# Binary Search Trees

# Why Use Binary Trees?

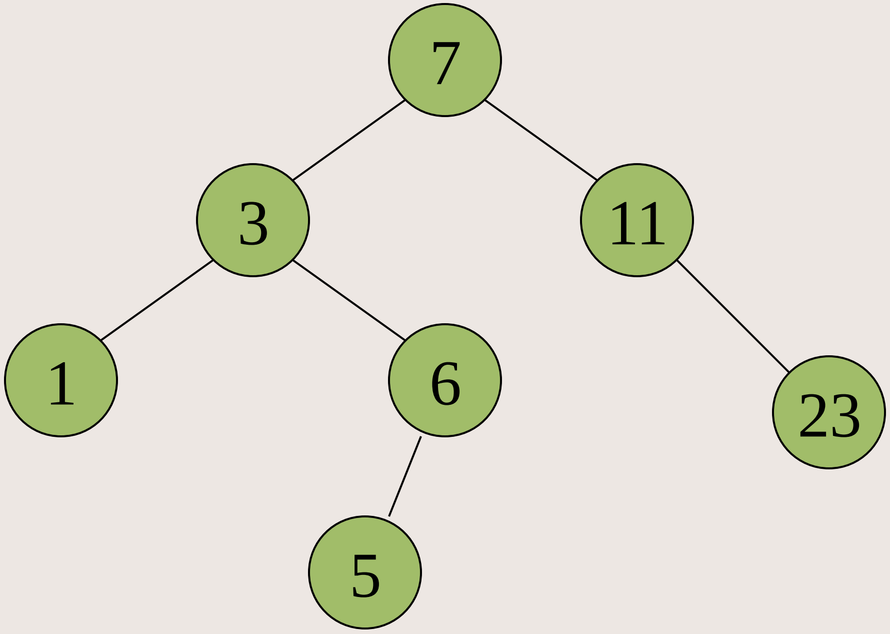
- Searches are an important application.
- What other searches have we considered?
    - brute force search (with array or linked list)
        - O(N)
    - binarySearch with a **pre-sorted** array (**not** a list!)
        - O(log(N))
- Binary Search Trees are also O(log(N)) on average.
    - So why use 'em?
        - Because sometimes a tree is the more natural structure.
        - Because insert and delete are also fast, O(logN). Not true for arrays.

# So It's a Trade Off

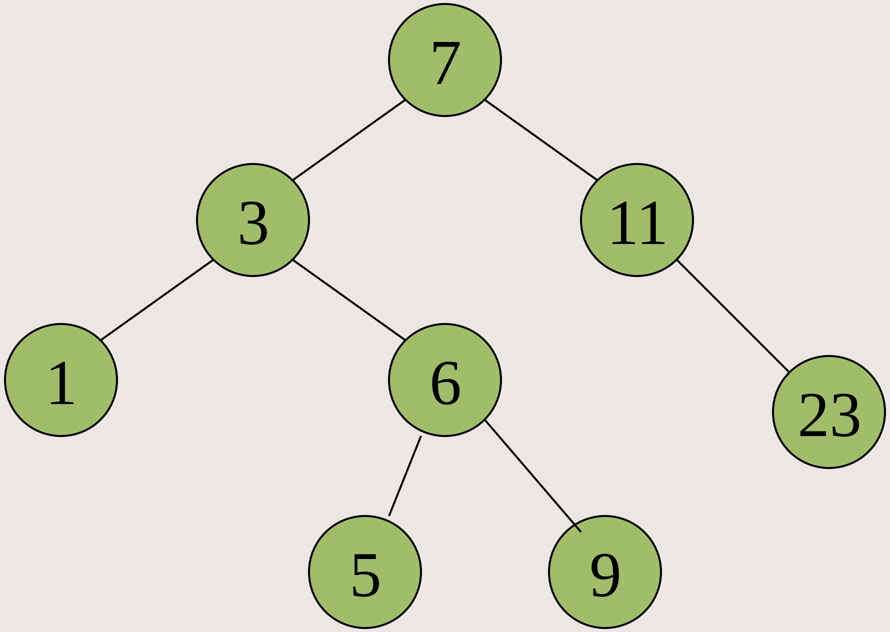
- Array Lists
  - O(N) insert
  - O(N) delete
  - O(N) search (assuming not pre-sorted)
- Linked Lists
  - O(1) insert
  - O(1) delete
  - O(N) search
- Binary Search Tree
  - O(log(N)) insert
  - O(log(N)) delete
  - O(log(N)) search
    - on average, but occasionally (rarely) as bad as O(N).

