

Informatics 1
Introduction to Computation
Lecture 16

Search in Trees

Sam Lindley
The University of Edinburgh

Binary trees

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

```
t :: Tree Int
```

```
t = Node (Node (Node Nil 4 Nil)
               2
               (Node Nil 5 Nil))
        1
        (Node (Node (Node Nil 8 Nil)
                    6
                    (Node Nil 9 Nil))
              3
              (Node Nil 7 Nil))
```

Binary trees

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

```
t :: Tree Int
```

```
t = Node (Node (Node Nil 4 Nil)
```

2

```
(Node Nil 5 Nil))
```

1

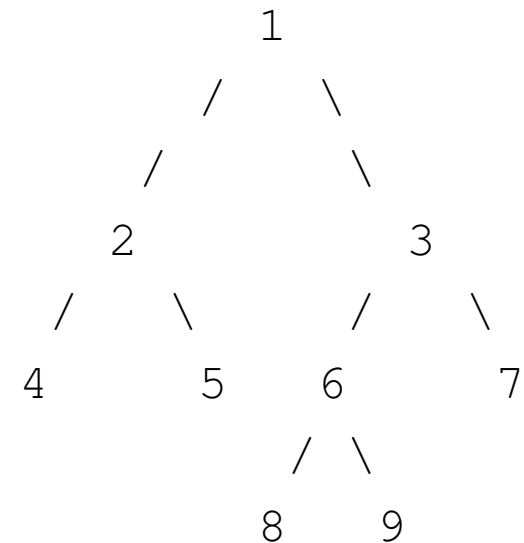
```
(Node (Node (Node Nil 8 Nil)
```

6

```
(Node Nil 9 Nil))
```

3

```
(Node Nil 7 Nil))
```



Binary trees

```
inf :: Tree Int
```

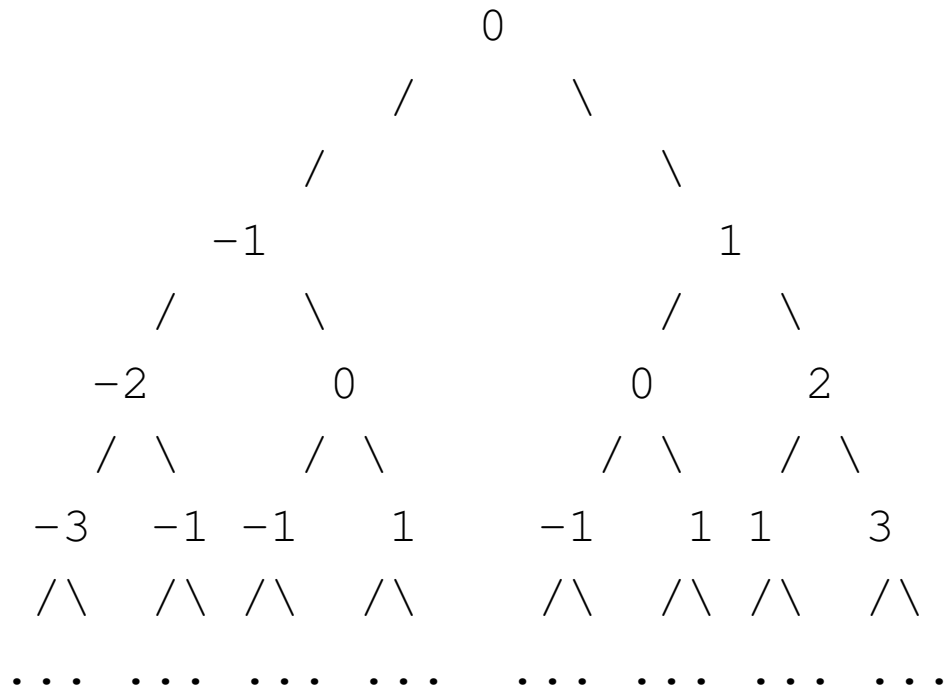
```
inf = inffrom 0
```

```
  where
```

```
    inffrom x = Node (inffrom (x-1)) x (inffrom (x+1))
```

Binary trees

```
inf :: Tree Int
inf = inffrom 0
  where
    inffrom x = Node (inffrom (x-1)) x (inffrom (x+1))
```



