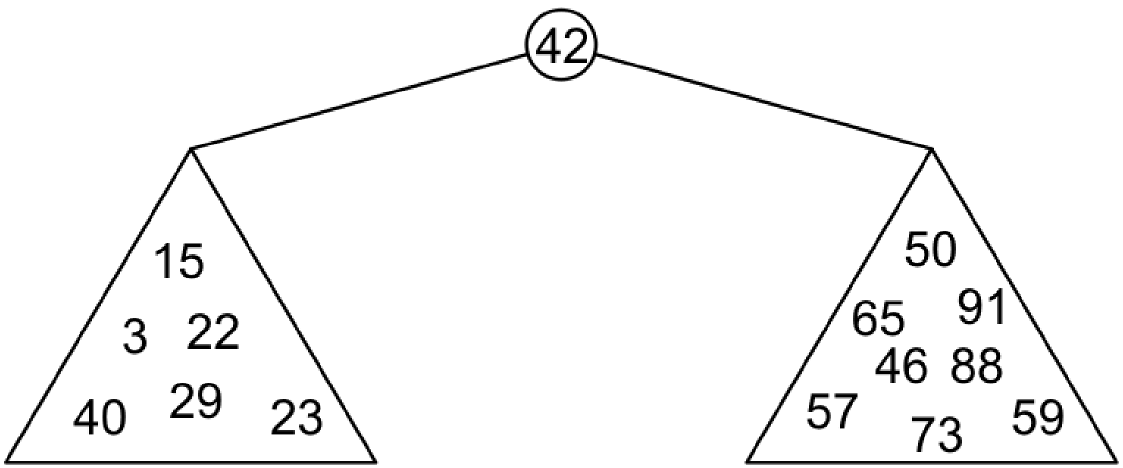
Binary Search Trees

- Our visual idea of these binary search trees could be something like this:



Binary Search Trees

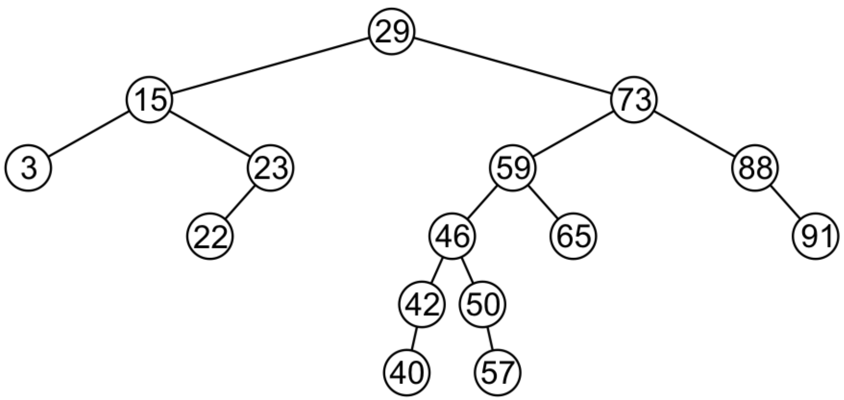
- Our visual idea of these binary search trees could be something like this:

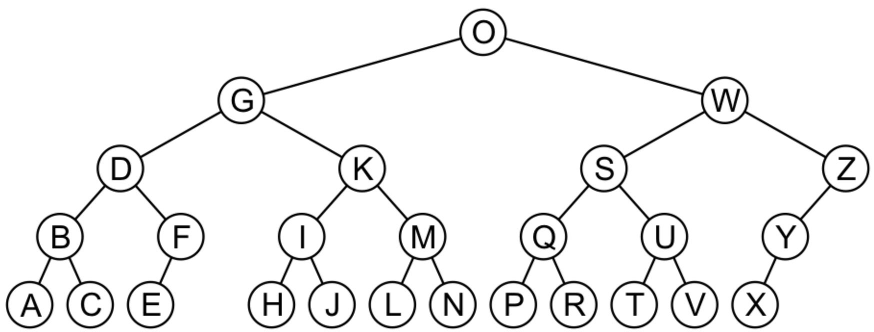


- Of course, each of the sub-trees must be binary search trees themselves!

Binary Search Trees

- Other examples of binary search trees:





Binary Search Trees

- So, let's rewind a little bit, all the way until our class back in 2012-01-18, discussing containers and relations...

