Trees

Rob Hackman

Fall 2025

University of Alberta

Table of Contents

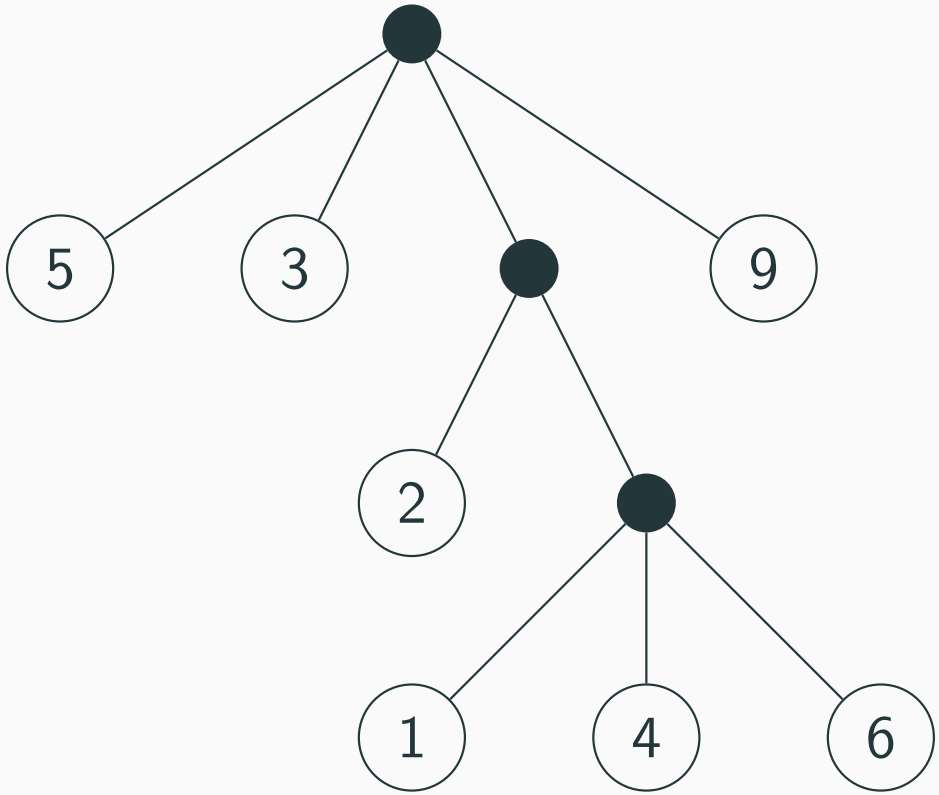
Reconsidering `SuperLLists`

In our previous example we wrote a function to sum all the numbers in a `SuperLList` of numbers which led to a very large trace, since instead of only recursively calling ourselves once for each item in our `LList` had to recursively call ourselves for each non-integer item in *each* `LList`.

It could be useful to depict such a data type as a diagram, where each `LList` is represented simply as a dot with lines to the items within it, and each `int` is represented simply as an integer.

Drawing a `SuperLList`

In this way we may draw the `SuperLList`
`(5, 3, (2, (1, 4, (6))), 9)` as shown below.



Trees

Drawing out `SuperLList` in such a manner makes it much easier to identify what each item of each `LList` is. It also shows the recursive nature of our data type clearly.

Trees

Drawing out `SuperLList` in such a manner makes it much easier to identify what each item of each `LList` is. It also shows the recursive nature of our data type clearly.

In fact what is being shown, and what can be represented with multidimensional `LLists` is a common data structure in Computing called a *tree*.

Trees

Drawing out `SuperLList` in such a manner makes it much easier to identify what each item of each `LList` is. It also shows the recursive nature of our data type clearly.

In fact what is being shown, and what can be represented with multidimensional `LLists` is a common data structure in Computing called a *tree*.

The data structure is called a tree because of the way items contained within each other are shown to branch out from one another, much like a trees branches and leaves grow.

The `IntTree` type

We will now define a data type we will call an `IntTree` to aid us in learning the terminology about trees. Our definition for `IntTree` is as follows:

```
An IntTree is one of:
 - ()
 - tuple(int, tuple of IntTree)
```

In this way each `IntTree` is either the empty tuple, or a tuple of two elements — an integer and a (possibly empty) `tuple` of `IntTrees`.

Example `IntTrees`

Each pair that makes a non-empty `IntTree` is known as a *node*. The `int` is the value stored at that node and the `tuple` of `IntTrees` in the pair are the nodes that are contained within that `node`, which are known as the *children* of that node.

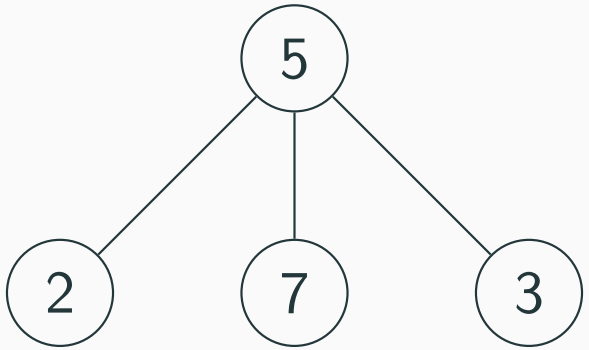
For example the node `(5, ())` represents a node `5` with no children at all. A node with no children is known as a *leaf* node.

From our data definition *any* such pair represents an `IntTree`. So the picture that represents this tree is shown below.

Example `IntTrees`

So any pair of an `int` and the empty `tuple` represents a leaf node. A non-leaf node is a node that has at least one child, such a node is called an *internal node*.

So the node `(5, ((2, ()), (7, ()), (3, ())))` is an internal node with the value 5 and three children, each of those children are leaf nodes with the values 2, 7, and 3 respectively. A visual representation of this tree is shown below.

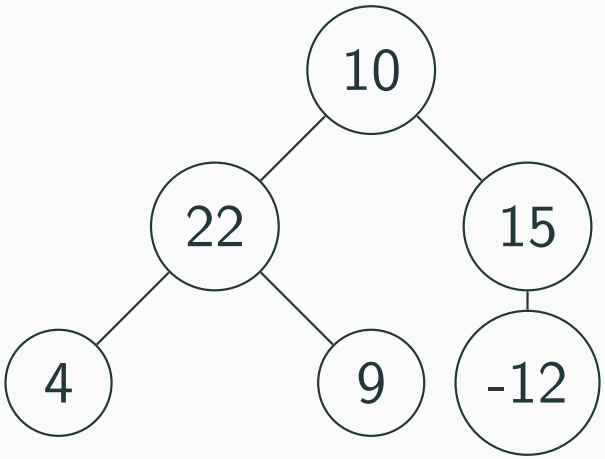


Example `IntTrees`

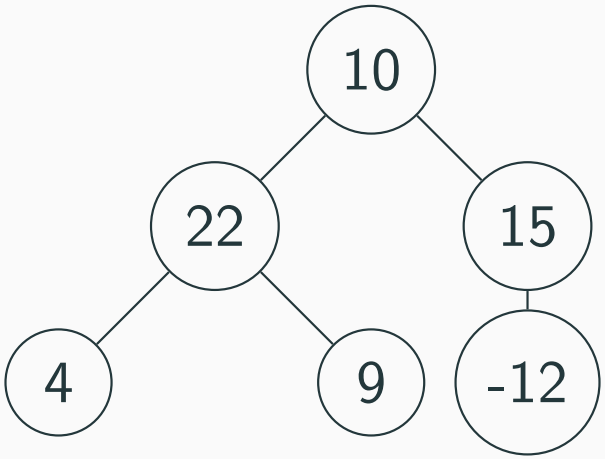
A nodes children do not only have to be leaf nodes, however. Children nodes can also have children.