Depth-First Search

```
depthFirst :: Eq a => (a -> Bool) -> Tree a -> Maybe a
depthFirst p Nil = Nothing
depthFirst p (Node t1 x t2)
    | p x                = Just x
    | depthFirst p t1 == Nothing = depthFirst p t2
    | otherwise          = depthFirst p t1
```

Depth-First Search

```
depthFirst :: Eq a => (a -> Bool) -> Tree a -> Maybe a
depthFirst p Nil = Nothing
depthFirst p (Node t1 x t2)
    | p x                = Just x
    | depthFirst p t1 == Nothing = depthFirst p t2
    | otherwise          = depthFirst p t1
```

```
depthFirst (>4) 1
    / \
   2   ...
  / \
 4   5
```

```
= depthFirst (>4) 2
    / \
   4   5
```

```
= depthFirst (>4) 5    [since depthFirst (>4) 4 = Nothing]
```

```
= Just 5
```

Depth-First Search

```
depthFirst :: Eq a => (a -> Bool) -> Tree a -> Maybe a
depthFirst p Nil = Nothing
depthFirst p (Node t1 x t2)
    | p x                = Just x
    | depthFirst p t1 == Nothing = depthFirst p t2
    | otherwise          = depthFirst p t1
```

```
df_traverse :: Tree a -> [a]
df_traverse Nil = []
df_traverse (Node t1 x t2)
    = x : (df_traverse t1) ++ (df_traverse t2)
```

```
depthFirst' :: Eq a => (a -> Bool) -> Tree a -> Maybe a
depthFirst' p t
    = head( [Just x | x <- df_traverse t, p x] ++ [Nothing] )
```

Depth-First Traversal

```
df_traverse :: Tree a -> [a]
df_traverse Nil = []
df_traverse (Node t1 x t2)
    = x : (df_traverse t1) ++ (df_traverse t2)
```

```
df_traverse  1
             /  \
            2    3
           / \  / \
          4  5 ... ..
```

```
= 1 : df_traverse  2  ++ df_traverse  3
             /  \             /  \
            4    5           ... ..
```

```
= 1 : 2 : df_traverse 4 ++ df_traverse 5  ++ df_traverse  3
                                           /  \
                                           ... ..
```

```
= 1 : 2 : 4 : [] ++ [] ++ 5 : [] ++ []  ++ df_traverse  3
                                           /  \
                                           ... ..
```

$$= [1, 2, 4, 5, \dots]$$

Depth-First Search

```
depthFirst' :: Eq a => (a -> Bool) -> Tree a -> Maybe a
depthFirst' p t
    = head( [Just x | x <- df_traverse t, p x] ++ [Nothing] )
```

```
depthFirst' (>4)
      1
     / \
    2   3
   / \  / \
  4  5 ... ..
```

```
= head( [Just x | x <- df_traverse 1, x>4] ++ [Nothing] )
```

```
      / \
     2   3
    / \  / \
   4  5 ... ..
```

```
= head( [Just x | x <- [1, 2, 4, 5, ...], x>4] ++ [Nothing] )
```

```
= head( [Just 5, ...] ++ [Nothing] )
```

```
= Just 5
```

Depth-First vs Breadth-First

```
df_traverse :: Tree a -> [a]
df_traverse Nil = []
df_traverse (Node t1 x t2)
    = x : (df_traverse t1) ++ (df_traverse t2)
```

```
bf_traverse :: Tree a -> [a]
bf_traverse t = bft [t]
    where
        bft [] = []
        bft xs = [x | Node _ x _ <- xs]
                  ++ bft (concat [ [t1,t2] | Node t1 _ t2 <- xs ])
```

Breadth-First Search

```
bf_traverse :: Tree a -> [a]
bf_traverse t = bft [t]
  where
    bft [] = []
    bft xs = [x | Node _ x _ <- xs]
              ++ bft (concat [ [t1,t2] | Node t1 _ t2 <- xs ])

breadthFirst :: (a -> Bool) -> Tree a -> Maybe a
breadthFirst p t
  = head( [Just x | x <- bf_traverse t, p x] ++ [Nothing] )
```

