Tree-Structured Data

Joseph C Osborn

April 28, 2025

Outline

Lists vs Strings

We've seen that in Haskell, a string is a list of characters.
Strings, since they specify their inner type (and the inner type is from the finite set of characters), are "flat" data structures.
Lists, on the other hand, can contain lists of another type inside of them, and they have a "tree structure":

```
[[[1, 2, 3], [4, 5]], [[6, 7], [], []]]
```

- Outer list
  - First inner list
    - `[1,2,3]`
    - `[4,5]`
  - Second inner list
    - `[6,7]`
    - `[]`
    - `[]`

Trees vs Lists

- We can certainly make trees out of lists, if we know the number of levels in advance.
  - e.g. `[[[Int]]]` in the last example
- But what if we want a tree-like structure that can go arbitrarily deep?

Compare:

```
data List a = Nil | Cons a
data Tree a = Branch a [Tree a]
```

Define an example of a nested list and a corresponding tree, and compare their types.

Binary Trees in Haskell

`Branch [Tree a]` is a very general constructor, for trees where branches can have arbitrary numbers of child branches and leafs. Sometimes, we want some assurances that we're working with a tree of a more predictable shape.
In these cases we use a type like this:

```
data BTree a = Empty | Branch a (BTree a) (BTree a)
```

Define an example BTree using this constructor.

Tree to List

```
data BTree a = Empty | Branch a (BTree a) (BTree a)
```

Let's write a function to convert a BTree to a list:

```
traverse Empty = []
traverse Branch a (BTree l) (BTree r) =
  traverse l ++ [a] ++ traverse r
```

Find an element in a tree

```
data BTree a = Empty | Branch a (BTree a) (BTree a)
```

Let's implement a search function that goes through the whole tree to find an element satisfying a predicate.
Don't use `traverse`!

```
find :: (elt -> Bool) -> BTree elt -> Maybe elt
find _ Empty = Nothing
find f (Branch a left right) = ...
```

Find an element in a tree

```haskell
data BTree a = Empty | Branch a (BTree a) (BTree a)

find :: (elt -> Bool) -> BTree elt -> Maybe elt
find _ Empty = Nothing
find f (Branch a left right)
  | f a = Just a
  | otherwise = case find f left of
      Just x -> Just x
      Nothing -> find f right
```

Insert an element in a tree

```
data BTree a = Empty | Branch a (BTree a) (BTree a)
```

Let's finish this function, which adds `elt` at the leftmost spot.

```
insert :: elt -> BTree elt -> BTree elt
insert x Empty = ...
insert x (Branch _ left right) = ...
```

Insert an element in a tree

```
data BTree a = Empty | Branch a (BTree a) (BTree a)

insert :: elt -> BTree elt -> BTree elt
insert x Empty = Branch x Empty Empty
insert x (Branch _ left _right) = insert x left
```

Tree Surgery

```
data BTree a = Empty | Branch a (BTree a) (BTree a)
```

How about this one, which removes and returns the leftmost value from the tree?

```
remove_leftmost :: BTree elt -> (BTree elt, Maybe elt)
remove_leftmost Empty = (Empty, Nothing)
remove_leftmost (Branch a left right) =
  case remove_leftmost left of
    (Empty, Nothing) ->  -- This is the leftmost item!
    (tree, Just x) ->  -- Left child had some item
```

Tree Surgery

```
data BTree a = Empty | Branch a (BTree a) (BTree a)
```

How about this one, which removes and returns the leftmost value from the tree?

```
remove_leftmost :: BTree elt -> (BTree elt, Maybe elt)
remove_leftmost Empty = (Empty, Nothing)
remove_leftmost (Branch a left right) =
  case remove_leftmost left of
    (Empty, Nothing) -> (right, Just a) -- This is the leftm
    (tree, Just x) -> (Branch a tree right, Just x) -- Left
```

Remove an element from a tree

```
data BTree a = Empty | Branch a (BTree a) (BTree a)
```

How about this one, which removes all the elements equivalent to x?

```
remove :: (Eq elt) => elt -> BTree elt -> BTree elt
remove _ Empty = Empty
remove x (Branch a left right) =
  | a == x = -- ? what to do here?
  | otherwise = Branch a (remove x left) (remove x right)
```

Remove an element from a tree

```
data BTree a = Empty | Branch a (BTree a) (BTree a)
```

How about this one, which removes all the elements equivalent to `x`?
Hint: We want to recursively remove on both sides, but then what do we put in place of a removed element?
May as well use that `remove_leftmost` function!

```
remove :: (Eq elt) => elt -> BTree elt -> BTree elt
remove _ Empty = Empty
remove x (Branch a left right) =
  | a == x = case ((remove x left), (remove x right)) of
       (Empty, r) -> r
       (l, r) -> -- now what?
  | otherwise = Branch a (remove x left) (remove x right)
```

