https://www.cs.cornell.edu/courses/cs2112/2014fa/lectures/lecture.html?id=trees

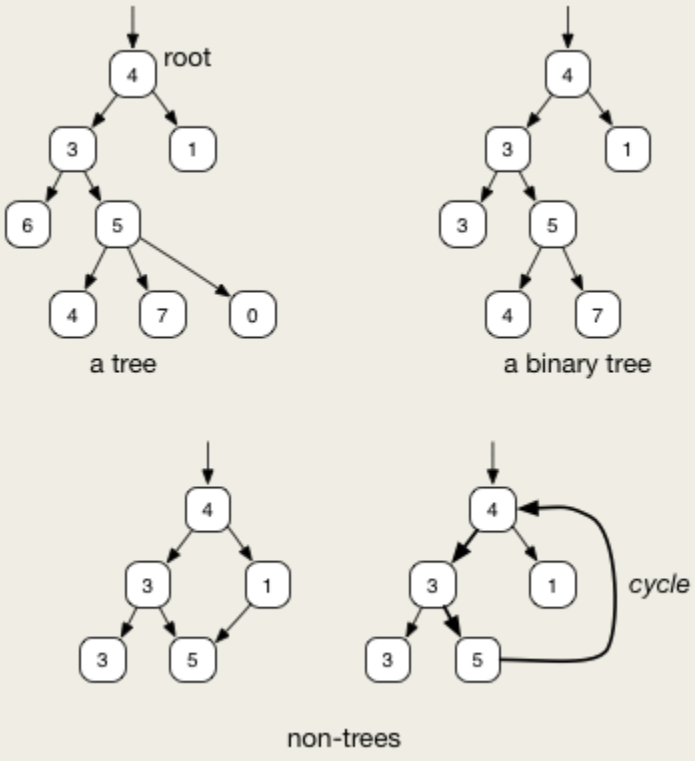
CS 2112 : Object-Oriented Design and Data Structures – Honors

Fall 2014

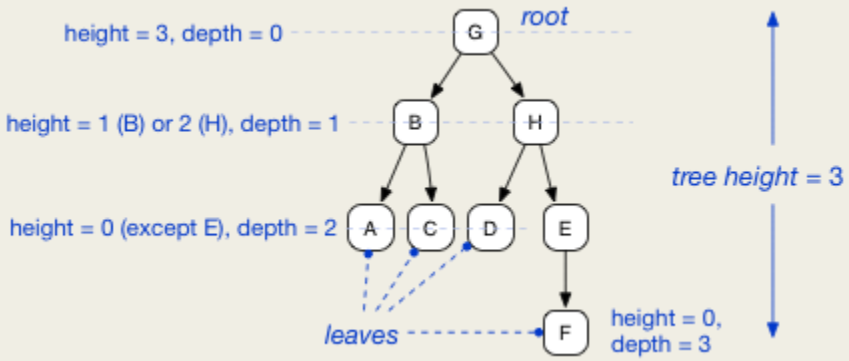
Trees

Trees are a very useful class of data structures. Like (singly-)linked lists, trees are form an **acyclic** graph of node objects in which each node may have some attached information. Whereas linked list nodes have zero or one successor nodes, tree nodes may have more. Having multiple successors (called **children**) makes trees more versatile as a way to represent information and often more efficient as a way to find information within the data structure.

Trees are recursive data structures. A tree has a single **root node** that is the starting point for walking the graph of tree nodes; all nodes are **reachable** from the root. All tree nodes except the root have exactly one predecessor node, called its **parent**. We often draw the root with an incoming arrow to distinguish it. It is convenient to draw trees as growing downward:



Because a tree is a recursive data structure, each child of node in the tree is the root of a **subtree**. For example, in the following tree, node G has two children B and H, each of which is the root of a subtree. This tree is a **binary** tree in which each node has up to two children. We say that a binary tree has a **branching factor** of 2. In a binary tree, the children are usually ordered, so B is the left child and root of the left subtree, and H is the right child and root of the right subtree. Nodes that have no children are called **leaves**.