# Search Tree Concept

- Every node stores a value.
  - Every left subtree (i.e., every node below and to the left) has a value less than that node.
  - Every right subtree has a value greater than that node.



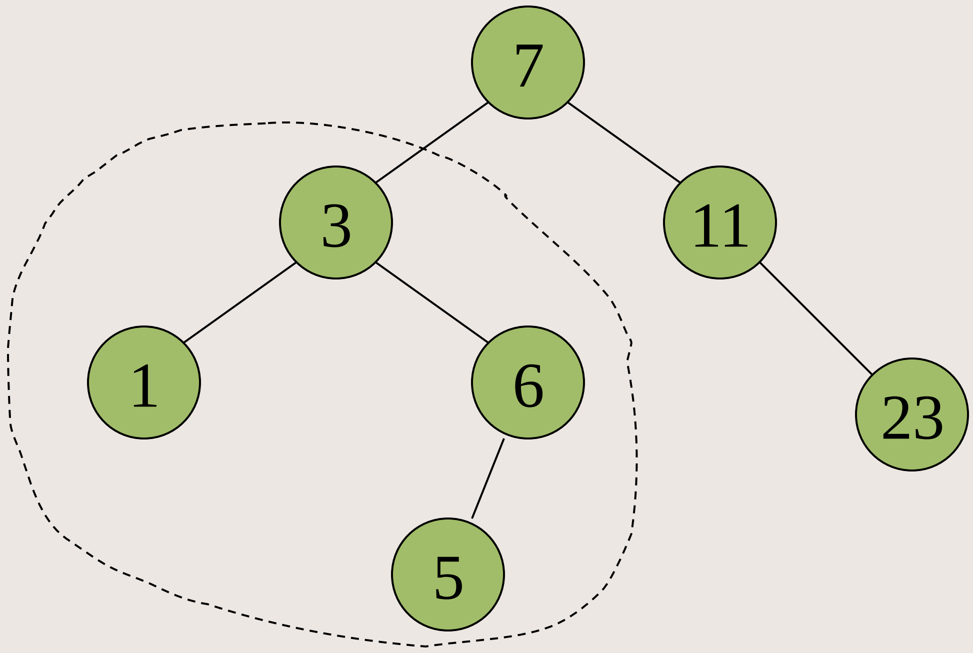
# Question: Is This a Binary Search Tree?




No. Why not?