Binary Search Trees

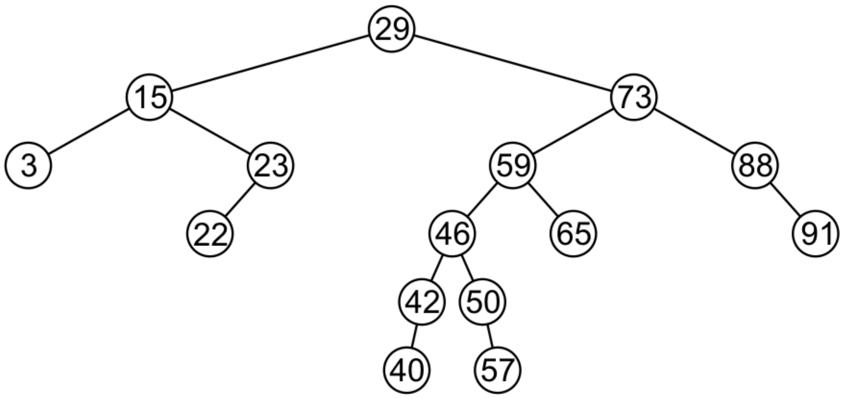
- The point being — some of the operations that we may want to do on linearly ordered data (especially *sorted* data) could be:
  - Find the smallest and largest elements.
  - Iterate over all the elements, in order.
  - Find the next and previous elements to a given value, which may or may not be in the container.

Binary Search Trees

- The point being — some of the operations that we may want to do on linearly ordered data (especially *sorted* data) could be:
  - Find the smallest and largest elements.
  - Iterate over all the elements, in order.
  - Find the next and previous elements to a given value, which may or may not be in the container.
- Can we do these in a binary search tree?

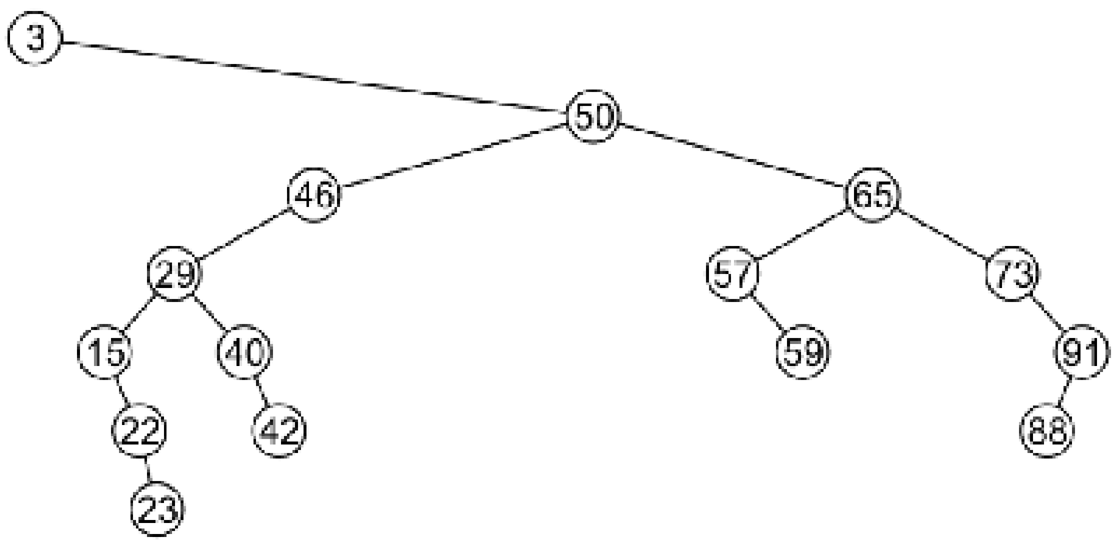
Binary Search Trees

- Let's try the first one — find the smallest and largest elements in the tree (easy, right?)