For example we can define the following `IntTree` per our data definition, and depict it pictorially as shown below.

```
(10, ((22, ((4, ()), (9, ()))),
      (15, ((-12, ()),))))
```

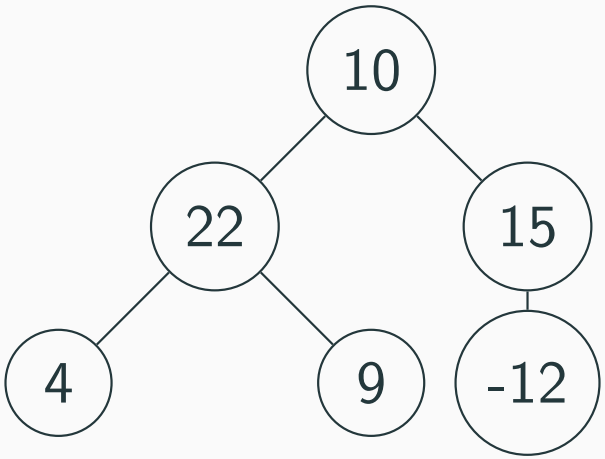


Example `IntTrees`



Note how the node that has value 10 had two children, each of which are also internal nodes. As nodes that are contained within a node are said to be the children of the node containing them, the node that contains children is said to be the *parent* of the nodes it directly contains.

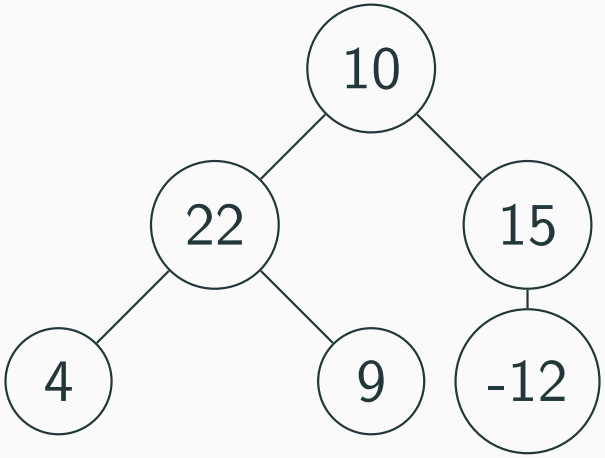
Example `IntTrees`



For ease, we will refer to nodes directly by the value they store. In the diagram above we would say the node 10 is the parent of the nodes 22 and 15. Conversely, the nodes 22 and 15 and the children of the node 10.

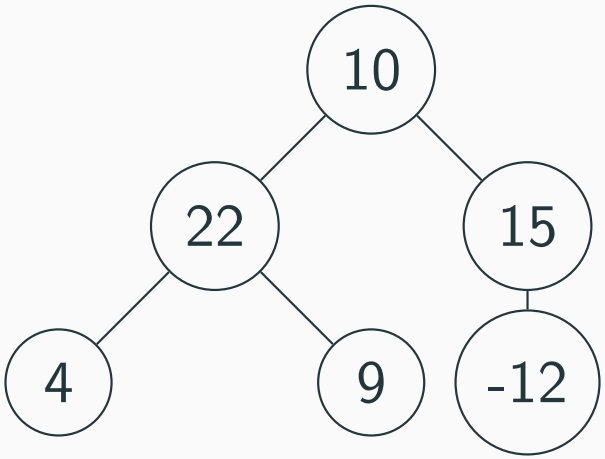
Similarly, the node 22 is the parent of the nodes 4 and 9, and the nodes 4 and 9 are the children of node 22.

Example `IntTrees`



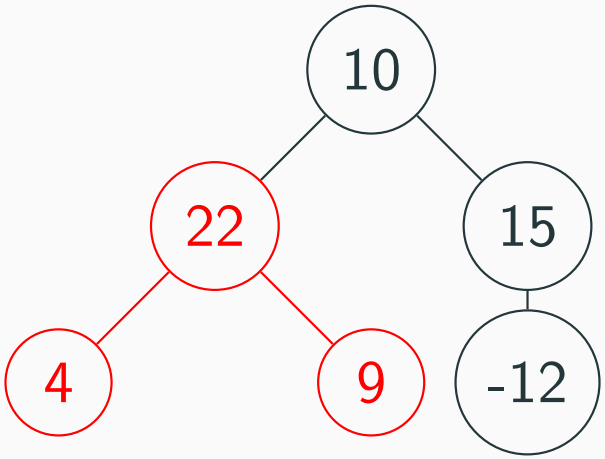
Since our tree is represented by a single node, the node which represents our tree is called the *root*. In the diagram above our root is the node 10.

Example `IntTrees`



Since by design our recursive data definition means that *any* node can represent an `IntTree` that means that each node of our tree can be thought of as representing its own tree. As such, we can speak about the *subtree* rooted at a particular node, which simply means the tree that would be represented by a particular node.

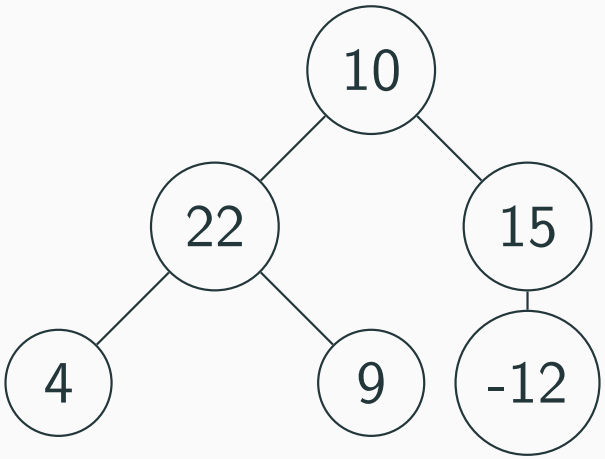
Example `IntTrees`



For example, we could talk about the subtree rooted at the node 22, which is simply the tree `(22, ((4, ()), (9, ())))`.

In the diagram the subtree rooted at node 22 has been coloured red, to demonstrate what tree this subtree represents.

Example `IntTrees`



For any node ω that is contained within the subtree rooted at node N (other than N itself), it is said that ω is a *descendant* of node N. Similarly, it is said that node N is an *ancestor* of node ω.

More familiarly, the descendants of a node are its children, or any descendants of its children. Similarly, the ancestors of a node are its parents, or any ancestors of its parents. These relationships are analogous to those in human families.

Terminology review

Several terms were defined over the last several slides. We present them again here for clarity.

- Leaf node — a node with no children
- Internal node — a node with at least one child
- Root node — the node that represents the given tree, from which all nodes in the tree are descended

Terminology review

Several terms were defined over the last several slides. We present them again here for clarity.

- Child node — for any node B that is contained within a node A it is said that B is the child of A
- Parent node — for any node B that is contained within a node A it is said that A is the parent of B

Terminology review

Several terms were defined over the last several slides. We present them again here for clarity.

- Subtree — a subtree is simply the tree represented by some node of a tree that is not the root of that tree
- Ancestor — for any node B that is contained within the subtree rooted at node A and $B \neq A$ then it is said that node A is an ancestor of node B.
- Descendant — for any node B that is contained within the subtree rooted at node A and $B \neq A$ then it is said that node B is a descendant of node A.

Working on trees

Since trees are an implicitly recursive data solving problems that operate on trees comes naturally to us given our experience in working recursively. As with all of our work recursively we use our data definition to guide the structure of our code.

Working on trees

Since trees are an implicitly recursive data solving problems that operate on trees comes naturally to us given our experience in working recursively. As with all of our work recursively we use our data definition to guide the structure of our code.

One use case of the types of trees we've seen is in representing the taxonomy of animals. In biological taxonomy there are different ranks, and each rank contains several of the lower ranks. The taxonomical rankings are as follows:

Working on trees

Since trees are an implicitly recursive data solving problems that operate on trees comes naturally to us given our experience in working recursively. As with all of our work recursively we use our data definition to guide the structure of our code.

