

Search Trees and Sorting

COMP1100 — Introduction to Programming and Algorithms

Ben Lippmeier

Australian National University

Semester 2, 2006

Trees

Along with tuples and lists, trees are another basic structure which is used widely in computer science.

There are several variations on the basic tree structure, and versions are found anywhere a large body of data must be stored, searched or organised.

Some examples are:

- Binary trees
- B-trees
- AVL trees
- Red-Black trees
- Binary Space Partitioning (BSP) trees

Binary trees

Binary trees are one of the simplest tree structures, and are used to store *ordered* values. They are called binary trees because each node always has *two* sub-trees.

A **Binary Tree** is either:

- **empty**, or
- it consists of a **node** containing some value and **left and right subtrees** where the subtrees are also **binary trees**.

Here is the data type for the polymorphic binary trees that we saw a few lectures ago.

```
data Tree a
  = Null
  | Node a (Tree a) (Tree a)
```

We call it *polymorphic* because it can store values of any type. eg, a binary tree of Int would have type `Tree Int`.

Terminology

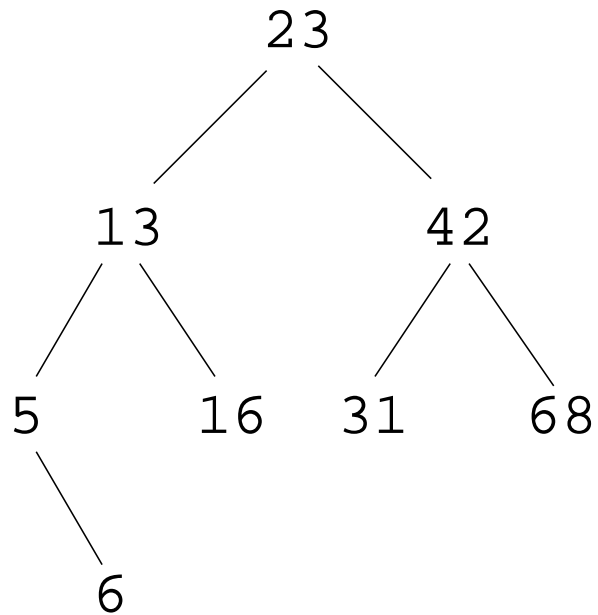
- The node at the top of the tree is called the *root*.
- The subtrees of a node are called *branches*.
- A node whose subtrees are both Null is called a *leaf*.
- We call a value contained within a node a *key*.
- Nodes which are not leaves are sometimes called *internal nodes*.
- The *depth* of a tree is the maximum number of nodes that can be encountered during a downward traversal.

Invariant

For any given node, call its key **k**.

- The keys in the left hand subtree of that node are always *less than k*.
- The keys in the right hand subtree are always *more than k*.

Obviously, for this to hold the keys must have a total ordering defined for them.



```

(Node 23 (Node 13 (Node 5 Null
                  (Node 6 Null Null))
          (Node 16 Null Null))
 (Node 42 (Node 31 Null Null)
          (Node 68 Null Null)))
  
```

Counting the elements

Finding the size of a binary tree is much the same as finding the length of a list.

- If a list is empty, then its length is zero.
- If a list contains elements, then we count the *head* as one then add that to the length of the *tail*.

```
length :: [a] -> Int
```

```
length [] = 0
```

```
length (x:xs) = 1 + length xs
```

For a tree:

- If a tree is empty, then its size is zero.
- If a tree has nodes, then we count the first node as one, then add that to the size of the two branches.

```
treeSize :: Tree a -> Int
treeSize Null      = 0
treeSize (Node x left right)
              = 1 + treeSize left + treeSize right
```

Inserting a node

When we add an element to a *list*, we use the *cons* function.

```
(: )      :: a -> [a]      -> [a]
```

The *cons* function takes a element, a list, and produces a new list.

