CPSC 221
Basic Algorithms and Data Structures

May 29, 2015

Administrative stuff

Midterm exam is 5 days from today

Details are in the lecture slides for May 27

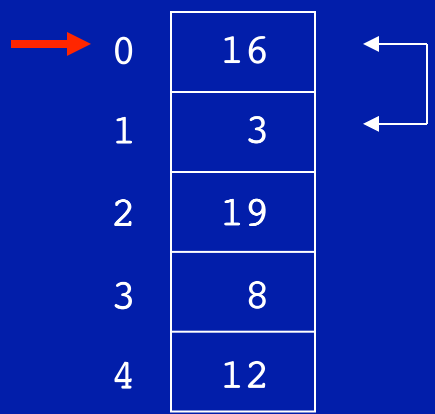
Reminder: the exam will not be held in this classroom

Binary search with iteration

5	10	15	20	25	30	35	40	45	50	55	60	65	70	75
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

```
int binSearchIt(int array[], int target, int left, int right)
{
    int result = -1;
    while (! (right < left))
    {
        int mid = (left + right) / 2;
        if (array[mid] == target)
        {
            result = mid;
            right = left - 1;  // kill the loop
        }
        else if (target < array[mid])
            right = mid - 1;
        else
            left = mid + 1;
    }
    return result;
}
cout << binSearch(array, 55, 0, 14) << endl;
```

Selection sort

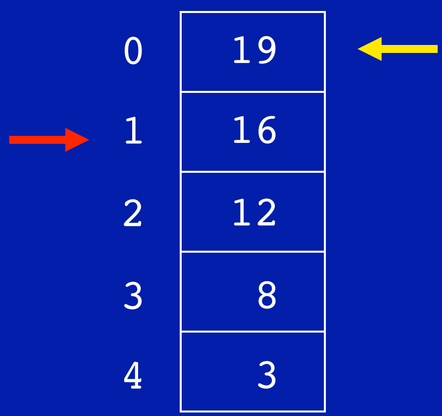


The smallest value so far is 3

Its index is 1

Let's say we want to sort the values in the array at the left in increasing order. One way to approach the problem would be to use an algorithm called selection sort. We start by setting a pointer to the first element in the array; this is where the smallest value in the array will be placed. Then we'll look at every value in this unsorted array and find the minimum value. Once we've found the minimum value, we swap that value with the one we selected at the beginning.

Insertion sort



Pseudocode algorithm:

1. for each array element from the second element to the last element
2. Insert the selected element where it where it belongs in the array by shifting all values larger than the selected element back by one location

Your textbook has a more detailed pseudocode algorithm for insertion sort that behaves sort of like what we just did.

Here's some C++ code that might do what we we're talking about too:

Mergesort

Mergesort takes a different approach to the problem. It falls in the class of algorithms called “divide and conquer”.

In mergesort, the problem space is continually split in half by applying the algorithm recursively to each half, until the base case is reached.

A simple algorithm for mergesort is:

```
mergesort(unsorted_list)
   Divide the unsorted_list into two sublists of half the size of the unsorted_list
   Apply mergesort to each of the unsorted sublists
   Merge those two now-sorted sublists back into one sorted list
```

where ‘list’ could be any sequential data structure

Quicksort

If we want to sort big sequences quickly, without the extra memory and copying back and forth of mergesort, the answer is quicksort. In practice, it is the fastest sorting algorithm known. While its worst-case run time is $O(n^2)$, its average run time is $O(n \lg n)$.

A simple algorithm for mergesort is:

quicksort(unsorted_list)
  Choose one element to be the *pivot*
  Partition (i.e. reorder) the list so that all elements < the pivot are to the left of the pivot, and all elements > the pivot are to the right of the pivot
  Apply quicksort recursively to the sublist to the left and the sublist to the right

where ‘list’ could be any sequential data structure

Is that all there is?

S	B	Q	A	E	P	G	L	C	N	U	K	R	X	W
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

All these sorting algorithms will turn this...

Is that all there is?

S	B	Q	A	E	P	G	L	C	N	U	K	R	X	W
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B	C	E	G	K	L	N	P	Q	R	S	U	W	X

All these sorting algorithms will turn this into this (some faster than others). So we can sort with O(n lg n) speed, and search with O(lg n) speed. That's really good. But we still have some issues. What might those be?

Issue #1

A	B	C	D	...	Z
0	1	2	3	...	6999999999

Where are we gonna put that 7 billion element array in memory? Finding that much contiguous available memory might be problematic. A data structure that was more flexible, more dynamic, and much less dependent on contiguous memory could be helpful. What kind of structure might that be?

Issue #1

A	B	C	D	...	Z
0	1	2	3	...	6999999999

Where are we gonna put that 7 billion element array in memory? Finding that much contiguous available memory might be problematic. A data structure that was more flexible, more dynamic, and much less dependent on contiguous memory could be helpful. What kind of structure might that be? Some sort of linked list structure seems like a possibility.

Issue #2

A	B	C	D	...	Z
0	1	2	3	...	6999999999

Now that we've sorted our 7 billion element array in O(n lg n) time, what do we do if we want to add one more element in the right place?

Issue #2

A	B	C	D	...	Z	Y
0	1	2	3	...	6999999999	7000000000

Now that we've sorted our 7 billion element array in O(n lg n) time, what do we do if we want to add one more element in the right place? How about if we add that element to the end of the existing array and then apply quicksort to the whole thing again?

Issue #2

A	B	C	D	...	Z	Y
0	1	2	3	...	6999999999	7000000000

Now that we've sorted our 7 billion element array in O(n lg n) time, what do we do if we want to add one more element in the right place? How about if we add that element to the end of the existing array and then apply quicksort to the whole thing again? What input causes quicksort to perform with its absolute worst time complexity (i.e. $O(n^2)$)?

Issue #2

A	B	C	D	...	Z	Y
0	1	2	3	...	6999999999	7000000000

So if we're doing 1 billion comparisons per second, we can sort the original unsorted array in less than 4 minutes, according to what we learned last time. But once the array is sorted, if we want to add an element that belongs just before the current last item in the array, that'll take more than 1500 years?

Issue #2

A	B	C	D	...	Z	Y
0	1	2	3	...	6999999999	7000000000

So if we're doing 1 billion comparisons per second, we can sort the original unsorted array in less than 4 minutes, according to what we learned last time. But once the array is sorted, if we want to add an element that belongs just before the current last item in the array, that'll take more than 1500 years? Isn't it nice that you know something about algorithm analysis now? Let's try another approach...

Issue #2

A	B	C	D	...	Z	Y
0	1	2	3	...	6999999999	7000000000

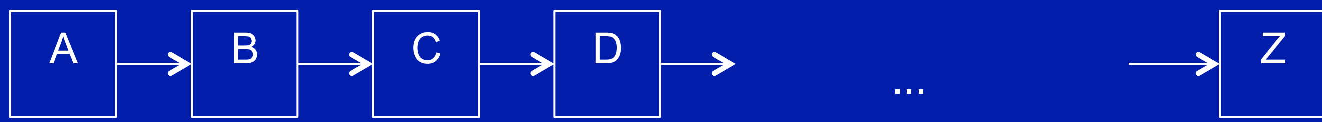
How about we just insert the element in its appropriate place by searching the sorted array until we find the right place, then make a space by moving all the elements past that place one space toward the back, like one pass of insertion sort. What's the time complexity of insertion now?

Issue #2

A	B	C	D	...	Z	Y
0	1	2	3	...	6999999999	7000000000

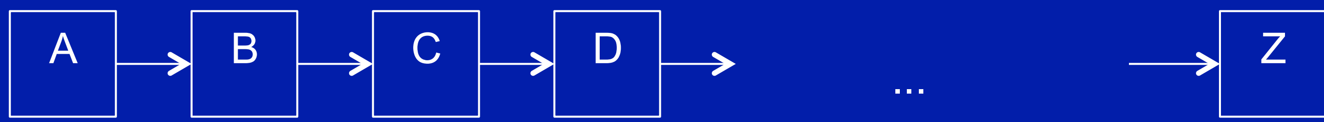
How about we just insert the element in its appropriate place by searching the sorted array until we find the right place, then make a space by moving all the elements past that place one space toward the back, like one pass of insertion sort. What's the time complexity of insertion now? Using binary search, we can find the insertion point in O(lg n) comparisons, but moving elements will take O(n) movements. O(n) beats O(n^2), but...

There's got to be a better way



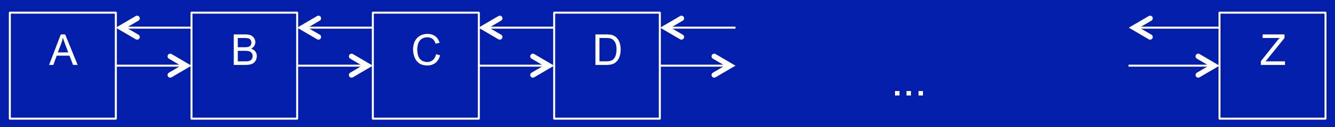
Maybe we could speed up the insertion time even more. Before we hinted at a linked list structure for flexibility. Can we do binary search on a singly-linked list to get that O(lg n) time complexity?

There’s got to be a better way



Maybe we could speed up the insertion time even more. Before we hinted at a linked list structure for flexibility. Can we do binary search on a singly-linked list to get that O(lg n) time complexity? No, we have to be able to jump in both directions in the sequence to make binary search work.

There's got to be a better way

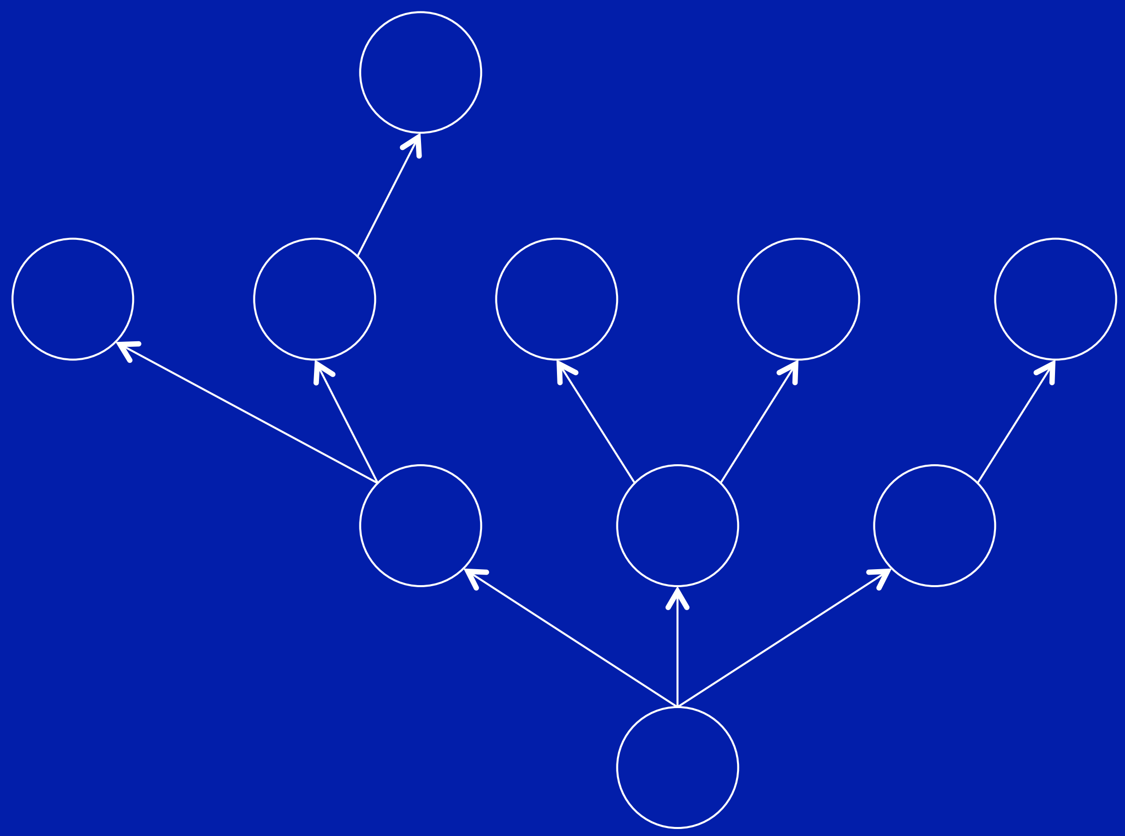


A doubly-linked list lets us move in both directions. So binary search should be possible, but we can't move the pointers (left, right, mid) with simple index arithmetic. We would have to do link traversals, and as long as we're traversing the links in the list, we might as well look at the values at the nodes as we pass by, and this just turns into linear search. What is the linked structure that makes it all come together?