One use case of the types of trees we've seen is in representing the taxonomy of animals. In biological taxonomy there are different ranks, and each rank contains several of the lower ranks. The taxonomical rankings are as follows:

- Kingdom is the highest rank we will consider
- Phylum is the second-highest rank we will consider
- Class is the third-highest rank we will consider
- Family is the fourth-highest rank we will consider
- Genus is the fifth-highest rank we will consider
- Species is the lowest rank we will consider

Biological taxonomy

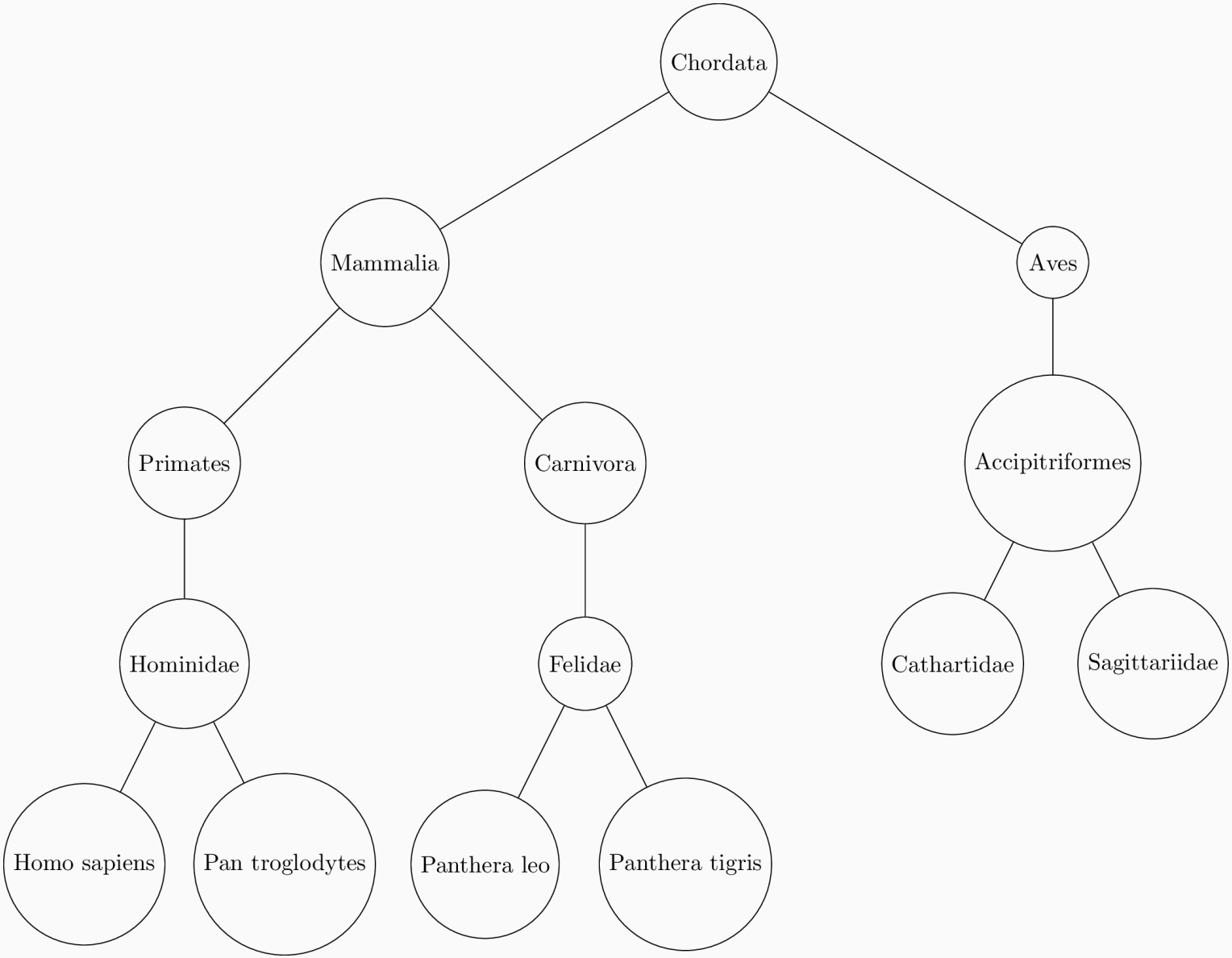
In biological taxonomy each item of each rank is classified as being a member of a higher rank.

For example the class *Mammalia* (mammals) is a member of the phylum *Chordata* which itself is a member of the kingdom *Animalia*. Each item at each rank contains several items of the lower ranks. For example, the class *Aves* (bird) is also a member of the phylum *Chordata*.

As such, we can represent biological taxonomical classifications as a tree.

Taxonomical tree

Below is an example taxonomical tree.



Defining `TaxoTree`

The tree shown on the previous slide was rooted at the phylum Chordata, and showed several taxonomical groupings that are members of phylum Chordata. We will now define a data type for storing these trees called the `TaxoTree`

```
A TaxoTree is one of:
 - ()
 - (str, tuple of TaxoTree)
```

Now, with this definition we can write a function that operates on a `TaxoTree`.

Taxonomical path

We now will write a function `taxoPath` that takes two parameters:

- A `TaxoTree`, which is the tree to operate on
- A `str` which is the classification to look for in the tree

The function `taxoPath` then searches for the given string in the given `TaxoTree` and returns a `LList` of the values that reside on the path to the value. Which is to say, the string represents a taxonomical group, and you are returning a `LList` of all taxonomical groups that group belongs to that are shown in the tree.

Taxonomical path example

For example, in the taxonomical tree that was pictured earlier in the slides, if we called `taxoPath` on that tree and the string `"Hominidae"` it would return the `LList` `("Chordata", "Mammalia", "Primates", "Hominidae")`.

If we called `taxoPath` on that same tree and the string `"Aves"` it would return the `LList` `("Chordata", "Aves")`.

If `taxoPath` is called on a `TaxoTree` and a string which is not contained within that `TaxoTree` then `taxoPath` would return the empty `LList`.

Writing `taxoPath`

We begin by referring to our data definition to aid us in writing our function.

```
A TaxoTree is one of:
 - ()
 - (str, tuple of TaxoTree)
```

As with any recursive data type we have a base case and at least one recursive case. So we must write our code to consider this.

```
def taxoPath(tree, s):
```

Writing `taxoPath`

Starting with our base case we note that if a `TaxoTree` is the empty tree that contains no nodes then of course it does not contain the string `s` so we simply return the empty `LList`.

```
def taxoPath(tree, s):
  if tree == ():
    return empty()
```

Writing `taxoPath`

If the tree we are given is not empty then we must check if this node is the node that contains the value we are looking for, if it is then our job is complete.

```
def taxoPath(tree, s):
  if tree == ():
    return empty()
  if tree[0] == s:
    return LL(s)
```

Writing `taxoPath`

If the current node is not the node that stores the value we are looking for then there are two possibilities:

- None of the subtrees rooted at any of our children contain the value we are looking for
- One of our subtrees contains the value we are looking for

Due to the nature of the taxonomical groupings the value we're looking for can be in at most one of our subtrees.

```
def taxoPath(tree, s):
  if tree == ():
    return empty()
  if tree[0] == s:
    return LL(s)
```

Writing `taxoPath`

As such, we must search the subtrees rooted at each of the current node's children to determine if any one of them contains value we are looking for.

So, we have an entire `tuple` of `TaxoTrees` and we want to know for each of them the taxonomical path to a given classification...

Luckily we have a function to do just that!

```
def taxoPath(tree, s):
  if tree == ():
    return empty()
  if tree[0] == s:
    return LL(s)
```

Writing `taxoPath`

So we simply want to map the function that searches a `TaxoTree` for the specific string `s` onto our `tuple` of `TaxoTrees`.

