Trees

- **Today's Topics**
  - Trees
  - Kinds of trees - branching factor
  - functions over trees
  - patterns of recursion - the fold for trees
  - Arithmetic expressions
  - Infinite trees

Trees

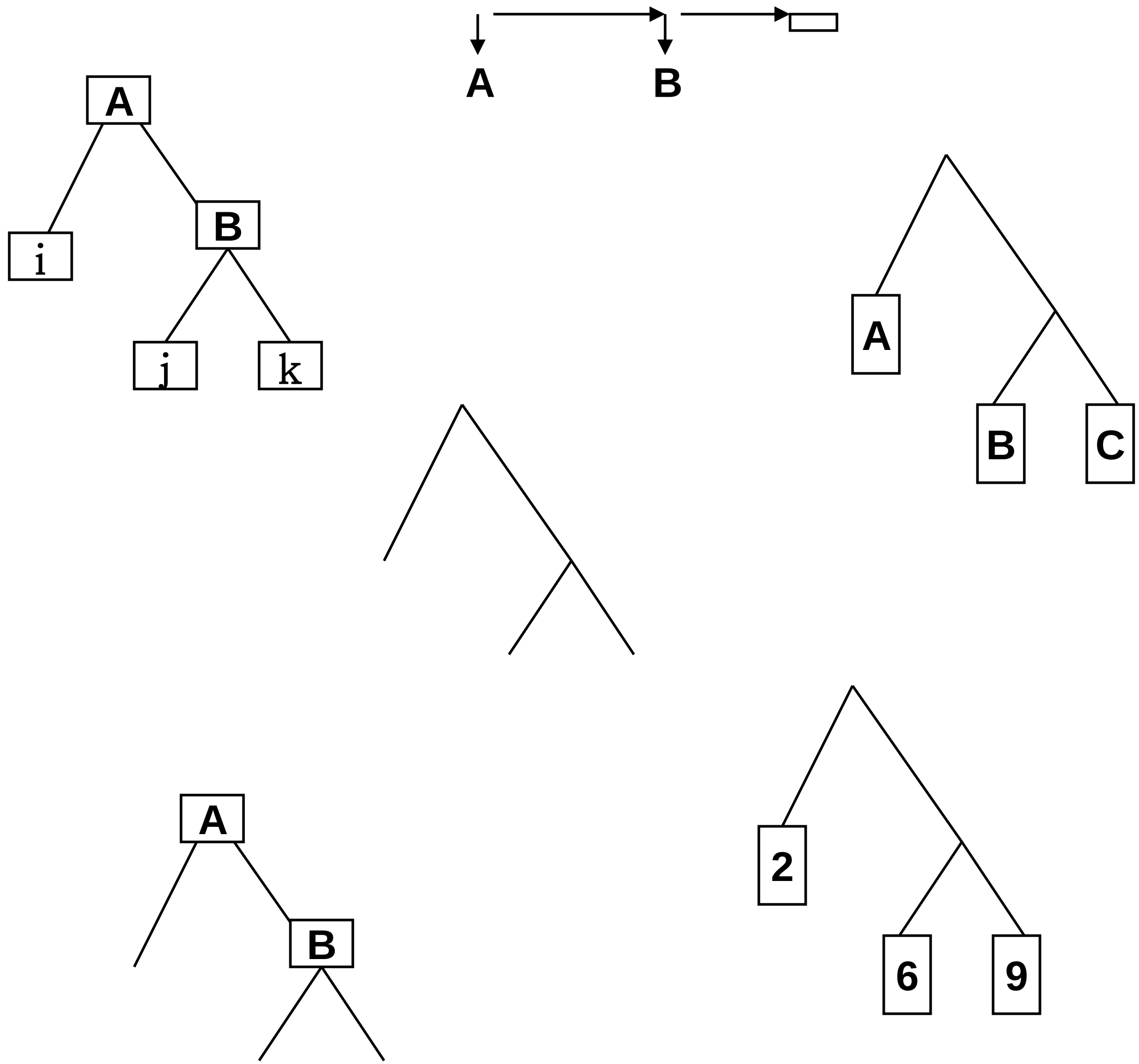
- **Trees are important data structures in computer science**
- **Trees have interesting properties**
  - They usually are finite, but unbounded in size
  - Sometimes contain other types inside
  - Sometimes the things contained are polymorphic
  - differing “branching factors”
  - different kinds of leaf and branching nodes
- **Lots of interesting things can be modeled by trees**
  - lists (linear branching)
  - arithmetic expressions
  - parse trees (for languages)
- **In a lazy language it is possible to have infinite trees**