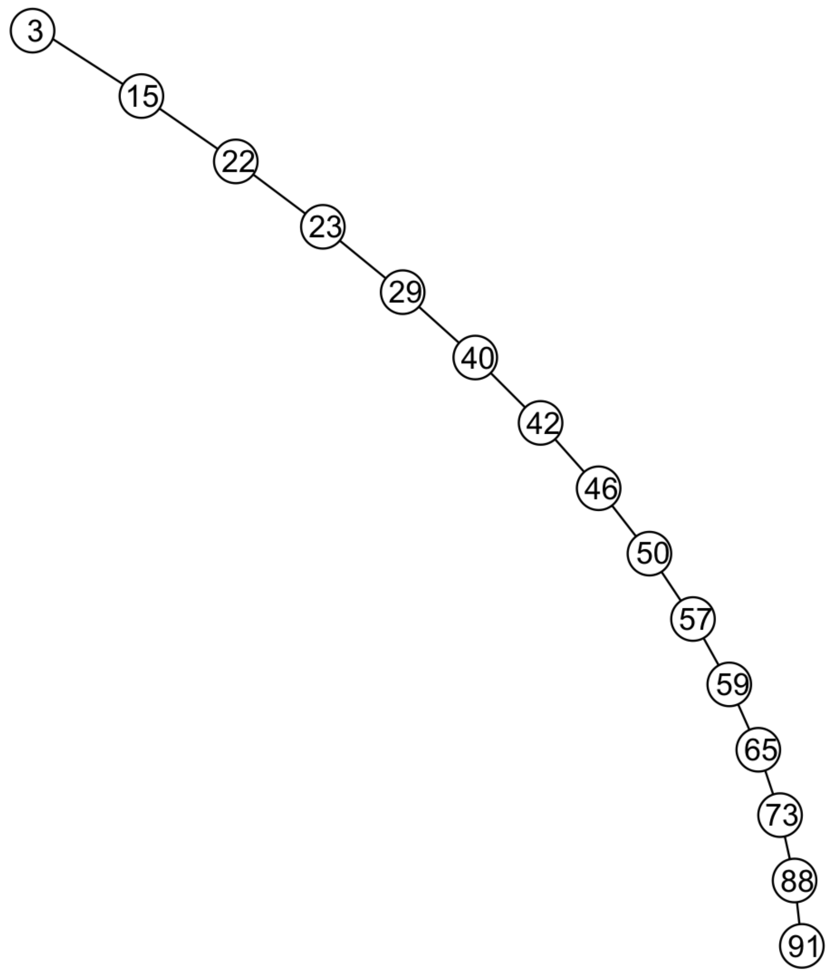
Binary Search Trees

- Would that work for a tree that is completely unbalanced, and that has no left sub-tree right from the root node?



Binary Search Trees

- What about for this one?



Binary Search Trees

- Next — how do we iterate (for example, print) the elements in order?

Binary Search Trees

- Next — how do we iterate (for example, print) the elements in order?
  - Notice that this involves visiting *every* element in the tree — suggesting a tree traversal strategy...

Binary Search Trees

- Next — how do we iterate (for example, print) the elements in order?
  - Notice that this involves visiting *every* element in the tree — suggesting a tree traversal strategy...
  - So, which traversal strategy would work here?

Binary Search Trees

- Next — how do we iterate (for example, print) the elements in order?
  - Notice that this involves visiting *every* element in the tree — suggesting a tree traversal strategy...
  - So, which traversal strategy would work here?
    - Breadth-first?

Binary Search Trees

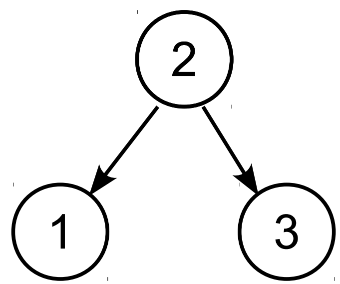
- Next — how do we iterate (for example, print) the elements in order?
  - Notice that this involves visiting *every* element in the tree — suggesting a tree traversal strategy...
  - So, which traversal strategy would work here?
    - Breadth-first? No way — right?

Binary Search Trees

- Next — how do we iterate (for example, print) the elements in order?
  - Notice that this involves visiting *every* element in the tree — suggesting a tree traversal strategy...
  - So, which traversal strategy would work here?
    - Breadth-first? No way — right?
    - Depth-first — pre-order? post-order?

Binary Search Trees

- We can see that neither pre- nor post-order work for this task! Simple counter-example:



Pre-order: 2 – 1 – 3

Post-order: 1 – 3 – 2

Binary Search Trees

- We'll discuss in class the approach to solve this!