So we create closure for a unary function to map, which is a function that takes a tree and simply returns the result of calling `taxoPath` on that tree and the captured string s.

```
def taxoPath(tree, s):
  if tree == ():
    return empty()
  if tree[0] == s:
    return LL(s)
  pathToS = lambda tr: taxoPath(tr, s)
  childPaths = map(pathToS, tree[1])
```

Writing `taxoPath`

Our map produces for us a `LList` of `LLists` which are the results of applying `taxoPath` to each our child trees.

Since only one child may contain the value we're looking for that means at most one of these `LLists` will be non-empty.

So we need only to determine if we have a non-empty `LList` or not!

```
def taxoPath(tree, s):
  if tree == ():
    return empty()
  if tree[0] == s:
    return LL(s)
  pathToS = lambda tr: taxoPath(tr, s)
  childPaths = map(pathToS, tree[1])
```

Writing `taxoPath`

So we simply filter out all the empty `LLists` from the recursive results of our children. If there are no non-empty results then there is no path to the given value from this node and we should return the empty `LList`.

```
def taxoPath(tree, s):
  if tree == ():
    return empty()
  if tree[0] == s:
    return LL(s)
  pathToS = lambda tr: taxoPath(tr, s)
  childPaths = map(pathToS, tree[1])
  nonEmpties = filter(lambda l: not isEmpty(l), childPaths)
  if isEmpty(nonEmpties):
    return empty()
```

Writing `taxoPath`

If there *is* a non-empty result, then that result is the `LList` of classifications we want, in addition though we must add *this* node's classification to the result, since we started at this node.

```
def taxoPath(tree, s):
  if tree == ():
    return empty()
  if tree[0] == s:
    return LL(s)
  pathToS = lambda tr: taxoPath(tr, s)
  childPaths = map(pathToS, tree[1])
  nonEmpties = filter(lambda l: not isEmpty(l), childPaths)
  if isEmpty(nonEmpties):
    return empty()
  return cons(tree[0], first(nonEmpties))
```

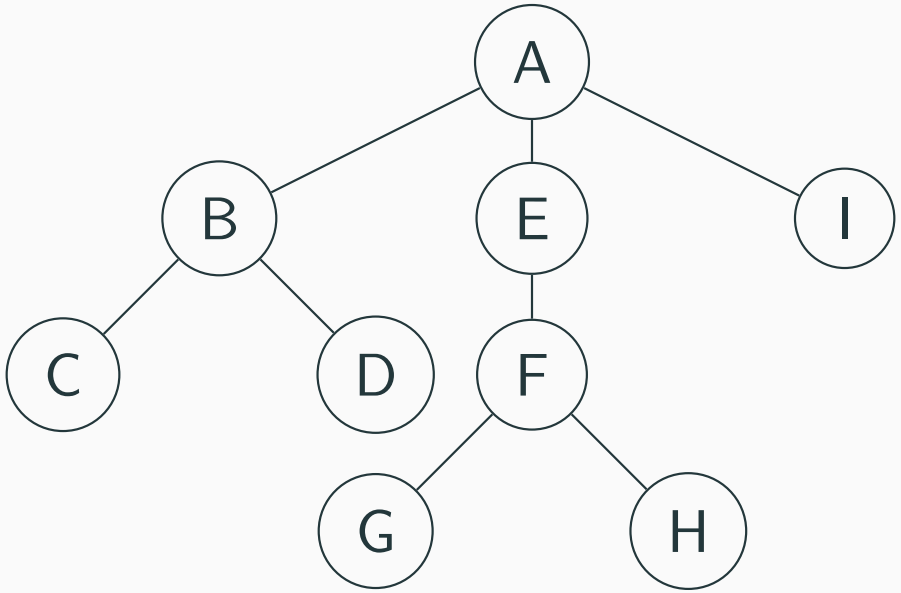
Trees — breaking down problems

Due to the recursive nature of trees the solutions to problems involving trees naturally tend to be recursive.

We can easily solve many problems on trees by remembering that the function we are writing operates on a tree, and each of our children represents a tree — the subtree rooted at that child. So we can recursively call our function on children nodes to get a recursive result that can help us build our answer.

This technique is how we solved `taxoPath`.

Knowledge Check



Knowledge Check: What node is the parent of node G? Which nodes are the ancestors of node H? Which nodes are the children of node A? Draw the subtree rooted at node E. List all the leaf nodes. List all the internal nodes.

Tree representation

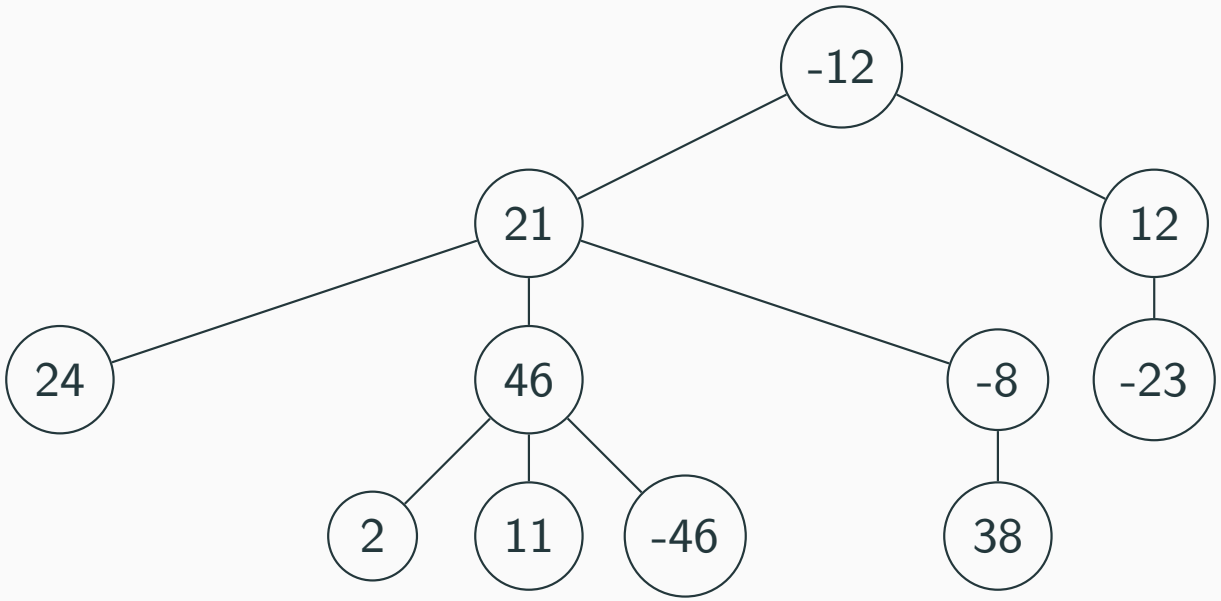
Practice Question: Recall the data definition we made for `IntTrees`

```
An IntTree is one of:
 - ()
 - tuple(int, tuple of IntTree)
```

Given that data definition write the literal that represents the tree depicted on the next slide:

Tree representation

```
An IntTree is one of:
 - ()
 - tuple(int, tuple of IntTree)
```



From data to diagram

Practice Question: Consider the following tuple in Python that defines an `IntTree`

```
t = (-25, ((49, ((-26, ()), (25, ()), (-45, ()), (50, ()))),
    (34, ()), (6, ((-4, ()), (-23, ()))))
```

Draw the tree that is represented by this `IntTree`.

Finding leaf nodes

Practice Question: Recall that a leaf node is a node that has no children.

Write a function `leafValues` that takes a single `IntTree` parameter and returns a LList of integers of all the values stored in leaf nodes in the tree. For this question you do not need to worry about the order of the values in your LList, just that all items are included.