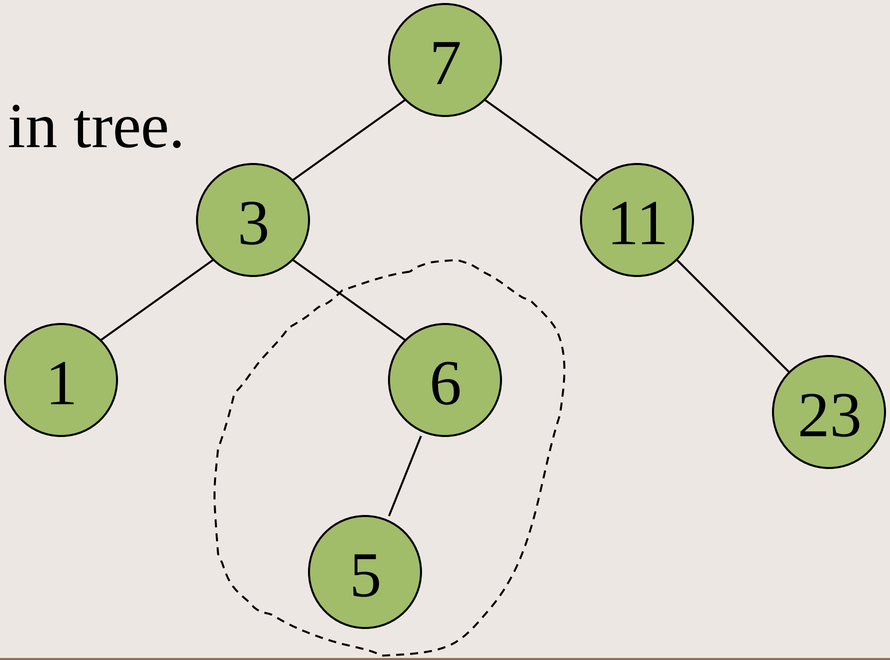
# So S'pose We Wanna' Search

- Search for 4
  1. start at root 7.
  2. move to 3 on left, because 4 < 7.



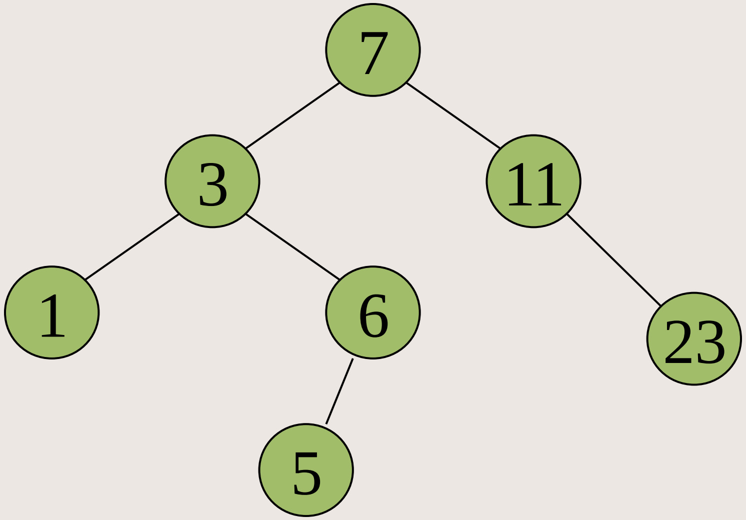
# So S'pose We Wanna' Search (cont.)

- Search for 4.
  3. Now move to right because 4 > 3.
  4. Now move to left because 4 < 6.
  5. Now move to left because 4 < 5. But nowhere to go!
  5. So done. Not in tree.



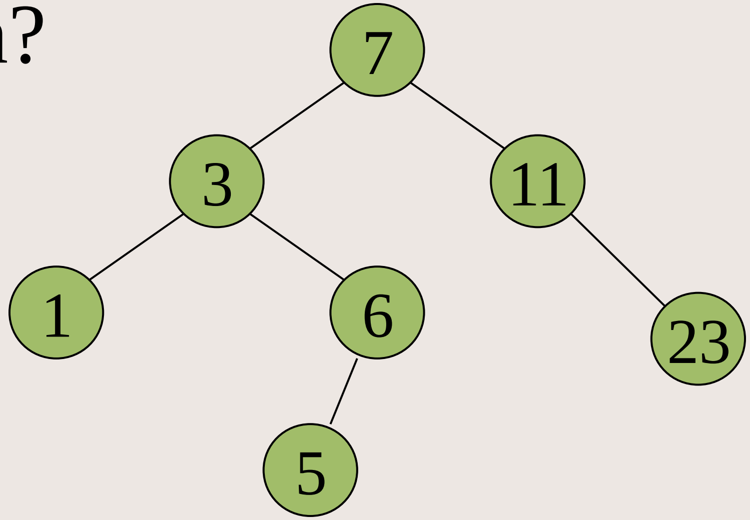
# How Many Steps Did That Take?