Breadth-First Traverse

```
bf_traverse :: Tree a -> [a]
```

```
bf_traverse t = bft [t]
```

```
  where
```

```
    bft [] = []
```

```
    bft xs = [x | Node _ x _ <- xs]
```

```
          ++ bft (concat [ [t1,t2] | Node t1 _ t2 <- xs ])
```

```
bf_traverse      1                                = bft [      1                                ]
```

```
      /  \
     2    3
    / \  / \
   4  5 6  7
      / \
     ... ..
```

```
      /  \
     2    3
    / \  / \
   4  5 6  7
      / \
     ... ..
```

```
= [1] ++ bft (concat [[ 2 , 3 ]]) = [1] ++ bft [ 2 , 3 ]
```

```
      / \  / \
     4  5 6  7
      / \
     ... ..
```

```
      / \  / \
     4  5 6  7
      / \
     ... ..
```

Breadth-First Traverse

```
bf_traverse :: Tree a -> [a]
bf_traverse t = bft [t]
  where
    bft [] = []
    bft xs = [x | Node _ x _ <- xs]
              ++ bft (concat [ [t1,t2] | Node t1 _ t2 <- xs ])

... = [1] ++ bft [ 2 , 3 ]
                / \   / \
                4  5  6  7
                / \
                ... ..

= [1] ++ [2,3] ++ bft (concat [[ 4 , 5 ], [ 6 , 7 ]])
                        / \
                        ... ..

= [1] ++ [2,3] ++ bft [ 4 , 5 , 6 , 7 ]
                        / \
                        ... ..

= [1] ++ [2,3] ++ [4,5,6,7] ++ ...
```

= [1, 2, 3, 4, 5, 6, 7, ...]

Breadth-First Search

```
breadthFirst :: (a -> Bool) -> Tree a -> Maybe a
breadthFirst p t
    = head( [Just x | x <- bf_traverse t, p x] ++ [Nothing] )
```

```
breadthFirst (>4)      1
                        /  \
                       2    3
                      / \  / \
                     4  5 ... ..
```

```
= head( [Just x | x <- bf_traverse 1 , x>4] ++ [Nothing] )
```

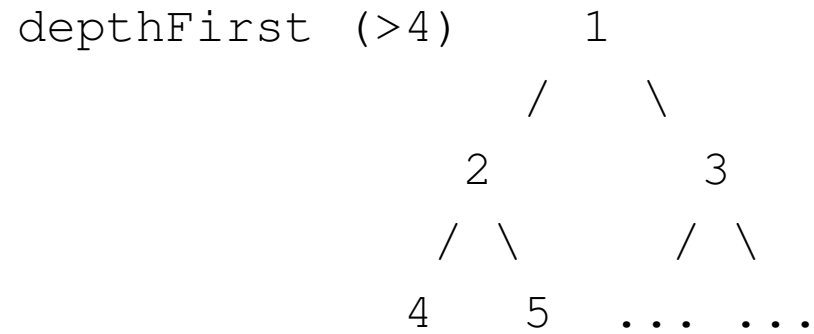
```
                        /  \
                       2    3
                      / \  / \
                     4  5 ... ..
```

```
= head( [Just x | x <- [1, 2, 3, 4, 5, 6, 7, ...], x>4] ++ [Nothing] )
```

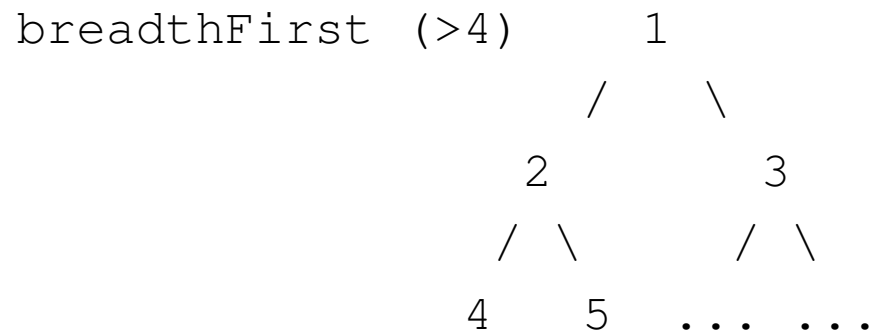
```
= head( [Just 5, Just 6, Just 7, ...] ++ [Nothing] )
```

```
= Just 5
```