Each node in a tree has a height and a depth. The depth is the length of the path from the root. The height is the length of the path from the node to the deepest leaf reachable from it. The height of a tree is the height of its root node, or the depth of the deepest leaf.

Conventionally we represent a binary tree using a class with instance variables for the left and right children:

```
class BinaryNode<T> {
    T data;
    BinaryNode<T> left, right; // may be null
}
```

For trees with a larger branching factor, the children may be stored in another data structure such as an array or linked list:

```
class NAryNode<T> {
    T data;
    NAryNode<T>[] children;
}

class NAryNode<T> {
    T data;
    LinkedList<NAryNode<T>> children;
}
```

Analogously to doubly-linked lists, tree nodes in some tree data structures maintain a pointer to their parent node (if any). A parent pointer can be useful for walking upward in a tree, though it takes up extra memory and creates additional maintenance requirements.

Why trees?

There are two main reasons why tree data structures are important:

- First, some information has a naturally tree-like structure, so storing it in trees makes sense. Examples of such information include parse trees, inheritance and subtyping hierarchies, game search trees, and decision trees.

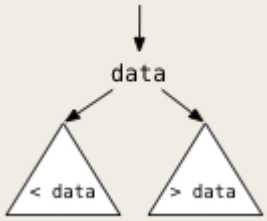
- Second, trees make it possible to find information within the data structure relatively quickly. If a significant fraction of the nodes have more than one child, it can be arranged that all nodes are fairly close to the root. For simplicity, let's think about a **full tree** in which all leaves are at equal depth and all non-leaves have two children. In a full binary tree of depth h, there are $1 + 2 + 4 + ... 2^h = 2^{h+1} - 1$ nodes. Calling this number of nodes n, we have $h = \lg(n+1)-1$, so h is $O(\lg n)$. (Recall that lg x is the logarithm of x base 2). Thus, if we are looking for information in a full binary tree, it can be reached along a path whose length is logarithmic in the total information stored in the tree.

  For large n, logarithmic time is a big speedup over the linear-time performance offered by data structures like linked lists and arrays. For n = 1,000,000, lg n = 20, a speedup of 50,000 when constant factors are ignored. For n = 1,000,000,000, lg = 30, a speedup of more than 30,000,000.

Binary search trees

Of course, the preceding analysis relies on us knowing which path to take through the tree to find information. This can be arranged by organizing the tree as a **binary search tree**.

A binary search tree is a binary tree that satisfies a data structure invariant: for each node in the tree, all elements in the node's left subtree are less than the element at the node, and all elements in the node's right subtree are greater than the node's element. Pictorially, we can visualize the tree relative to a given node as follows:



We can express this data structure invariant as a class invariant:

```
class BinaryNode<T extends Comparable<T>> {
    T data;
```

```
    /** Invariant: {@code left} and {@code right} are the root of subtrees,
     *  where all elements in the left subtree are less than {@code data},
     *  and all elements in the right subtree are greater. The value {@code
null}
     *  represents an empty subtree.
     */
    BinaryNode<T> left, right;
}

interface Comparable<T> {
    /** Return 0 if {@code this == y}, a positive number if {@code this > y},
     * and a negative number if {@code this < y}.
     */
    int compareTo(T y);
}
```

Since the invariant is defined on a recursive type, it applies recursively throughout the tree, ensuring that data structure invariant holds for every node.

For the invariant to make sense, we must be able to compare two elements to see which is greater (in some ordering). The ordering is specified by the operation `compareTo()`. One way to ensure that the type `T` has such an operation is to specify in the class declaration that `T` **`extends`** `Comparable<T>`, where `Comparable` is the generic interface shown above. The keyword **`extends`** merely signifies that `T` is a *subtype* of `Comparable<T>`; it is perfectly sufficient for `T` to be a class that "implements" the interface. The compiler will prevent us from instantiating the class `BinaryNode` on any type T that is not a declared subtype of `Comparable<T>`, and therefore the code of `BinaryNode` can assume that `T` has the `compareTo()` method.