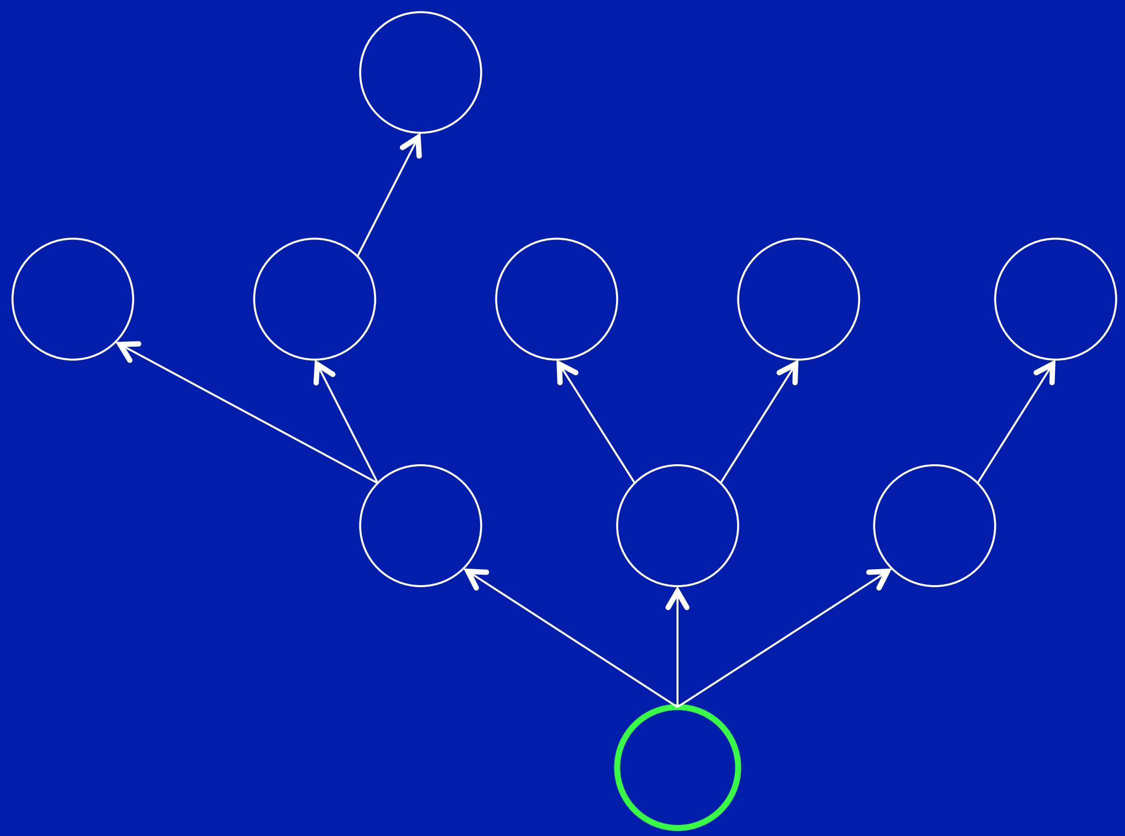
The better way

The tree

Trees

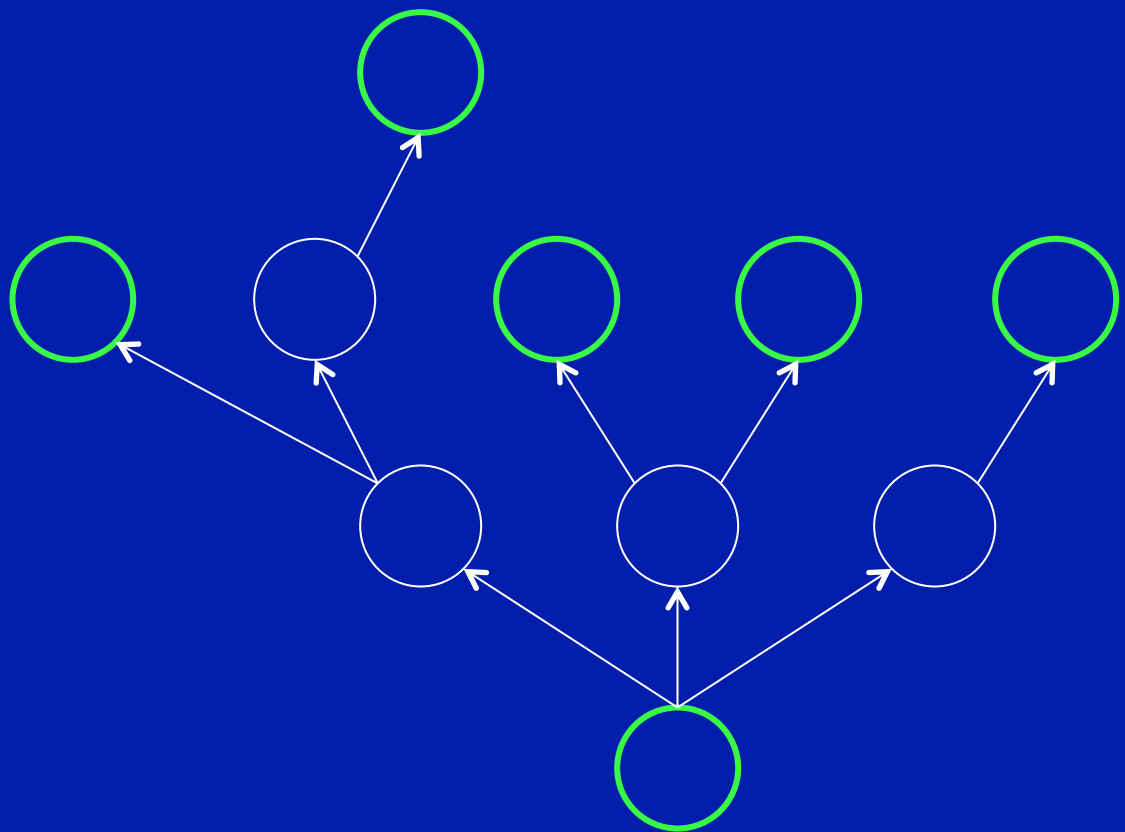


Trees



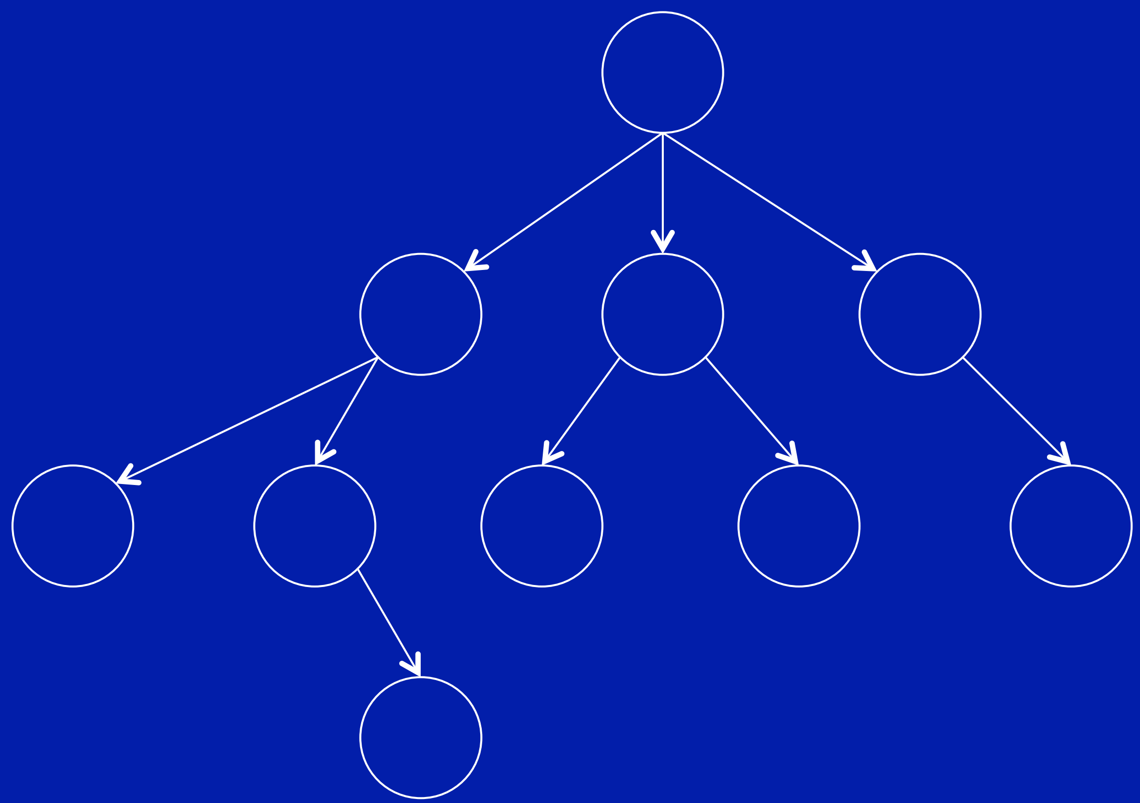
In computing, a tree has a root (or root node)

Trees



In computing, a tree has a root (or root node) and leaves (or leaf nodes) just like a botanical tree. But the resemblance between our trees and nature's best sort of ends there.

Trees



For example, we like our trees upside down. Non-computing people tend to think this is weird. Really, we’re just trendsetters...

Trees

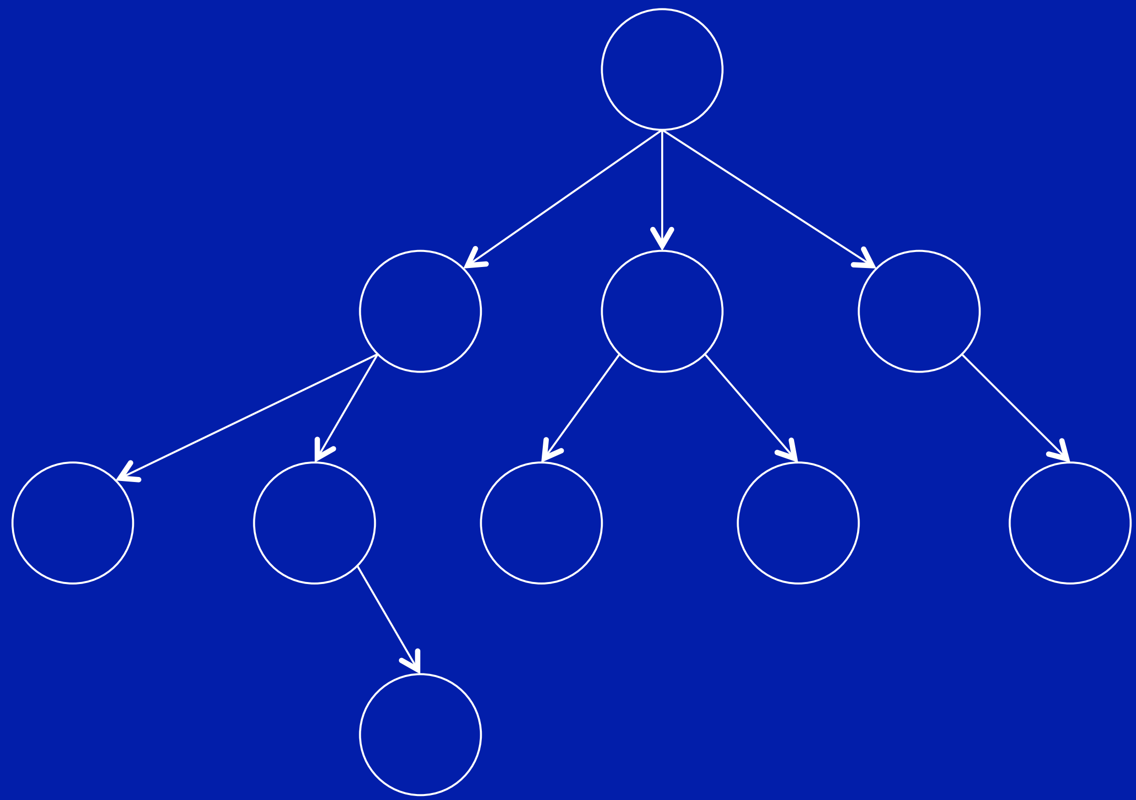
For example, we like our trees upside down. Non-computing people tend to think this is weird. Really, we're just trendsetters...

Trees

Anyone who has ever used a computer has experience with trees, though they may not know it. That's how file directory structures are represented.

If you came here through CPSC 110, you already have hands-on programming experience with trees, because you all did that really cool file directory example in class (followed most likely by file directory labs and homework assignments).

Trees



A tree is a (possibly non-linear) data structure made up of nodes or vertices and edges without having any cycle. The tree with no nodes is called the **null** or **empty** tree. A tree that is not empty consists of a root node and potentially many levels of additional nodes that form a hierarchy. - Wikipedia

Binary trees

For reasons which will become obvious, we're primarily interested in binary trees (and variations on the binary-tree theme).

A binary tree is a tree that is either

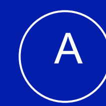
- empty (null for us), or
- a node called the root node and two binary trees called the left subtree and a right subtree

(Remember that there can't be multiple paths from the root to any leaf node in a tree.)

Binary trees

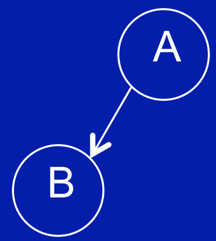
So this is a binary tree.

Binary trees



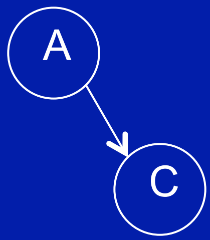
This is a binary tree too.

Binary trees



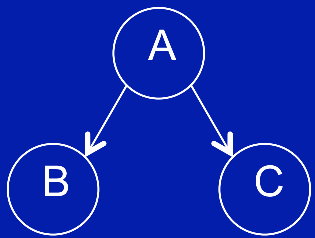
And this.

Binary trees



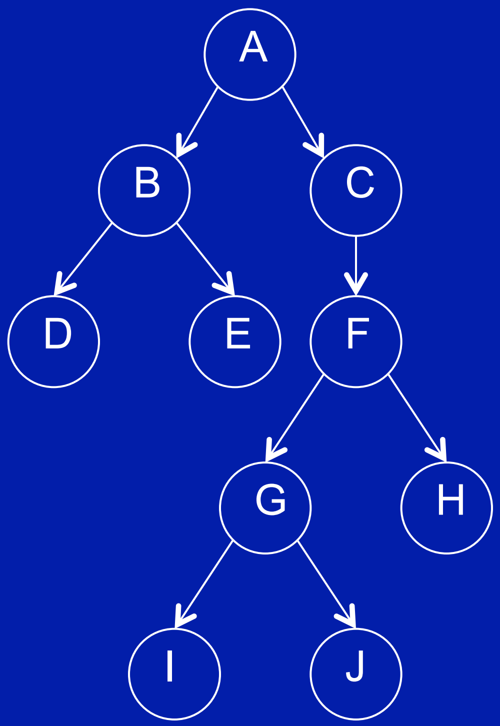
Here's another one.

Binary trees



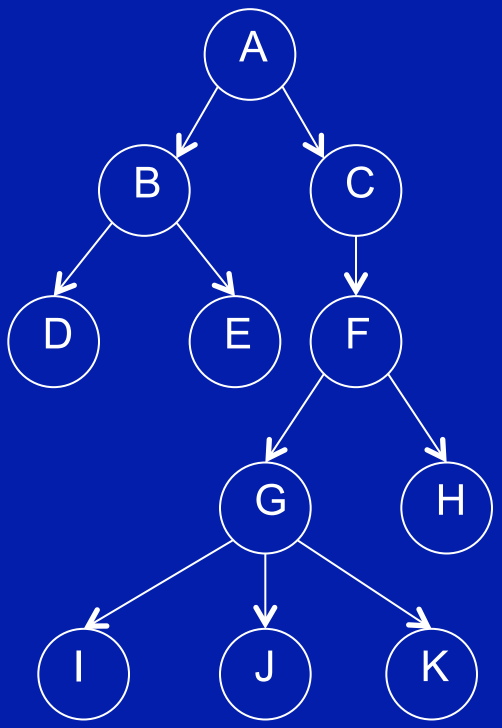
Of course this is a binary tree.

Binary trees



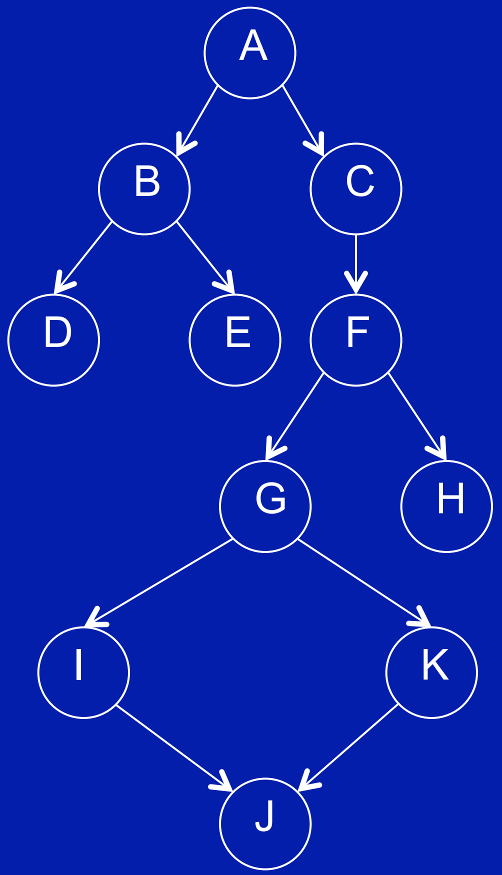
Here is yet another.

Binary trees



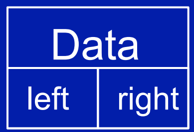
This is a tree, but it's not binary.

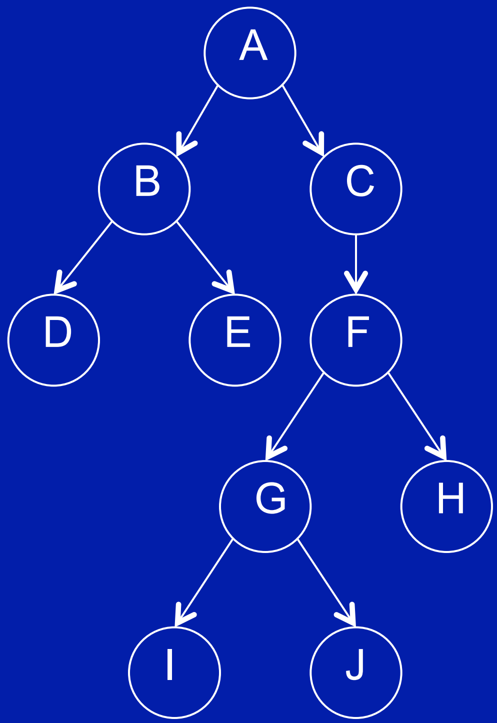
Binary trees



This might be binary, but it's not a tree.

Binary trees

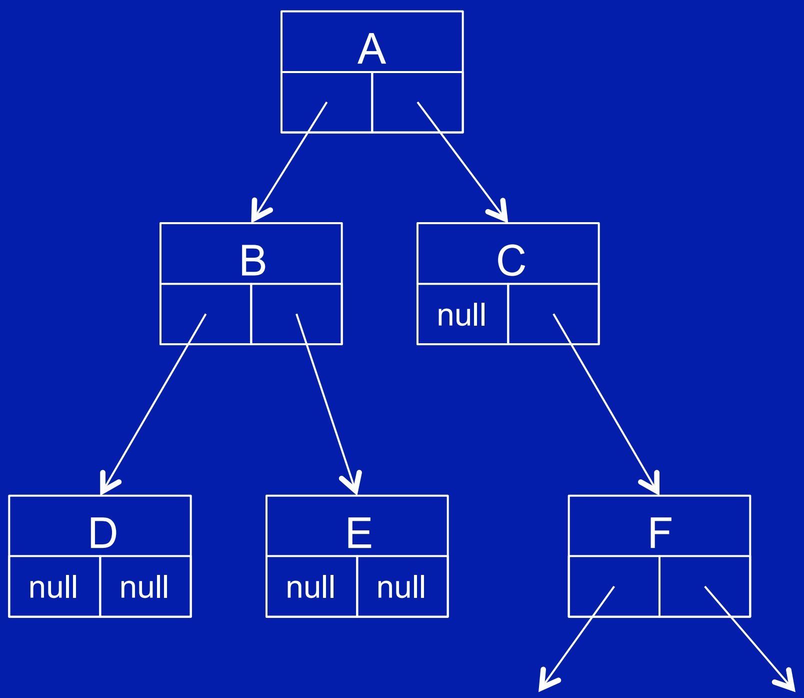


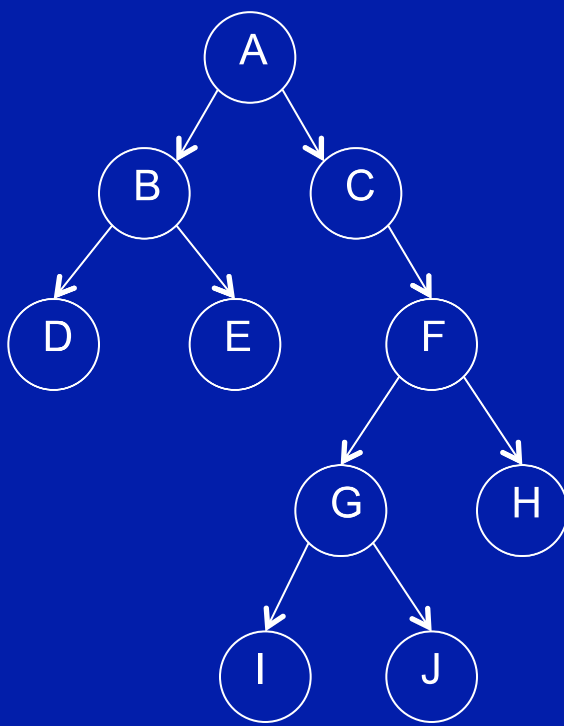



```
struct Node
{
  DType data;
  Node * left;
  Node * right;
}
```

Each binary tree node holds some data, a pointer to its left subtree and a pointer to its right subtree.

Binary trees

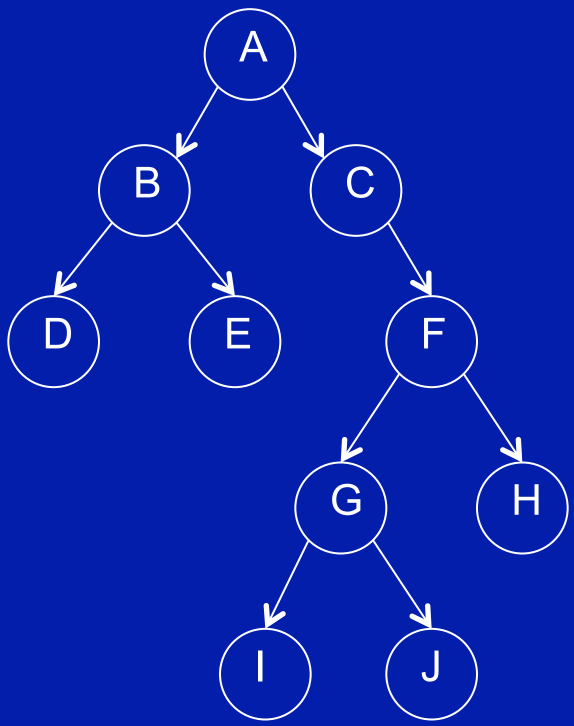




Same tree as in the previous slide but the pointer from C to F is now clearly a right pointer.

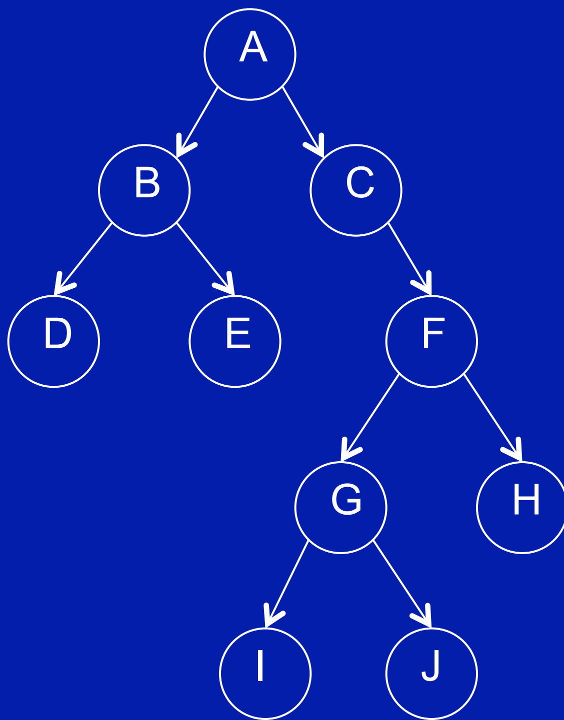
Each binary tree node holds some data, a pointer to its left subtree and a pointer to its right subtree.

