Introduction to Haskell Programming

Data structures – Binary search trees

S P Suresh

Chennai Mathematical Institute

Haskell @ CMI

Creating a binary tree

- Create a binary tree from a list

Creating a binary tree

- Create a binary tree from a list
- As **balanced** as possible

Creating a binary tree

- Create a binary tree from a list
- As **balanced** as possible
- **Strategy**:

Creating a binary tree

- Create a binary tree from a list
- As **balanced** as possible
- **Strategy**:
    - Split the list in two halves

Creating a binary tree

- Create a binary tree from a list
- As **balanced** as possible
- **Strategy**:
  - Split the list in two halves
  - Recursively create a binary tree from each half

Creating a binary tree

- Create a binary tree from a list
- As **balanced** as possible
- **Strategy**:
  - Split the list in two halves
  - Recursively create a binary tree from each half
  - Join them together

Creating a binary tree

- Creating a balanced tree from a list

```
createTree :: [a] → BTree a
createTree [] = Nil
createTree xs = Node tl y tr where
    m               = length xs `div` 2
    (tl, y:ys)      = (createTree (take m xs), drop m xs)
    tr              = createTree ys

levels (createTree [0..14]) = [7,3,11,1,5,9,13,0,2,4,6,8,10,12,14]
height (createTree [0..14]) = 4
```

Creating a binary tree

- `length`, `take` and `drop` take linear time

Creating a binary tree

- `length`, `take` and `drop` take linear time
- $T(n) = 2T(n/2) + O(n)$ and hence $T(n) = O(n \log n)$

Creating a binary tree

- `length`, `take` and `drop` take linear time
- $T(n) = 2T(n/2) + O(n)$ and hence $T(n) = O(n \log n)$
- Can we improve?

Creating a binary tree

- `length`, `take` and `drop` take linear time
- $T(n) = 2T(n/2) + O(n)$ and hence $T(n) = O(n \log n)$
- Can we improve?
  - Culprit is the repeated use of `length`, `take` and `drop`

Creating a binary tree

- `length`, `take` and `drop` take linear time
- $T(n) = 2T(n/2) + O(n)$ and hence $T(n) = O(n \log n)$
- Can we improve?
  - Culprit is the repeated use of `length`, `take` and `drop`
  - Problem remains even if we use `splitAt`

Creating a binary tree

- `length`, `take` and `drop` take linear time
- $T(n) = 2T(n/2) + O(n)$ and hence $T(n) = O(n \log n)$
- Can we improve?
  - Culprit is the repeated use of `length`, `take` and `drop`
  - Problem remains even if we use `splitAt`
  - Can we avoid that?

Deriving a more efficient `createTree`

- Avoid `length` in recursive calls

Deriving a more efficient `createTree`

- Avoid **`length`** in recursive calls
- New version of `createTree`

```
createTree :: [a] → BTree a
createTree xs        = ct (length xs) xs where
    ct 0 xs          = Nil
    ct n xs          = Node tl y tr where
        m            = n `div` 2
        (tl, y:ys)   = (ct m (take m xs), drop m xs)
        tr           = ct (n-m-1) ys
```

Deriving a more efficient `createTree`

- Avoid **`length`** in recursive calls
- New version of `createTree`

```
createTree :: [a] → BTree a
createTree xs        = ct (length xs) xs where
    ct 0 xs          = Nil
    ct n xs          = Node tl y tr where
        m            = n `div` 2
        (tl, y:ys)   = (ct m (take m xs), drop m xs)
        tr           = ct (n-m-1) ys
```

- Let `go n xs = (ct n (take n xs), drop n xs)`

Deriving a more efficient `createTree`

- Avoid **`length`** in recursive calls
- New version of `createTree`

```
createTree :: [a] → BTree a
createTree xs        = ct (length xs) xs where
    ct 0 xs          = Nil
    ct n xs          = Node tl y tr where
        m            = n `div` 2
        (tl, y:ys)   = (ct m (take m xs), drop m xs)
        tr           = ct (n-m-1) ys
```

- Let `go n xs = (ct n (take n xs), drop n xs)`
- Then `createTree xs = fst (go (length xs) xs)`

