Data Structures

Augustin Cosse.

Spring 2021

April 2, 2021

Converting lists into Arrays

- The **java.util.Collection** interface includes two methods for generating an array that has the same elements as the collection

toArray()	Returns an array of elements of type Object containing all the elements in this collection
toArray(A)	Returns an array of elements of the same element type as A containing all the elements in this collection

- If the collection is a list, the returned array will have its elements stored in the same order as the list

- In particular, if we have a useful array based method that we want to use on a list or other type of collection, we can do so by using that collection's toArray() method to produce an array representation of the collection.

Converting Arrays into Lists

- Similarly, it is often useful to be able to convert an array into an equivalent list. The **java.util.Arrays** class includes the following methods

asList(A) Returns a list representation of the array A, with the same element type as the elements of A

- The list returned by this method uses the array A as its internal representation for the list. So this list is guaranteed to be an array based list and any changes made to it will automatically reflect in A

Converting Arrays into Lists

- Because of this connection between the array and newly created list, the asList method should always be used with caution
- when used with care, this method can however often save us a lot of time as shown by the random shuffle example below

```
Integer[] arr = {1,2,3,4,5,6,7,8};
List<Integer> listArr = Arrays.asList(arr);
Collections.shuffle(listArr);
```

- It is worth noting that the array A sent to the asList() method should be a reference type (hence our use of the Integer rather than int) type above

Sorting a positional list

- When discussing arrays, we introduced the insertionSort() algorithm (recalled below)

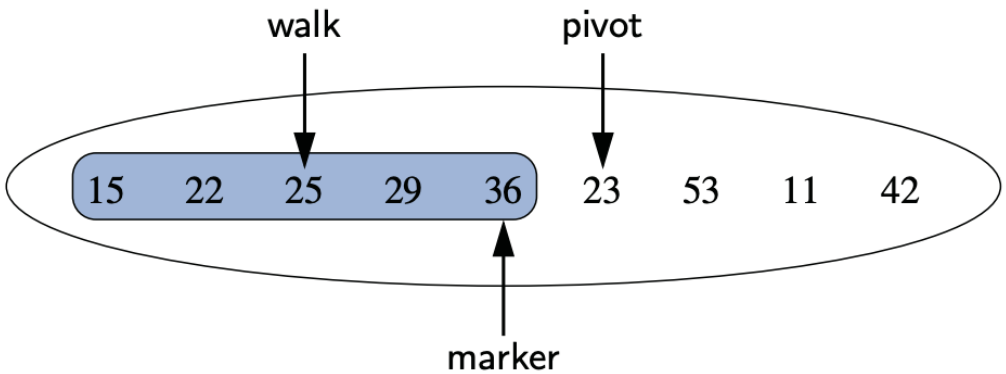
```
public static void insertionSort(char[] data){
  int n = data.length;
  for(int k=1, k<n; k++){
    char cur = data[k];
    int j = k;
    while(j>0 && data[j-1]>cur ){
      data[j] = data[j-1];
      j--;
}
    data[j] = curr;
}}
```

Sorting a positional list

- We can design a similar algorithm on Positional Lists. To do so, we maintain a variable named `marker` that represents the rightmost position of the currently sorted portion of a list

- During each pass, we consider the position just past the marker as the pivot and consider where the pivot element belongs relatove to the sorted portion.

- We also use another variable, named `walk`, to move leftward from the marker,as long as there remains a preceding element with value larger than the pivot's. The connfiguration of thes variables is shown below



Sorting a positional list

- The sorting algorithm can then be implemented as follows

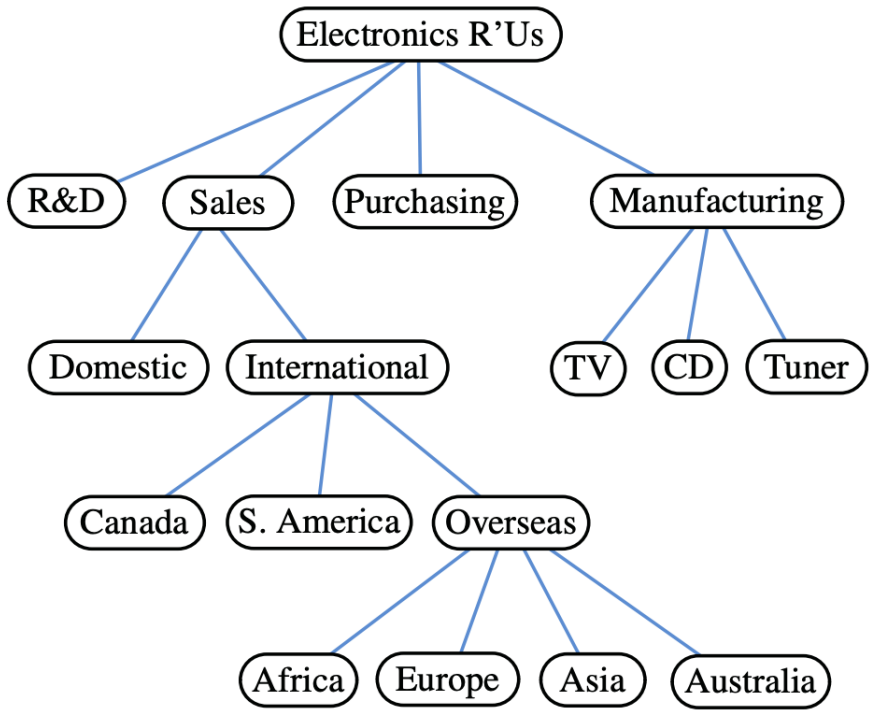
```
public static void insertionSort(PositionalList<Integer> list) {
  Position<Integer> marker = list.first( );
   // last sorted position
  while (marker != list.last( )) {
    Position<Integer> pivot = list.after(marker);
    int value = pivot.getElement( ); // number to be placed
    if (value > marker.getElement( )) // pivot is sorted
      marker = pivot;
    else { // must relocate pivot
      Position<Integer> walk = marker;
     // find leftmost item greater than value
      while (walk != list.first( ) &&
                list.before(walk).getElement( ) > value)
        walk = list.before(walk);
      list.remove(pivot); // remove pivot entry and
      list.addBefore(walk, value); // reinsert value in front
    }}  }
```

Trees

- Trees are on the most important non linear structures in computing
- Trees structures can be considered a breakthrough in data organization in the sense that they allow us to implement a host of algorithms much faster than when using linear data structures, such as arrays or linked lists
- Trees also provide a natural organization for data, and consequently have become ubiquitous structures in file systems, graphical user interfaces, databases, websites and many other computer systems
- When we say that trees are nonlinear we are referring to an organizational relationship that is richer than the simple "before" and "after" between objects in sequences.

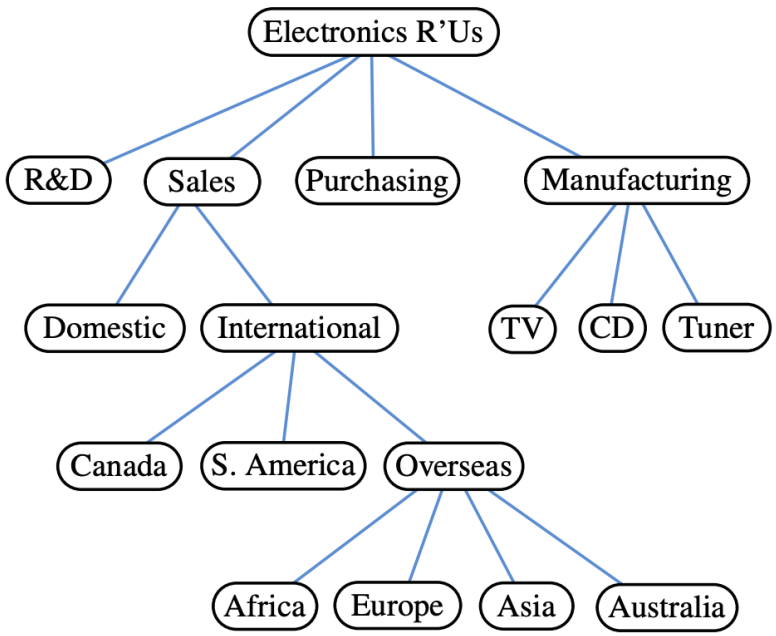
Trees

- The relationships in a tree are hierarchical with some object being above and some below others
- Actually, the main terminology for tree data structures comes from family trees with the term parent, child, ancestor and descendant



Trees

- With the exception of the top element, each element in a tree has a parent element and zero or more children elements
- A tree is usually visualized by drawing the connections between parents and children using straight lines
- We typically call the top element the root of the tree, but it is drawn as the highest element, with the other elements being connected below