Tree traversal



Sometimes, we want to operate on the values contained in a binary tree. We walk through the tree in a prescribed order and visit or process the value at the nodes as they are encountered. This is called tree traversal.

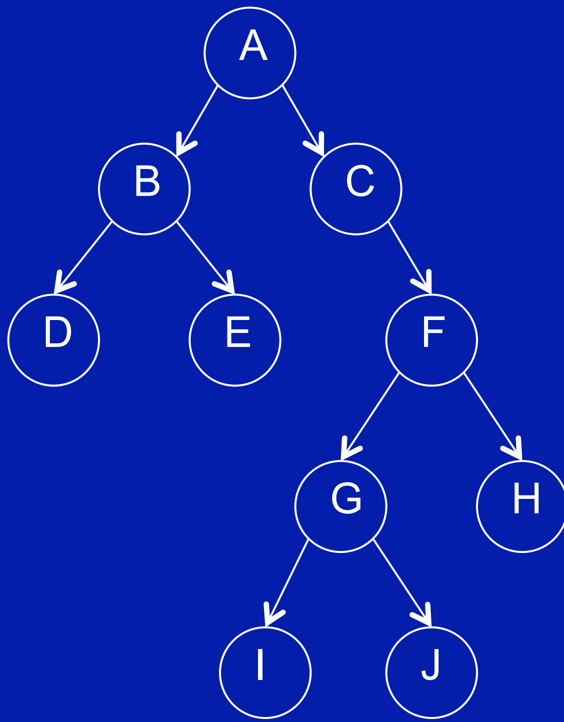
Tree traversal



We'll talk about three kinds of tree traversal: preorder, postorder, and inorder. The pre-, post-, and in- indicate when the root node is processed in relation to its subtrees.

Preorder traversal

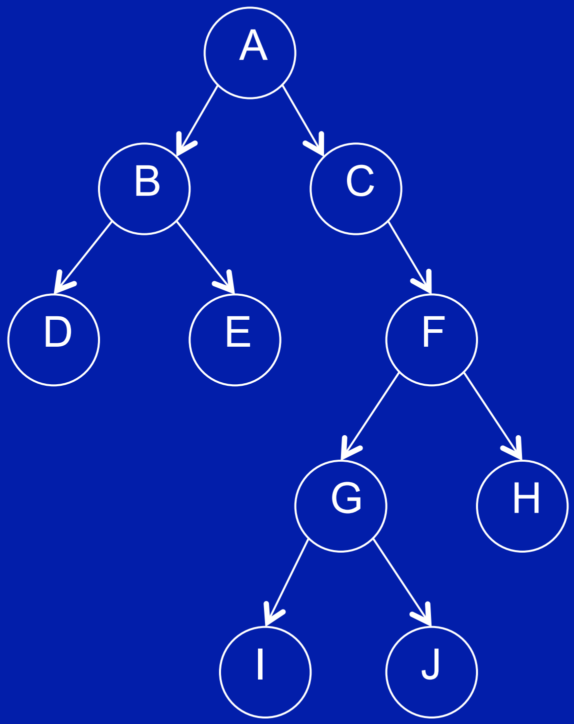
```
if the tree is empty
  return;
else
  visit (process) the root;
  apply preorder to the
    left subtree;
  apply preorder to the
    right subtree;
```



Let's say that visit or process in this case means "print the value". In what order will the values be printed?

Preorder traversal

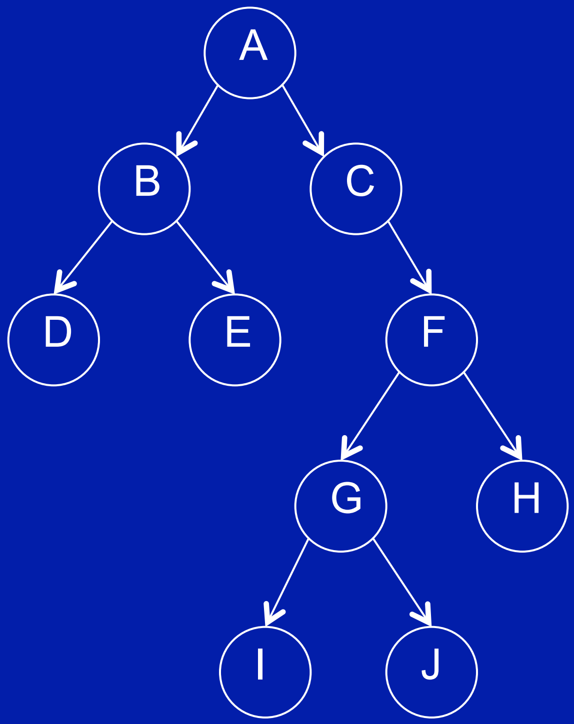
```
if the tree is empty
  return;
else
  visit (process) the root;
  apply preorder to the
    left subtree;
  apply preorder to the
    right subtree;
```



Let’s say that visit or process in this case means “print the value”. In what order will the values be printed?
A

Preorder traversal

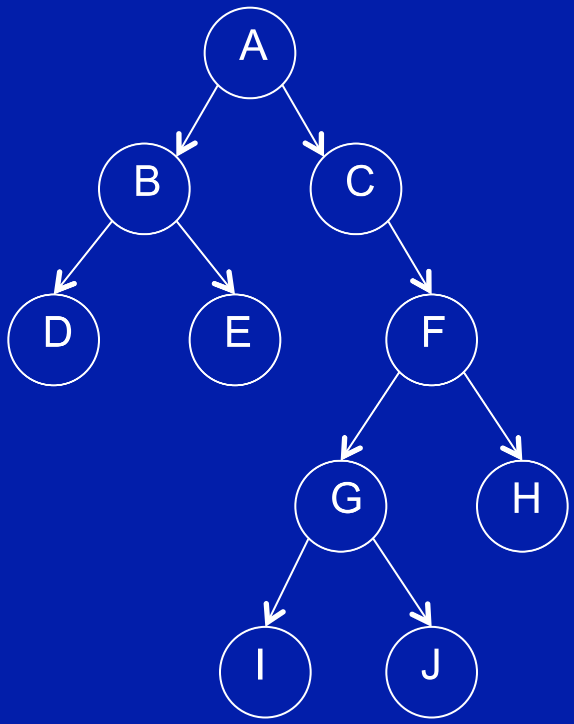
```
if the tree is empty
  return;
else
  visit (process) the root;
  apply preorder to the
    left subtree;
  apply preorder to the
    right subtree;
```



Let's say that visit or process in this case means "print the value". In what order will the values be printed?
A B

Preorder traversal

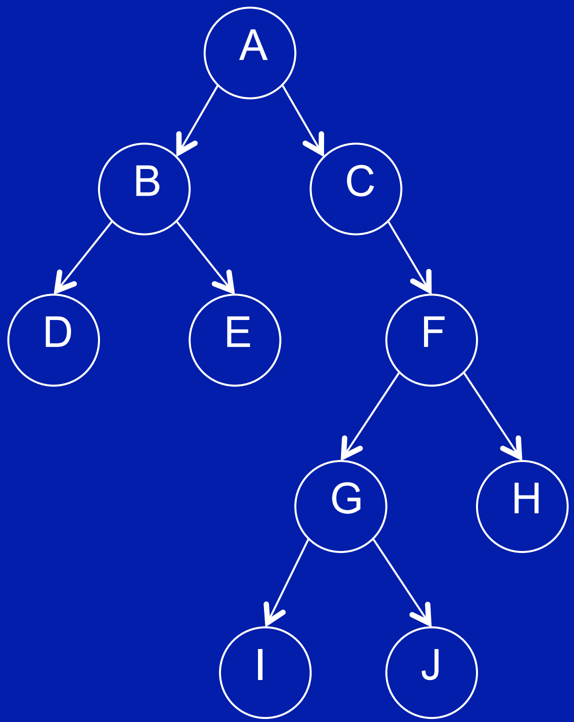
```
if the tree is empty
  return;
else
  visit (process) the root;
  apply preorder to the
    left subtree;
  apply preorder to the
    right subtree;
```



Let’s say that visit or process in this case means “print the value”. In what order will the values be printed?
A B D

Preorder traversal

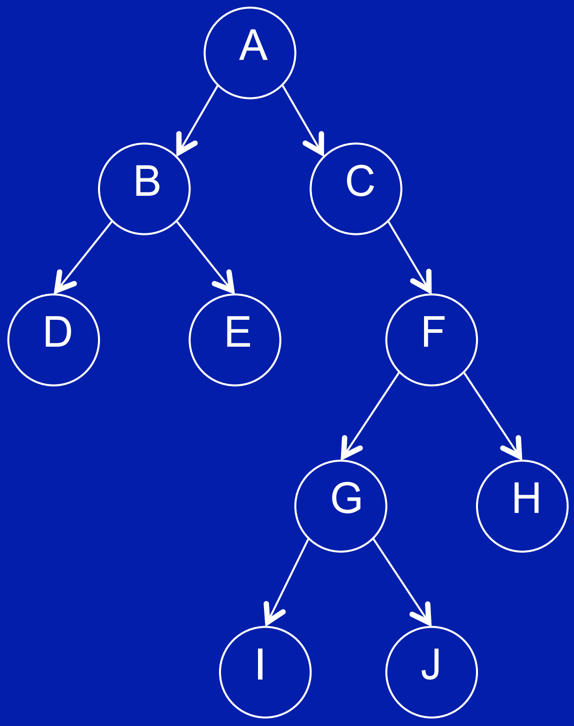
```
if the tree is empty
  return;
else
  visit (process) the root;
  apply preorder to the
    left subtree;
  apply preorder to the
    right subtree;
```



Let's say that visit or process in this case means "print the value". In what order will the values be printed?
A B D E

Preorder traversal

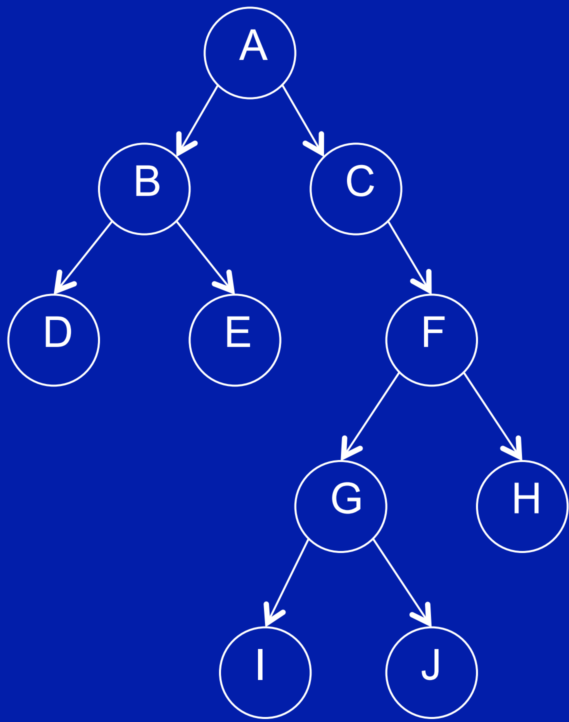
```
if the tree is empty
  return;
else
  visit (process) the root;
  apply preorder to the
    left subtree;
  apply preorder to the
    right subtree;
```



Let’s say that visit or process in this case means “print the value”. In what order will the values be printed?
A B D E C

Preorder traversal

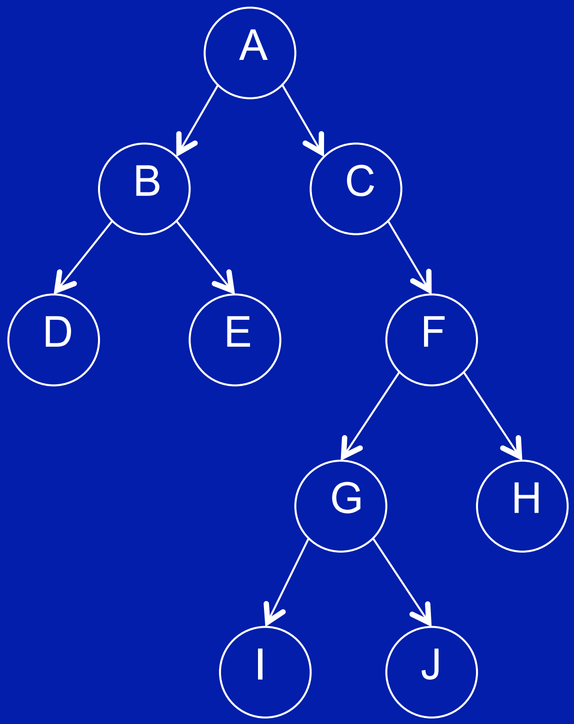
```
if the tree is empty
  return;
else
  visit (process) the root;
  apply preorder to the
    left subtree;
  apply preorder to the
    right subtree;
```



Let’s say that visit or process in this case means “print the value”. In what order will the values be printed?
A B D E C F

Preorder traversal

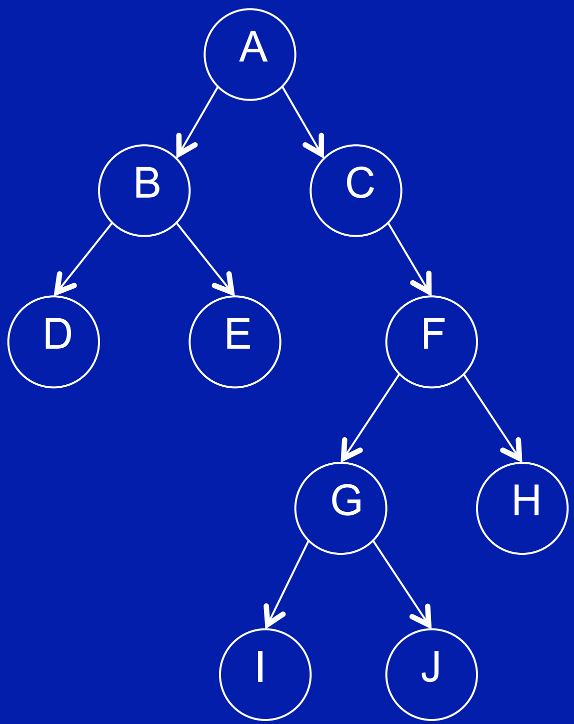
```
if the tree is empty
  return;
else
  visit (process) the root;
  apply preorder to the
    left subtree;
  apply preorder to the
    right subtree;
```



Let’s say that visit or process in this case means “print the value”. In what order will the values be printed?
A B D E C F G

Preorder traversal

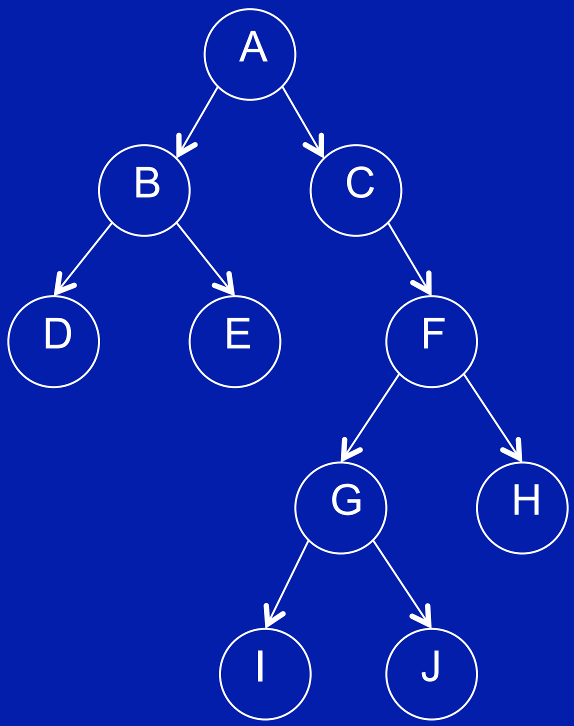
```
if the tree is empty
  return;
else
  visit (process) the root;
  apply preorder to the
    left subtree;
  apply preorder to the
    right subtree;
```



Let's say that visit or process in this case means "print the value". In what order will the values be printed?
A B D E C F G I

Preorder traversal

```
if the tree is empty
  return;
else
  visit (process) the root;
  apply preorder to the
    left subtree;
  apply preorder to the
    right subtree;
```

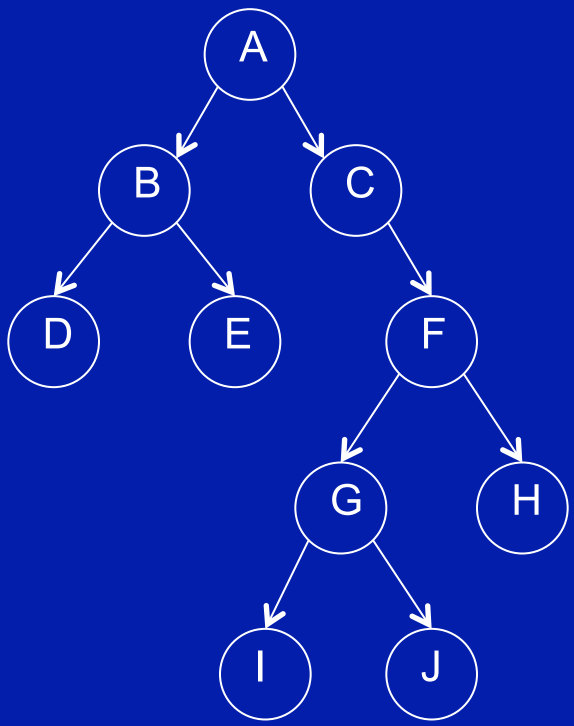


Let’s say that visit or process in this case means “print the value”. In what order will the values be printed?