- 7 to 3 to 6 to 5. Three steps (after the root).
- Will never be worse than the distance from the root to the furthest leaf (*height*!).
- On average splits ~twice at each node.



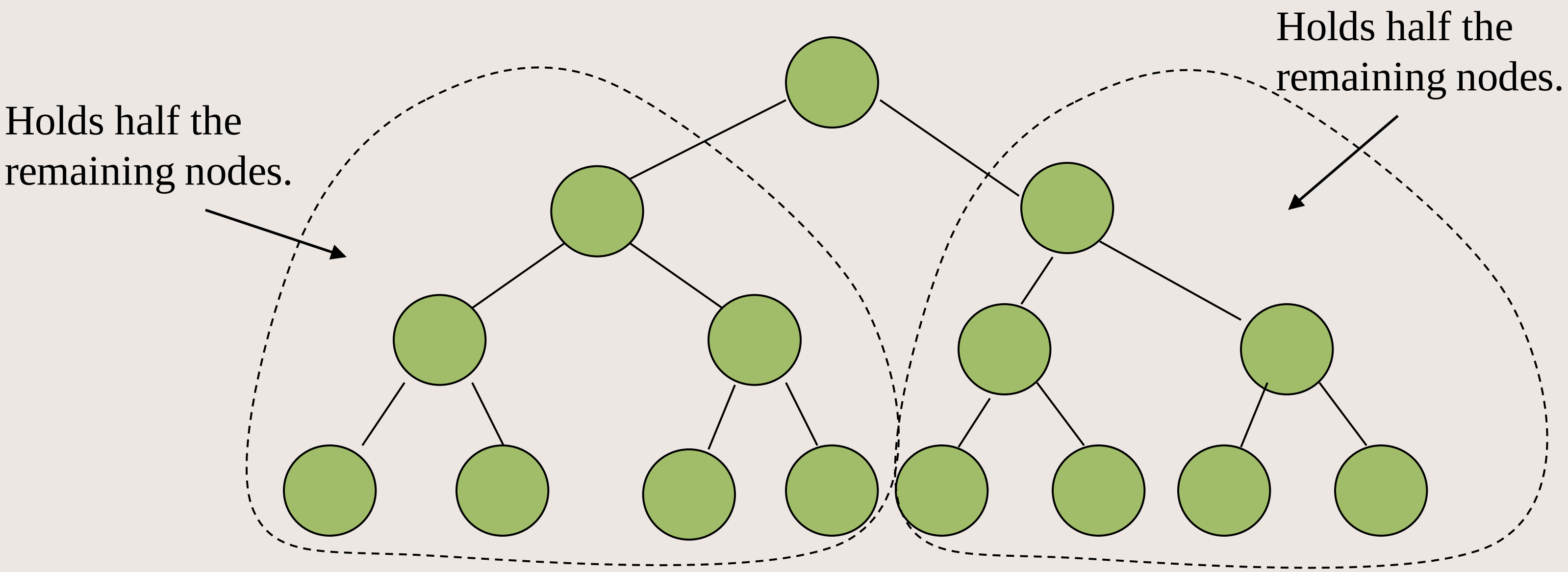
# So Time To Search?

- So double the number of nodes at each layer.
- It's like "doubling the counter variable each time through a for loop." How long does that take to run?
- $\boxed{\text{Log(N)}}$



# Big-O of Search

- S'pose tree bifurcates at every node. (This is an assumption that could be relaxed later).
- Each time we step down a layer in the tree, the # of nodes we have to search is cut in half.



# Big-O of Search (cont. 1)

- For example, first we have to
  - search 15 nodes = $2^4 – 1$
  - then 7 nodes = $2^3 – 1$
  - then 3 nodes = $2^2 – 1$
  - then 1 node = $2^1 - 1$
- Now, note that tree has $\sim 2^{h+1}-1$ nodes where h = height of the tree.
  - The height is the longest path (number of *edges*) from the root to the farthest leaf.