Finding leaf nodes

```
def leafValues(t):
    '''
    Returns a LList of all the leaf values in the IntTree t

    t - IntTree
    returns - LList of int

    Example:
      t = leafNodes((-22, ((-41, ((-10, ()),
                    (-1, ()), (-43, ()), (22, ()))),
                    (-41, ((-14, ()),)))))
      leafNodes(t) -> (-10, -1, -43, 22, -14)
    '''
```

A few more definitions

Before moving on to the next topic we need a bit more terminology.

- The *height* of a tree is the length of the *longest* path from the root to a leaf node, as such the height of a tree with a single node is 0. We define the height of the empty tree as -1.
- The *depth* of a node is the length of the path to that node from the root of the tree, as such the depth of the root node is 0.

Table of Contents

General trees

So far the trees we have been discussing are all examples of trees that are called *general* trees.

A general tree is a tree in which each node can have *any* number of children.

Our trees were all general trees since they were always a value and an arbitrary length tuple of child trees. Since our children tuple could have any number of items in it, each node could have any number of children.

n-ary trees

In contrast to general trees are *N-ary trees*. For some given natural number N an N-ary tree is a tree where every node has *no more* than N children.

When defining a data type that represents an N-ary tree the data definition would enforce that each node has no more than N children.

Example N-ary tree data definitions

Below are some example data definitions for N-ary trees when N is two, three, and four.

```
A BinaryTree of X is one of:
  - ()
  - (X, BinaryTree, BinaryTree)

A TernaryTree of X is one of:
  - ()
  - (X, TernaryTree, TernaryTree, TernaryTree)

A QuaternaryTree of X is one of:
  - ()
  - (X, QuaternaryTree, QuaternaryTree,
       QuaternaryTree, QuaternaryTree)
```

Examining the `BinaryTree`

Considering our `BinaryTree` definition we see how the maximum number of children is enforced.

```
A BinaryTree of X is one of:
  - ()
  - (X, BinaryTree, BinaryTree)
```

Examining the `BinaryTree`

Considering our `BinaryTree` definition we see how the maximum number of children is enforced.

```
A BinaryTree of X is one of:
  - ()
  - (X, BinaryTree, BinaryTree)
```

Either a `BinaryTree` is the empty tree with no nodes or it is a node. Each node is represented as a triplet of values, the first being the value stored at this node and the second and third being the two children of the node.

Examining the `BinaryTree`

Considering our `BinaryTree` definition we see how the maximum number of children is enforced.

```
A BinaryTree of X is one of:
  - ()
  - (X, BinaryTree, BinaryTree)
```

There are four options for the two `BinaryTree` children:

- Both children are the empty tree, which indicates this is a leaf node with no children
- The first child is non-empty and the second is empty, so this is a node with one child
- The first child is empty and the second is non-empty, so this is a node with one child
- Both children are non-empty, so this is a node with two children

Examining the `BinaryTree`

Considering our `BinaryTree` definition we see how the maximum number of children is enforced.

```
A BinaryTree of X is one of:
  - ()
  - (X, BinaryTree, BinaryTree)
```

Given these are the only four possibilities for our two children, it is clear that any node that meets our data definition cannot have more than two children.

Binary tree terminology

Binary trees are a data type often used in computing science, and as such we have terminology specifically for referring to them.

Binary tree terminology

Binary trees are a data type often used in computing science, and as such we have terminology specifically for referring to them.

- The first child of a node in a binary tree is called its *left* child, and the second child node of a binary tree is called its *right* child.

Binary tree terminology

Binary trees are a data type often used in computing science, and as such we have terminology specifically for referring to them.

- The first child of a node in a binary tree is called its *left* child, and the second child node of a binary tree is called its *right* child.
- The subtree rooted at a node's left child is called its left subtree, the subtree rooted at a node's right child is called its right subtree

Binary tree terminology

Binary trees are a data type often used in computing science, and as such we have terminology specifically for referring to them.

- The first child of a node in a binary tree is called its *left* child, and the second child node of a binary tree is called its *right* child.
- The subtree rooted at a node's left child is called its left subtree, the subtree rooted at a node's right child is called its right subtree
- A binary tree is said to be *full* if every node has exactly zero or two children

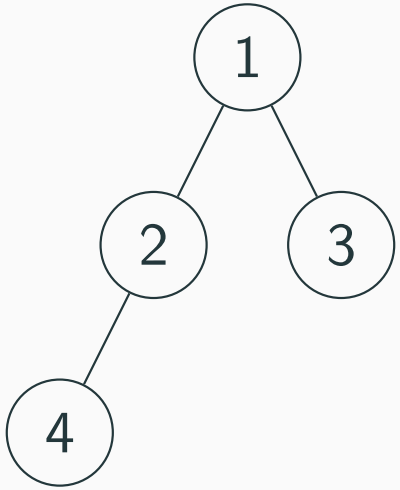
Binary tree terminology

Binary trees are a data type often used in computing science, and as such we have terminology specifically for referring to them.

- The first child of a node in a binary tree is called its *left* child, and the second child node of a binary tree is called its *right* child.
- The subtree rooted at a node's left child is called its left subtree, the subtree rooted at a node's right child is called its right subtree
- A binary tree is said to be *full* if every node has exactly zero or two children
- A binary tree is said to be *balanced* if the magnitude of the difference between the heights of the left and right subtrees of *every* node is no more than one.

Binary tree examples

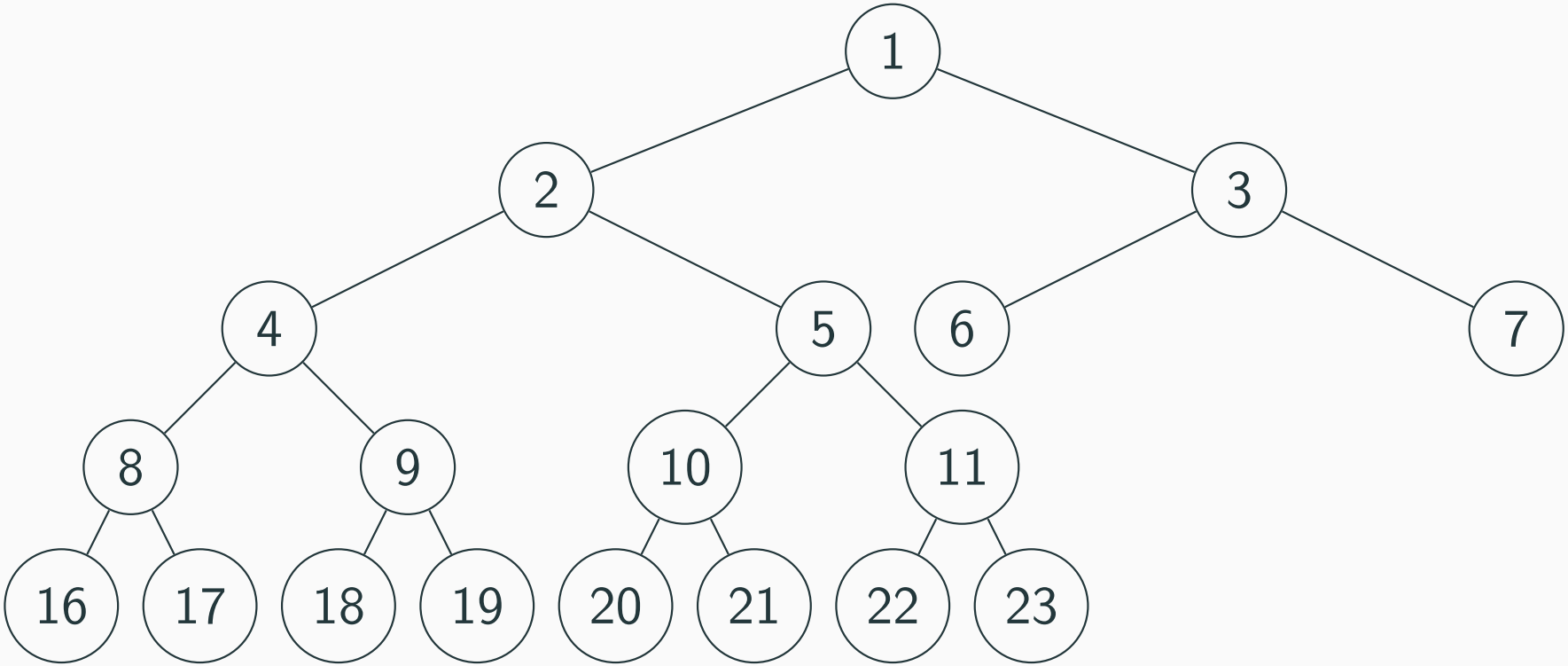
The tree below is balanced but is not full.