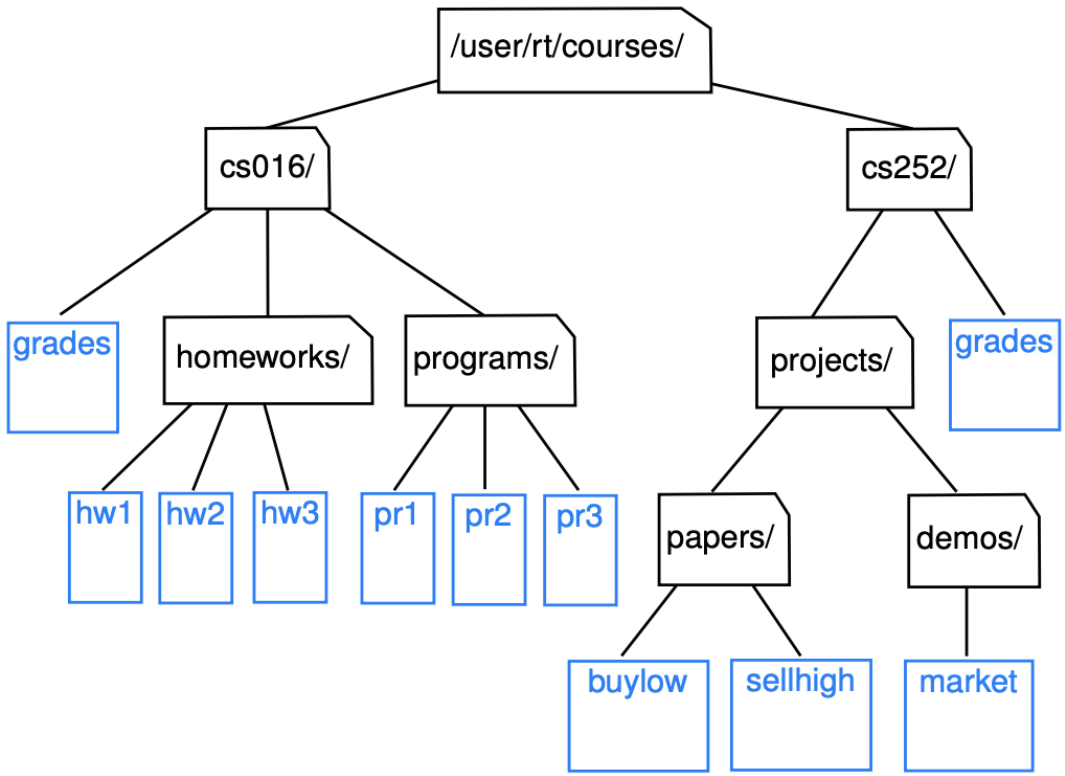


Trees

- Formally we define a tree T as a set of nodes such that the nodes have a parent-child relationship that satisfies the following properties
  - If T is nonempty, it has a special node, called the root of T, that has no parent
  - Each node ν of T different from the root has a unique parent node w; every node with parent w is a child of w
- Note that according to this definition, a tree can be empty, meaning that it does not have any node.
- Finally, this convention allows us to define a tree recursively such that a tree is either empty or consists of a node r called the root of T, and a (possibly empty) set of subtrees whose roots are the parents of the children of r

Trees

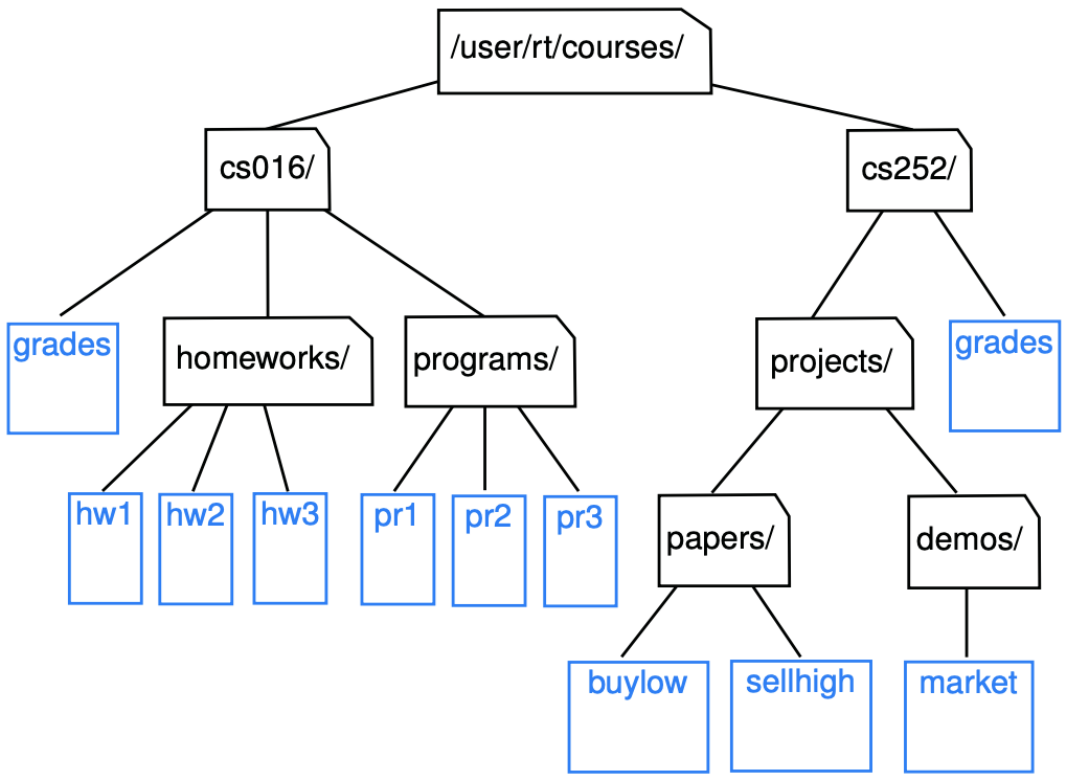
- Two nodes that are the children of the same parent are called siblings. A node ν is external if ν has no children.
- A node ν is internal if it has one of more children
- External nodes are also known as leaves



Trees

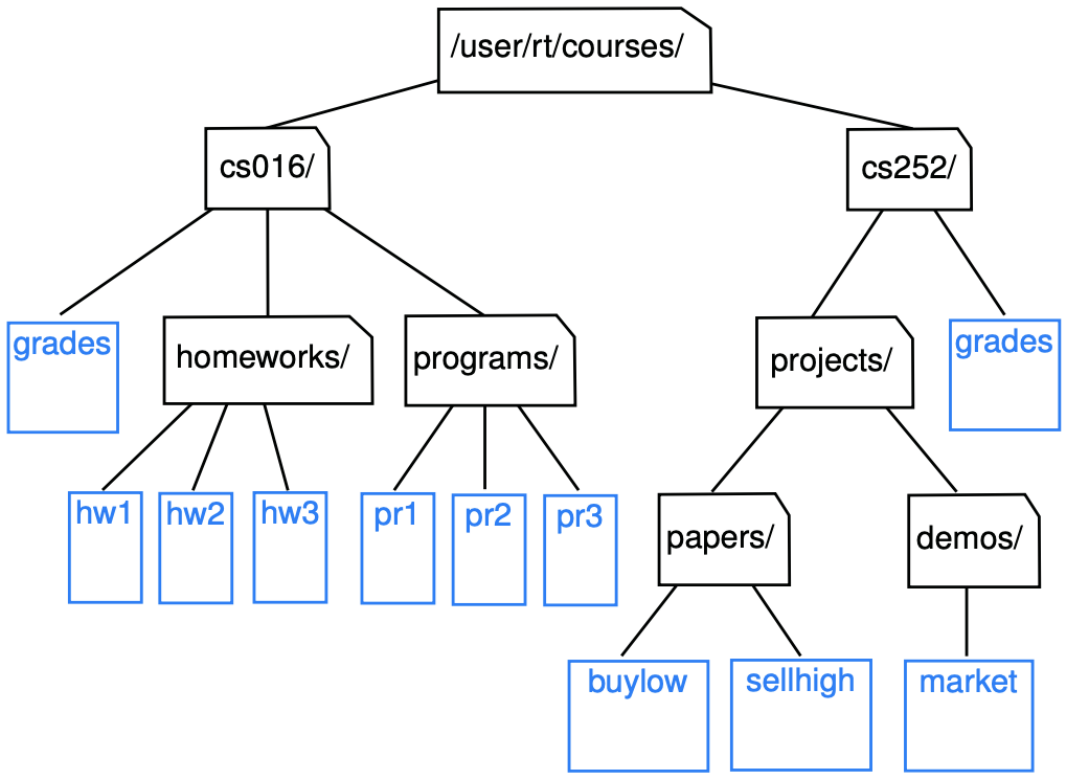
- A node u is an ancestor of a node ν if $u = \nu$ or u is an ancestor of the parent of ν

- Conversely, we say that a node ν is a descendant of a node u if u is an ancestor of ν. For example, in the tree below, `cs252/` is an ancestor of `papers/` and `pr3` is a descendant of `cs016/`.