A B D E C F G I J

Preorder traversal

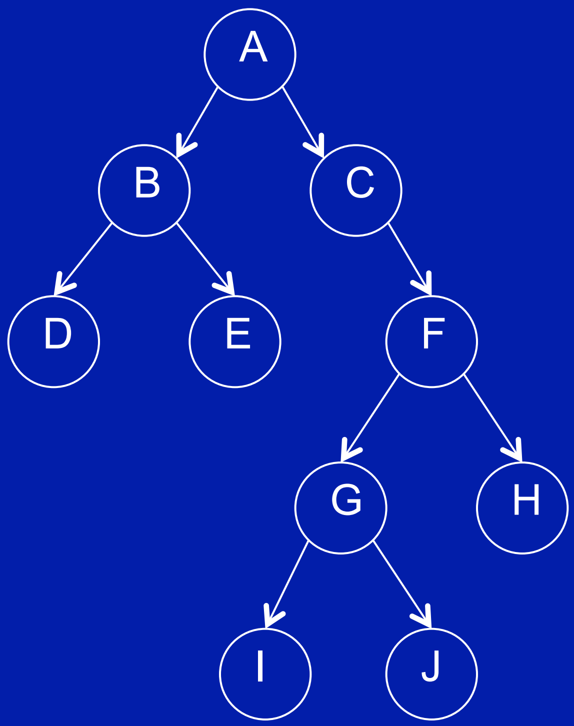
```
if the tree is empty
  return;
else
  visit (process) the root;
  apply preorder to the
    left subtree;
  apply preorder to the
    right subtree;
```



Let's say that visit or process in this case means "print the value". In what order will the values be printed?
A B D E C F G I J H

Inorder traversal

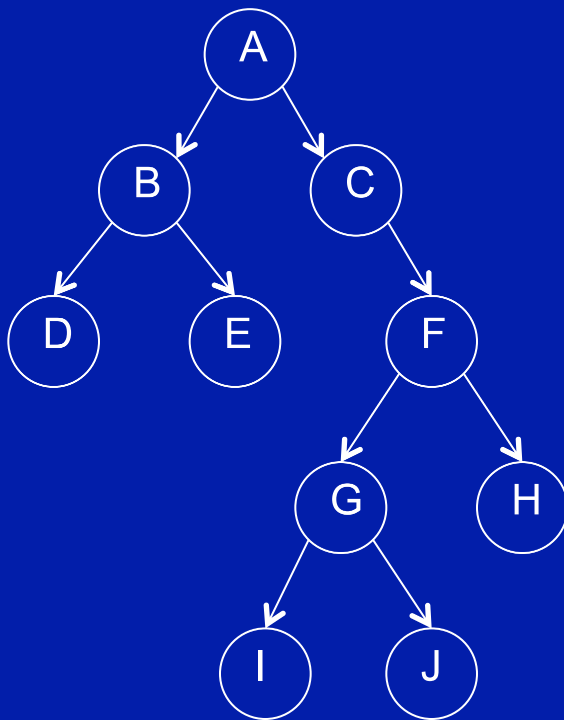
```
if the tree is empty
  return;
else
  apply inorder to the
    left subtree;
  visit (process) the root;
  apply inorder to the
    right subtree;
```



Let’s say that visit or process in this case means “print the value”. In what order will the values be printed?

Inorder traversal

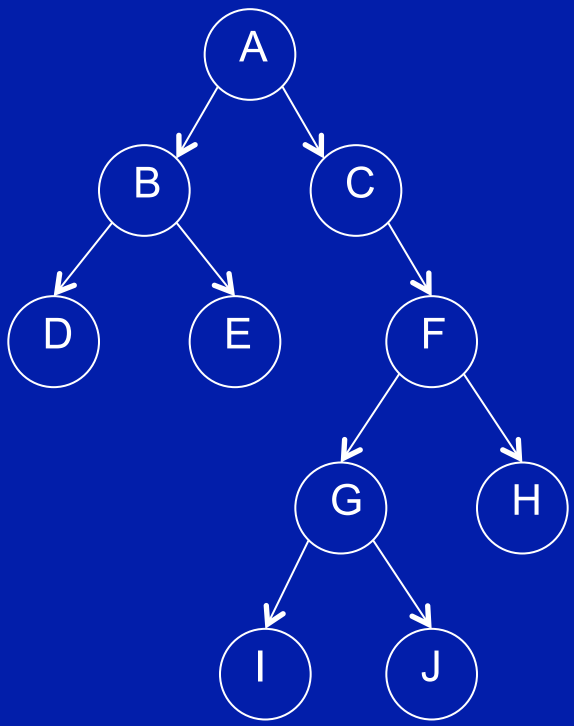
```
if the tree is empty
  return;
else
  apply inorder to the
    left subtree;
  visit (process) the root;
  apply inorder to the
    right subtree;
```



Let's say that visit or process in this case means "print the value". In what order will the values be printed?
D

Inorder traversal

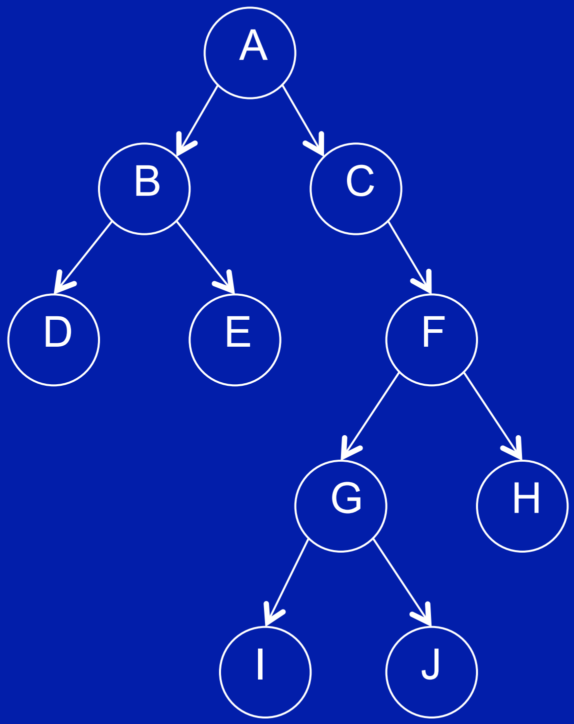
```
if the tree is empty
  return;
else
  apply inorder to the
    left subtree;
  visit (process) the root;
  apply inorder to the
    right subtree;
```



Let's say that visit or process in this case means "print the value". In what order will the values be printed?
D B

Inorder traversal

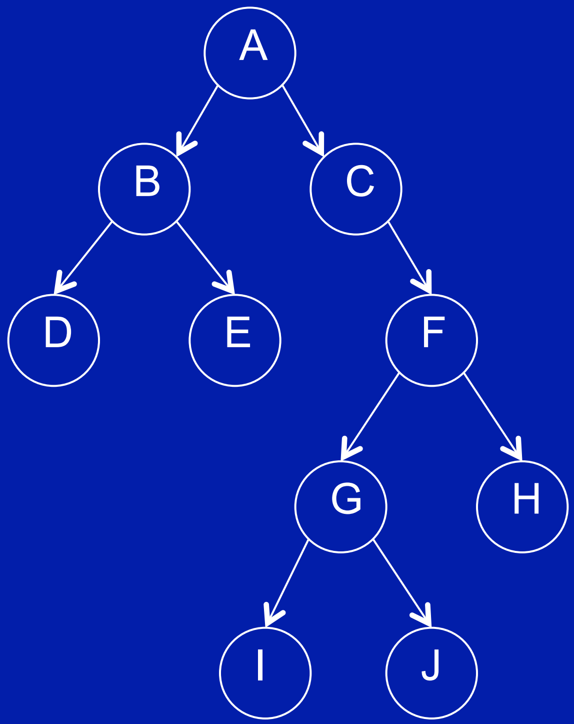
```
if the tree is empty
  return;
else
  apply inorder to the
    left subtree;
  visit (process) the root;
  apply inorder to the
    right subtree;
```



Let’s say that visit or process in this case means “print the value”. In what order will the values be printed?
D B E

Inorder traversal

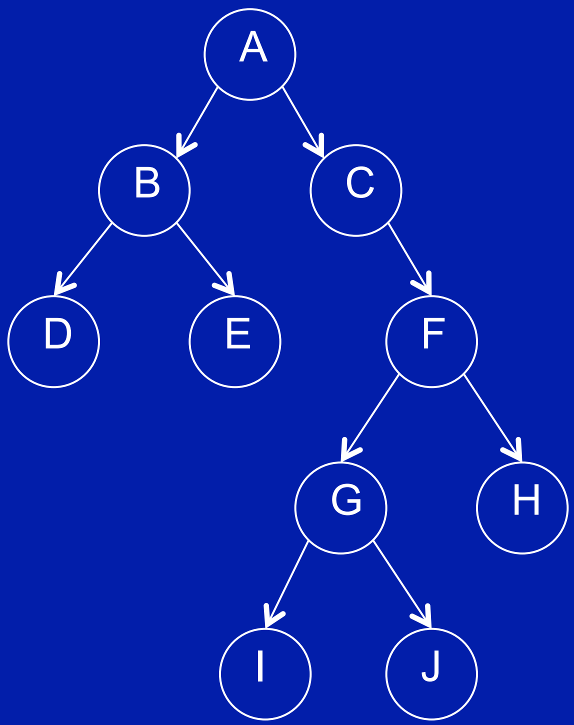
```
if the tree is empty
  return;
else
  apply inorder to the
    left subtree;
  visit (process) the root;
  apply inorder to the
    right subtree;
```



Let's say that visit or process in this case means "print the value". In what order will the values be printed?
D B E A

Inorder traversal

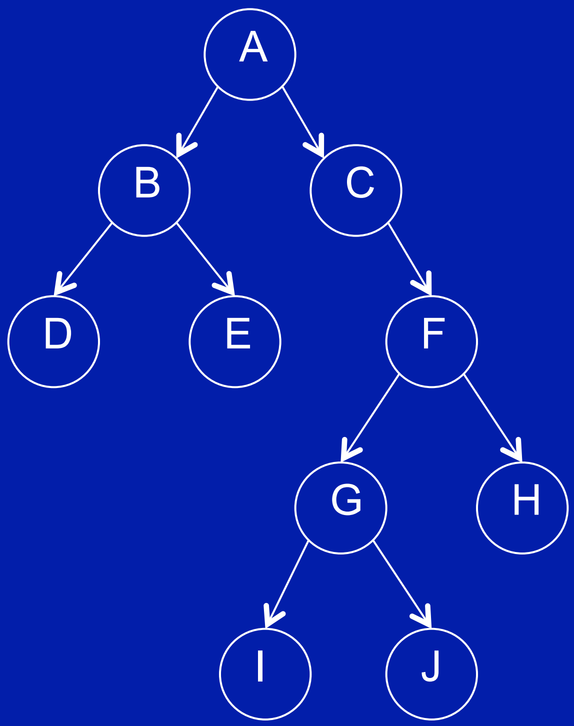
```
if the tree is empty
  return;
else
  apply inorder to the
    left subtree;
  visit (process) the root;
  apply inorder to the
    right subtree;
```



Let's say that visit or process in this case means "print the value". In what order will the values be printed?
D B E A C

Inorder traversal

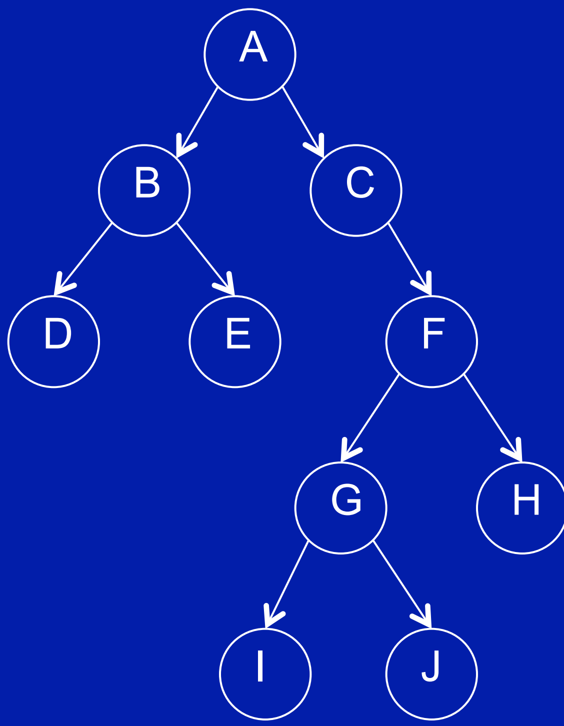
```
if the tree is empty
  return;
else
  apply inorder to the
    left subtree;
  visit (process) the root;
  apply inorder to the
    right subtree;
```



Let’s say that visit or process in this case means “print the value”. In what order will the values be printed?
D B E A C I

Inorder traversal

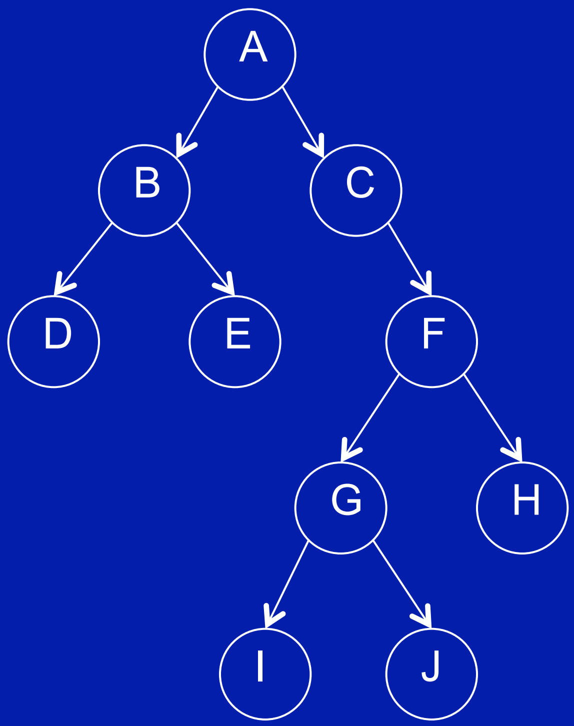
```
if the tree is empty
  return;
else
  apply inorder to the
    left subtree;
  visit (process) the root;
  apply inorder to the
    right subtree;
```



Let's say that visit or process in this case means "print the value". In what order will the values be printed?
D B E A C I G

Inorder traversal

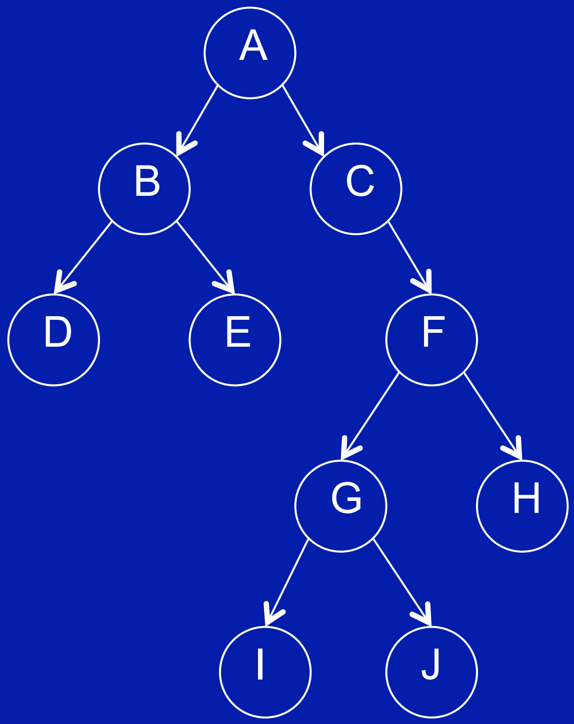
```
if the tree is empty
  return;
else
  apply inorder to the
    left subtree;
  visit (process) the root;
  apply inorder to the
    right subtree;
```



Let's say that visit or process in this case means "print the value". In what order will the values be printed?
D B E A C I G J

Inorder traversal

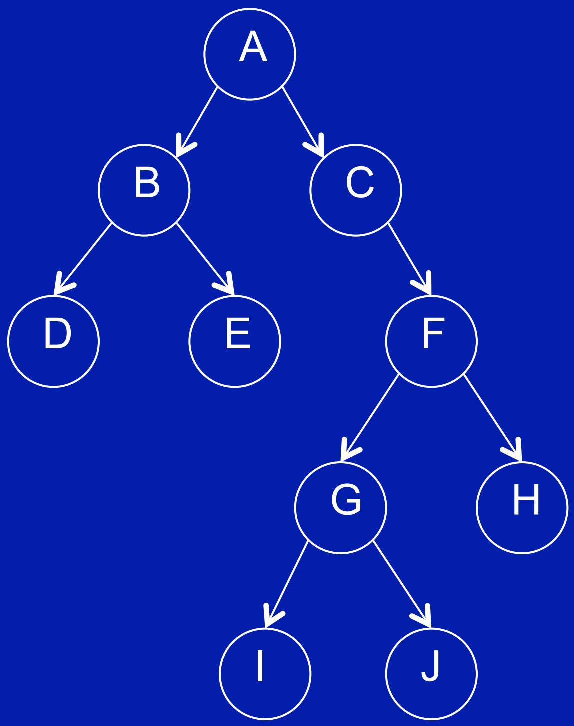
```
if the tree is empty
  return;
else
  apply inorder to the
    left subtree;
  visit (process) the root;
  apply inorder to the
    right subtree;
```



Let’s say that visit or process in this case means “print the value”. In what order will the values be printed?
D B E A C I G J F

Inorder traversal

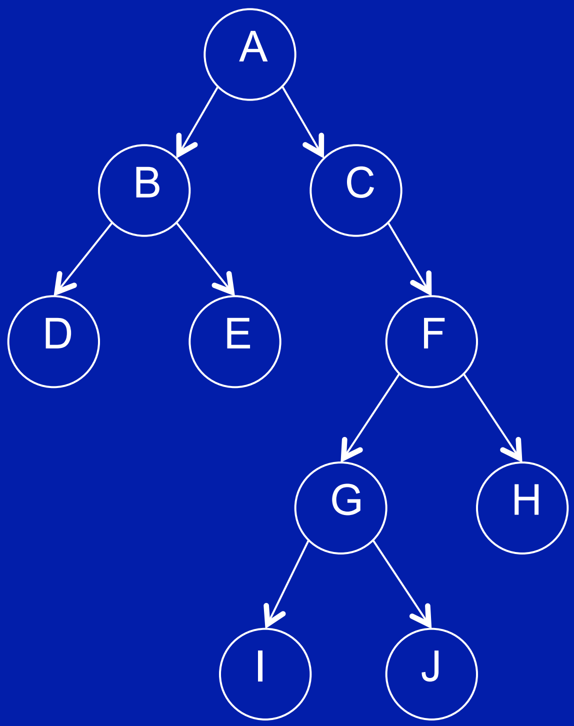
```
if the tree is empty
  return;
else
  apply inorder to the
    left subtree;
  visit (process) the root;
  apply inorder to the
    right subtree;
```



Let's say that visit or process in this case means "print the value". In what order will the values be printed?
D B E A C I G J F H

Postorder traversal

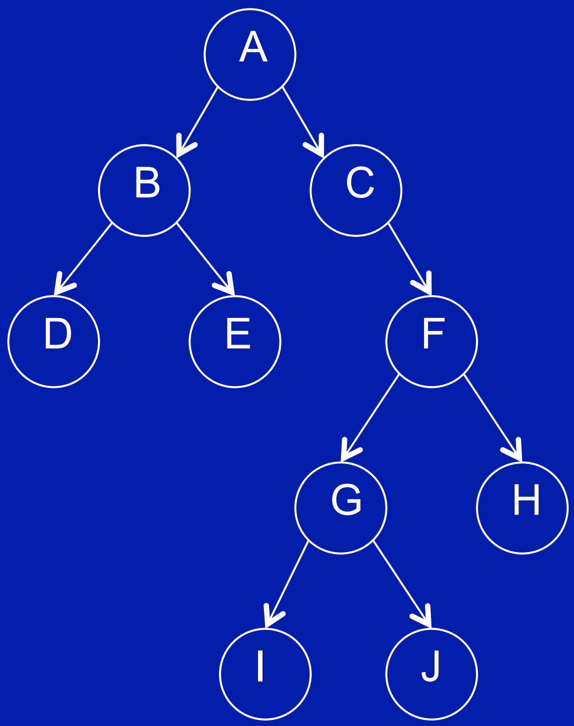
```
if the tree is empty
  return;
else
  apply postorder to the
    left subtree;
  apply postorder to the
    right subtree;
  visit (process) the root;
```



Let’s say that visit or process in this case means “print the value”. In what order will the values be printed?