Remove an element from a tree

```
data BTree a = Empty | Branch a (BTree a) (BTree a)
```

How about this one, which removes all the elements equivalent to x?
Hint: We want to recursively remove on both sides, but then what do we put in place of a removed element?
May as well use that remove_leftmost function!

```
remove :: (Eq elt) => elt -> BTree elt -> BTree elt
remove _ Empty = Empty
remove x (Branch a left right) =
  | a == x = case ((remove x left), (remove x right)) of
      (Empty, r) -> r
      (l, r) -> case remove_leftmost l of
        (lrem, Just swap) -> Branch swap lrem r
        (lrem, Nothing) -> error "No leftmost element of emp
  | otherwise = Branch a (remove x left) (remove x right)
```

Functor and fmap

Finally, we'll revisit `fmap` from last time. Just like we can map through Maybe, we can map through a BTree:

```
data BTree a = Empty | Branch a (BTree a) (BTree a)
```

```
fmap :: (a -> b) -> BTree a -> BTree b
```

Give it a try! This one isn't as hard as the last one.

Functor and fmap

Finally, we'll revisit `fmap` from last time. Just like we can map through Maybe, we can map through a BTree:

```
data BTree a = Empty | Branch a (BTree a) (BTree a)

fmap :: (a -> b) -> BTree a -> BTree b
fmap _f Empty = Empty
fmap f (Branch a l r) = Branch (f a) (fmap f l) (fmap f r)
```

Binary Search Trees

Let's refine our binary trees a little more. We'll say a subset of BTrees are "binary search trees".
That means they're used for searching, they can only hold orderable things, and they meet this property:
For all trees: an empty tree has the BST property; and a branch has the property if all the elements on the left branch are smaller than its value, and all the elements on the right branch are bigger than its value, and both branches are also BSTs.
For this lecture we'll say that BSTs contain no duplicates.
BSTs are really useful for quickly finding if an object is in a collection of objects, and for maintaining a record of the minimum and maximum objects in the set.

Find an element in a BST

```
bst_member :: (Ord x) => x -> BTree x -> Bool
bst_member _ Empty = False
bst_member x (Branch a l r)
  | x == a = True
  | x < a = bst_member x l
  | otherwise = bst_member x r
```

Min, max

Min is a "find leftmost" function, max is a "find rightmost" function, and "remove min" is just our `remove_leftmost` function from before!
Give min and max a try!

Insert an element in a BST

```
bst_insert :: (Ord x) => x -> BTree x -> BTree x
bst_insert a Empty = Branch a Empty Empty
bst_insert x (Branch a l r)
  | x == a = -- ?
  | x < a = -- ??
  | otherwise = -- ???
```

Insert an element in a BST

```
bst_insert :: (Ord x) => x -> BTree x -> BTree x
bst_insert a Empty = Branch a Empty Empty
bst_insert x (Branch a l r)
  | x == a = Branch a l r
  | x < a = Branch a (bst_insert x l) r
  | otherwise = Branch a l (bst_insert x r)
```

Induction on a BST

BTrees are inductively defined, so we can do induction on them too:
For any BTree b, P(b). Let b be given; by induction on b.

- (b = Empty). P(Empty)
- (b = Branch a l r). IH_l: P(l). IH_r: P(r).
  - WTP: P(Branch a l r).

Proof: $\text{insert}_{\text{preserves bst}}$

for any element x and BTree b, if b is a BST then insert x b is a BST.
(Recall: "Is a BST" means "a is bigger than any element in left and smaller than any element in right")

- By induction on b.
- Empty: insert x Empty is just Branch x Empty Empty which is trivially a BST.
- Branch a l r: IH1: "If l is a BST then insert x l is a BST". IH2: "If r is a BST then insert x r is a BST".
  - Assume b is a BST.
  - By cases on x vis-a-vis a:
  - x=a. b is a BST, and insert x b = b in this case.
  - x<a. r is a BST, l is a BST, and by IH1 if l is a BST then insert x l is a BST. We know x < a so b is a BST.
  - x>a. Likewise, but with r/IH2 instead of l/IH1.

Remove an element from a BST

```
bst_remove :: (Ord x) => x -> BTree x -> BTree x
bst_remove _ Empty = Empty
bst_remove x (Branch a l r)
  | x == a = -- use remove_leftmost and bst_insert!
  | x < a = Branch a (bst_remove x l) r
  | otherwise = Branch a l (bst_remove x r)
```

Proof: $\text{remove}_{\text{preserves bst}}$

for any element x and BTree b, if b is a BST then remove x b is a BST.
This is a very similar proof to the last one, but depends on the previous proof.