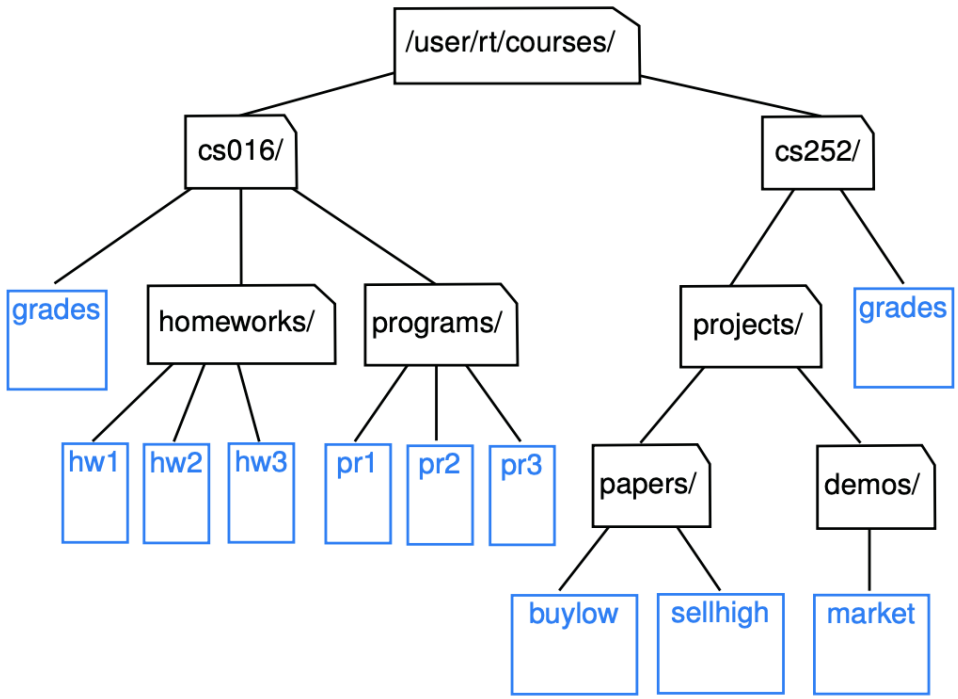
Trees

- The subtree of T rooted at a node ν is the tree consisting of all the descendants of ν in T (including ν itself). In the tree below, the subtree rooted at `cs016/` consists of the nodes `cs016/`, `grades`, `homeworks/`, `programs/`, `hw1`, `hw2`, `hw3`, `pr1`, `pr2` and `pr3`.



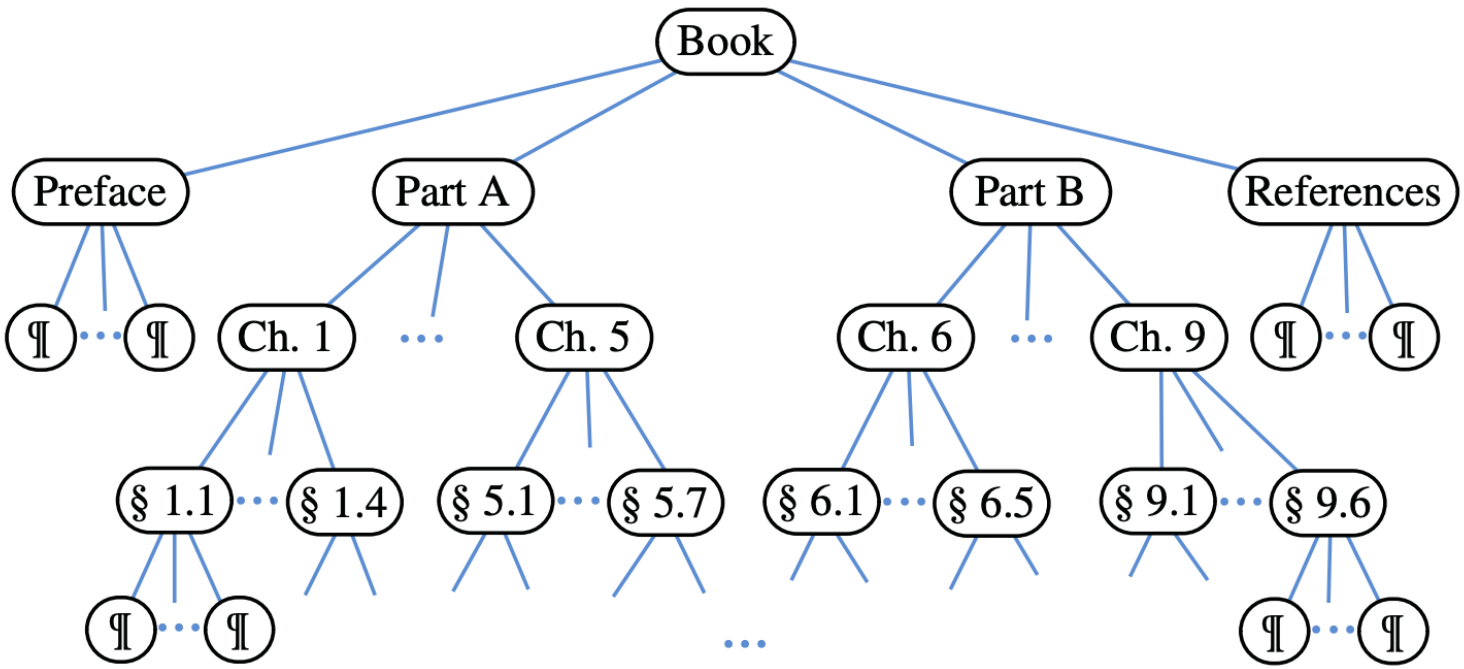
Trees

- An edge of the tree T is a pair of nodes (u, ν) such that u is the parent ν, or vice versa.
- A path of T is a sequence of nodes such that any two consecutive nodes in the sequence form an edge.
- As an example, the tree below contains the paths (`cs252/`, `projects/`, `demos/`, `market`)



Trees

- A tree is ordered if there is a meaningful linear order among the children of each node; that is, we purposefully identify the children of a node as being the first, second, third, and so on
- Such an order is usually visualized by arranging siblings left to right, according to their order



The tree abstract data type

- As we did with positional lists, we will define the tree ADT using the concept of position as an abstraction for a node of a tree.
- An element is stored at each position and positions satisfy parent-child relationships that define the tree sculpture.
- A position object for a tree supports the following methods

getElement()	Returns the element stored at this position

The tree abstract data type

- The tree ADT then supports a set of accessor methods, allowing a user to navigate the various positions of a tree T

root()	Returns the position of the root of the tree (or null if empty)
parent(p)	Returns the position of the parent of position p (or null if p is the root)
children(p)	Returns an iterable collection containing the children of position p (if any)
numChildren(p)	Returns the number of children of position p

- If a tree T is ordered, then children(p) reports the children of p in order.

The tree abstract data type

- In addition to the accessor methods, the following additional methods

isInternal(p)	Returns true if position p has at least one child
isExternal(p)	Returns true if position p does not have any children
isRoot(p)	Returns trueif position p is the root of the tree

- The methods above make programming with trees easier and more readable, since we can use themin the conditionals of **if** statements and **while** loops

The tree abstract data type

- Trees support a number of more general methods, unrelated to the specific structure of the tree which include

size()	Returns the number of positions (hence elements) that are contained in the tree
isEmpty()	Returns true if the tree does not contain any positions (and thus no element)
iterator()	Returns an iterator for all elements in the tree (so that the tree itself is iterable)
positions()	Returns an iterable collection of all positions of the tree

A Tree interface in Java

- To define a tree interface, we rely on the Position interface
- We declare the tree interface to extend Java's iterable interface (hence including the required iterator method)

```
public interface Tree<E> extends Iterable<E> {
  Position<E> root( );
  Position<E> parent(Position<E> p) throws IllegalArgumentException;
  Iterable<Position<E>> children(Position<E> p)
                          throws IllegalArgumentException;
  int numChildren(Position<E> p) throws IllegalArgumentException;
  boolean isInternal(Position<E> p) throws IllegalArgumentException;
  boolean isExternal(Position<E> p) throws IllegalArgumentException;
  boolean isRoot(Position<E> p) throws IllegalArgumentException;
  int size( );
  boolean isEmpty( );
  Iterator<E> iterator( );
  Iterable<Position<E>> positions( );}
```

A Tree interface in Java

- To define a tree interface, we rely on the Position interface
- We declare the tree interface to extend Java's iterable interface (hence including the required iterator method)

```
public interface Tree<E> extends Iterable<E> {
  Position<E> root( );
  Position<E> parent(Position<E> p) throws IllegalArgumentException;
  Iterable<Position<E>> children(Position<E> p)
                          throws IllegalArgumentException;
  int numChildren(Position<E> p) throws IllegalArgumentException;
  boolean isInternal(Position<E> p) throws IllegalArgumentException;
  boolean isExternal(Position<E> p) throws IllegalArgumentException;
  boolean isRoot(Position<E> p) throws IllegalArgumentException;
  int size( );
  boolean isEmpty( );
  Iterator<E> iterator( );
  Iterable<Position<E>> positions( );}
```

An Abstract Tree Base class

- As we saw in previous courses, while an interface is a type definition that includes public declarations of various methods, an interface cannot include definitions for any of those methods
- In contrast, an abstract class may define concrete implementations for some of its methods while leaving other abstract methods without definition
- An abstract class is designed to serve as a base class, through inheritance, for one or more concrete imeplementations of an interface.
- When some of the functionalityof an interface is implemented in an abstract class, less work remains to complete a concrete implementation

An Abstract Tree Base class

- In the case of the Tree interface, we will define an abstract Tree base class, demonstrating how many tree based algorithm can be defined independently of the low-level representation of a tree data structure