Postorder traversal

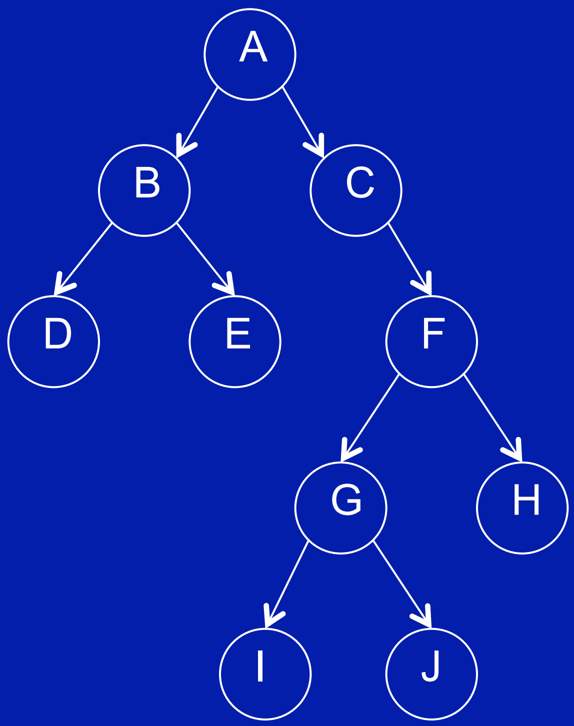
```
if the tree is empty
  return;
else
  apply postorder to the
    left subtree;
  apply postorder to the
    right subtree;
  visit (process) the root;
```



Let’s say that visit or process in this case means “print the value”. In what order will the values be printed?
D

Postorder traversal

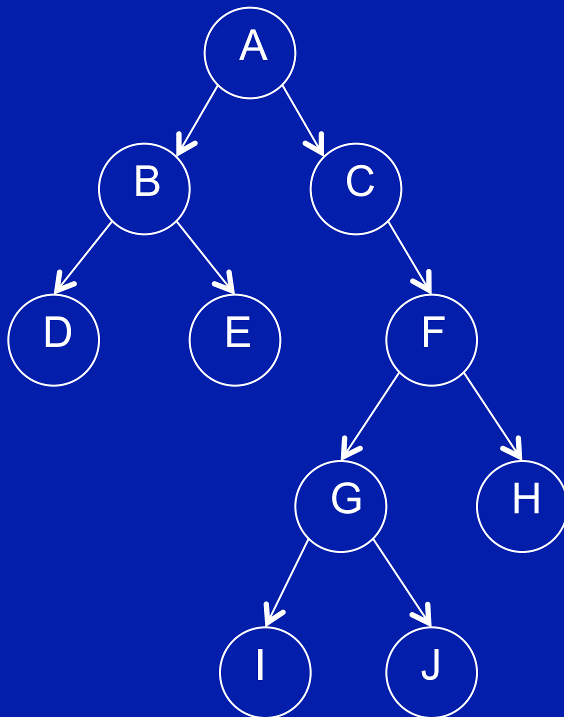
```
if the tree is empty
  return;
else
  apply postorder to the
    left subtree;
  apply postorder to the
    right subtree;
  visit (process) the root;
```



Let's say that visit or process in this case means "print the value". In what order will the values be printed?
D E

Postorder traversal

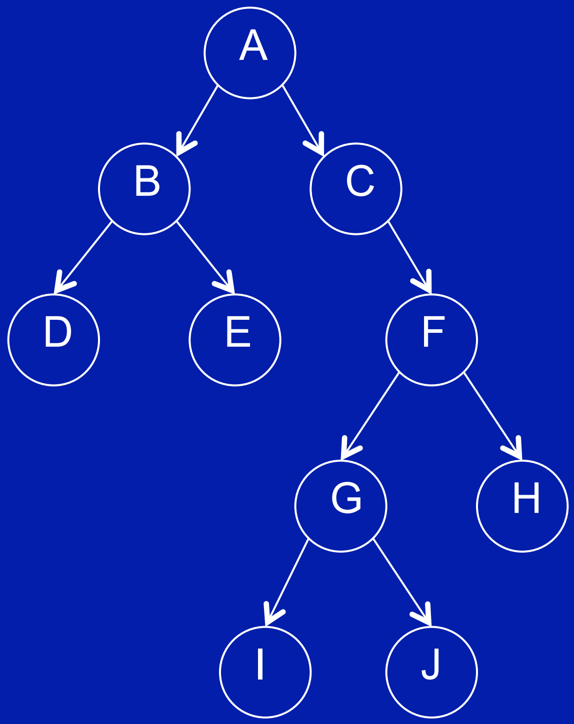
```
if the tree is empty
  return;
else
  apply postorder to the
    left subtree;
  apply postorder to the
    right subtree;
  visit (process) the root;
```



Let’s say that visit or process in this case means “print the value”. In what order will the values be printed?
D E B

Postorder traversal

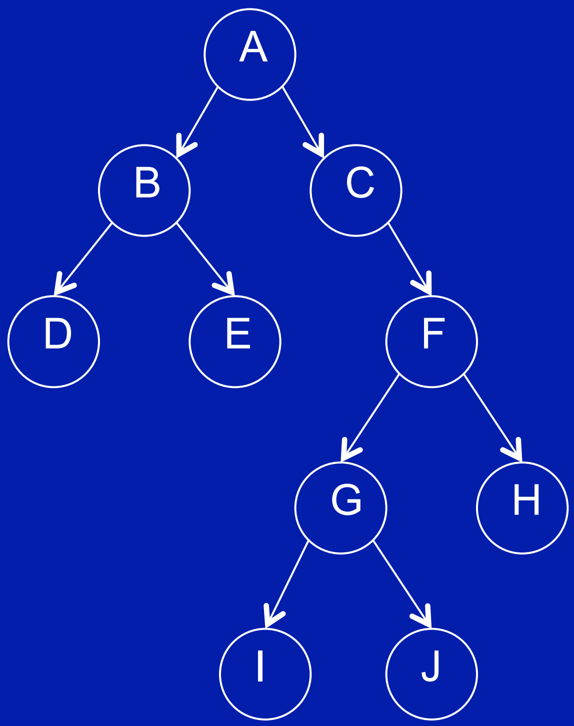
```
if the tree is empty
  return;
else
  apply postorder to the
    left subtree;
  apply postorder to the
    right subtree;
  visit (process) the root;
```



Let’s say that visit or process in this case means “print the value”. In what order will the values be printed?
D E B I

Postorder traversal

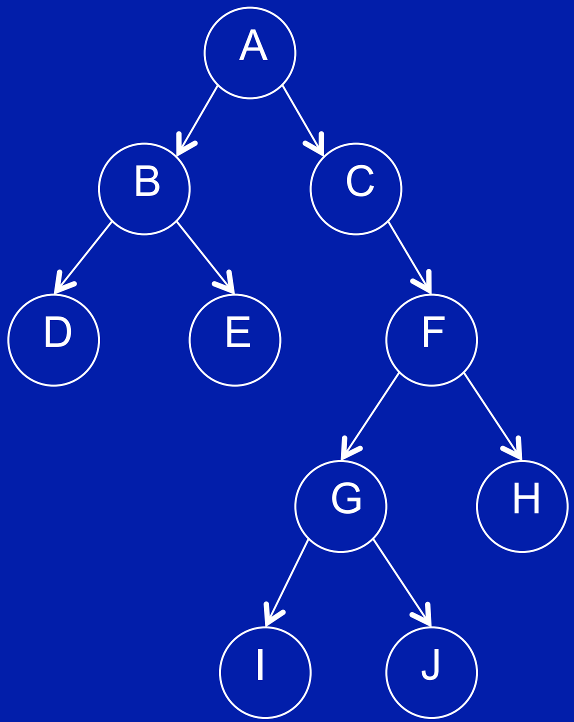
```
if the tree is empty
  return;
else
  apply postorder to the
    left subtree;
  apply postorder to the
    right subtree;
  visit (process) the root;
```



Let’s say that visit or process in this case means “print the value”. In what order will the values be printed?
D E B I J

Postorder traversal

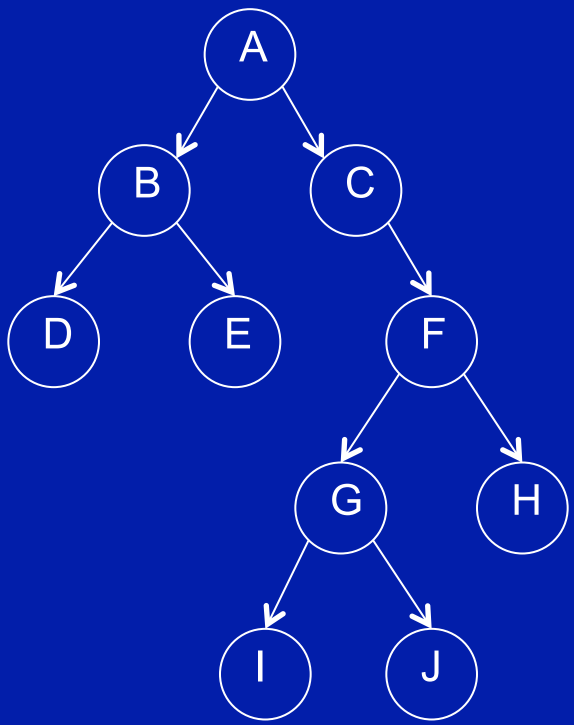
```
if the tree is empty
  return;
else
  apply postorder to the
    left subtree;
  apply postorder to the
    right subtree;
  visit (process) the root;
```



Let’s say that visit or process in this case means “print the value”. In what order will the values be printed?
D E B I J G

Postorder traversal

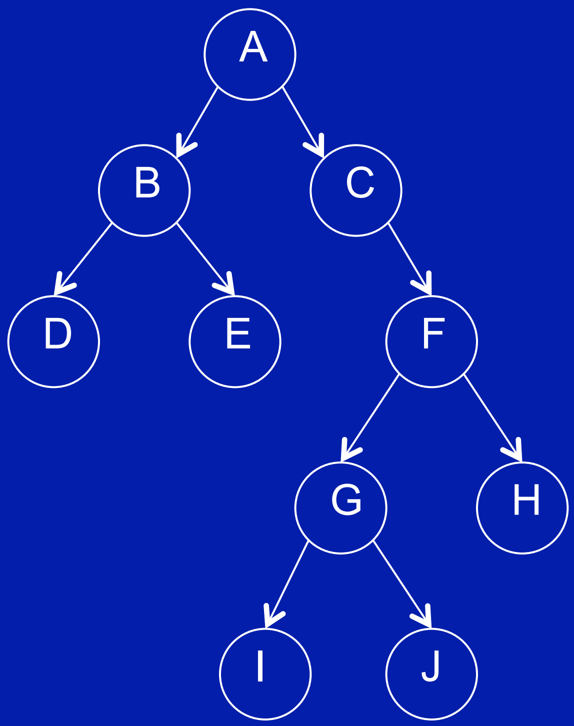
```
if the tree is empty
  return;
else
  apply postorder to the
    left subtree;
  apply postorder to the
    right subtree;
  visit (process) the root;
```



Let’s say that visit or process in this case means “print the value”. In what order will the values be printed?
D E B I J G H

Postorder traversal

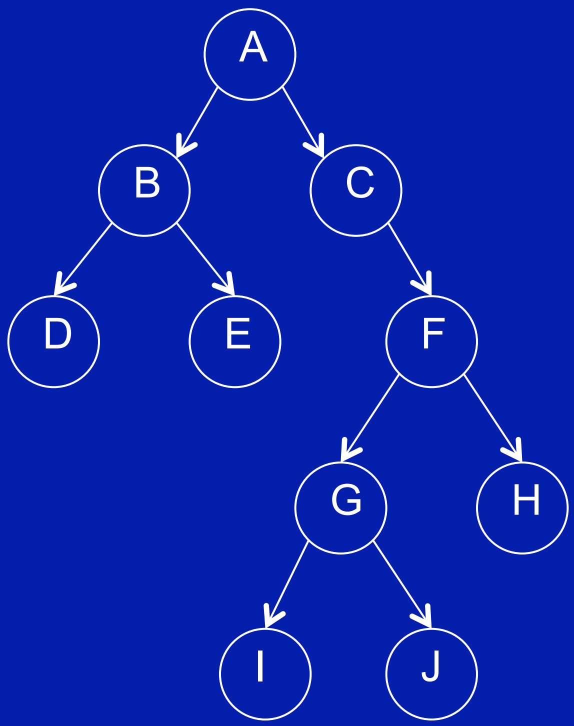
```
if the tree is empty
  return;
else
  apply postorder to the
    left subtree;
  apply postorder to the
    right subtree;
  visit (process) the root;
```



Let’s say that visit or process in this case means “print the value”. In what order will the values be printed?
D E B I J G H F

Postorder traversal

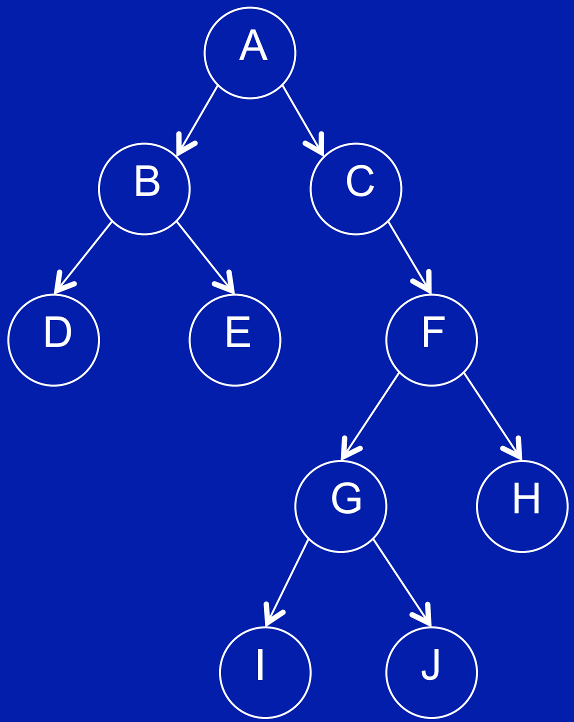
```
if the tree is empty
  return;
else
  apply postorder to the
    left subtree;
  apply postorder to the
    right subtree;
  visit (process) the root;
```



Let's say that visit or process in this case means "print the value". In what order will the values be printed?
D E B I J G H F C

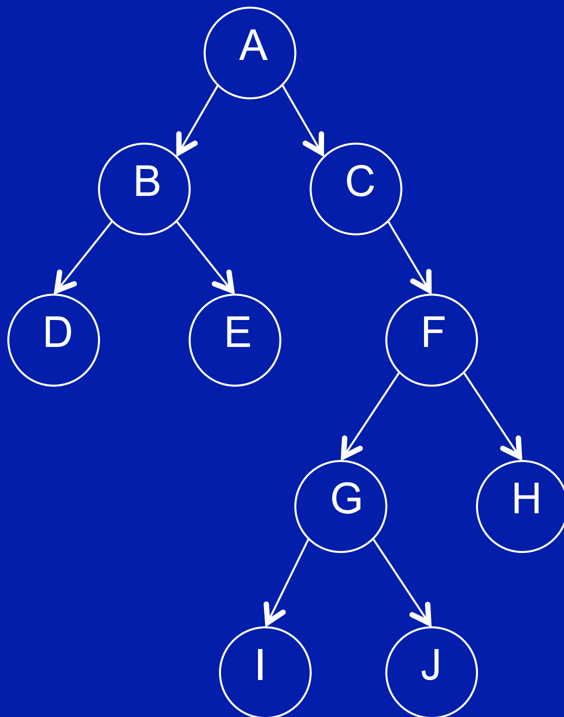
Postorder traversal

```
if the tree is empty
  return;
else
  apply postorder to the
    left subtree;
  apply postorder to the
    right subtree;
  visit (process) the root;
```



Let's say that visit or process in this case means "print the value". In what order will the values be printed?
D E B I J G H F C A

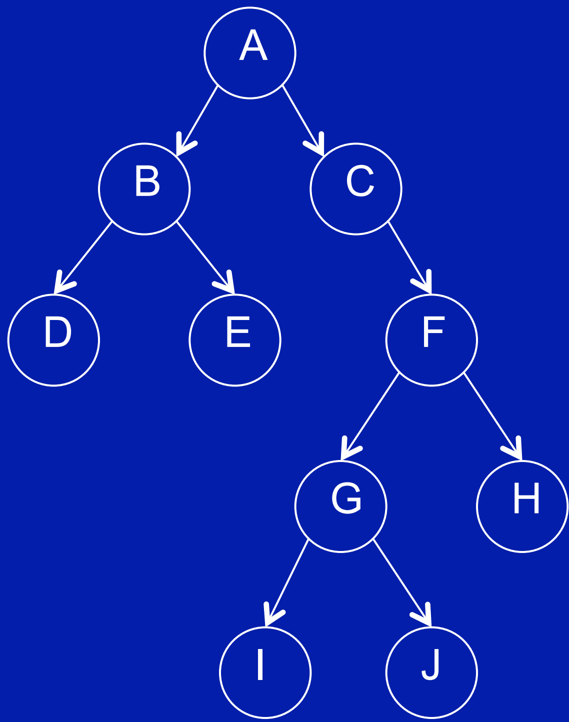
Another traversal



How could I get the traversal (printing) to happen in this order?: A B C D E F G H I J

Level order traversal

```
add root to queue
while queue not emtpy:
  take node from queue
  process the node
  if node->left isn't null
    then put node->left on
    queue
  if node->right isn't null
    then put node->right on
    queue
```



How could I get the traversal (printing) to happen in this order?: A B C D E F G H I J

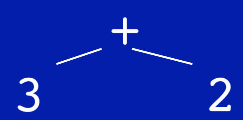
This is called level order traversal.