Searching in a binary search tree

Now, consider what happens when we try to find a path down through the tree looking for an arbitrary element x. We compare x to the root data value. If x is equal to the data value, then we've already found it. If x is less than the data value and it's in the tree, it *must* be in the left subtree, so we can walk down to the left subtree and look for the element there. Conversely, if x is greater than the data value and it's in the tree, it must by in the right subtree, so we should look for it there. In either case, if the subtree where x must be is empty (null), the element must not be in the tree. This algorithm can be expressed compactly as a (tail-)recursive method on `BinaryNode`:

```
boolean contains(T x) {
    int c = x.compareTo(data);
    if (c == 0) return true;
    BinaryNode<T> child = (c < 0) ? left : right;
    if (child == null) return false;
    return child.contains(x);
}
```

We've used Java's "ternary expression" here to make the code more compact (and to show off another coding idiom!) The expression b `?` e_1 `:` e_2 is a **conditional expression** whose value is the result of either e_1 or e_2, depending on whether the boolean expression b evaluates to `true` or `false`. Since the method is tail-recursive, we can also write it as a loop. Here is a version where the root of the tree is passed in explicitly as a parameter `n`:

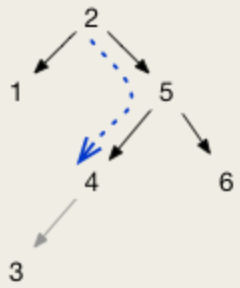
```
static boolean contains(T x, BinaryNode<T> n) {
    while (n != null) {
        int c = x.compareTo(n.data);
        if (c == 0) return true;
        n = (c < 0) ? left : right;
    }
    return false;
}
```

Adding an element to a binary search tree

To add an element so we can find it later, it has to be added along the search path that will be used. Therefore, we add an element by search for the element. If found, it need not be added; if not found, it is added as a child of the leaf node reached along the search path. Again, this can be written easily as a tail-recursive method:

```
/** Add element x to the binary search tree, unless it is already there.
    Return whether if the element was added.
 */
boolean add(T x) {
    if c = x.compareTo(data);
    if (c == 0) return false;
    if (c < 0) {
        if (left != null) return left.add(x);
        left = new BinaryNode<T>(x);
    } else {
        if (right != null) return right.add(x);
        right = new BinaryNode<T>(x);
    }
    return true;
}
```

To see how this algorithm works, consider adding the element 3 to the tree shown with the black arrows in the following diagram. We start at the root (2) and go to the right (5) because 3 < 2. In the recursive call we then go to the left (2<5) to node 4. Since 3<4, we try to go to the left but observe `left==null`, and therefore create a left child containing 3, shown by the gray arrow.



Using trees to implement maps

A map abstraction lets us associate values with keys, and look up the value corresponding to a given key. This can be implemented using a tree by using the elements as keys, but adding the associated value to the same tree node as its key. When the key is found, so is the associated value. An alternate way to view this implementation is that each element stored in the tree is really a pair of a **key** and a **value**, where two elements are ordered according to their keys alone.

Supporting duplicate elements

For some applications, it may be useful to store multiple elements that are considered equal to each other. Suppose elements are key–value pairs, but we want to allow a key to be associated with *multiple* values. To allow equal elements to be stored in the same tree, we need to relax the BST invariant slightly. Given a node containing value x, we must know whether to go to the left or right to find the other occurrences of x. To build a tree where we go left, we relax the BST invariant so that the left subtree contains elements less than *or equal* to x, whereas the right subtree contains elements strictly greater than x.

N-ary search trees

It is possible to define search trees with more than two children per node. The higher branching factor means paths through the tree are shorter, but considerably complicates all of the algorithms involved. B-trees are an example of an N-ary search tree structure. In N-ary search tree, each node contains up to N-1 elements e0..eN-2 and has up to N children c0, arranged so that the subtrees of the children contain only elements between successive elements at the node. If a node has n children, the node contains n-1 elements obeying the following invariant:

$$c_0 < e_0 < c_1 < e_1 < c_2 < e_2 \ldots < e_{n-2} < c_{n-1}$$

Given an element to be searched for, we do a binary search on the elements e_i and if, it is not found, the invariant indicates the appropriate child subtree to search.