- For node 1
  - The left subtree has height 1
  - The right subtree has height 0
- For node 2
  - The left subtree has height 0
  - The right subtree has height -1
- All leaf nodes have empty left and right children, so for any leaf node its left and right subtree are always both height -1

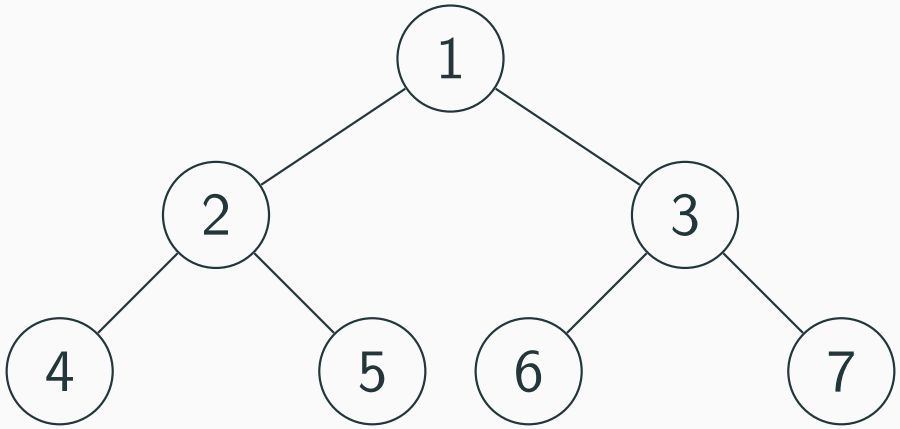
Binary tree examples

The tree below is full but not balanced.



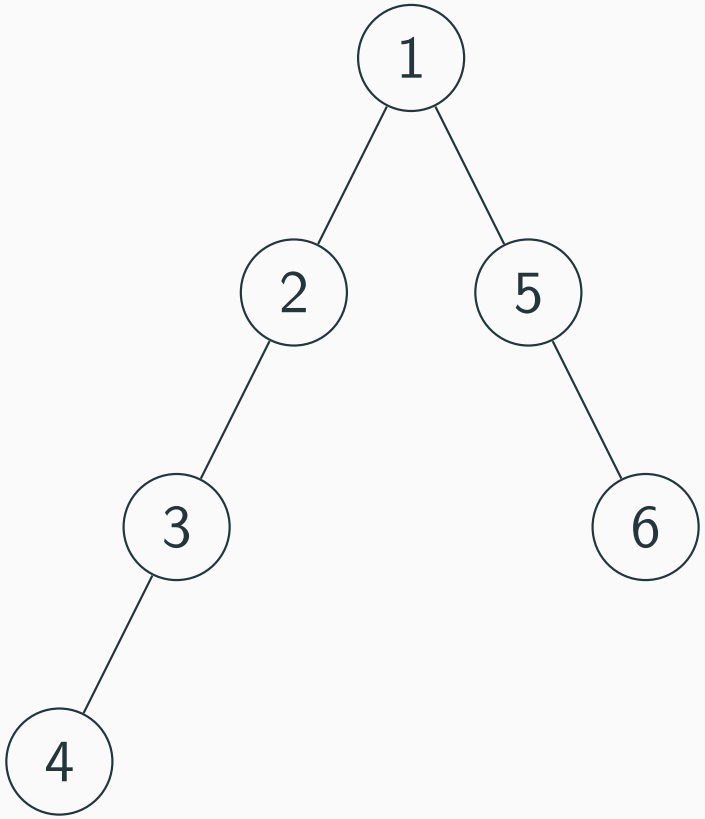
Binary tree examples

The tree below is both balanced and full.



Knowledge Check

Knowledge Check: Is the following binary tree balanced? Is it full?



A note on N-ary trees

N-ary trees are a more specific form of general trees, and binary trees are just one particular N-ary tree.

In this way, N-ary trees are actually simpler than the general trees we worked with first.

Even though binary trees are simpler they have many common applications in computer science, so studying them deeply in addition to general trees is worthwhile.

Binary tree path finding

When we wrote the function `taxoPath` we were finding the *path* through a general tree to a specific value. We now write the function `findPath` that takes a `BinaryTree` and a value and returns a `LList` that represents the path to the given value in the given `BinaryTree` if one exists, or the empty `LList` if one does not.

We will define our path to a value in a `BinaryTree` as a `LList` that contains only the strings `"left"` and `"right"`, denoting starting from the root the series of subtrees to follow until the value is found.

Writing `findPath`

Writing the `findPath` function will be very similar to our `taxoPath` function.

Again, we start with the base case from our data definition, which is an empty `BinaryTree`.

```
def findPath(bt, val):
  if bt == ():
    return ()
```

Writing `findPath`

When our `BinaryTree` is not empty, just as in the case of `taxoPath`, we must see if the value we are looking for is stored at this node.

If the value we are looking for is stored at this node then the path to it is *also* empty, since we do not need to move left or right — we're already there!

```
def findPath(bt, val):
  if bt == ():
    return empty()
  if bt[0] == val:
    return empty()
```

Writing `findPath`

Now, we must check if there exists a path to our value in either our left or right subtree.

We'll start by checking our left subtree first.

```
def findPath(bt, val):
  if bt == ():
    return empty()
  if bt[0] == val:
    return empty()
  leftPath = findPath(val[1])
  if leftPath == ???:
```

Writing `findPath`

Hold on! Given the result from `findPath` on our left subtree how can we check if a path to the given value was found or not?

In `taxoPath` we looked for a recursive result that was non-empty, since the result when we found the value was not empty. However, in this case the correct path to produce when the tree's root is the value we're looking for itself would be the empty `LList`.

```
def findPath(bt, val):
  if bt == ():
    return empty()
  if bt[0] == val:
    return empty()
  leftPath = findPath(val[1])
  if leftPath == ???:
```

Writing `findPath`

How can we distinguish if the return value from our function `findPath` was empty because the value was not found, or empty because the value in our left child was the value we're looking for?

```
def findPath(bt, val):
  if bt == ():
    return empty()
  if bt[0] == val:
    return empty()
  leftPath = findPath(val[1])
  if leftPath == ???:
```

Writing `findPath`

Well, we can simply check if our left child contains the value we're looking for. If it does then the returned empty `LList` is indicating the path, if it does not directly contain the value we're looking for then it would have had to return a non-empty path if it found the value beneath it.

```
def findPath(bt, val):
  if bt == ():
    return empty()
  if bt[0] == val:
    return empty()
  leftPath = findPath(val[1])
  if isEmpty(leftPath) and leftPath[0] == val:
    return cons("left", leftPath)
```

Writing `findPath`

However, this code does not work when the left child is the empty tree. Because the empty tree is represented by the empty tuple this code would try to index the empty tuple by zero and would crash.