Binary expression trees

Here's one way in which tree traversal can be useful.
Arithmetic expressions can be represented as binary trees.
Consider the expression (3 + 2) * 5 – 1

Going left to right through the expression, we can start to build the expression tree.

(3 + 2) yields:

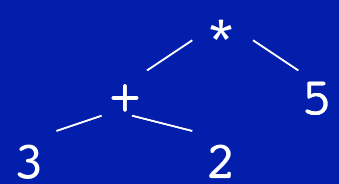


Binary expression trees

Here's one way in which tree traversal can be useful.
Arithmetic expressions can be represented as binary trees.
Consider the expression (3 + 2) * 5 – 1

Going left to right through the expression, we can start to build the expression tree.

* 5 adds this:

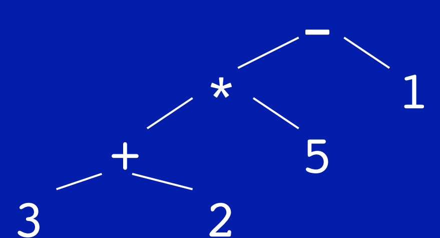


Binary expression trees

Here's one way in which tree traversal can be useful. Arithmetic expressions can be represented as binary trees. Consider the expression (3 + 2) * 5 – 1

Going left to right through the expression, we can start to build the expression tree.

and – 1 results in this:



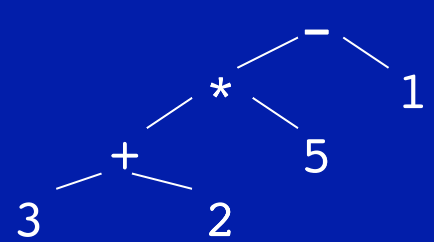
Binary expression trees

Here’s one way in which tree traversal can be useful. Arithmetic expressions can be represented as binary trees. Consider the expression (3 + 2) * 5 – 1

Going left to right through the expression, we can start to build the expression tree.

If we print this tree using postorder traversal, we get

`3 2 + 5 * 1 -`

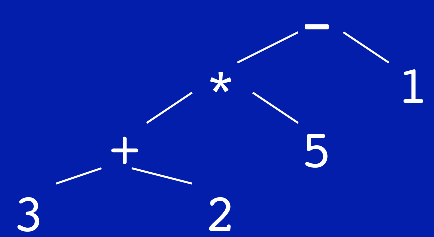


Binary expression trees

Why that's just Reverse Polish Notation! Try it out with your RPN calculator.

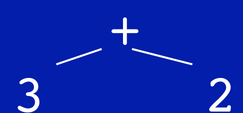
Turns out that it's much easier to write an algorithm to evaluate an expression in RPN than in traditional infix notation. No parentheses are needed, calculations can be done immediately, and you can use a stack to do the calculations (it's just conceptually simpler).

```
3 2 + 5 * 1 -
```



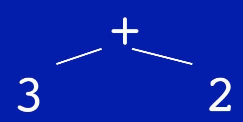
Binary expression trees

Now consider the expression 3 + 2 * 5 – 1
Things begin the same way. 3 + 2 gives:



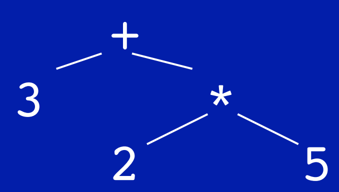
Binary expression trees

Now consider the expression 3 + 2 * 5 – 1
But the * 5 changes things.
The * has higher precedence than the +, so 2 * 5 has to happen first. It has to go lower in the expression tree:



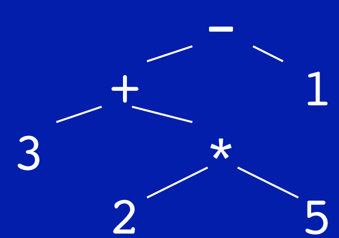
Binary expression trees

Now consider the expression 3 + 2 * 5 – 1
But the * 5 changes things.
The * has higher precedence than the +, so 2 * 5 has to happen first. It has to go lower in the expression tree:



Binary expression trees

Now consider the expression 3 + 2 * 5 – 1
The – 1 has lower precedence , so it goes higher in the expression tree:



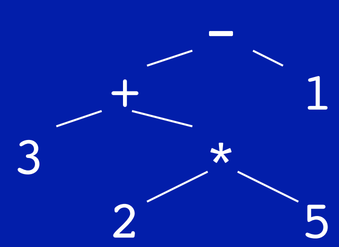
Binary expression trees

Now consider the expression 3 + 2 * 5 – 1

The – 1 has lower precedence , so it goes higher in the expression tree:

A postorder tree traversal gives this RPN expression:

```
3 2 5 * + 1 -
```



Binary expression trees

Now consider the expression 3 + 2 * 5 – 1
The – 1 has lower precedence , so it goes higher in the expression tree:

A postorder tree traversal gives this RPN expression:
Again, you can try this on your RPN calculator.

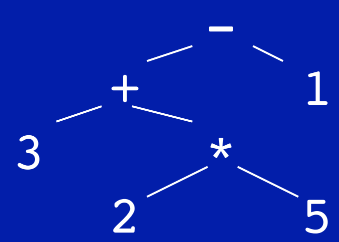
```
      3 2 5 * + 1 -

               -
            /     \
        +           1
      /   \
    3       *
          /   \
        2       5
```

Binary expression trees

DISCLAIMER:
This is an example of the utility of binary trees and postorder tree traversal.

It is not, by any stretch of the imagination, an accurate explanation of how compilers parse arithmetic expressions. For the real story, take a compilers course (CPSC 411).

```
3 2 5 * + 1 -
```



Binary search trees

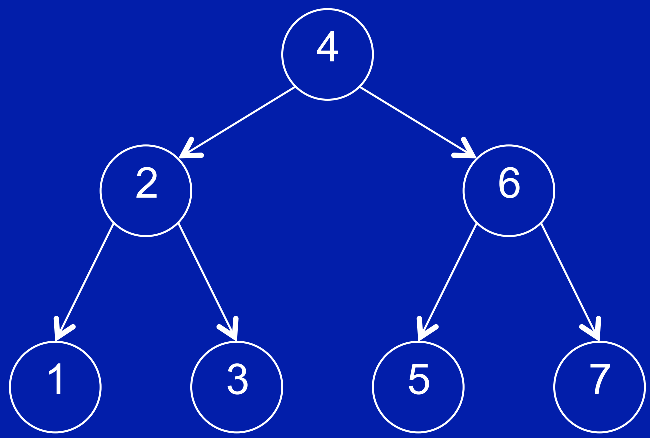
Binary trees are cool, but what we are really interested in is a class of binary trees called binary search trees.

A binary search tree is a binary tree in which every node is

- empty or
- the root of a binary tree in which all the values in the left subtree are less than the value at the root, and all the values in the right subtree are greater than the value at the root.

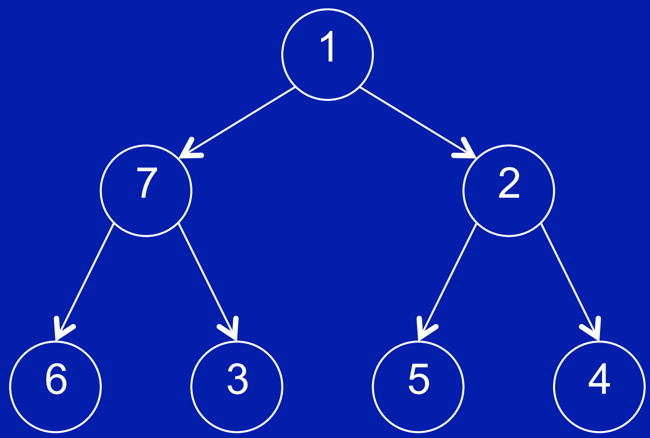
Binary search trees

This is a binary search tree:



Binary search trees

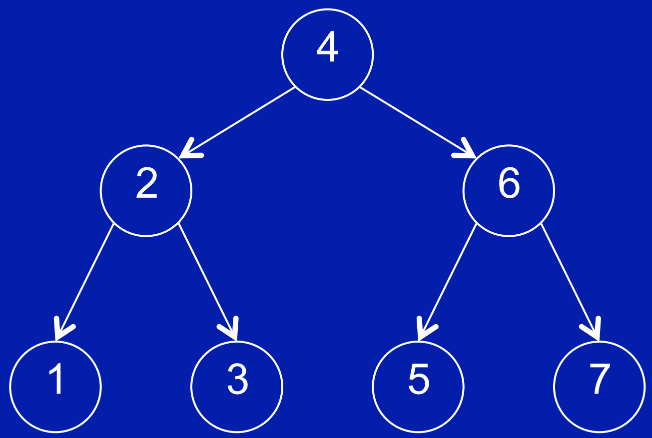
This is not:



Binary search trees

Finding a target value in a binary search tree (BST) is like applying binary search to a sorted array.

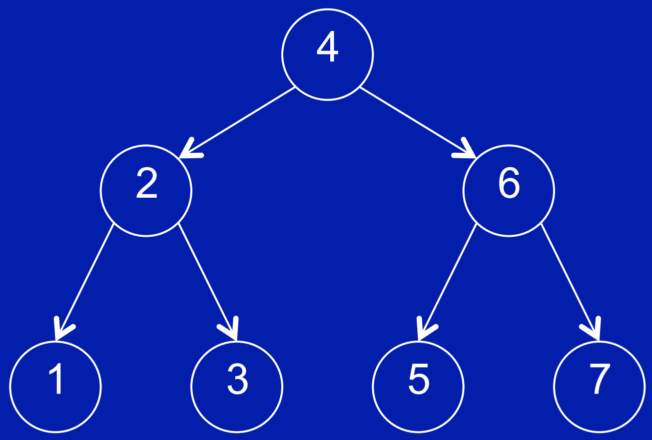
```
if the tree is empty
  (i.e. the root is null)
  then the value is not
  found so return failure;
else if
  the target value = the
  value at the root node
  then return success;
else if
  the target value < the
  value at the root node
  then return the result
  of searching the left
  subtree;
else
  return the result of
  search the right subtree;
```



Binary search trees

Finding a target value in a binary search tree (BST) is like applying binary search to a sorted array.

```
if the tree is empty
  (i.e. the root is null)
  then the value is not
  found so return failure;
else if
  the target value = the
  value at the root node
  then return success;
else if
  the target value < the
  value at the root node
  then return the result
  of searching the left
  subtree;
else
  return the result of
  search the right subtree;
```

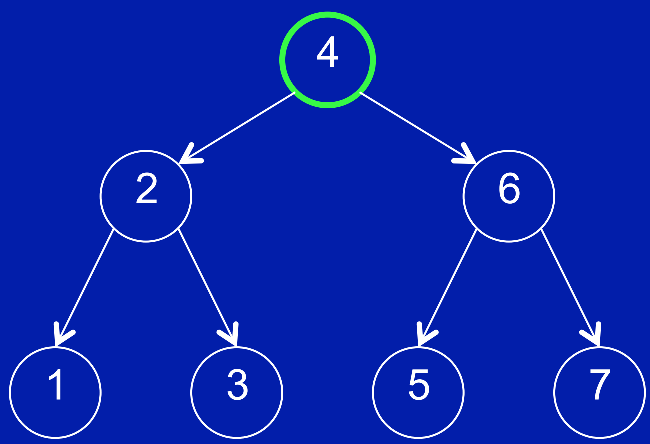


Search for 3.

Binary search trees

Finding a target value in a binary search tree (BST) is like applying binary search to a sorted array.

```
if the tree is empty
  (i.e. the root is null)
  then the value is not
  found so return failure;
else if
  the target value = the
  value at the root node
  then return success;
else if
  the target value < the
  value at the root node
  then return the result
  of searching the left
  subtree;
else
  return the result of
  search the right subtree;
```

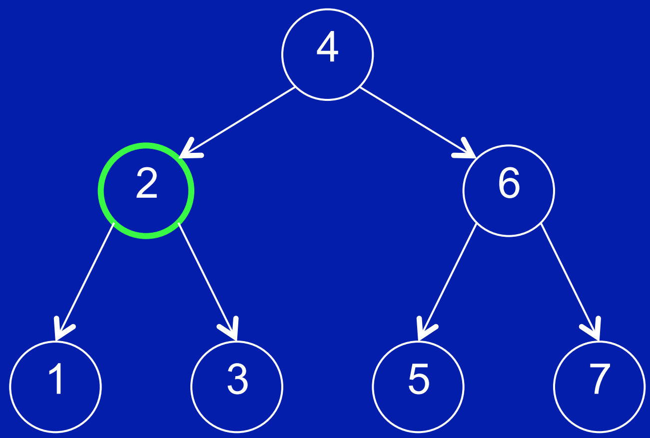


Search for 3.
Is it here?

Binary search trees

Finding a target value in a binary search tree (BST) is like applying binary search to a sorted array.

```
if the tree is empty
  (i.e. the root is null)
  then the value is not
  found so return failure;
else if
  the target value = the
  value at the root node
  then return success;
else if
  the target value < the
  value at the root node
  then return the result
  of searching the left
  subtree;
else
  return the result of
  search the right subtree;
```

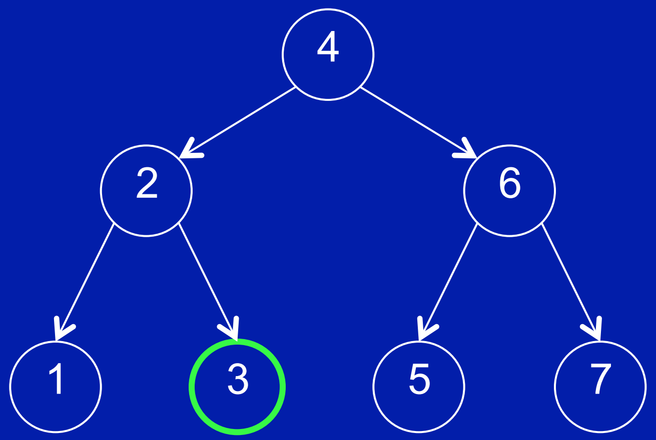


Search for 3.
Is it here? No.
Is it here?

Binary search trees

Finding a target value in a binary search tree (BST) is like applying binary search to a sorted array.

```
if the tree is empty
  (i.e. the root is null)
  then the value is not
  found so return failure;
else if
  the target value = the
  value at the root node
  then return success;
else if
  the target value < the
  value at the root node
  then return the result
  of searching the left
  subtree;
else
  return the result of
  search the right subtree;
```

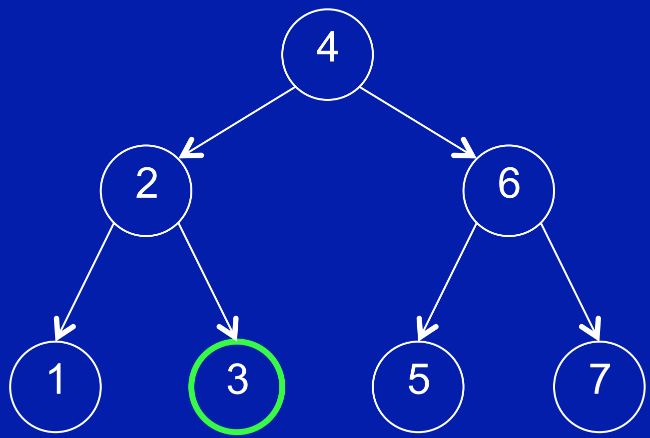


Search for 3.
Is it here? No.
Is it here? No.
Is it here?

Binary search trees

Finding a target value in a binary search tree (BST) is like applying binary search to a sorted array.

```
if the tree is empty
  (i.e. the root is null)
  then the value is not
  found so return failure;
else if
  the target value = the
  value at the root node
  then return success;
else if
  the target value < the
  value at the root node
  then return the result
  of searching the left
  subtree;
else
  return the result of
  search the right subtree;
```

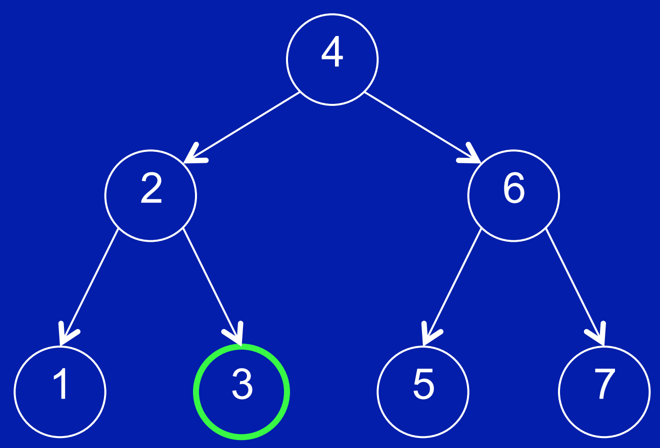


Search for 3.
Is it here? No.
Is it here? No.
Is it here? Yes.
What's your guess as to time complexity of finding a target in this BST?

Binary search trees

Finding a target value in a binary search tree (BST) is like applying binary search to a sorted array.

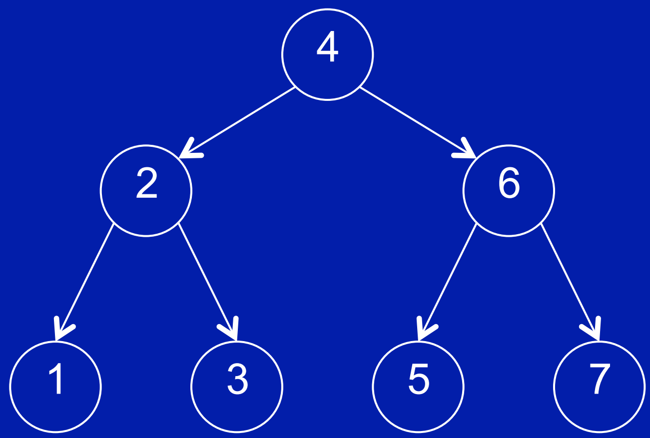
```
if the tree is empty
  (i.e. the root is null)
  then the value is not
  found so return failure;
else if
  the target value = the
  value at the root node
  then return success;
else if
  the target value < the
  value at the root node
  then return the result
  of searching the left
  subtree;
else
  return the result of
  search the right subtree;
```



O(lg n) is a pretty good guess. Why?