Parent pointers

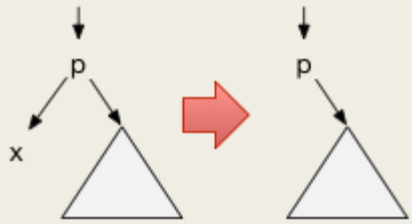
Thus far we have only had pointers going downward in the tree. It is sometimes handy to have pointers going from nodes to their parents, much as nodes in a doubly linked list contain pointers to their predecessor nodes.

Removing elements from a binary search tree

Removing elements from a tree is generally more complicated than adding them, because the elements to be removed are not necessarily leaves. The algorithm starts by first finding the node containing the value x to be removed, and its parent node p. There are three cases to consider:

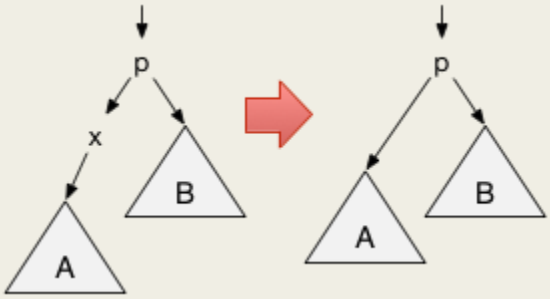
1. *Node x is a leaf.*

   In this case, we can **prune** the node x from the tree by setting the pointer to it from p to `null`. The other subtree of p (shown as a white triangle, but it may be empty) is unchanged.



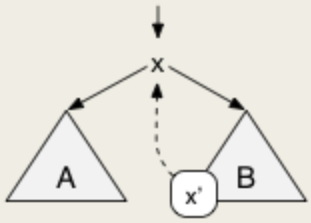
2. *Node x has one child.*

   We **splice out** node x from the tree by redirecting the pointer from p to x to now point to the single child of x. Since the BST invariant guarantees that A < x < p < B, splicing out x preserves the BST invariant.



3. *Node x has two children.*

   In this case it's not easy to remove node x. Instead, we **replace** the data element in the node with the element from, depending on the implementation, either the very next element in the tree or the immediately previous element. Suppose, without loss of generality, we always use the very next element x'. Since it's the immediately next element, its node can't possibly have a left child. Therefore, we either *prune* the x' node (if it is a leaf) or *splice it out* (if it is not), and then overwrite the data in the x node with x'.

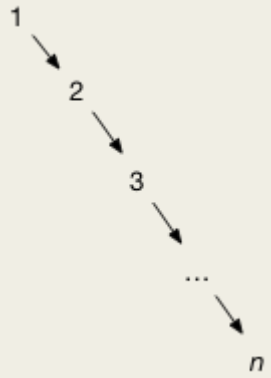


To find the element x', we start by walking down the tree one step to the right, since the element x' must be in the right subtree of x. The element x' must be the smallest element in the right subtree. We find the smallest element in the subtree by simply walking down to the left as far as possible. Note that this smallest element may have a right child, in which case we reattach that child to the parent of the node being removed. One interesting case in which that happens is when the right child of x has no left child.

Asymptotic performance of binary search trees

The time required to search for elements in a search tree is in the worst cas proportional to the longest path from the root of the tree to a leaf, or the height of the tree, h. Therefore, tree operations take O(h) time.

With the simple binary search tree implementation we've seen so far, the worst performance is seen when elements are inserted in order, e.g.: `add(1)`, `add(2)`, `add(3)`, ... `add(n)`. The resulting binary tree will only have right children, and will be functionally identical to a linked list!