- We start with the following simple implementation of the AbstractTree class

```
public abstract class AbstractTree<E>
                        implements Tree<E> {
  public boolean isInternal(Position<E> p) {
      return numChildren(p) > 0; }
  public boolean isExternal(Position<E> p) {
      return numChildren(p) == 0; }
  public boolean isRoot(Position<E> p) {
      return p == root( ); }
  public boolean isEmpty( ) {
      return size( ) == 0; }}
```

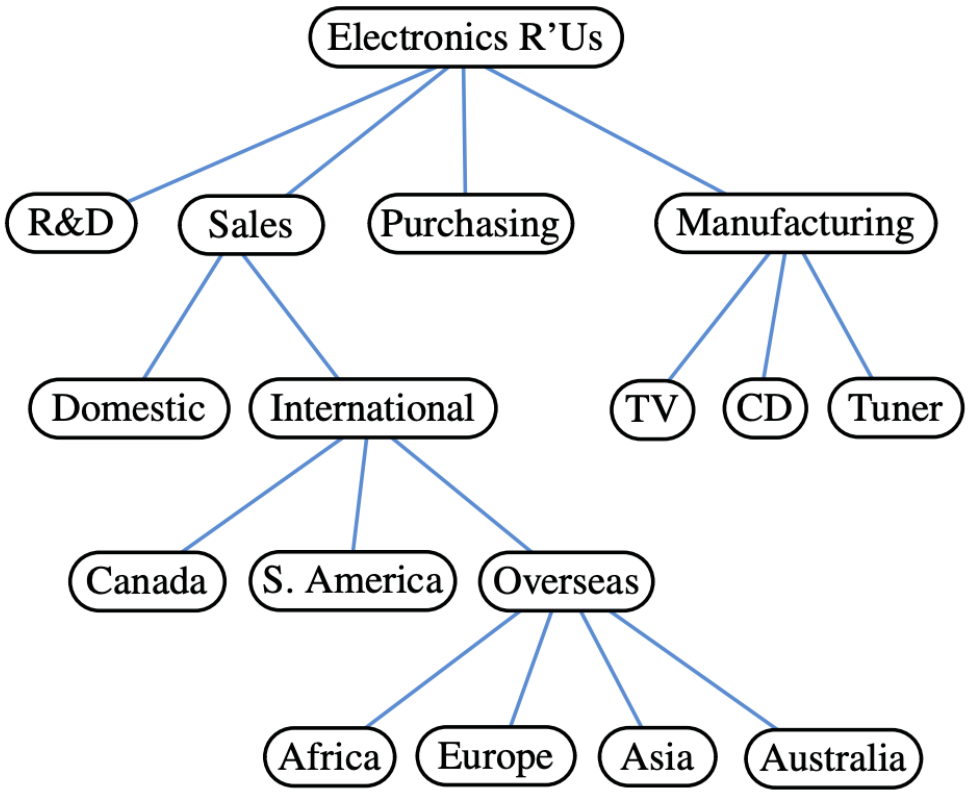
An Abstract Tree Base class

- Recall that any class that implements an interface must provide an implementation for all the methods of the interface, otherwise, the class has to be marked as abstract.
- Here there are no abstract methods in the body of the AbstractTree class. However, some of the methods from the Tree interface are not given any implementation (this the case with **numChildren** for example), hence the keyword **abstract**

Computing depth and height in a Tree

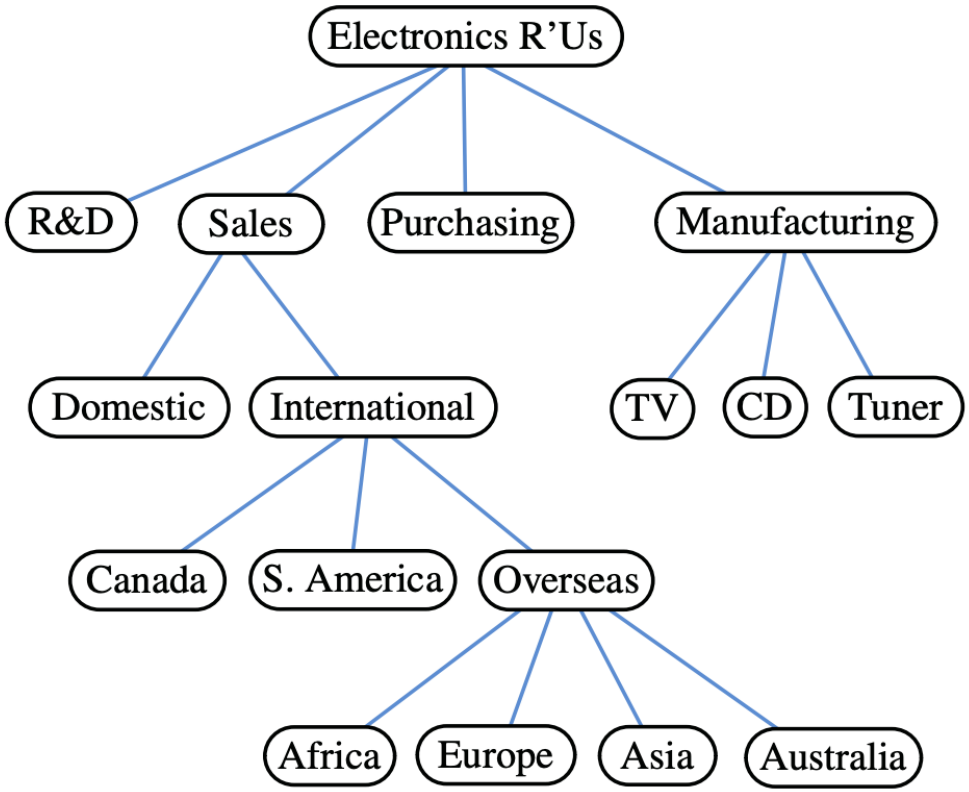
- Let p be a position within the tree T. The depth of p is the number of ancestors of p, other than p itself. For example, in the tree below, the node storing international has depth 2

- Note that this definition implies that the depth of the root of T is 0



Computing depth and height in a Tree

- The depth can also be recursively defined as follows:
  - If p is the root, then the depth of p is 0
  - Otherwise, the depth of p is one plus the depth of the parent of p



Computing depth and height in a Tree

- Based on this definition, we can define a simple recursive algorithm for computing the depth of a position p in the tree

```
public int depth(Position<E> p) {
  if (isRoot(p))
    return 0;
  else
    return 1 + depth(parent(p));
}
```

- The method calls itself recursively on the parents of p and 1 to the value returned
- The running time of `depth(p)` for position p is $O(d_p + 1)$ where d_p denotes the depth of p in the tree, because the algorithm performs a constant time recursive step for each ancestor of p

Computing depth and height in a Tree

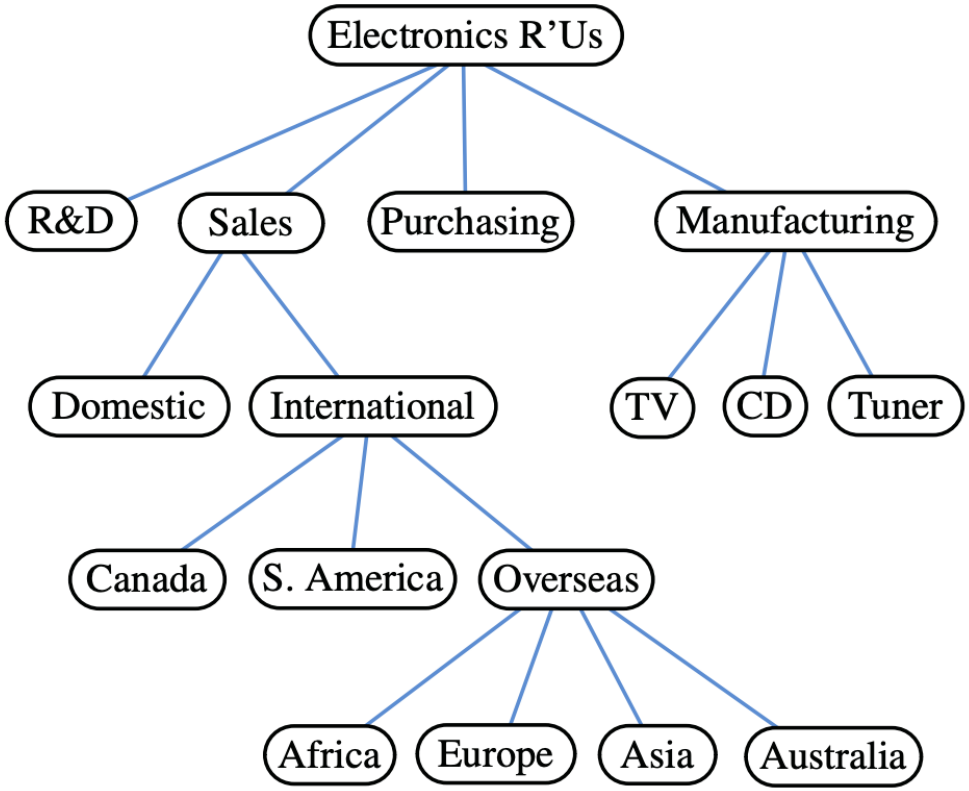
- As a result, the `depth` algorithm runs in $O(n)$ worst case time where n is the total number of positiions of T because a position of T may have depth $n-1$ if all nodes form a single branch

```
public int depth(Position<E> p) {
  if (isRoot(p))
    return 0;
  else
    return 1 + depth(parent(p));
}
```