So, we must first check if our left tree is empty.

```
def findPath(bt, val):
  if bt == ():
    return empty()
  if bt[0] == val:
    return empty()
  leftPath = findPath(bt[1])
  if isEmpty(leftPath) and bt[1][0] == val:
    return cons("left", leftPath)
```

Writing `findPath`

So we add a check to make sure our left child is non-empty before indexing it.

```
def findPath(bt, val):
  if bt == ():
    return empty()
  if bt[0] == val:
    return empty()
  leftPath = findPath(bt[1])
  if isEmpty(leftPath) and bt[1] != () and bt[1][0] == val:
    return cons("left", leftPath)
```

Writing `findPath`

However, the condition `bt[1] != () and bt[1][0] == val` tells us our left child directly contains the value we're looking for, so we don't even need to compare the `leftPath` to know the answer!

```
def findPath(bt, val):
  if bt == ():
    return empty()
  if bt[0] == val:
    return empty()
  if bt[1] != () and bt[1][0] == val:
    return LL("left")
```

Writing `findPath`

However, our left subtree may still contain the value we're looking for at a node more nested within it than its root node. If we recursively call ourselves now though we are guaranteed that the only reason the empty `LList` would be returned is if the value did not exist, since we know it is *at least* one step away from our left child.

```
def findPath(bt, val):
  if bt == ():
    return empty()
  if bt[0] == val:
    return empty()
  if bt[1] != () and bt[1][0] == val:
    return LL("left")
  leftPath = findPath(bt[1], val)
  if not isEmpty(leftPath):
    return cons("left", leftPath)
```

Writing `findPath`

If we did not find the value in our left subtree, we must repeat the same process for our right subtree!

```
def findPath(bt, val):
  ... # All code from before
  if bt[2] != () and bt[2][0] == val:
    return LL("right")
  rightPath = findPath(bt[2], val)
  if not isEmpty(rightPath):
    return cons("right", rightPath)
```

Writing `findPath`

Lastly, we must consider what to do if our node is not contained within our left or right subtree. If that is the case then the node is not contained within the tree the function has been called on, and we should return the empty `LList`

```
def findPath(bt, val):
  ... # All code from before
  if bt[2] != () and bt[2][0] == val:
    return LL("right")
  rightPath = findPath(bt[2], val)
  if not isEmpty(rightPath):
    return cons("right", rightPath)
  return empty()
```

Writing `findPath`

```
def findPath(bt, val):
  if bt == ():
    return empty()
  if bt[0] == val:
    return empty()
  if bt[1] != () and bt[1][0] == val:
    return LL("left")
  leftPath = findPath(bt[1], val)
  if not isEmpty(leftPath):
    return cons("left", leftPath)
  if bt[2] != () and bt[2][0] == val:
    return LL("right")
  rightPath = findPath(bt[2], val)
  if not isEmpty(rightPath):
    return cons("right", rightPath)
  return empty()
```

A note on `findPath`

The function `findPath` was made needlessly complex by the method in which we had to check if the value was contained within our subtrees, since the return value from our function alone did not convey all the information it needed to.

Soon we will review this problem and see a change that could be made to the specification of this function to greatly simplify its use recursively.

Table of Contents

Traversing a binary tree

Often we need to write functions that do some work on every single value stored within a binary tree. The process of accessing each node of a binary tree is referred to as *traversing* the tree. We can traverse any tree, but specifically when it comes to binary trees we refer to specific orders in which we traverse the tree.

`BinaryTree` **to** `LList`

One such process we may apply to the values of a binary tree is to simply stick all of those values into a sequence.

So we are going to write a function that takes a `BinaryTree of ints` and returns a `LList of ints` that are all the integers in the binary tree.

But what order should the values from the `BinaryTree` appear in our resultant `LList`? There are many options to choose from... we will consider three different.

Three traversals

The order in which we place the elements in our resultant `LList` will be the order in which we traverse the elements of the tree. We will discuss three different orders of traversal:

Three traversals

The order in which we place the elements in our resultant `LList` will be the order in which we traverse the elements of the tree. We will discuss three different orders of traversal:

- Pre-order traversal — A traversal in which first the item at the current node is considered, then the left subtree is recursively traversed, then right subtree is recursively traversed.
    - Named **pre**-order because the current node is processed **prior** to the processing of either subtree.

Three traversals

The order in which we place the elements in our resultant `LList` will be the order in which we traverse the elements of the tree. We will discuss three different orders of traversal:

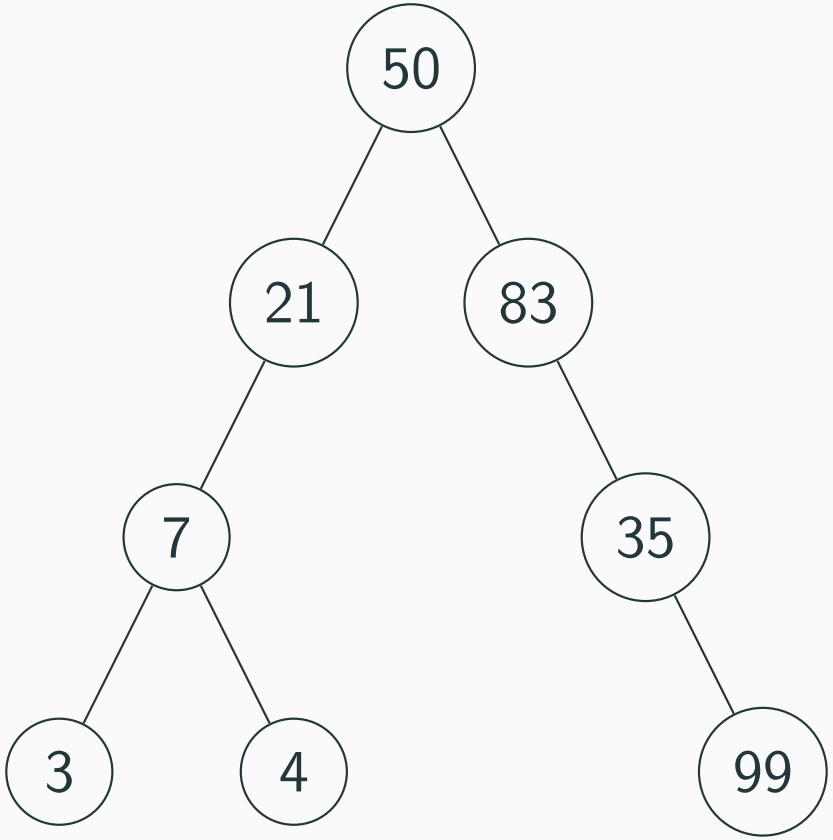
- In-order traversal — A traversal in which the left subtree is recursively traversed, then the current node is processed, then the right subtree is recursively traversed.
    - Named **in**-order because the current node is processed **in-between** the processing of the left and right subtrees.

Three traversals

The order in which we place the elements in our resultant `LList` will be the order in which we traverse the elements of the tree. We will discuss three different orders of traversal:

- Post-order traversal — A traversal in which the left subtree is recursively traversed, then the right subtree is recursively traversed, and then the current node is processed.
    - Named **post**-order because the current node is processed **after** the processing of both subtrees.