Binary search trees

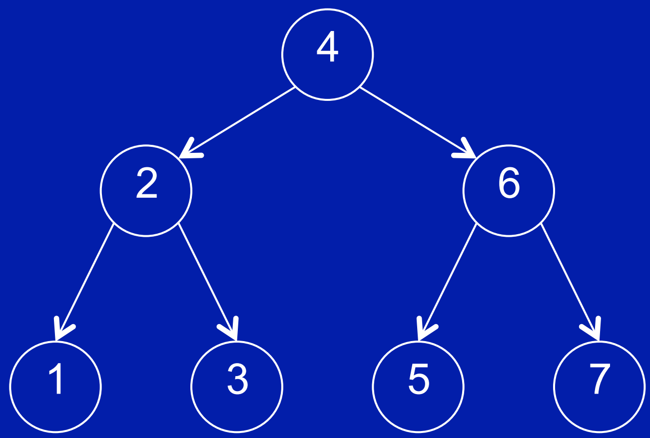
When we add the ‘find’ operation to our BST data structure, we have a new abstract data type. There are other operations that we might want to use frequently with this ADT, but the two we really want to know about now are ‘insert’, because that’s what started this conversation, and ‘delete’ or ‘remove’, because if we’re going to insert things, we also want to delete things.



Binary search trees

insert(target) works like this:

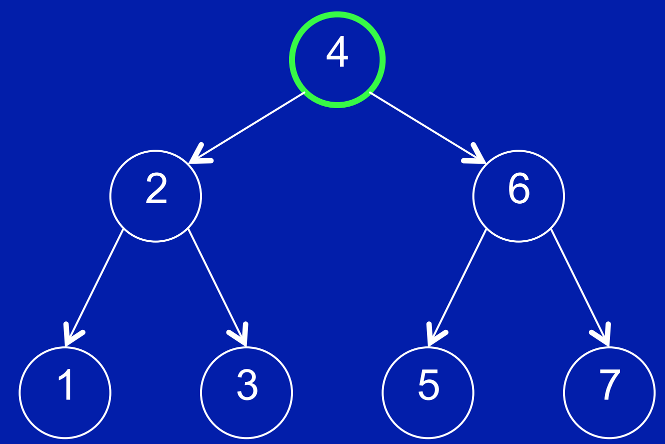
```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success;
else if
  the target value = the
  value at the root then
  the target is already
  in the BST, return failure;
else if
  the target < the value at
  the root then call insert on
  the left subtree;
else
  call insert on the right
  subtree;
```



Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success;
else if
  the target value = the
  value at the root then
  the target is already
  in the BST, return failure;
else if
  the target < the value at
  the root then call insert on
  the left subtree;
else
  call insert on the right
  subtree;
```

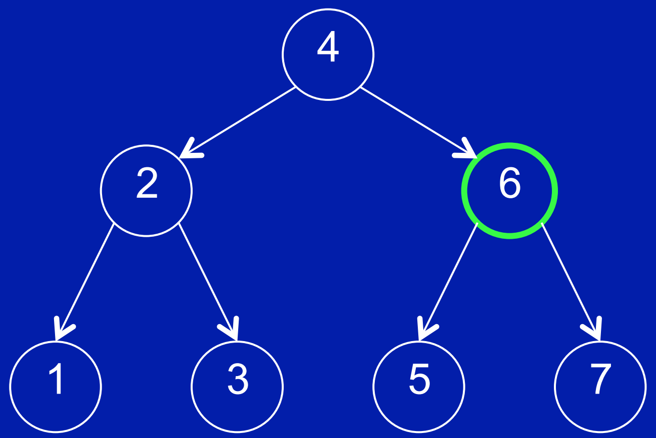


Insert 8.
Is this the place?

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success;
else if
  the target value = the
  value at the root then
  the target is already
  in the BST, return failure;
else if
  the target < the value at
  the root then call insert on
  the left subtree;
else
  call insert on the right
  subtree;
```

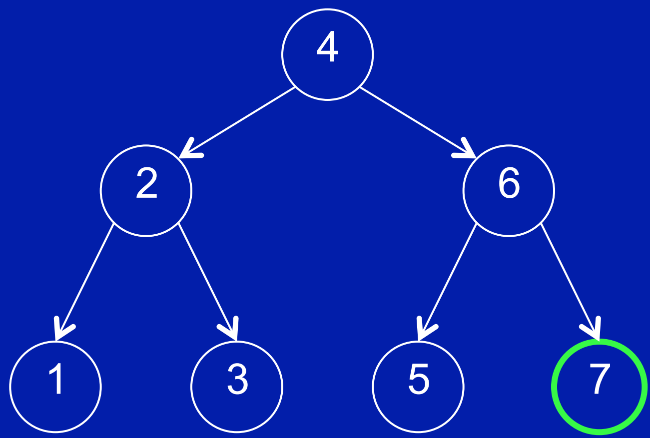


Insert 8.
Is this the place? No.
Is this the place?

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success;
else if
  the target value = the
  value at the root then
  the target is already
  in the BST, return failure;
else if
  the target < the value at
  the root then call insert on
  the left subtree;
else
  call insert on the right
  subtree;
```

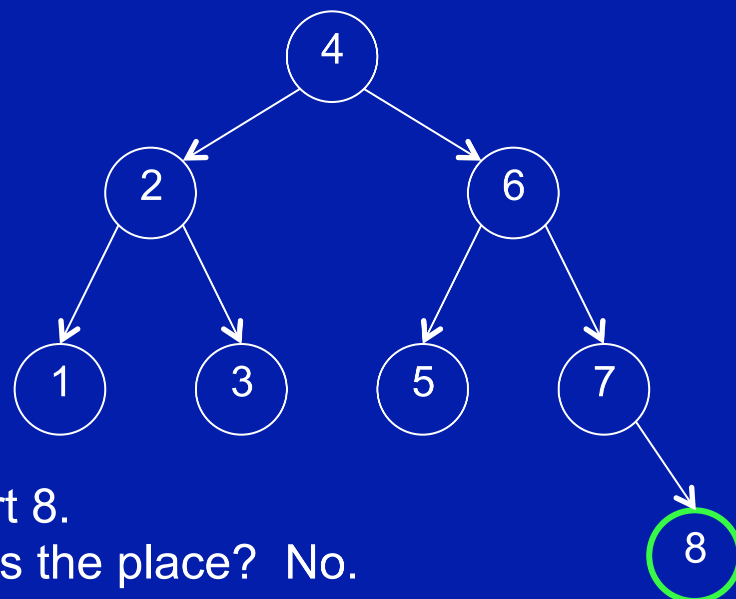


Insert 8.
Is this the place? No.
Is this the place? No.
Is this the place?

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success;
else if
  the target value = the
  value at the root then
  the target is already
  in the BST, return failure;
else if
  the target < the value at
  the root then call insert on
  the left subtree;
else
  call insert on the right
  subtree;
```

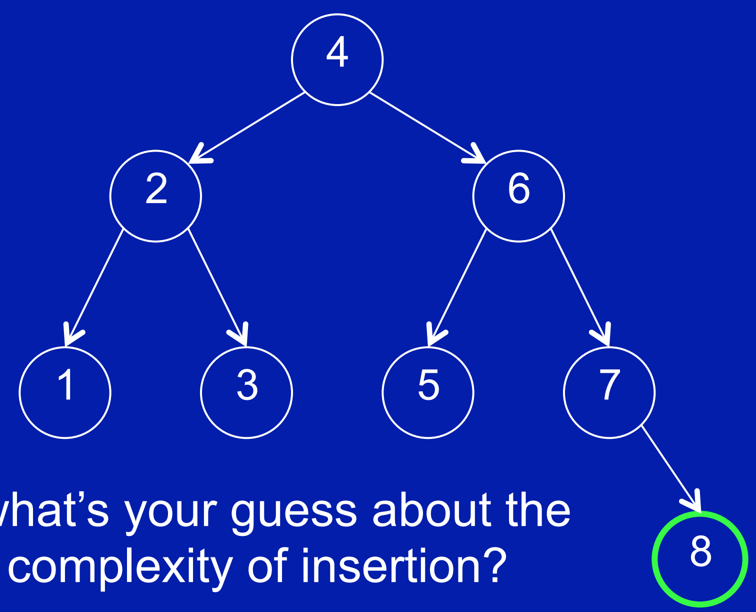


Insert 8.
Is this the place? No.
Is this the place? No.
Is this the place? No.
But the right subtree of 7 is empty, so we create a new node which is now the root of right subtree of 7.

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success;
else if
  the target value = the
  value at the root then
  the target is already
  in the BST, return failure;
else if
  the target < the value at
  the root then call insert on
  the left subtree;
else
  call insert on the right
  subtree;
```

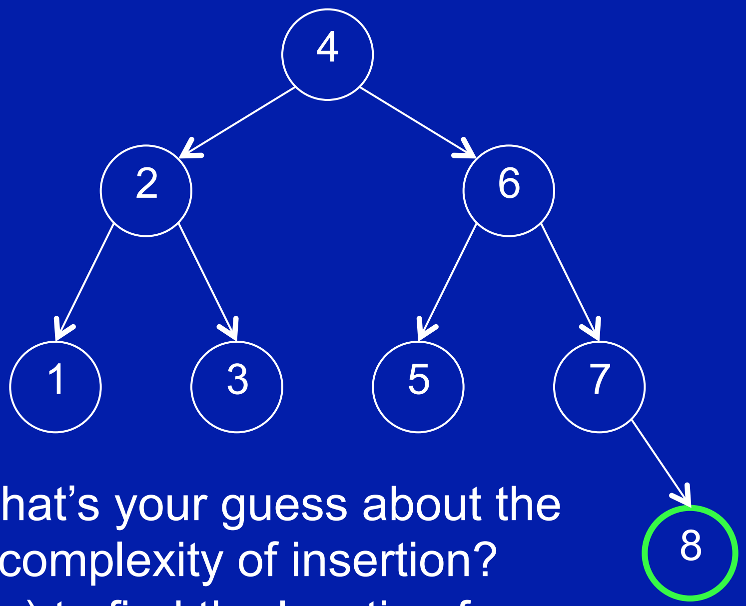


So what's your guess about the time complexity of insertion?

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success;
else if
  the target value = the
  value at the root then
  the target is already
  in the BST, return failure;
else if
  the target < the value at
  the root then call insert on
  the left subtree;
else
  call insert on the right
  subtree;
```



So what's your guess about the time complexity of insertion?
O(lg n) to find the location for insertion plus some fixed time to create the new node and add the link.

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success;
else if
  the target value = the
  value at the root then
  the target is already
  in the BST, return failure;
else if
  the target < the value at
  the root then call insert on
  the left subtree;
else
  call insert on the right
  subtree;
```

How do you build a binary search tree from scratch?

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success;
else if
  the target value = the
  value at the root then
  the target is already
  in the BST, return failure;
else if
  the target < the value at
  the root then call insert on
  the left subtree;
else
  call insert on the right
  subtree;
```

Building a binary search tree from an unsorted sequence of values is just the repeated application of insert.

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success;
else if
  the target value = the
  value at the root then
  the target is already
  in the BST, return failure;
else if
  the target < the value at
  the root then call insert on
  the left subtree;
else
  call insert on the right
  subtree;
```

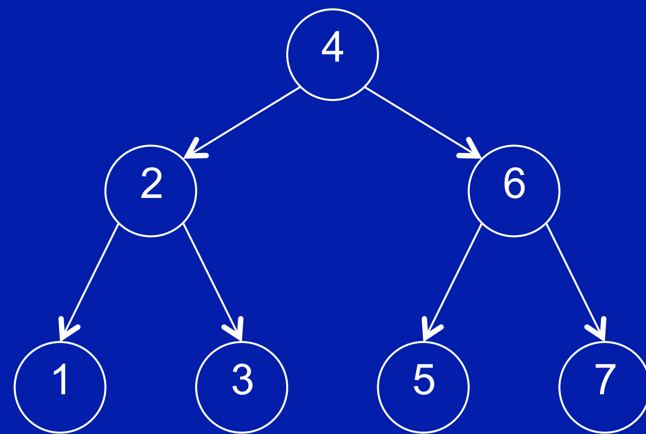
Building a binary search tree from an unsorted sequence of values is just the repeated application of insert.

What do you get if you insert
4 6 2 5 3 1 7 in that order?

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success;
else if
  the target value = the
  value at the root then
  the target is already
  in the BST, return failure;
else if
  the target < the value at
  the root then call insert on
  the left subtree;
else
  call insert on the right
  subtree;
```



Building a binary search tree from an unsorted sequence of values is just the repeated application of insert.

What do you get if you insert
4 6 2 5 3 1 7 in that order?

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success;
else if
  the target value = the
  value at the root then
  the target is already
  in the BST, return failure;
else if
  the target < the value at
  the root then call insert on
  the left subtree;
else
  call insert on the right
  subtree;
```

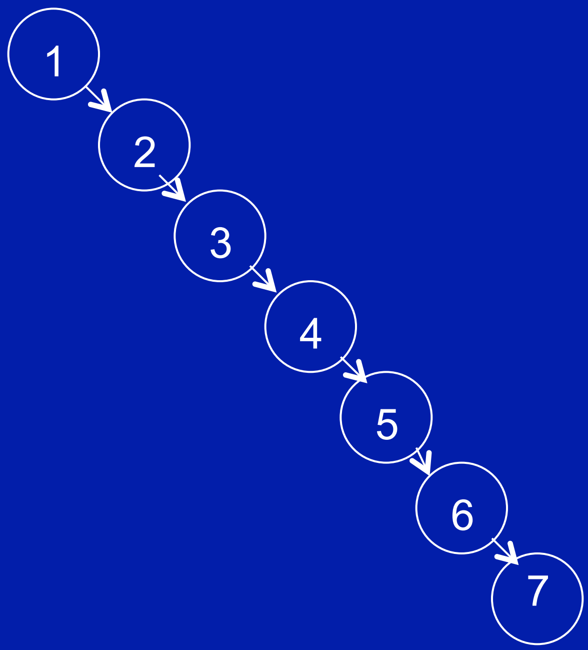
Building a binary search tree from an unsorted sequence of values is just the repeated application of insert.

What do you get if you insert 1 2 3 4 5 6 7 in that order?

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success;
else if
  the target value = the
  value at the root then
  the target is already
  in the BST, return failure;
else if
  the target < the value at
  the root then call insert on
  the left subtree;
else
  call insert on the right
  subtree;
```



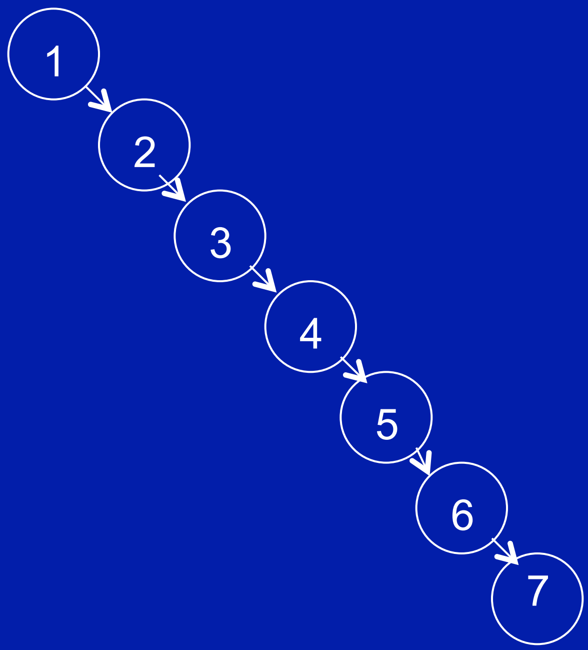
Building a binary search tree from an unsorted sequence of values is just the repeated application of insert.

What do you get if you insert 1 2 3 4 5 6 7 in that order?

Binary search trees

insert(target) works like this:

```
if the tree is empty
  then put the target to
  be inserted in a new
  node which is now the
  root of the BST and
  return success;
else if
  the target value = the
  value at the root then
  the target is already
  in the BST, return failure;
else if
  the target < the value at
  the root then call insert on
  the left subtree;
else
  call insert on the right
  subtree;
```



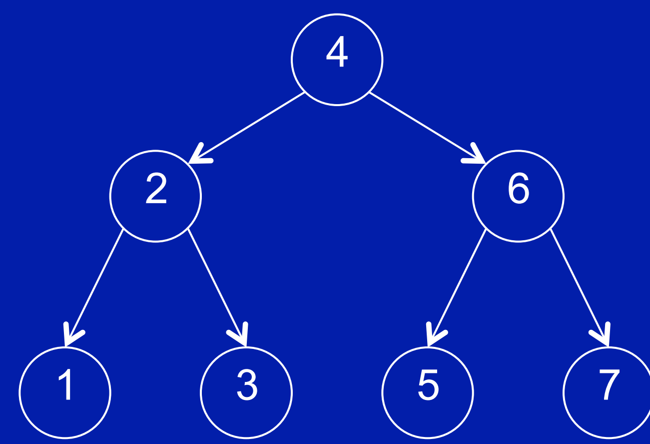
Building a binary search tree from an unsorted sequence of values is just the repeated application of insert.

What do you get if you insert
1 2 3 4 5 6 7 in that order?
Do we have an issue here?

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

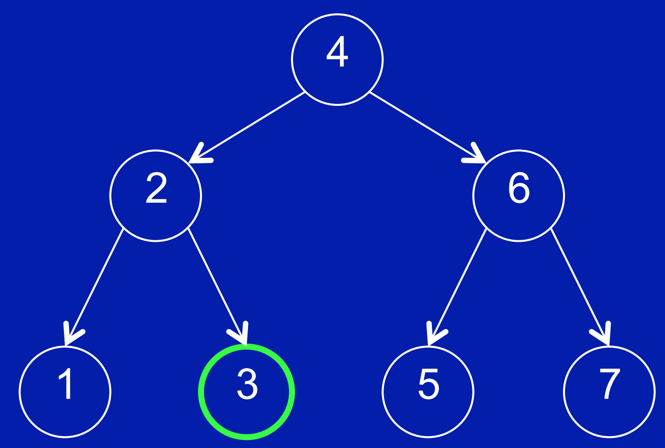
```
use binary search to find
  the target in the BST;
if the target to be deleted
  is a leaf node then its
  parent's pointer to that
  leaf node is set to null;
```



Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST;
if the target to be deleted
  is a leaf node then its
  parent's pointer to that
  leaf node is set to null;
```

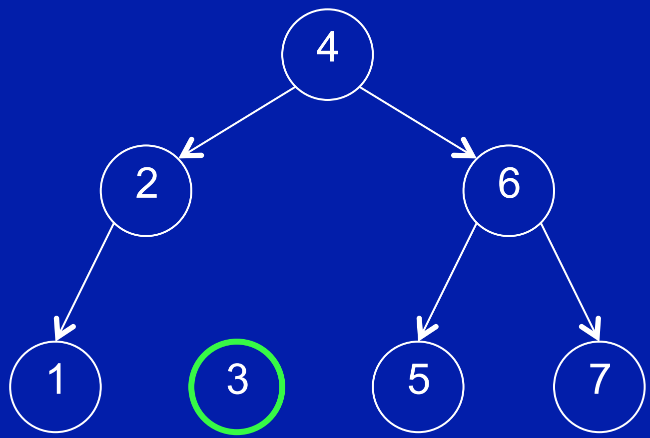


Delete 3.

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST;
if the target to be deleted
  is a leaf node then its
  parent's pointer to that
  leaf node is set to null;
```

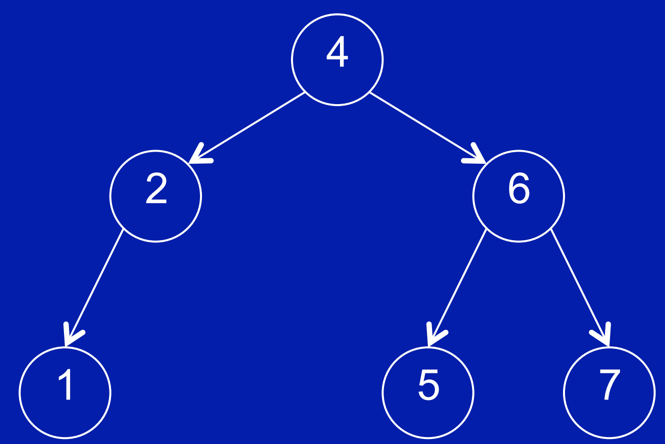


Delete 3.