Computing depth and height in a Tree

- From the definition of the depth of a node p, we can define the height of a tree to be equal to the maximum of the depths of its positions

- The tree below has height 4 as the node storing Africa has depth 4



Computing depth and height in a Tree

- Just as for the depth, we can provide a method that computes the height of a tree

```
private int heightBad( ) {
// works, but quadratic worst-case time
  int h = 0;
  for (Position<E> p : positions( ))
    if (isExternal(p)) // only leaf positions
      h = Math.max(h, depth(p));
  return h;
}
```

- The **positions()** method can be implemented such that the entire iteration runs in $O(n)$ time where n is the total number of positions in T

Computing depth and height in a Tree

- Because **heightBad** calls algorithm **depth** on each leaf of T, its running time is $O(n + \sum_{p \in L}(d_p + 1))$ where L is the set of leaf positions of T
- In the worst case, the sum $\sum_{p \in L}(d_p + 1)$ is proportional to n^2, consequently the algorithm **heightBad** runs in $O(n^2)$ worst case time

```
private int heightBad( ) {
// works, but quadratic worst-case time
  int h = 0;
  for (Position<E> p : positions( ))
    if (isExternal(p)) // only leaf positions
      h = Math.max(h, depth(p));
  return h;
}
```

Computing depth and height in a Tree

- We can however compute the height of a tree more efficiently by considering a recursive definition
- To do this, we parametrize a function based on the position within the tree and calculate the height of the subtree rooted at this position
- Formally, we define the height of a position p in a tree T as follows :
  - If p is a leaf, then the height of p is zero
  - Otherwise, the height of p is one more than the maximumof the heights of p's children

Computing depth and height in a Tree

- Following this recursive approach, we can thus say that the height of tree T is given by the maximum depth among all the leaves of T

- The method **height** below can be considered as a (more efficient) substitute for the original simpler method **heightBad**

```
public int height(Position<E> p) {
  int h = 0; // base case if p is external
  for (Position<E> c : children(p))
    h = Math.max(h, 1 + height(c));
  return h;
}
```

- To determine the total complexity of this new algorithm, we count the number of operations needed on the non recursive part of the call

Computing depth and height in a Tree

- Clearly there is a constant amount of work per position plus the overhead of computing the maximum among positions.
- Although we do not have a concrete implementation for **children(p)**, we can assume for now that such a call can be executed in $O(c_p + 1)$ time where c_p denotes the number of children of p
- The algorithm **height(p)** thus spent $O(c_p + 1)$ time at each position p to compute the maximum and its complexity is given by $O(\sum_p (c_p + 1)) = O(n + \sum_p c_p)$

```
public int height(Position<E> p) {
  int h = 0; // base case if p is external
  for (Position<E> c : children(p))
    h = Math.max(h, 1 + height(c));
  return h;}
```

Computing depth and height in a Tree

- To complete the analysis, we rely on the following proposition

Proposition

Let T be a tree with n postions, and let c_P denote the number of children of a position p in T, then summing over the positions of T, $\sum_p c_P = n - 1$

Proof.

Each position in T, with the exception of the root, is the child of a node and therefore contributes one unit to the sum $\sum_p c_p$ □

```
public int height(Position<E> p) {
  int h = 0; // base case if p is external
  for (Position<E> c : children(p))
    h = Math.max(h, 1 + height(c));
  return h;}
```

Computing depth and height in a Tree

- Combining this with the complexity $O(n + \sum_p c_p)$ we get a total running time of $O(n)$

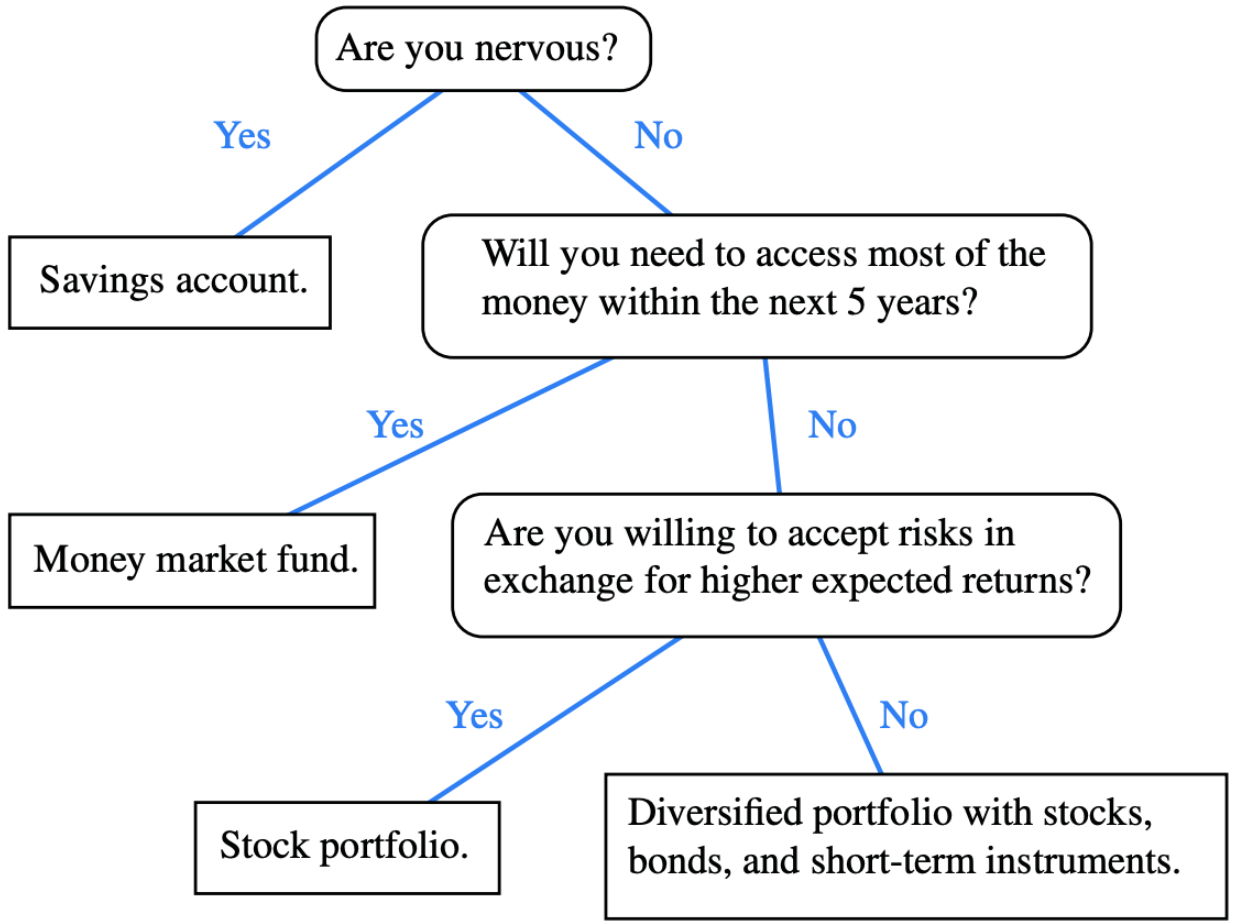
```
public int height(Position<E> p) {
  int h = 0; // base case if p is external
  for (Position<E> c : children(p))
    h = Math.max(h, 1 + height(c));
  return h;}
```

Binary Trees

- A Binary tree is an ordered tree with the following properties
  - Every node has at most two children
  - Each child node is labeled as being either a left child or a right child
  - A left child precedes a right child in the order of children of a node
- The subtree rooted at a left or right child of an internal node ν is called a left subtree or right subtree, respectively, of ν.
- A binary tree is proper if each node has either zero or two children. Some textbooks also refer to such trees as being full binary trees. Hence in a proper binary tree, every internal node has exactly two children.
- A binary tree that is not proper is improper