Binary Search Trees

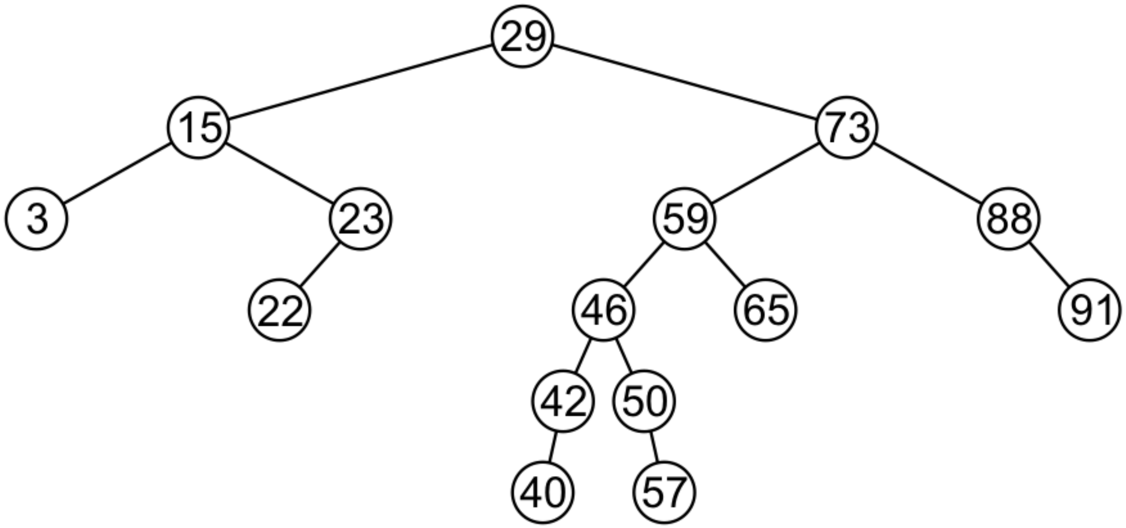
- Next — find the next and previous elements to a given value, which may or may not be in the container.

Binary Search Trees

- Next — find the next and previous elements to a given value, which may or may not be in the container.
  - Let's start with an “easier” version of the above: let's try to find a given value (or determine that it isn't in the container).

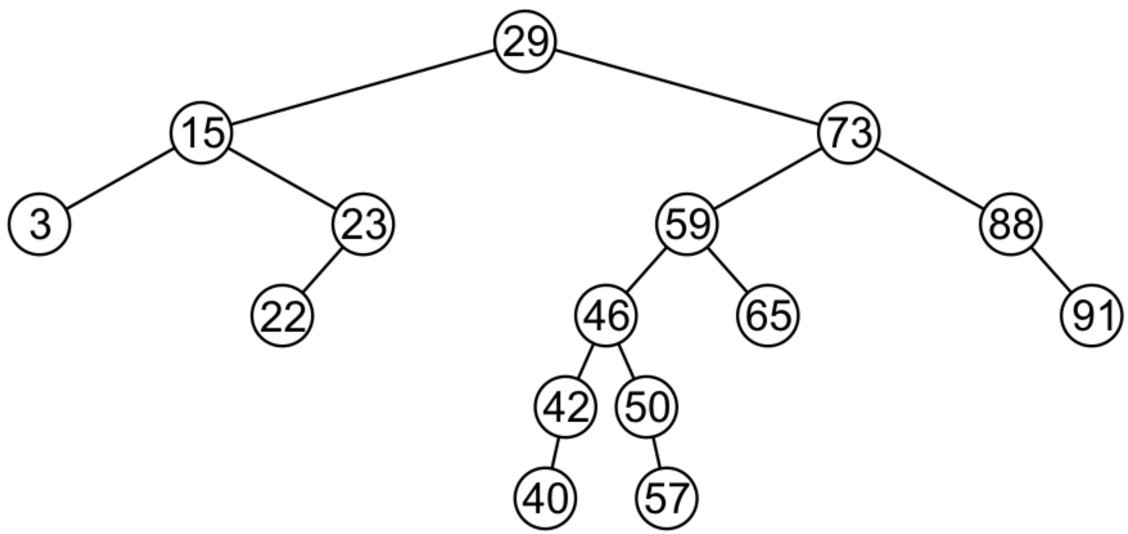
Binary Search Trees

- An example — let's find 50, 21, 65, 15, 45:



Binary Search Trees

- Ok, now the tricky one — find the first value (next in order) after 20, first value before 80:
  - Any ideas?



Binary Search Trees

- An interesting detail — when searching, we're traversing a path (since we're always moving to one of the children); since the length of the longest path is the height *h* of the tree, then finding an element takes O(*h*).
  - And since $h = \lg n$ (where n is the number of elements), then we're good

Binary Search Trees

- An interesting detail — when searching, we're traversing a path (since we're always moving to one of the children); since the length of the longest path is the height *h* of the tree, then finding an element takes O(*h*).
  - And since $h = \lg n$ (where n is the number of elements), then we're good — right?

Binary Search Trees

- An interesting detail — when searching, we're traversing a path (since we're always moving to one of the children); since the length of the longest path is the height h of the tree, then finding an element takes $O(h)$.
  - And since $h = \lg n$ (where n is the number of elements), then we're good — right?
  - No, of course *wrong*! (why?)