When we write our `treeInsert` function we'll use the same order for the arguments. `treeInsert` will take a key, a tree, and produce a new tree which includes the provided key.

```
treeInsert :: Ord a  
           => a -> Tree a -> Tree a
```

We'll need to use ordering on keys so we can find the correct place in the tree to do the insert, hence the type class constraint `Ord a`.

The base case:

When we add an element to an empty tree we get a single node and two empty subtrees `Node x Null Null`. ie, a leaf.

```
treeInsert x Null = Node x Null Null
```

The step case:

We can't add a new key directly to a node. We'll need to descend into the tree until we reach a leaf. Remember the invariant. We use the keys in internal nodes to determine which subtree to insert into.

```
treeInsert x (Node x' left right)
  | x < x'      = Node x' (treeInsert x left) right
  | x > x'      = Node x' left  (treeInsert x right)
```

Putting it together

If the key is already in the tree, we could either ignore it or throw an error.

We'll throw an error in this case.

```
treeInsert :: Ord a => a -> Tree a -> Tree a
treeInsert x Null      = Node x Null Null
treeInsert x (Node x' left right)
  | x < x'      = Node x' (treeInsert x left) right
  | x > x'      = Node x' left (treeInsert x right)
  | otherwise   = error "treeInsert: duplicate key."
```

(TreeInsert.hs)

Building a tree from a list

Haskell has a nice representation for lists. We would like a function to insert all the elements from a list into a given tree.

- To insert zero elements, return the original tree.
- To insert a non-zero number of elements, insert the first one then insert the rest.

```
treeBuild :: Ord a => Tree a -> [a] -> Tree a
treeBuild tree []          = tree
treeBuild tree (x:xs)
    = treeBuild (treeInsert x tree) xs
```

Finding a node

Like inserting a node, a function which finds a node must determine where in the tree a key *should be*.

- We won't find our key in an empty tree, so return `Nothing`.
- For a given node, either the key we're looking for is *in that node* or else it could be in one of the subtrees.

```
treeLookup :: Ord a => a -> Tree a -> Maybe a
treeLookup x Null          = Nothing
treeLookup x (Node x' left right)
  | x == x'                = Just x'
  | x < x'                 = treeLookup x left
  | x > x'                 = treeLookup x right
```

Flattening a tree

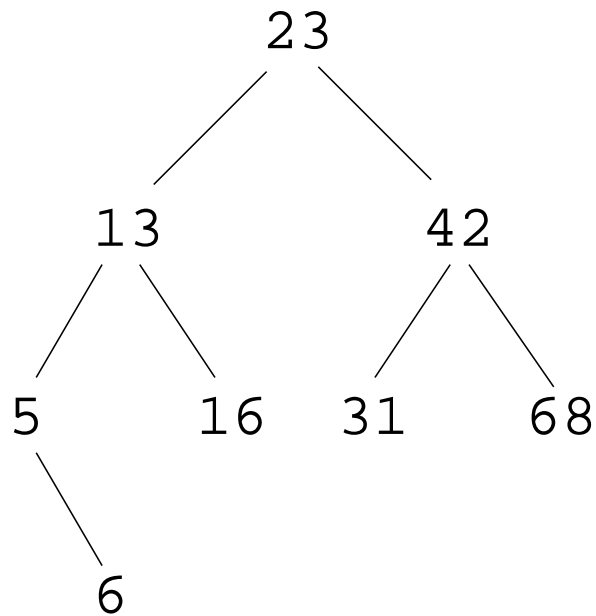
Flattening (or ‘collapsing’) a tree is the reverse of building one. When we collapse a tree we get a list of its keys.

- An empty tree has no keys.
- The keys from a flattened node are the ones from the left-hand subtree, the node, and the right-hand subtree.

```
treeFlatten :: Tree a -> [a]
treeFlatten Null      = []
treeFlatten (Node x left right)
    = treeFlatten left ++ [x] ++ treeFlatten right
```

When we flatten a tree, we *walk* over it from left to right.