Depth-First Search vs Breadth-First Search

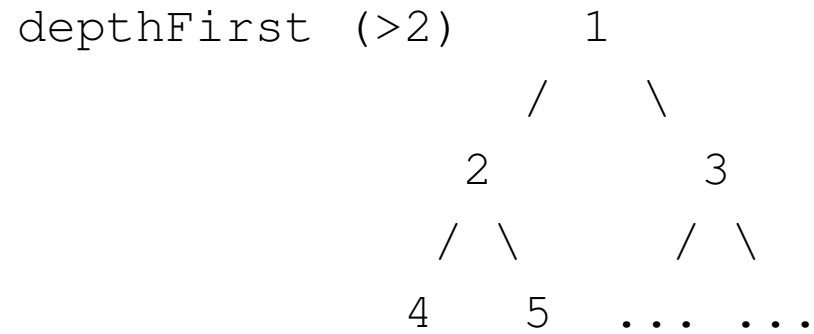


= Just 5

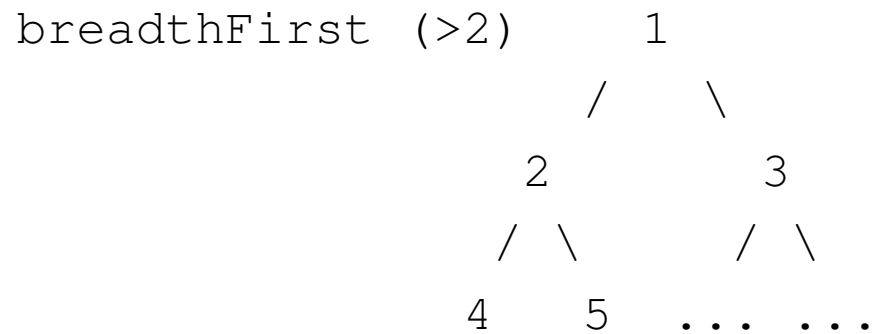


= Just 5

Depth-First Search vs Breadth-First Search



= Just 4



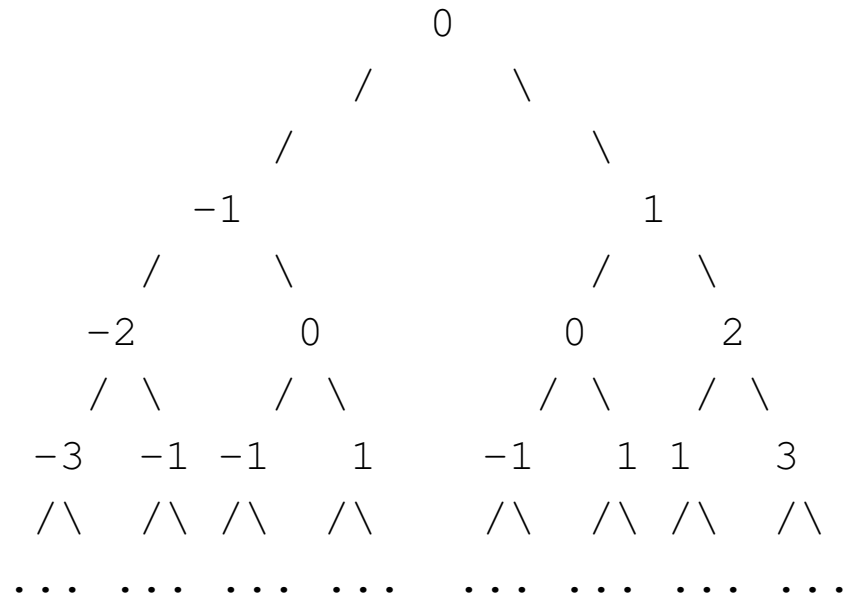
= Just 3

Infinite case

```
> depthFirst (>0) inf
*** Exception: stack overflow
```

```
> depthFirst' (>0) inf
[runs forever]
```

```
> breadthFirst (>0) inf
Just 1
(0.01 secs, 93,200 bytes)
```