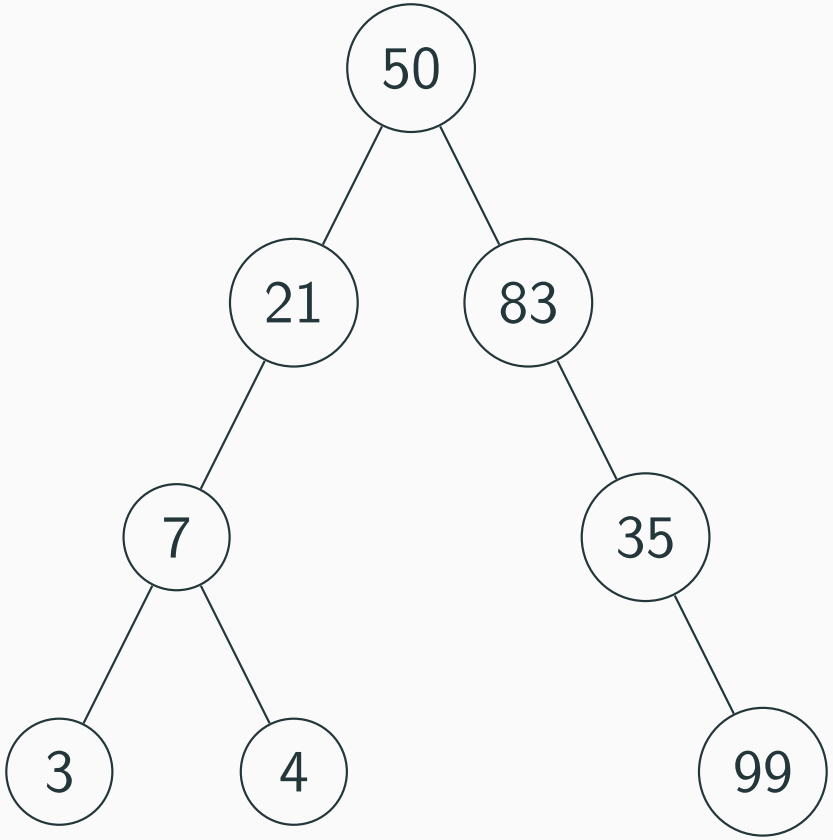
Visualizing traversals



If we produced a `LList` of the values in the tree shown above by doing a pre-order traversal we would get the `LList`:

```
LL(50, 21, 7, 3, 4, 83, 35, 99)
```

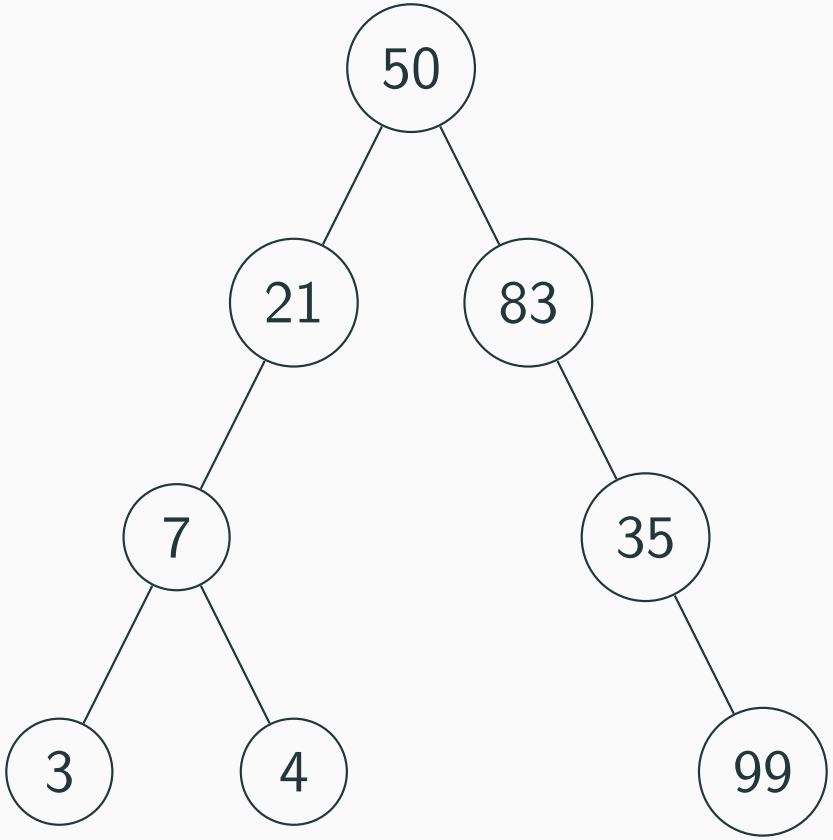
Visualizing traversals



If we produced a `LList` of the values in the tree shown above by doing an in-order traversal we would get the `LList`:

```
LL(3, 7, 4, 21, 50, 83, 35, 99)
```

Visualizing traversals



If we produced a `LList` of the values in the tree shown above by doing a post-order traversal we would get the `LList`:

```
LL(3, 4, 7, 21, 99, 35, 83, 50)
```

Pre-order traversal of a `BinaryTree`

We will start by writing the function `preBTreeToLList` which takes a single `BinaryTree` parameter and returns a `LList` of all the elements of that `BinaryTree` in *pre-order* traversal order.

```
def preBTreeToLList(t):
```

Pre-order traversal of a `BinaryTree`

Again we work backwards from our data definition. Our base case is the empty tree, and the `LList` with all the values from an empty tree in it is the empty `LList`

```
def preBTreeToLList(t):
  if t == ():
    return empty()
```

Pre-order traversal of a `BinaryTree`

Next, if we have a non-empty tree then we have a triplet of three things — the current nodes value, the left subtree and the right subtree.

Conveniently, the definition of pre-order traversal tells us exactly what to do. First process the current node, then process the left subtree, and then process the right subtree.

But in our case what does it mean to "process" a value or a tree?

```
def preBTreeToLList(t):
  if t == ():
    return empty()
```

Pre-order traversal of a `BinaryTree`

Our goal is to build a `LList` of values, so to process something is to turn it into a `LList` of values.

Creating the `LList` of our current value is easy, it's just the `LList` that contains that value.

```
def preBTreeToLList(t):
  if t == ():
    return empty()
  current = LL(t[0])
```

Pre-order traversal of a `BinaryTree`

Since our function returns a `LList` of the values encountered in a pre-order traversal of a tree, we can simply call our function recursively on our left subtree to get a `LList` of all the values in our left subtree in the correct order!

```
def preBTreeToLList(t):
  if t == ():
    return empty()
  current = LL(t[0])
  leftList = preBTreeToLList(t[1])
```

Pre-order traversal of a `BinaryTree`

We must remember that our goal is to produce a *single* `LList` of integers though, so we must combine the answers we got from processing our current node and processing our left subtree.

To combine two `LLists` we simply want to concatenate them.

```
def preBTreeToLList(t):
  if t == ():
    return empty()
  current = LL(t[0])
  leftList = preBTreeToLList(t[1])
  concatenate = lambda l1, l2: foldr(l1, cons, l2)
  resSoFar = concatenate(current, leftList)
```

Pre-order traversal of a `BinaryTree`

Now, all that remains is to process our right subtree, combine, and return the result!

```
def preBTreeToLList(t):
  if t == ():
    return empty()
  current = LL(t[0])
  leftList = preBTreeToLList(t[1])
  concatenate = lambda l1, l2: foldr(l1, cons, l2)
  resSoFar = concatenate(current, leftList)
  rightList = preBTreeToLList(t[2])
  return concatenate(resSoFar, rightList)
```

In-order traversal

To write the function `inBTreeToLList` which returns a `LList` of the values in a `BinaryTree` in the order produced by in-order traversal, we simply need to take the exact same function but change the order in which we process items!

The `inBTreeToLList` function

```
def inBTreeToLList(t):
  if t == ():
    return empty()
  leftList = inBTreeToLList(t[1])
  current = LL(t[0])
  concatenate = lambda l1, l2: foldr(l1, cons, l2)
  resSoFar = concatenate(leftList, current)
  rightList = inBTreeToLList(t[2])
  return concatenate(resSoFar, rightList)
```

Post-order traversal

Practice Question: Write the function `postBTreeToLList` that takes a single `BinaryTree` parameter and returns a `LList` of all the values in that binary tree. The values in the `LList` must be in the order produced by a post-order traversal of the tree.