Deriving a more efficient `createTree`

- Calculating a direct definition of `go`, assuming `0 < n ⩽ length xs`

```
go n xs
= (ct n (take n xs), drop n xs)
= (Node tl y tr,     drop n xs) where
    m            = n `div` 2
    (tl, y:rest) = (ct m (take m (take n xs)), drop m (take n xs))
    tr           = ct (n-m-1) rest
```

Deriving a more efficient `createTree`

- If `m < n`, and `drop m xs = y:ys`, then ...

```
take m (take n xs) = take m xs

y:rest  = drop m (take n xs) = take (n-m) (drop m xs)
        = take (n-m) (y:ys) = y: take (n-m-1) ys
So rest = take (n-m-1) ys

drop n xs   = drop (n-m) (drop m xs)
            = drop (n-m) (y:ys)
            = drop (n-m-1) ys
```

Deriving a more efficient `createTree`

- Substituting appropriately ...

```
go n xs

= (Node tl y tr, drop (n-m-1) ys) where
    m           = n `div` 2
    (tl, y:ys)  = (ct m (take m xs), drop m xs)
    tr          = ct (n-m-1) (take (n-m-1) ys)

= (Node tl y tr, zs) where
    m           = n `div` 2
    (tl, y:ys)  = go m xs
    (tr, zs)    = go (n-m-1) ys
```

Creating a binary tree

- Creating a tree in linear time

```
createTree :: [a] → BTree a
createTree l = fst (go (length l) l) where
    go :: Int → [a] → (BTree a, [a])
    go 0 xs = (Nil, xs)
    go n xs = (Node tl y tr, zs) where
        m          = n `div` 2
        (tl, y:ys) = go m xs
        (tr, zs)   = go (n-m-1) ys
```

Creating a binary tree

- Creating a tree in linear time

```
createTree :: [a] → BTree a
createTree l = fst (go (length l) l) where
    go :: Int → [a] → (BTree a, [a])
    go 0 xs = (Nil, xs)
    go n xs = (Node tl y tr, zs) where
        m          = n `div` 2
        (tl, y:ys) = go m xs
        (tr, zs)   = go (n-m-1) ys
```

- $T(n) = 2T(n/2) + c$ and hence $T(n) = (2n - 1)c$

The `Set` ADT

- Maintain a collection of distinct elements and support the following operations

The `Set` ADT

- Maintain a collection of distinct elements and support the following operations
  - `insertInto` – insert a given value into the set

The `Set` ADT

- Maintain a collection of distinct elements and support the following operations
  - `insertInto` – insert a given value into the set
  - `deleteFrom` – delete a given value from the set

The `Set` ADT

- Maintain a collection of distinct elements and support the following operations
  - `insertInto` – insert a given value into the set
  - `deleteFrom` – delete a given value from the set
  - `search` – check whether a given value is an element of the set

The `Set` ADT

- Maintain a collection of distinct elements and support the following operations
  - `insertInto` – insert a given value into the set
  - `deleteFrom` – delete a given value from the set
  - `search` – check whether a given value is an element of the set
  - `fromList` – turn a list into a set

The `Set` ADT

- Maintain a collection of distinct elements and support the following operations
  - `insertInto` – insert a given value into the set
  - `deleteFrom` – delete a given value from the set
  - `search` – check whether a given value is an element of the set
  - `fromList` – turn a list into a set
- Straightforward implementation

```
module Set(Set, insertInto, deleteFrom,
               search, fromList) where
data Set a = Set [a]
```

The Set ADT

```
search :: Eq a ⇒ a → Set a → Bool
search x (Set xs) = x `elem` xs

insertInto :: Eq a ⇒ a → Set a → Set a
insertInto x (Set xs) = Set (if x `elem` xs
                                then xs else x:xs)

deleteFrom :: Eq a ⇒ a → Set a → Set a
deleteFrom x (Set xs) = Set (filter (/=x) xs)

fromList :: [a] → Set a
fromList = Set . nub
```

`Set`: complexity

- `search` takes $O(n)$ time

`Set`: complexity

- `search` takes $O(n)$ time
- `insertInto` takes $O(n)$ time

`Set`: complexity

- `search` takes $O(n)$ time
- `insertInto` takes $O(n)$ time
- `deleteFrom` takes $O(n)$ time

`Set`: complexity

- `search` takes $O(n)$ time
- `insertInto` takes $O(n)$ time
- `deleteFrom` takes $O(n)$ time
- A sequence of n operations takes $O(n^2)$ time