Examples

```
data List a = Nil | MkList a (List a)

data Tree a = Leaf a | Branch  (Tree a) (Tree a)

data IntegerTree = IntLeaf Integer
                 | IntBranch IntegerTree IntegerTree

data SimpleTree  = SLeaf
                 | SBranch SimpleTree SimpleTree

data InternalTree a = ILeaf
                    | IBranch a (InternalTree a)
                                (InternalTree a)

data FancyTree a b =  FLeaf a
                   | FBranch b (FancyTree a b)
                               (FancyTree a b)
```

Match up the trees

- **`IntegerTree`**
- **`Tree`**
- **`SimpleTree`**
- **`List`**
- **`InternalTree`**
- **`FancyTree`**



Functions on Trees

- **Transforming one kind of tree into another**

```
mapTree :: (a->b) -> Tree a -> Tree b
mapTree f (Leaf x)       = Leaf (f x)
mapTree f (Branch t1 t2) = Branch (mapTree f t1)
                                  (mapTree f t2)
```

- **Collecting the items in a tree**

```
fringe               :: Tree a -> [a]
fringe (Leaf x)       = [x]
fringe (Branch t1 t2) = fringe t1 ++ fringe t2
```

- **what kind of information is lost using `fringe?`**

More functions

```
treeSize                    :: Tree a -> Integer
treeSize (Leaf x)            = 1
treeSize (Branch t1 t2) = treeSize t1 + treeSize t2


treeHeight                :: Tree a -> Integer
treeHeight (Leaf x)          = 0
treeHeight (Branch t1 t2) = 1 + max (treeHeight t1)
                                    (treeHeight t2)
```

Capture the pattern of recursion

```
foldTree :: (a -> a -> a) -> (b -> a) -> Tree b -> a
foldTree bf lf (Leaf x)        = lf x
foldTree bf lf (Branch t1 t2) =
     bf (foldTree bf lf t1) (foldTree bf lf t2)

mapTree2 f = foldTree Branch (Leaf . f)

fringe2 = foldTree (++) (\ x -> [x])

treeSize2 = foldTree (+) (const 1)

treeHeight2 = foldTree (\ x y -> 1 + max x y)
                       (const 0)
```

Flattening Trees

```
data Tree a
   = Leaf a | Branch  (Tree a) (Tree a)

flatten :: Tree a -> [a]
flatten (Leaf x) = [x]
flatten (Branch x y) = flatten x ++ flatten y
```

What is the complexity of flattening a deep fully filled out tree?

Flattening with accumulating parameter

```
data Tree a
   = Leaf a | Branch  (Tree a) (Tree a)

flatten :: Tree a -> [a]
flatten t = flat t []

flat (Leaf x) xs = x:xs
flat (Branch a b) xs = flat a (flat b xs)
```

Arithmetic Expressons

```
data Expr2 = C2 Float
           | Add2 Expr2 Expr2
           | Sub2 Expr2 Expr2
           | Mul2 Expr2 Expr2
           | Div2 Expr2 Expr2
```

- **using infix constructor functions**

```
data Expr = C Float
          | Expr :+ Expr
          | Expr :- Expr
          | Expr :* Expr
          | Expr :/ Expr
```

Infix constructor operators start with a colon (:) , just like constructor functions start with an upper case letter

Example uses

```
e1 = (C 10 :+ (C 8 :/ C 2)) :* (C 7 :- C 4)

evaluate :: Expr -> Float
evaluate (C x) = x
evaluate (e1 :+ e2) = evaluate e1 + evaluate e2
evaluate (e1 :- e2) = evaluate e1 - evaluate e2
evaluate (e1 :* e2) = evaluate e1 * evaluate e2
evaluate (e1 :/ e2) = evaluate e1 / evaluate e2


Main> evaluate e1
42.0
```

Infinite Trees

- **Can we make an Expr tree that represents the infinite expression: 1 + 2 + 3 + 4 ….**

```
sumFromN n = C n :+ (sumFromN (n+1))
sumAll = sumFromN 1
```

```
add1 (C n) = C (n+1)
add1 (x :+ y) = add1 x :+ add1 y
add1 (x :- y) = add1 x :- add1 y
add1 (x :* y) = add1 x :* add1 y
add1 (x :/ y) = add1 x :/ add1 y

sumAll2 = C 1 :+ (add1 sumAll2)
```

Observing Infinite Trees

- **We can observe an infinite tree by printing a finite prefix of it. We need a `take`-like function for trees.**

```
showE 0 _ = "..."
showE n (C m) = show m
showE n (x :+ y) = "(" ++ (showE (n-1) x) ++ "+"
                       ++ (showE (n-1) y) ++ ")"
```

```
Main> showE 5 sumAll2
"(1.0+(2.0+(3.0+(4.0+(...+...)))))"

Main> showE 5 sumAll
"(1.0+(2.0+(3.0+(4.0+(...+...)))))"
```