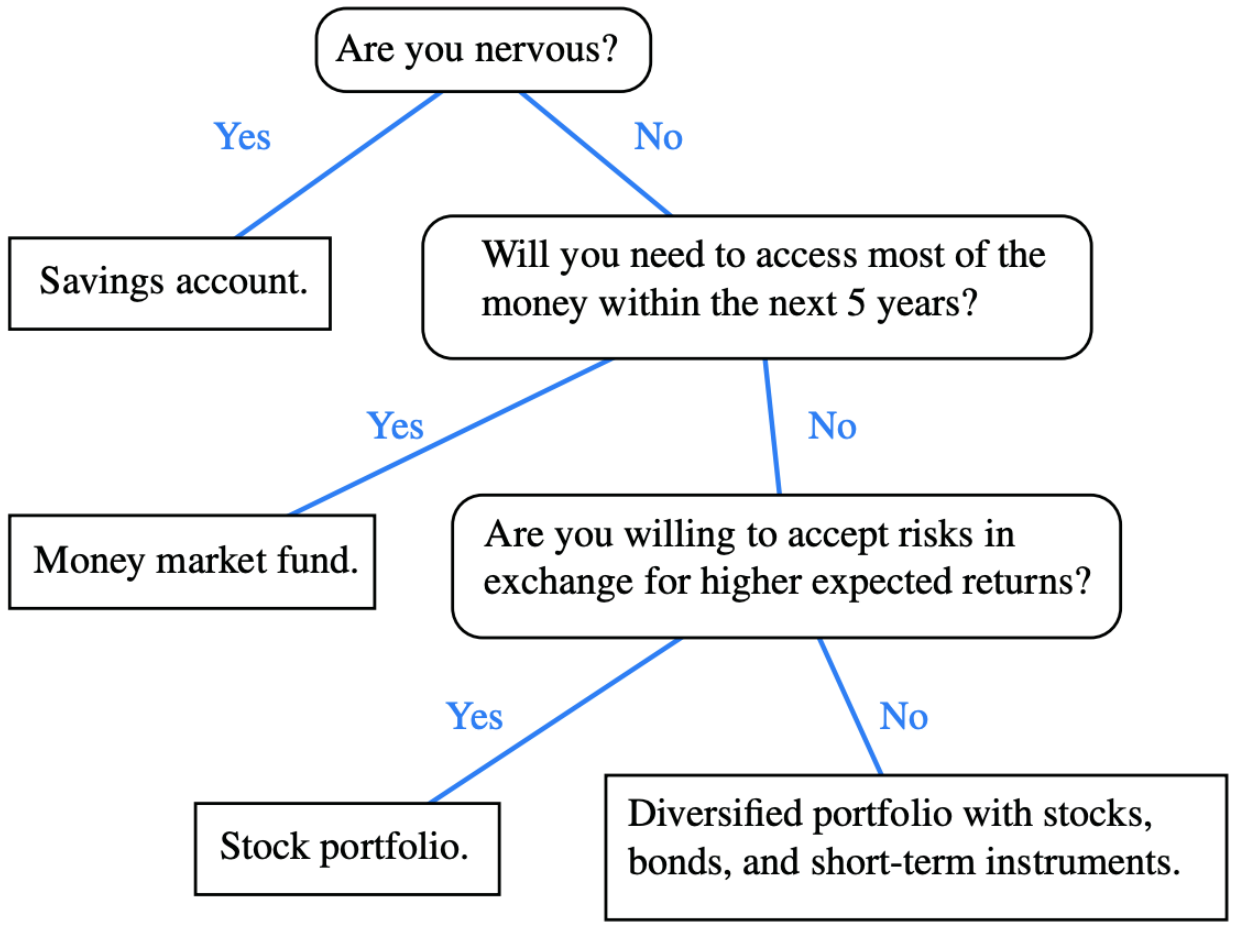
Binary Trees

- An important class of binary trees arises in contexts where we wish to represent a number of different outcomes that can result from answering a series of yes/no questions



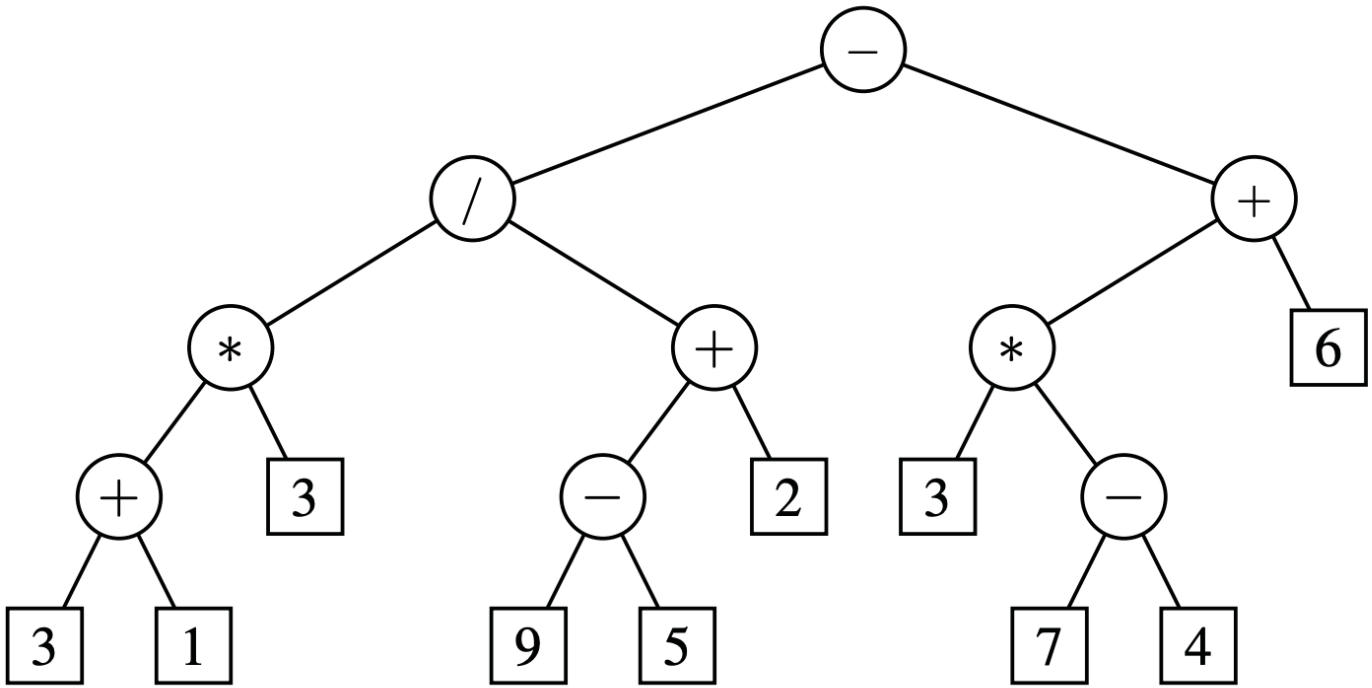
Binary Trees

- Each internal node is associated with a question. Starting at the root, we then go to the left or right child of the current node, depending on whether the answer to the question is "Yes" or "No"



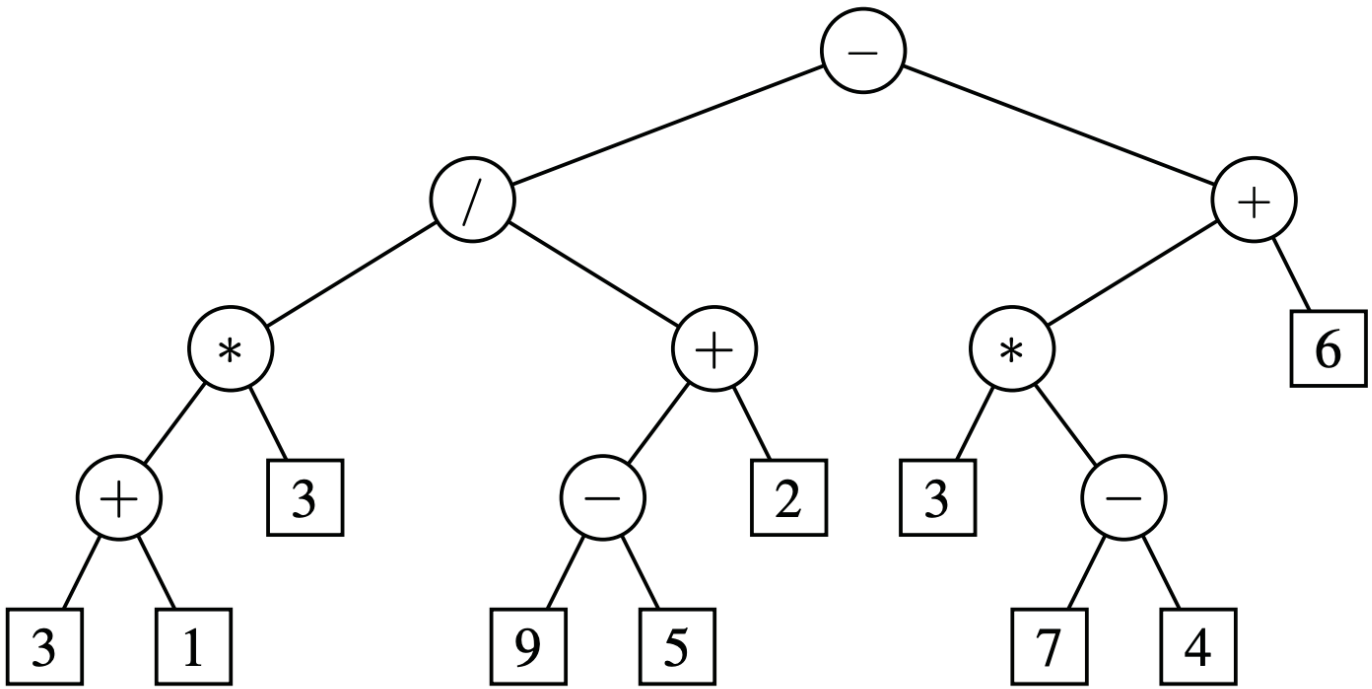
Binary Trees

- An arithmetic expression is another example of a concept that can be represented as a binary tree whose leaves are associated with variables or constants, and whose internal nodes are associated with one of the operators $+, -, *$ and $/$.