Best-First Search

Idea: Decide the order of nodes to visit using an **evaluation function**.

```
bestFirst :: (a -> Bool) -> (Tree a -> Int)
                                   -> Tree a -> Maybe a
bestFirst p f t = bfs p (insert t (empty f))

bfs :: (a -> Bool) -> PQ (Tree a) -> Maybe a
bfs p pq | isempty pq = Nothing
         | otherwise = if p x then Just x
                        else bfs p (insertnode t1
                                              (insertnode t2 pq'))
      where Node t1 x t2 = top pq
            pq'          = pop pq

insertnode :: Tree a -> PQ (Tree a) -> PQ (Tree a)
insertnode Nil pq = pq
insertnode t pq   = insert t pq
```

Uses a **priority queue** with functions empty, insert, top, pop, isempty

Priority Queue

```
data PQ a = MkPQ (a->Int) [a]

invariant :: PQ a -> Bool
invariant (MkPQ f xs) =
  and [ f x >= f y | (x,y) <- zip xs (tail xs) ]
  -- in descending order of evaluation function results

empty :: (a->Int) -> PQ a
empty f = MkPQ f []

insert :: a -> PQ a -> PQ a
insert x (MkPQ f ys) = MkPQ f (ins x ys)
  where ins x [] = [x]
        ins x (y:ys) | f x >= f y = x : y : ys
                      | f x < f y  = y : ins x ys

top :: PQ a -> a      -- return item with highest priority
top (MkPQ _ (x:xs)) = x

pop :: PQ a -> PQ a  -- remove item with highest priority
pop (MkPQ f (x:xs)) = MkPQ f xs

isempty :: PQ a -> Bool
isempty (MkPQ f xs) = null xs
```

Best-First Search

```
eval :: Tree Int -> Int  
eval Nil = 0  
eval (Node t1 x t2) = x
```

```
> breadthFirst (>19) inf  
Just 20  
(9.72 secs, 1,031,895,824 bytes)  
required examining 2097151 nodes
```

```
> bestFirst (>19) eval inf  
Just 20  
(0.02 secs, 156,592 bytes)  
required examining 21 nodes
```

Best-First Search

```
eval :: Tree Int -> Int
eval Nil = 0
eval (Node t1 x t2) = x
```

```
> breadthFirst (>19) inf
Just 20
(9.72 secs, 1,031,895,824 bytes)
required examining 2097151 nodes
```

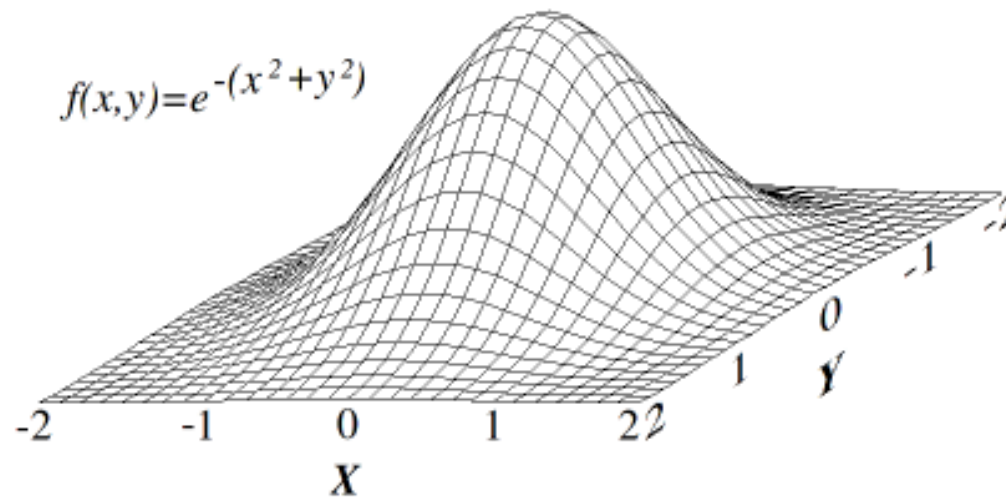
```
> bestFirst (>19) eval inf
Just 20
(0.02 secs, 156,592 bytes)
required examining 21 nodes
```

```
> breadthFirst (>100) inf
[requires examining 5070602400912917605986812821503 nodes]
```

```
> bestFirst (>100) eval inf
Just 101
(0.01 secs, 433,256 bytes)
required examining 101 nodes
```

A Limitation of Best-First Search

Local maximum is global maximum:



Local maximum which is not global maximum:

$$f(x,y)=e^{-(x^2+y^2)} + 2e^{-((x-1.7)^2+(y-1.7)^2)}$$