`Set`: complexity

- `search` takes $O(n)$ time
- `insertInto` takes $O(n)$ time
- `deleteFrom` takes $O(n)$ time
- A sequence of n operations takes $O(n^2)$ time
- We can do better if the elements admit an order

Binary search trees

- A **binary search tree** is another way of implementing the `Set` ADT

Binary search trees

- A **binary search tree** is another way of implementing the `Set` ADT
- A binary search tree is a binary tree

Binary search trees

- A **binary search tree** is another way of implementing the `Set` ADT
- A binary search tree is a binary tree
- Stores values of type `a` such that **`Ord`** `a`

Binary search trees

- A **binary search tree** is another way of implementing the `Set` ADT
- A binary search tree is a binary tree
- Stores values of type `a` such that `Ord a`
- In a binary search tree:

Binary search trees

- A **binary search tree** is another way of implementing the `Set` ADT
- A binary search tree is a binary tree
- Stores values of type `a` such that `Ord a`
- In a binary search tree:
  - values in the left subtree are smaller than the root

Binary search trees

- A **binary search tree** is another way of implementing the `Set` ADT
- A binary search tree is a binary tree
- Stores values of type `a` such that `Ord a`
- In a binary search tree:
  - values in the left subtree are smaller than the root
  - values in the right subtree are larger than the root

Binary search trees

- The binary search tree

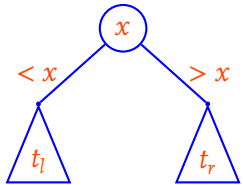
```
Node tl x tr
```

Binary search trees

- The binary search tree

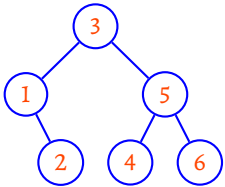
```
Node tl x tr
```

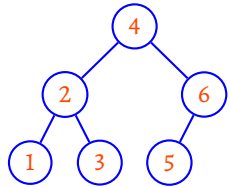
- Pictorially ...



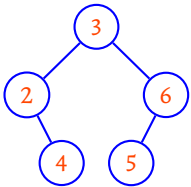
Binary search trees

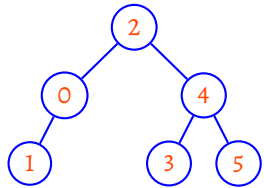
- Examples





- Non-examples





BST

- Binary search trees in Haskell:

```
data BST a = Nil | Node (BST a) a (BST a)
    deriving Eq
```

BST

- Binary search trees in Haskell:

```
data BST a = Nil | Node (BST a) a (BST a)
    deriving Eq
```

- The empty tree:

```
emptyBST :: BST a
emptyBST = Nil
```

BST

- Binary search trees in Haskell:

```
data BST a = Nil | Node (BST a) a (BST a)
    deriving Eq
```

- The empty tree:

```
emptyBST :: BST a
emptyBST = Nil
```

- Is a tree empty?

```
isEmpty :: BST a → Bool
isEmpty Nil = True
isEmpty _   = False
```

Is it a search tree?

- Just naming it BST does not make it a binary search tree

```
isBST :: Ord a ⇒ BST a → Bool
isBST = isSorted . inorder where
    isSorted [] = True
    isSorted xs = and (zipWith (⩽) xs (tail xs))
```

Is it a search tree?

- Just naming it `BST` does not make it a binary search tree

```
isBST :: Ord a ⇒ BST a → Bool
isBST = isSorted . inorder where
    isSorted [] = True
    isSorted xs = and (zipWith (⩽) xs (tail xs))
```

- Recall `inorder`

```
inorder :: BST a → [a]
inorder t                  = go t [] where
    go Nil xs              = xs
    go (Node tl x tr) xs   = go tl (x:go tr xs)
```

Is it a search tree?