# Big-O Search (cont. 2)

- So total number of nodes is

$$N = 2^{h+1}-1$$

Or $N \cong 2^{h+1}$ (true for big N)

- Now how many steps do we have to search? A max of "h+1" steps (4 steps in the example above).

- What is h? Solve for it!

$$\log(N) = (h+1) * \log(2)$$

$$h = (\log N/\log 2) - 1$$

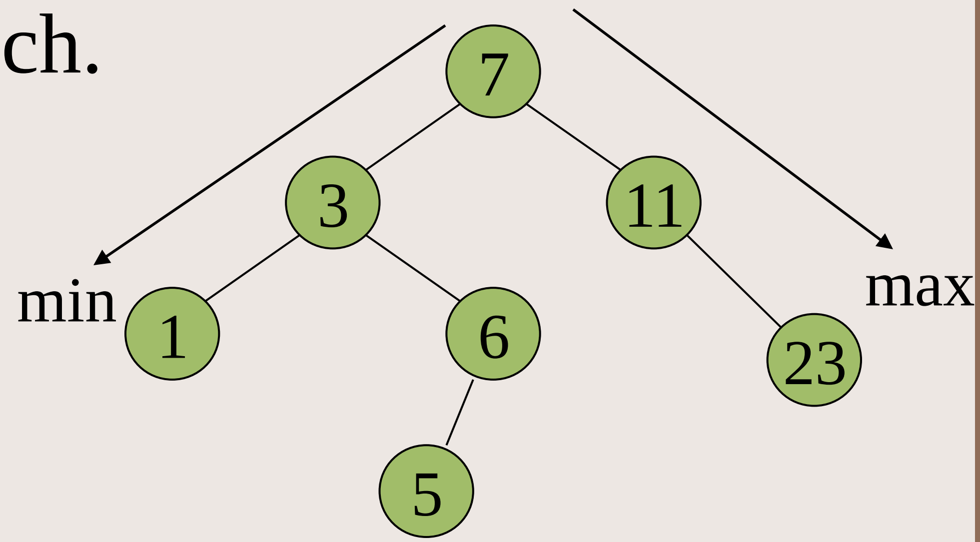
- So $h = O(\log N)$. Wow! That's how long it takes to do a search.

# findMin and findMax

- Can get minimum of a tree by always taking the left branch.

- Can get maximum of a tree by always taking right branch.

- Example…



# Inserting an Element Onto a Search Tree?

- Works just like “find”, but when reach the end of the tree, just insert.
- If the element is already on the tree, then add a counter to the node that keeps track of how many there are.
- Do an example with putting the following unordered array onto a binary search tree.
  - {21, 1, 34, 2, 6, -4, -5, 489, 102, 47}

# Insert Time

- How long to insert N elements?
  - Each insert costs O(log(N)).
  - There are N inserts.
  - Therefore, O(Nlog(N))
- Cool, our first example of something that takes NLogN time.

# Deleting From Search Tree?

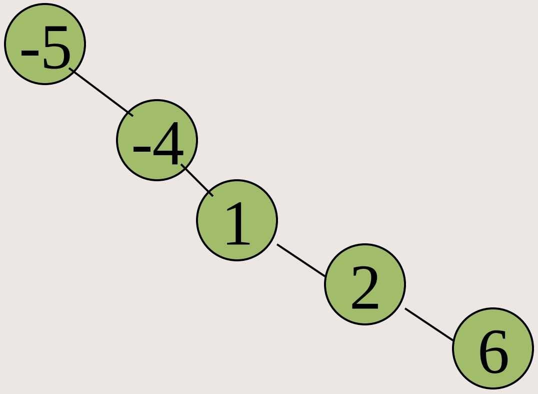
- Ugh.  Hard to *really* remove.
  - See what happens if erase a node.  Not a search tree.
  - Must adjust links…
  - There is a recursive approach (logN time), or can just reinsert all the elements in the subtree (NLogN time)
- Easiest to do "lazy deletion".
  - Usually are keeping track of duplicates stored in each node.
  - So just decrement that counter.  If goes below 1, then the node is empty.
  - If node is empty, ignore the node when doing find's etc.
  - So delete is

O(logN)

Tell me why?