Binary Search Trees

- Still, we saw that for perfect and complete trees, the height is $\Theta(\log n)$
- We can't hope to get complete trees here (*why?*), but maybe we'll be able to find things that are “close” to it, or in any case, that exhibit this same logarithmic behaviour for the height!

Binary Search Trees

- Still, we saw that for perfect and complete trees, the height is $\Theta(\log n)$
- We can't hope to get complete trees here (*why?*), but maybe we'll be able to find things that are “close” to it, or in any case, that exhibit this same logarithmic behaviour for the height!
  - We'll see (not today) that the required attribute is *balance* — if the tree is *balanced*, meaning that left and right sub-trees are guaranteed to be close (with respect to some measure such as number of nodes), then we're good!

Binary Search Trees

- The remarkable detail is that there are indeed techniques (we'll look at one of them) that allow us to guarantee that a binary search tree is always balanced.
  - That's the real saver — if we couldn't guarantee that, there would be no point in studying binary search trees (since we would have no guarantee of efficient — logarithmic time — operations)

Binary Search Trees

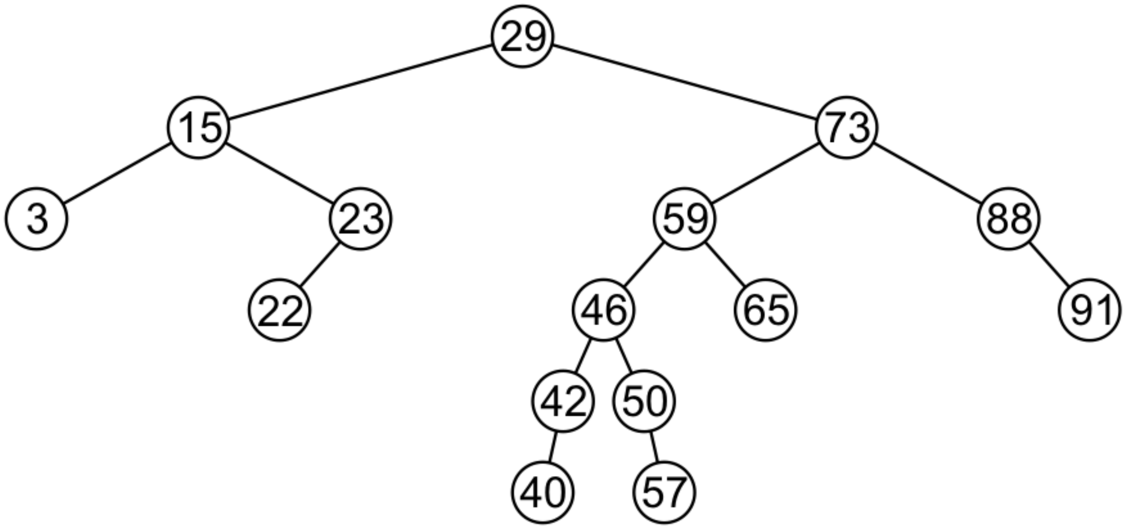
- How about inserting elements?
  - Since this is really a sequential container (we're storing linearly ordered data), we should have push_front and push_back — right?

Binary Search Trees

- How about inserting elements?
  - Since this is really a sequential container (we're storing linearly ordered data), we should have push_front and push_back — right?
  - No, we notice that an element goes at its corresponding position — we don't decide that it goes at the beginning or at the end.
  - So, what really makes sense is a single insert operation that places the element at the correct position in the tree!

Binary Search Trees

- Example: let's insert 20, 70, 30.



Binary Search Trees

- Last (and definitely not least!), we'll look at removing elements.
- Unlike insertions, which are easy since we always go down to a leaf node and insert on one of its empty nodes, removals can happen anywhere!
- We consider the three possible cases:
  - The node is a leaf node.
  - It has one child
  - It has two children

Binary Search Trees

- In all three cases, we first have to locate the node (this part is easy — we just saw the searching procedure).

Binary Search Trees

- If the node being removed is a leaf node, the operation is trivial — we just remove it (the operation is trouble-free)

Binary Search Trees

- If the node being removed has one child, we run into *a little bit* of trouble — removing the node leaves a node (possibly an entire sub-tree) floating ...

Binary Search Trees

- If the node being removed has one child, we run into *a little bit* of trouble — removing the node leaves a node (possibly an entire sub-tree) floating ...
- However, if it is the only child, then we can easily solve it — want to give it a try? (we'll discuss the answer in class)

Binary Search Trees

- So, it looks like if the node being removed has two children, then we run into *a lot* of trouble!
  - And here too, we'll discuss the solution in class!