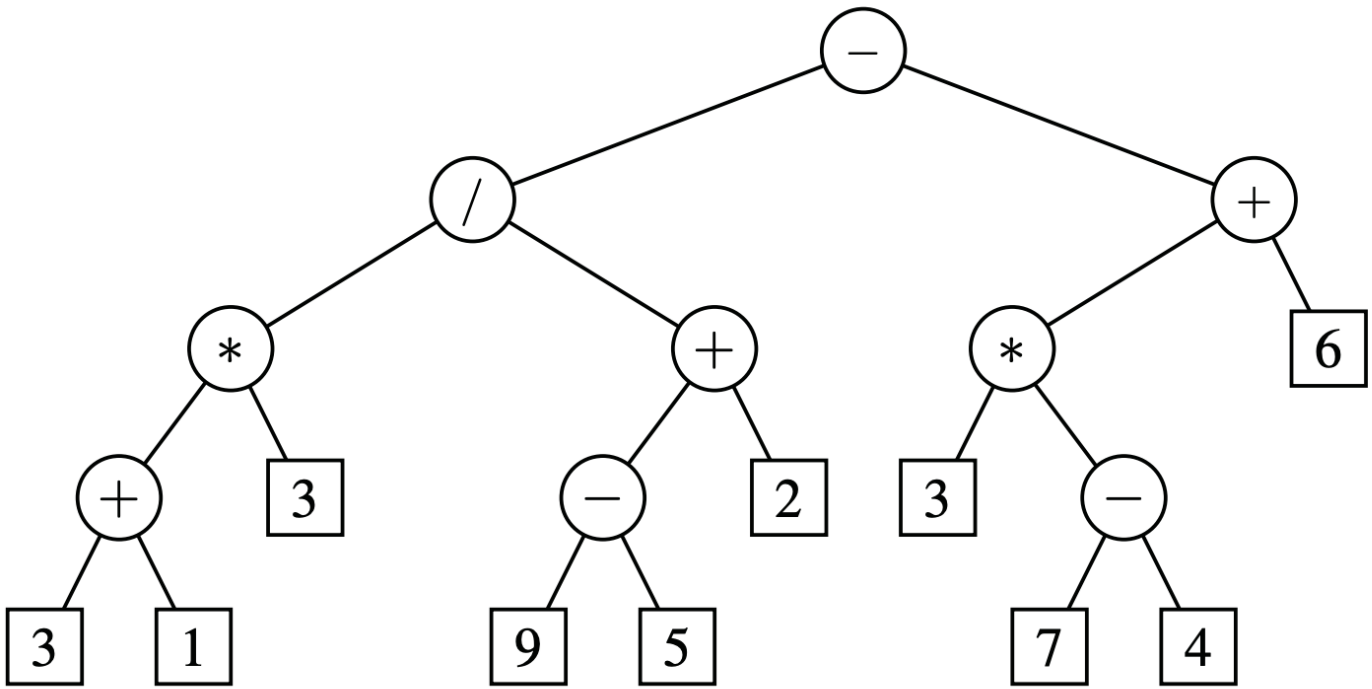
Binary Trees

- Each node in such a tree has a value associated with it: If a node is a leaf, then its value is that of its variable or constant. If a node is internal, then its value is defined by applying its operation to the values of its children



Binary Trees

- The binary tree below is used to represent the arithmetic expression $((((3+1)*3)/((9-5)+2))-((3*(7-4))+6))$



A Recursive Binary Tree

- Incidentally, we can also define a binary tree in a recursive way. In that case, a binary tree is either
  - An empty tree
  - A nonempty tree having a root node r, which stores an element, and two binary trees that are respectively the left and right subtrees of r. We note that one or both of those subtrees can be empty by this definition

The binary tree ADT

- As an abstract data type, a binary tree is a specialization of a tree that supports three additional accessors methods:

left(p)	Returns the position of the left child of p (or null if p has no left child)
right(p)	Returns the position of the right child of p (or null if p has no right child)
sibling(p)	Returns the position of the sibling of p (or null if p has no sibling)

Defining the binary tree Interface

- To define the binary tree interface, we extend the Tree interface as follows

```
/** An interface for a binary tree,
Each node has at most two children. */
public interface BinaryTree<E> extends Tree<E> {
/** Returns the Position of p's left child */
  Position<E> left(Position<E> p)
                throws IllegalArgumentException;
  /** Returns Position of p's right child */
  Position<E> right(Position<E> p)
               throws IllegalArgumentException;
  /** Returns Position of p's sibling */
  Position<E> sibling(Position<E> p)
              throws IllegalArgumentException;}
```

Defining the Abstract Binary Tree class

- To promote reusability, we define our Binary Tree class as an abstract class
- We further let this class inherit from the AbstractTree class

```
public abstract class AbstractBinaryTree<E>
                              extends AbstractTree<E>
  implements BinaryTree<E> {
  /**  Position of p's sibling (or null). */
  public Position<E> sibling(Position<E> p) {
    Position<E> parent = parent(p);
    if (parent == null) return null; //p must be root
    if (p == left(parent)) // p is a left child
      return right(parent);
    else // p is a right child
      return left(parent);
}// to be continued
}
```

Defining the Abstract Binary Tree class

- The sibbling method is derived from a combination of **left**, **right** and **parent**.

- We identify the sibling of a position p as the other "child" of p's parent. p does however not have a sibling if it is the root or if it is the only child of its parent

```
public abstract class AbstractBinaryTree<E>
                              extends AbstractTree<E>
  implements BinaryTree<E> {
  /**  Position of p's sibling (or null). */
  public Position<E> sibling(Position<E> p) {
    Position<E> parent = parent(p);
    if (parent == null) return null; //p must be root
    if (p == left(parent)) // p is a left child
      return right(parent);
    else // p is a right child
      return left(parent);}}
```

Defining the Abstract Binary Tree class

- We can also use the **left** and **right** methods to provide an implementation for the **numChildren** and the **children**

```
public abstract class AbstractBinaryTree<E>
                              extends AbstractTree<E>
  implements BinaryTree<E> {
// Part II
/** Returns the number of children of Position p. */
public int numChildren(Position<E> p) {
  int count=0;
  if (left(p) != null)
    count++;
  if (right(p) != null)
    count++;
  return count;}
// to be continued
}
```

Defining the Abstract Binary Tree class

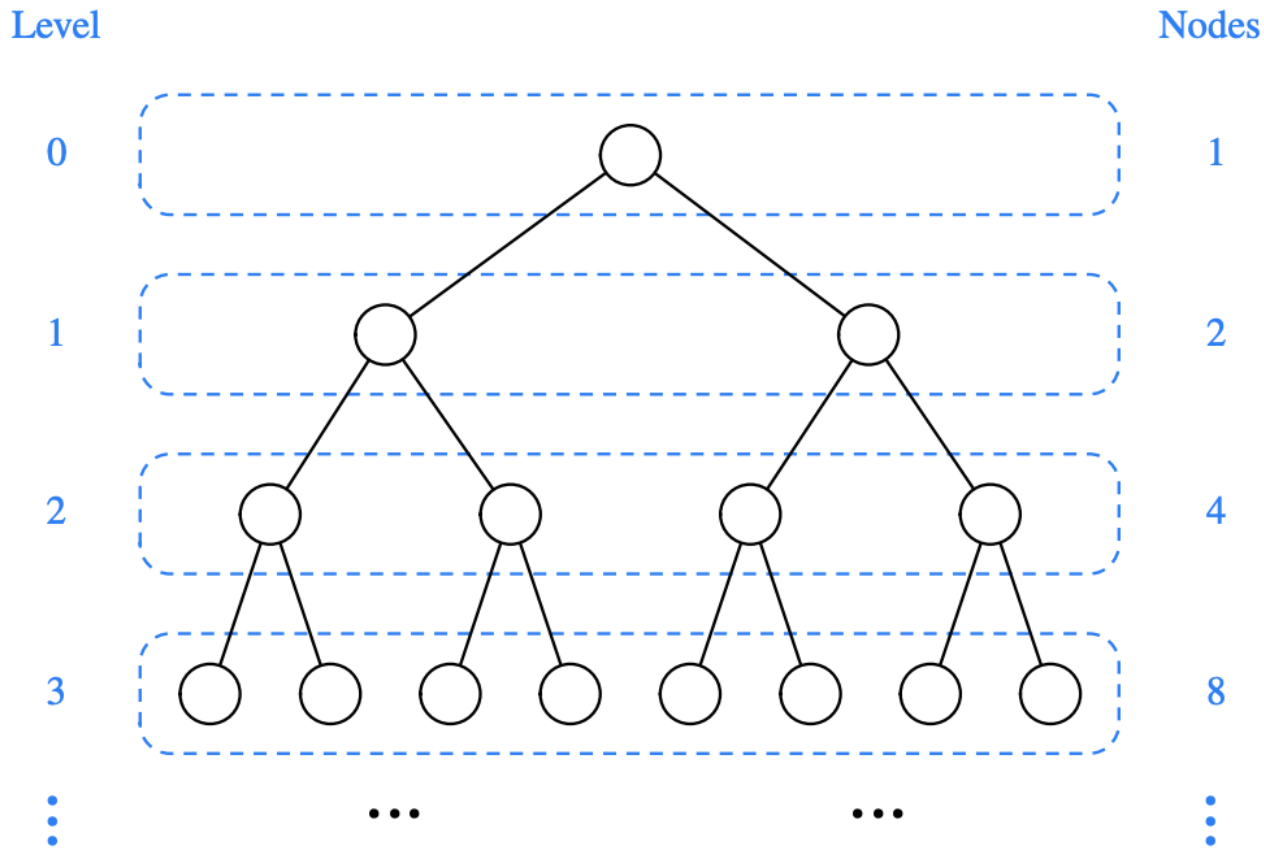
- The implementation of the **children** method relies on producing a snapshot. That is to say, we create an empty **java.util.ArrayList**, which is an iterable container and then add any children that exist, ordered so that a left child is reported before a right child.

```
public abstract class AbstractBinaryTree<E>
                              extends AbstractTree<E>
  implements BinaryTree<E> {
// Part II
public Iterable<Position<E>> children(Position<E> p) {
  List<Position<E>> snapshot = new ArrayList<>(2);
  if (left(p) != null)
    snapshot.add(left(p));
  if (right(p) != null)
    snapshot.add(right(p));
  return snapshot;
}}
```