# Humdinger.

- OK, so on average, insert and delete take O(logN) time.
- But remember the tree we created with
  {21, 1, 34, 2, 6, -4, -5, 489, 102, 47}?
- Let’s do the same thing with the array pre-sorted.
  {-5, -4, 1, 2, 6, 21, 34, 47, 102, 489}?
- Whoa, talk about unbalanced!




(Note: there isn’t a unique tree for each set of data!)

# Worst Case Scenario!

- Remember how everything depended on having an average of two children per node?
  - Well, it ain't happenin' here.
- In this case, the depth is N. So the worst case is that we could have O(N) time for each insert, delete, find.
  - So if insert N elements, took $O(N^2)$ time.
  - Ugh.

# Looking For Balance

- We want as many right branches as left branches.
- Whole “branch” of mathematics dealing with this.
    - (har, har, very punny!)
- Complicated, but for *random data*, is usually irrelevant for small trees.
- Phew, so we are ok (*usually*).

# Sample Implementation (Java)

```
public class BinaryNode
{

    public int value;
    public BinaryNode left;
    public BinaryNode right;

    public int numInNode;          //optional – keeps track of duplicates (and lazy deletion)

    /**constructor – can be null arguments*/
    public BinaryNode(int n, BinaryNode lt, BinaryNode rt)
    {
        value = n;
        left = lt;
        right = rt;
        numInNode = 1;
        deleted = false;
    }
}
```

# Sample Implementation (C)

```
struct BinaryNode;
typedef struct BinaryNode *BinaryNodePtr;

struct BinaryNode
{
	int value;
	BinaryNodePtr left;
	BinaryNodePtr right;

	int numInNode;	//optional – keeps track of duplicates
	int deleted;		//optional – marks whether node is deleted
}
```

# Sample Find (Java)

```
/**Usually start with the root node.*/
public BinaryNode find(int n, BinaryNode node)
{
    if(node == null)
        return null;
    if(n < node.value)
        return find(n, node.left);
    else if (n > node.value)
        return find(n, node.right);
    else
        return node;        //match!
}
```

could be null

Note: returns the node where n is living.

# Sample Find (C)

```
/*Usually start with the root node.*/
BinaryNodePtr find(int n, BinaryNodePtr node)
{
	if(node == NULL)
		return NULL;
	if(n < node->value)
		return find(n, node->left);
	else if (n > node->value)
		return find(n, node->right);
	else
		return node;			//match!
}
```

# Other Trees

- Many types of search trees
  - Most have modifications for balancing
    - B-Trees (not binary anymore)
    - AVL-trees (restructures itself on inserts/deletes)
    - splay-trees (ditto)
    - etc.

# So-So Dave Tree

- Each time insert a node, recreate the whole tree.
  1. Keep a separate array list containing the values that are stored on the tree.
     - so memory intensive!  Twice the storage.
  2. Now add the new value to the end of the array.
  3. Make a copy of the array. …
     - ooooooh, expensive. O(N).
  4. Randomly select an element from the copied array.
     - because randomly ordered, will tend to balance the new tree
     - O(1)
  5. Add to new tree and delete from array.
     - oooh, O(logN) insert
     - ughh, O(N) delete
  6. Repeat for each N.  So what's the total time????

O($N^2$) for inserts. Why?

# Can You Do Better?

- Improve the “So-So Dave Tree.”
  - p.s. It can be done!
  - p.p.s. Consider storing in something faster like a linked list, stack, or queue… You still have to work out details.
  - p.p.p.s. That’s the “Super Dave Tree.”
  - p.p.p.p.s. Bonus karma points if your solution isn’t the “Super Dave Tree” and is something radically different.
  - p.p.p.p.p.s. No karma points if you use a splay tree, AVL tree, or other common approach, but mega-educational points for learning this extra material.