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST;
if the target to be deleted
  is a leaf node then its
  parent's pointer to that
  leaf node is set to null;
```

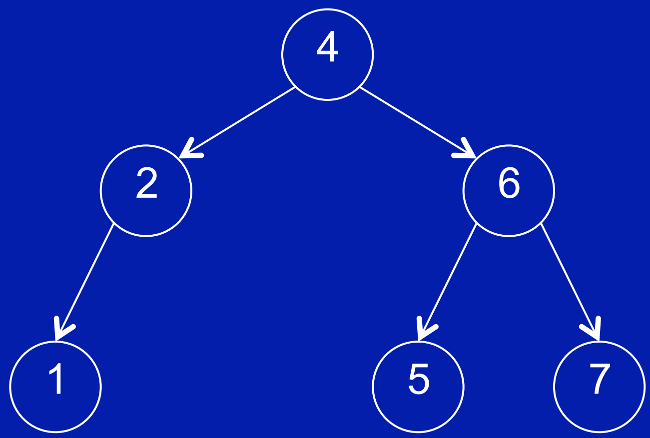


Delete 3.

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

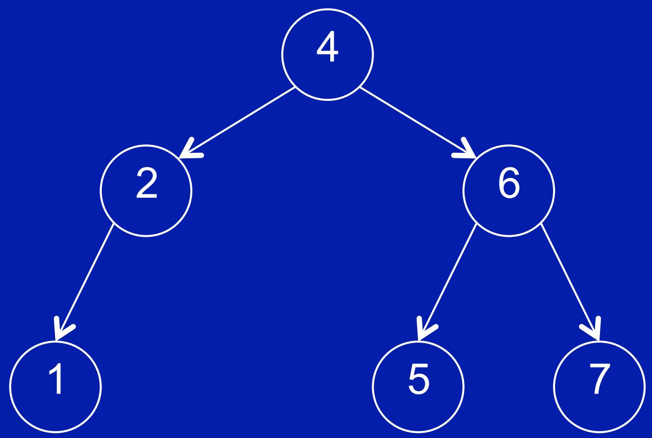
```
use binary search to find
  the target in the BST;
if the target to be deleted
  has only a left or a right
  child, then replace the
  target with the child;
```



Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST;
if the target to be deleted
  has only a left or a right
  child, then replace the
  target with the child;
```

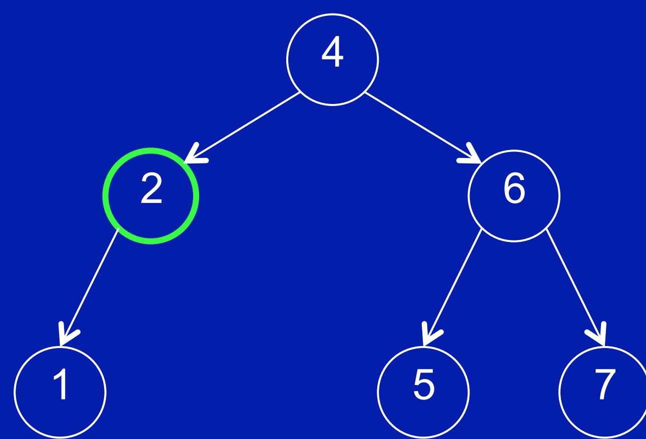


Delete 2.

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST;
if the target to be deleted
  has only a left or a right
  child, then replace the
  target with the child;
```

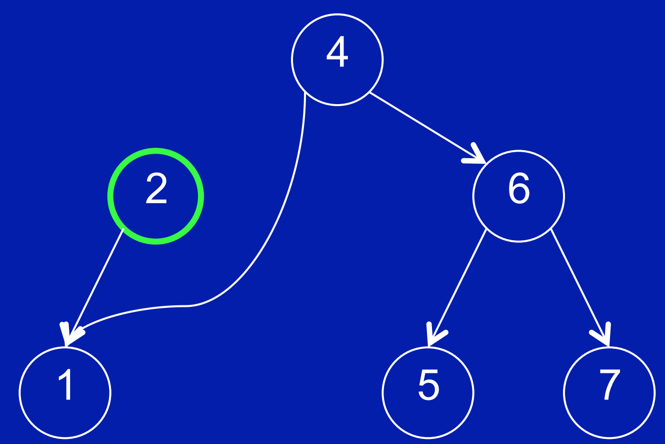


Delete 2.

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST;
if the target to be deleted
  has only a left or a right
  child, then replace the
  target with the child;
```

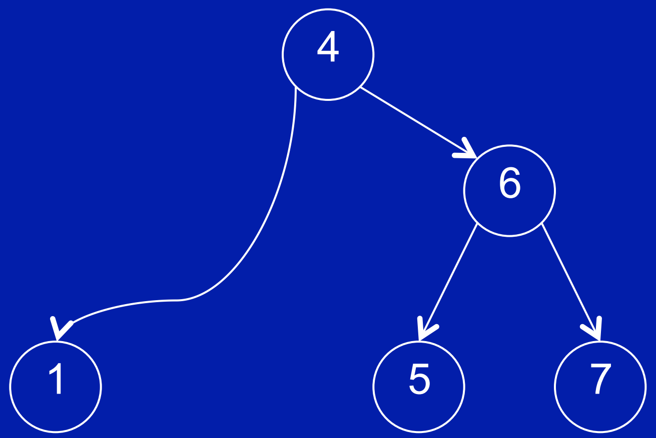


Delete 2.

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST;
if the target to be deleted
  has only a left or a right
  child, then replace the
  target with the child;
```

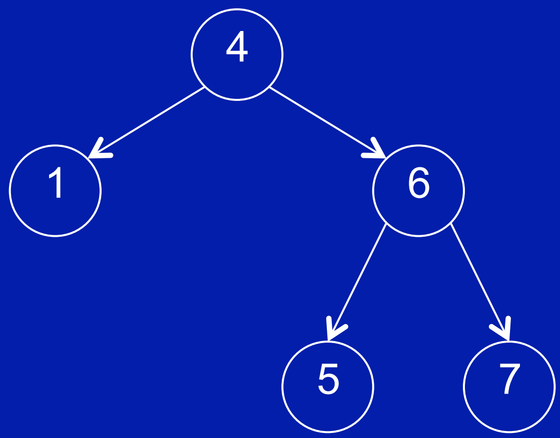


Delete 2.

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST;
if the target to be deleted
  has only a left or a right
  child, then replace the
  target with the child;
```

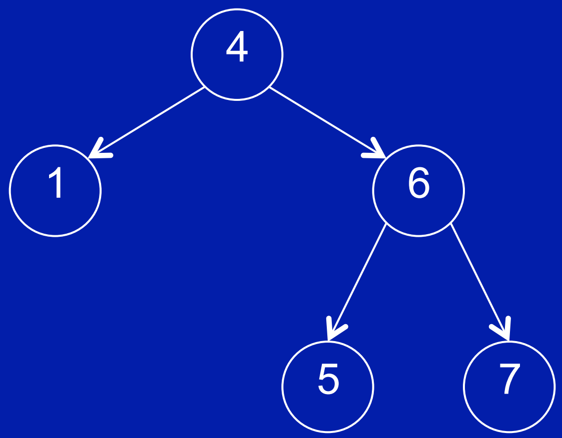


Delete 2.

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

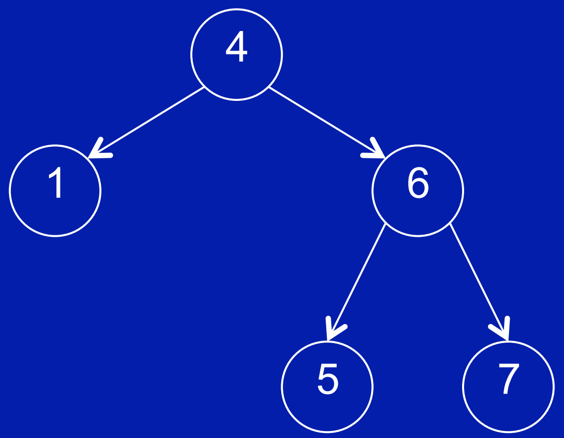
```
use binary search to find
  the target in the BST;
if the target to be deleted
  has two children, then
  find the largest value
  in the left subtree and
  replace the target node
  with this one;
```



Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST;
if the target to be deleted
  has two children, then
  find the largest value
  in the left subtree and
  replace the target node
  with this one;
```

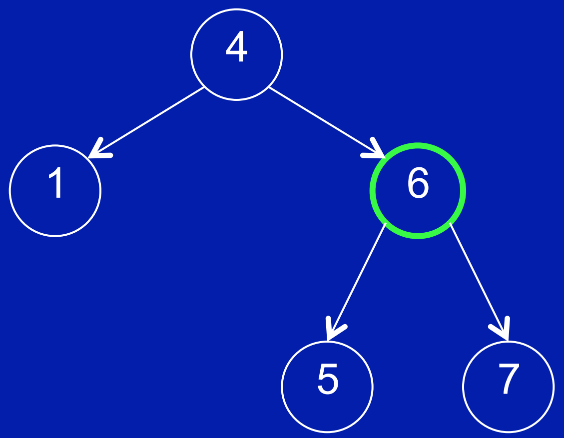


Delete 6.

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST;
if the target to be deleted
  has two children, then
  find the largest value
  in the left subtree and
  replace the target node
  with this one;
```

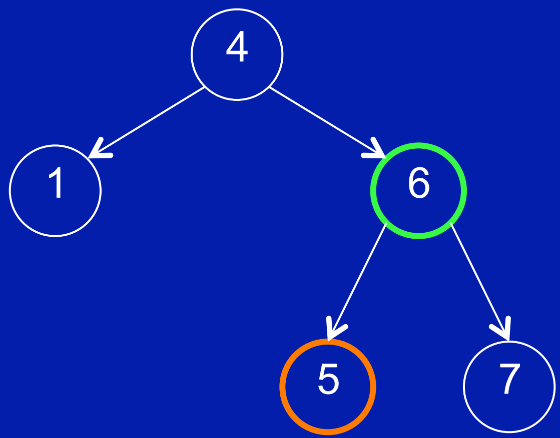


Delete 6.

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST;
if the target to be deleted
  has two children, then
  find the largest value
  in the left subtree and
  replace the target node
  with this one;
```

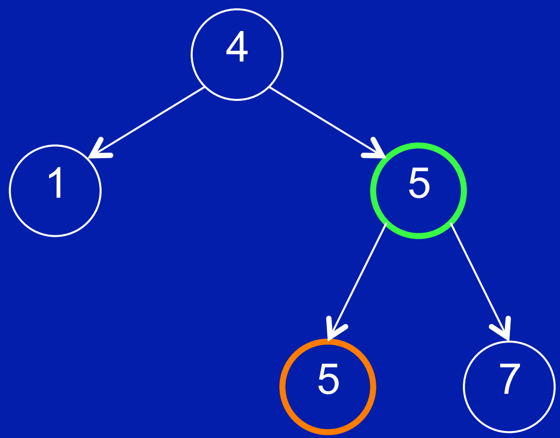


Delete 6.

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST;
if the target to be deleted
  has two children, then
  find the largest value
  in the left subtree and
  replace the target node
  with this one;
```

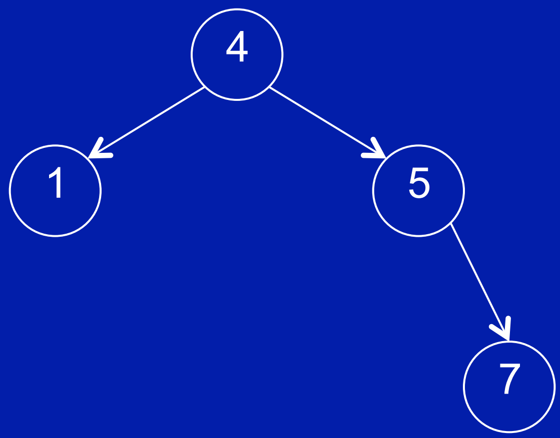


Delete 6.

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST;
if the target to be deleted
  has two children, then
  find the largest value
  in the left subtree and
  replace the target node
  with this one;
```

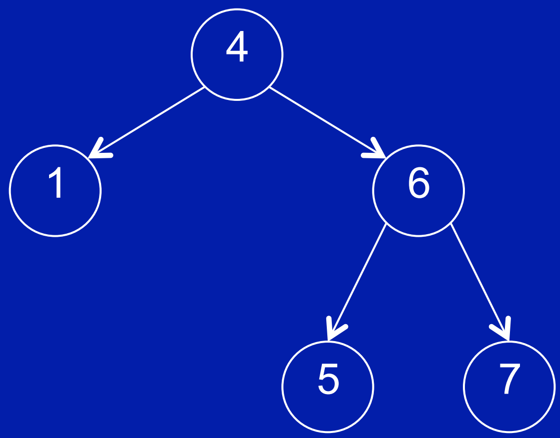


Delete 6.

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST;
if the target to be deleted
  has two children, then
  find the largest value
  in the left subtree and
  replace the target node
  with this one;
```

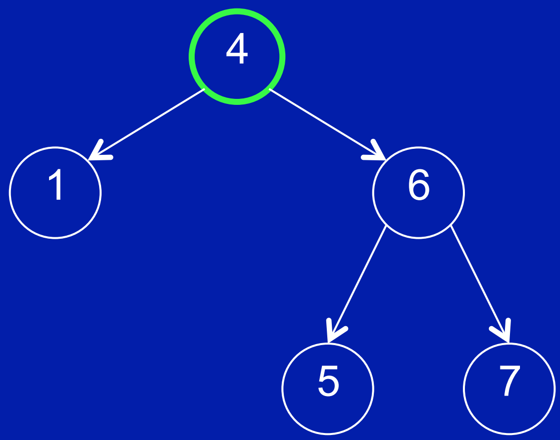


Can we also use the smallest value in the right subtree? Sure.
Delete 4.

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST;
if the target to be deleted
  has two children, then
  find the largest value
  in the left subtree and
  replace the target node
  with this one;
```

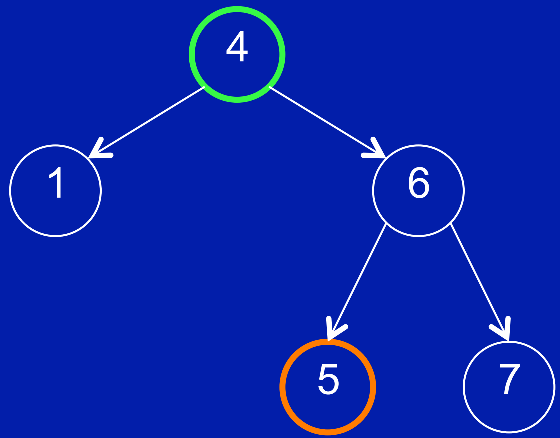


Can we also use the smallest value in the right subtree? Sure.
Delete 4.

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST;
if the target to be deleted
  has two children, then
  find the largest value
  in the left subtree and
  replace the target node
  with this one;
```

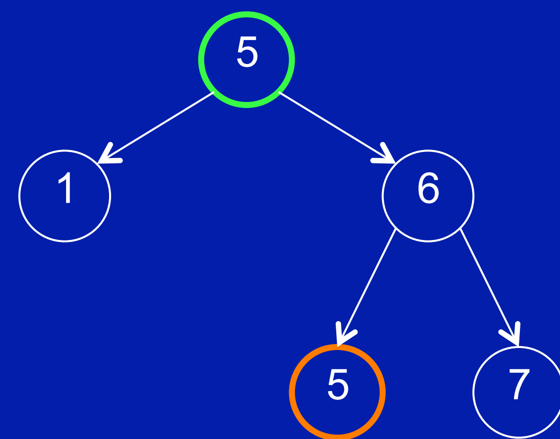


Can we also use the smallest value in the right subtree? Sure.
Delete 4.

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST;
if the target to be deleted
  has two children, then
  find the largest value
  in the left subtree and
  replace the target node
  with this one;
```

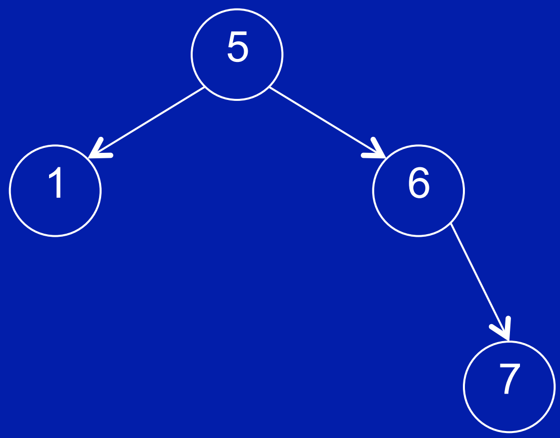


Can we also use the smallest value in the right subtree? Sure.
Delete 4.

Binary search trees

delete(target) is more complicated. Let's break it down into three different cases...

```
use binary search to find
  the target in the BST;
if the target to be deleted
  has two children, then
  find the largest value
  in the left subtree and
  replace the target node
  with this one;
```



Can we also use the smallest value in the right subtree? Sure.
Delete 4.
What if 5 had children? Then the effects would ripple down from there. More on that next time.