The invariant ensures that the keys on the left are always less than the keys on the right.



[5, 6, 13, 16, 23, 31, 42, 68]

Sorting

We just saw that flattening a tree always returns a list of the keys *in order*. If we wanted to sort some list of elements we could make use of this observation.

A *tree sort* is a method of sorting data by building a tree and then flattening it.

```
treeSort :: Ord a => [a] -> [a]
treeSort xx = treeFlatten (treeBuild Null xx)
```

(Tree.hs)

Representing tables as trees

Suppose we wanted to use a tree to represent a table from keys to values. We did a similar thing a few lectures ago with lists of tuples.

```
lookup :: Eq k => k -> [(k, v)] -> Maybe v
```

```
list = [(4, "Tangerine"), (5, "Orange"), (6, "Grape")]
```

```
lookup 5 list ==> Just "Orange"
```

We *could* add a value to each node along with its key.

The `Tree` type would then be parameterised over both the type of the key and the type of the value.

```
data TreeKV k v
  = Null
  | Node k v (TreeKV k v) (TreeKV k v)
```

```
lookupKV :: Ord k => k -> Tree k v -> Maybe v
```

This would work, but if we change the data type of our tree then we have to rewrite *all* of our existing tree functions. (Try it, good practice!)

Perhaps we could place a tuple at each node of the tree, instead of just a key.

```
data Tree a
  = Null
  | Node a (Tree a) (Tree a)
```

```
lookupT :: Ord k => k -> Tree (k, v) -> Maybe v
```

We would then have to rewrite `treeLookup` and `treeInsert`.

Other functions such as `treeSize` which don't inspect the data stored at each node would not have to change.

The Type-class way

There is a way to store keys *and* values at each node of the tree and still use our original tree functions. We declare a new type, a `KeyValue` pair and define ordering and equality to be functions on *keys only*.

```
data KeyValue k v = KV k v
```

```
instance Eq k => Eq (KeyValue k v) where  
  (KV k1 v1) == (KV k2 v2)      = k1 == k2
```

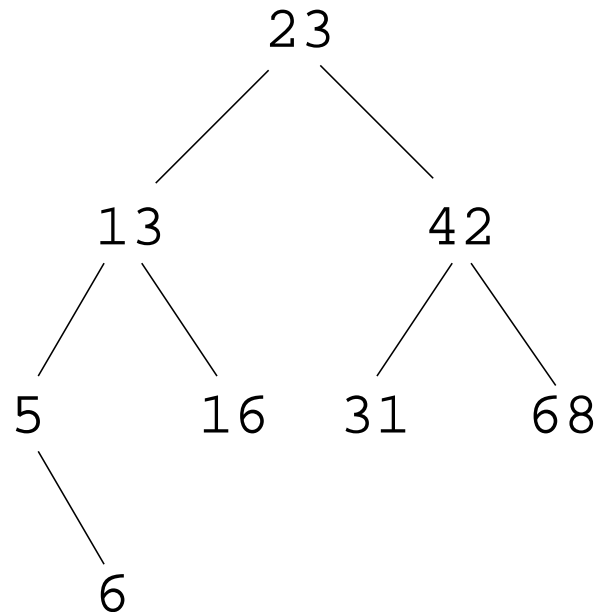
```
instance Ord k => Ord (KeyValue k v) where  
  compare (KV k1 v1) (KV k2 v2) = compare k1 k2
```

(TreeKV.hs)

Why trees?

The structure of a binary search tree reflects the ordering of the keys. We can use this ordering to help find a particular key.

For plain lists we must examine, on average, 50% of the elements in that list. Imagine trying to find a name in a phone book if the names weren't ordered!



Problems with simple binary trees

The final shape of a simple binary search tree depends on the order that we insert the keys.

Simple binary search trees do not perform well when we insert keys in their natural ordering. More advanced tree structures (B-Trees, AVL Trees, Red/Black Trees etc) attempt to keep the tree *balanced* to avoid this problem.