Properties of Binary Trees

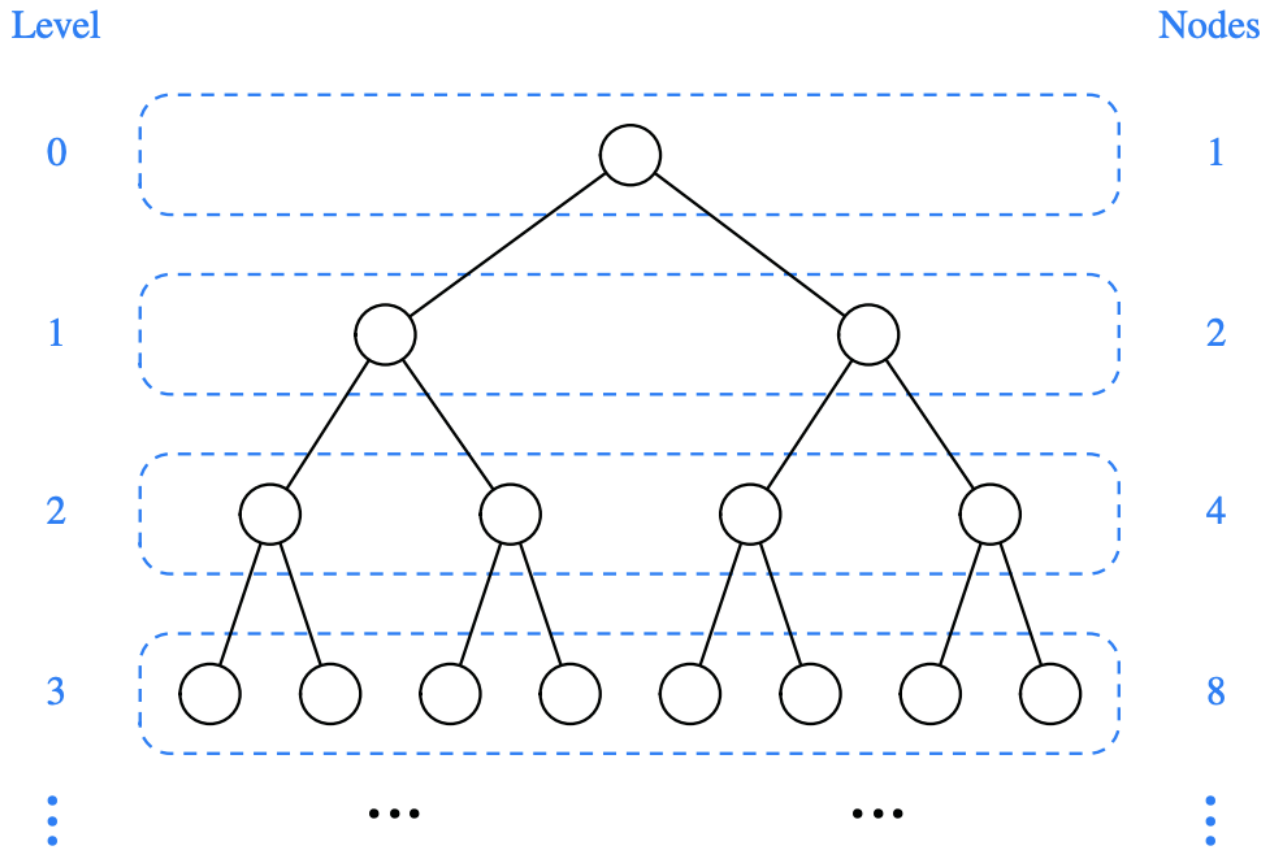
- Binary trees have several interesting properties deadling with relationships between their height and their number of nodes.

- We denote the set of all nodes of a tree T at the same depth d as the level d of T



Properties of Binary Trees

- In a binary tree, level 0 has at most one node (the root), level 1 has at most two nodes (children of the root), and so on.
- Generally speaking, level d has at most 2^d nodes



Properties of Binary Trees

- We can see that the maximum number of nodes on the levels of a binary tree grows exponentially as we go down the tree
- From this simple observation we can derive the following properties relating the height of a binary tree T with its number of nodes

Proposition

Let T be a nonempty binary tree, and let n, n_E, n_I and h denote the number of nodes, number of external nodes (i.e leafs), number of internal nodes, and height of T respectively. Then T has the following properties

- $h + 1 \leq n \leq 2^{h+1} - 1$
- $1 \leq n_E \leq 2^h$
- $h \leq 2^h$
- $\log(n + 1) - 1 \leq h \leq n - 1$

Properties of Binary Trees

Proposition

Let T be a nonempty binary tree, and let n, n_E, n_I and h denote the number of nodes, number of external nodes (i.e leafs), number of internal nodes, and height of T respectively. If T is proper, then T has the following additional properties

- $2h + 1 \leq n \leq 2^{h+1} - 1$
- $h + 1 \leq n_E \leq 2^h$
- $h \leq n_I \leq 2^h - 1$
- $\log(n + 1) - 1 \leq h \leq (n - 1)/2$

Properties of Binary Trees

- In addition to the previous properties, the number of external nodes and the number of internal nodes can be related through the following proposition

Proposition

In a nonempty proper binary tree T with n_E external nodes and n_I internal nodes, we have $n_E = N_I + 1$

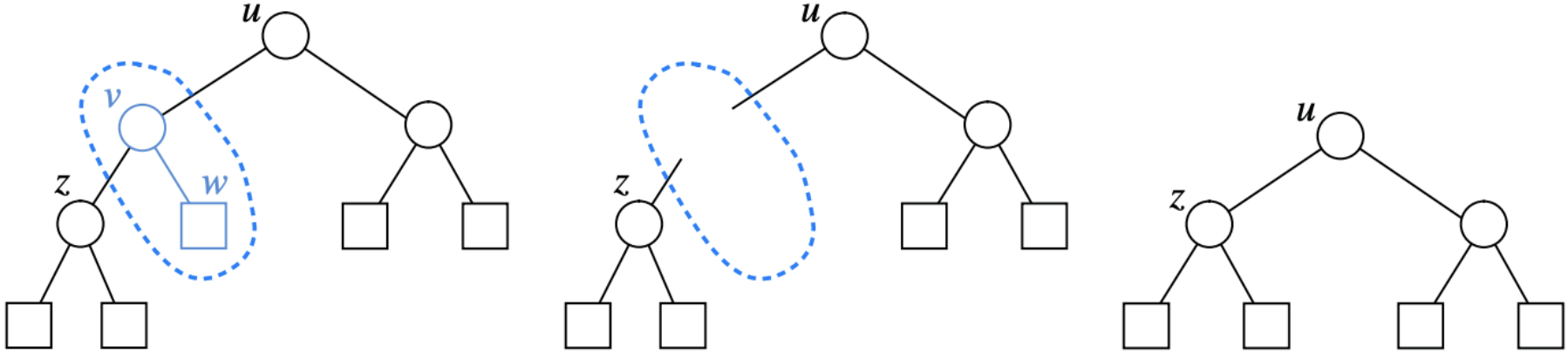
- To see this, we can remove nodes from the tree and divide them into two piles: an internal node pile and an external node pile until T becomes empty.

Properties of Binary Trees

Proposition

In a nonempty proper binary tree T with n_E external nodes and n_I internal nodes, we have $n_E = N_I + 1$

- If T has only one node ν, we remove ν and place it on the external node pile. Thus the external node pile has only one node and the internal node pile is empty.
- Otherwise we remove from T an (arbitrary) external node w and its parent v which is an internal node. We place w on the external node pile and v on the internal node pile. If v has a parent u, then we reconnect u with the former sibling z of w



Properties of Binary Trees

Proposition

In a nonempty proper binary tree T with n_E external nodes and n_I internal nodes, we have $n_E = N_I + 1$

- Note that the operation below removes one internal node and one external node and leaves the tree being a proper binary tree.
- Repeating this operation, we are eventually left with a final tree consisting of a single node.