For this tree, h is O(n), and tree operations are O(n). Our goal is logarithmic performance, which requires h is O(lg n). A tree in which h is O(lg n) is said to be **balanced**. Many techniques exist for obtaining balanced trees.

One simple-minded way to balance trees is to insert elements into the tree in a random order. This turns out to result in a tree whose expected height is O(lg n) (Proving this is outside the scope of this

course, but see CLRS, Chapter 12.4 for a proof). However, we need to know how to shuffle a collection of elements into a random order, which involves a small digression:

How to place a sequence of elements in random order

The Fisher–Yates algorithm (developed in 1938!) places N elements into random order. Recall that there are N! possible permutations of N elements; a perfectly random shuffle should have equal probability 1/N! of producing any given permutation. The algorithm works as follows. Assume we have the N elements in an array. We iterate from N-1 down to 0 deciding which element to place into a given array index. In each array index we randomly choose one of the elements in the array indices up to that point and swap it with the current array index.

```
T[] a;
int N = a.length;
Random r = new Random();
for (int i = N-1; i > 0; i--) {
    int j = r.nextInt(i+1);
    // swap a[j] and a[i]
    T temp = a[j];
    a[j] = a[i];
    a[i] = temp;
}
```

The first iteration generates one of N possible values, the second iteration one of N-1 possible values, and so on until the final iteration generates one of two possible values. Therefore, the total number of possible ways to execute is N×(N-1)×(N-2)×...×2 = N!. Furthermore, given a particular permutation, there is exactly one way for the algorithm to produce it. Therefore, all permutations are produced with equal probability, assuming the random number generator is truly random.

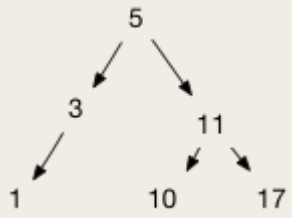
Traversing trees

Given a tree containing some number of elements, it is sometimes useful to **traverse** the tree, visiting each element and doing something with it, such as printing it out or adding it to another collection.

The most common traversal strategy is **in-order traversal**, in which each element is visited between the elements of the subtrees. In-order traverse can be expressed easily using recursion:

```
/** Apply the method visit() to every node in the tree, using an in-order
traversal. */
void traverse() {
    if (left != null) left.traverse();
    visit(data);
    if (right != null) right.traverse();
}
```

For example, consider using this algorithm on the following search tree:



The elements will be visited in the order 1, 3, 5, 10, 11, 17.

The traversal is not clearly tail-recursive and therefore cannot be easily converted into a loop, but this is not a problem unless the tree is very deep. If iterative traversal is required, it can be done if nodes contain parent pointers.

Notice that an in-order traversal of a binary search tree visits the element in sorted order. This observation gives us, in fact, an asymptotically efficient sorting algorithm. Given a collection of elements to sort, we add them into a BST, taking O(hn) time. If the elements are first shuffled into a random order, h is O(lg n) with high probability, so adding all the elements takes O(n lg n) time. Then, we can use an in-order traversal, taking O(n) time, to extract all the elements out in order. While no general sorting algorithm is more efficient asymptotically than O(n lg n), we will later see some other sorting algorithms that are just as asymptotically efficient but have lower constant factors.

Other traversals can be done. By moving the visiting of the element to both or after the descendants, we arrive at **preorder** and **postorder** traversals respectively:

```
/** Apply the method visit() to every node in the tree, using a preorder
traversal. */
void traverse() {
    visit(data);
    if (left != null) left.traverse();
    if (right != null) right.traverse();
}

/** Apply the method visit() to every node in the tree, using a postorder
traversal. */
void traverse() {
    if (left != null) left.traverse();
    if (right != null) right.traverse();
    visit(data);
}
```

A preorder traversal of the tree above visits the nodes in the order 5, 3, 1, 11, 10, 17: a node is visited before all of its descendants. A postorder traversal visits a node after all of its descendants; for this example, in the order 1, 3, 10, 17, 11, 5.

References

Cormen, Leiserson, Rivest, Stein. *Introduction to Algorithms.*