- Just naming it `BST` does not make it a binary search tree

```
isBST :: Ord a ⇒ BST a → Bool
isBST = isSorted . inorder where
    isSorted [] = True
    isSorted xs = and (zipWith (⩽) xs (tail xs))
```

- Recall `inorder`

```
inorder :: BST a → [a]
inorder t                  = go t [] where
    go Nil xs              = xs
    go (Node tl x tr) xs   = go tl (x:go tr xs)
```

- If `t` is a BST, `inorder t` is a sorted list

Minimum in a tree

- `mint` gives the minimum value in a non-empty tree:

```
mint :: Ord a ⇒ BST a → a
mint (Node tl x tr)  = min x (min y z)
where   y = if isEmpty tl then x else mint tl
        z = if isEmpty tr then x else mint tr
```

Minimum in a tree

- `mint` gives the minimum value in a non-empty tree:

```
mint :: Ord a ⇒ BST a → a
mint (Node tl x tr)  = min x (min y z)
where   y = if isEmpty tl then x else mint tl
        z = if isEmpty tr then x else mint tr
```

- `maxt` is similar (uses **`max`** instead of **`min`**)

Searching in a tree

- Searching for value v in a search tree

Searching in a tree

- Searching for value v in a search tree
- If the tree is empty, report `False`

Searching in a tree

- Searching for value `v` in a search tree
- If the tree is empty, report `False`
- If the tree is nonempty

Searching in a tree

- Searching for value `v` in a search tree
- If the tree is empty, report `False`
- If the tree is nonempty
  - If `v` is the value at the root, report `True`

Searching in a tree

- Searching for value v in a search tree
- If the tree is empty, report `False`
- If the tree is nonempty
  - If v is the value at the root, report `True`
  - If v is smaller than the value at the root, search in left subtree

Searching in a tree

- Searching for value v in a search tree
- If the tree is empty, report `False`
- If the tree is nonempty
  - If v is the value at the root, report `True`
  - If v is smaller than the value at the root, search in left subtree
  - If v is larger than the value at the root, search in right subtree

Searching in a tree

- Haskell code transcribes the search strategy:

```
searchBST :: Ord a ⇒ a → BST a → Bool
searchBST _ Nil             = False
searchBST v (Node tl x tr)  = case compare v x of
    LT → searchBST v tl
    EQ → True
    GT → searchBST v tr
```

Searching in a tree

- Haskell code transcribes the search strategy:

```
searchBST :: Ord a ⇒ a → BST a → Bool
searchBST _ Nil             = False
searchBST v (Node tl x tr)  = case compare v x of
    LT → searchBST v tl
    EQ → True
    GT → searchBST v tr
```

- **Worst case running time**: Height of the tree

Inserting in a tree

- Inserting an element into a tree follows a similar strategy:

```
insertBST :: Ord a ⇒ a → BST a → BST a
insertBST v Nil = Node Nil v Nil
insertBST v t@(Node tl x tr) = case compare v x of
    LT → Node (insertBST v tl) x tr
    EQ → t
    GT → Node tl x (insertBST v tr)
```

Inserting in a tree

- Inserting an element into a tree follows a similar strategy:

```
insertBST :: Ord a ⇒ a → BST a → BST a
insertBST v Nil = Node Nil v Nil
insertBST v t@(Node tl x tr) = case compare v x of
    LT → Node (insertBST v tl) x tr
    EQ → t
    GT → Node tl x (insertBST v tr)
```

- **Worst case running time**: Height of the tree

Inserting in a tree

- Inserting an element into a tree follows a similar strategy:

```
insertBST :: Ord a ⇒ a → BST a → BST a
insertBST v Nil = Node Nil v Nil
insertBST v t@(Node tl x tr) = case compare v x of
    LT → Node (insertBST v tl) x tr
    EQ → t
    GT → Node tl x (insertBST v tr)
```

- **Worst case running time**: Height of the tree
- `@` allows one to refer to a data value by a name

Inserting in a tree

- Inserting an element into a tree follows a similar strategy:

```
insertBST :: Ord a ⇒ a → BST a → BST a
insertBST v Nil = Node Nil v Nil
insertBST v t@(Node tl x tr) = case compare v x of
    LT → Node (insertBST v tl) x tr
    EQ → t
    GT → Node tl x (insertBST v tr)
```

- **Worst case running time**: Height of the tree
- `@` allows one to refer to a data value by a name
- From list to binary search tree ...and back to list

```
fromList = foldr insertBST emptyBST
treeSort = inorder . fromList
```

Deleting from a tree

- We first tackle a simpler problem

Deleting from a tree

- We first tackle a simpler problem
  - Delete the maximum value

Deleting from a tree

- We first tackle a simpler problem
  - Delete the maximum value
  - and return the value as well as the modified tree

Deleting from a tree

- We first tackle a simpler problem
  - Delete the maximum value
  - and return the value as well as the modified tree
- Maximum is the rightmost node:

```
deleteMax :: Ord a ⇒ BST a → (a, BST a)
deleteMax (Node tl x Nil) = (x, tl)
deleteMax (Node tl x tr) = let (y, ty) = deleteMax tr
                           in (y, Node tl x ty)
```

Deleting from a tree

- We first tackle a simpler problem
  - Delete the maximum value
  - and return the value as well as the modified tree
- Maximum is the rightmost node:

```
deleteMax :: Ord a ⇒ BST a → (a, BST a)
deleteMax (Node tl x Nil) = (x, tl)
deleteMax (Node tl x tr) = let (y, ty) = deleteMax tr
                           in (y, Node tl x ty)
```

- **Worst case running time**: Height of the tree

Deleting from a tree

- Deleting a value v from a search tree

Deleting from a tree

- Deleting a value v from a search tree
- If the tree is empty, nothing to do

Deleting from a tree

- Deleting a value v from a search tree
- If the tree is empty, nothing to do
- If the tree is nonempty

Deleting from a tree

- Deleting a value v from a search tree
- If the tree is empty, nothing to do
- If the tree is nonempty
  - If v is smaller than the value at the root, delete from left subtree

Deleting from a tree

- Deleting a value v from a search tree
- If the tree is empty, nothing to do
- If the tree is nonempty
  - If v is smaller than the value at the root, delete from left subtree
  - If v is larger than the value at the root, delete from right subtree

Deleting from a tree

- Deleting a value v from a search tree
- If the tree is empty, nothing to do
- If the tree is nonempty
  - If v is smaller than the value at the root, delete from left subtree
  - If v is larger than the value at the root, delete from right subtree
  - If v is equal to the value at the root, remove root and

Deleting from a tree

- Deleting a value v from a search tree
- If the tree is empty, nothing to do
- If the tree is nonempty
  - If v is smaller than the value at the root, delete from left subtree
  - If v is larger than the value at the root, delete from right subtree
  - If v is equal to the value at the root, remove root and
    - If left subtree is empty, just slide the right subtree up a level

Deleting from a tree

- Deleting a value v from a search tree
- If the tree is empty, nothing to do
- If the tree is nonempty
  - If v is smaller than the value at the root, delete from left subtree
  - If v is larger than the value at the root, delete from right subtree
  - If v is equal to the value at the root, remove root and
    - If left subtree is empty, just slide the right subtree up a level
    - If left subtree is non-empty, replace root by maximum element of left subtree

Deleting from a tree

- Deleting a value v from a search tree
- If the tree is empty, nothing to do
- If the tree is nonempty
  - If v is smaller than the value at the root, delete from left subtree
  - If v is larger than the value at the root, delete from right subtree
  - If v is equal to the value at the root, remove root and
    - If left subtree is empty, just slide the right subtree up a level
    - If left subtree is non-empty, replace root by maximum element of left subtree
    - Search tree property is preserved

Deleting from a tree

- Code for delete:

```
deleteBST :: Ord a ⇒ a → BST a → BST a
deleteBST v Nil = Nil
deleteBST v (Node tl x tr)
| v < x          = Node (deleteBST v tl) x tr
| v > x          = Node tl x (deleteBST v tr)
| v == x         = if isEmpty tl then tr
                                 else Node ty y tr
    where (y, ty) = deleteMax tl
```

Deleting from a tree

- Code for delete:

```
deleteBST :: Ord a ⇒ a → BST a → BST a
deleteBST v Nil = Nil
deleteBST v (Node tl x tr)
| v < x          = Node (deleteBST v tl) x tr
| v > x          = Node tl x (deleteBST v tr)
| v == x         = if isEmpty tl then tr
                                 else Node ty y tr
    where (y, ty) = deleteMax tl
```

- **Worst case running time**: Height of the tree

Balance

- Complexity of tree operations depends on the **height** of the tree

Balance

- Complexity of tree operations depends on the **height** of the tree
- In general, a tree might not be balanced

Balance

- Complexity of tree operations depends on the **height** of the tree
- In general, a tree might not be balanced
- Inserting in ascending or descending order results in highly skewed trees