Foundations of Computer Science
Lecture #10: Search

Anil Madhavapeddy
28th October 2022

Review: Curried Functions

```
> let prefix a b = a ^ b;;
val prefix : string -> string -> string = <fun>
```

```
                prefix a b      <=>  (prefix a) b
string -> string -> string           string -> (string -> string)
```

Review: Curried Functions

```
> let prefix a b = a ^ b;;
val prefix : string -> string -> string = <fun>
```

```
                 prefix a b       (prefix a) b
string -> string -> string  <=>  string -> (string -> string)
```

Expressions are evaluated from left to right (left -assoc.)

The -> symbol associates to the right

partial application: fix first arg.

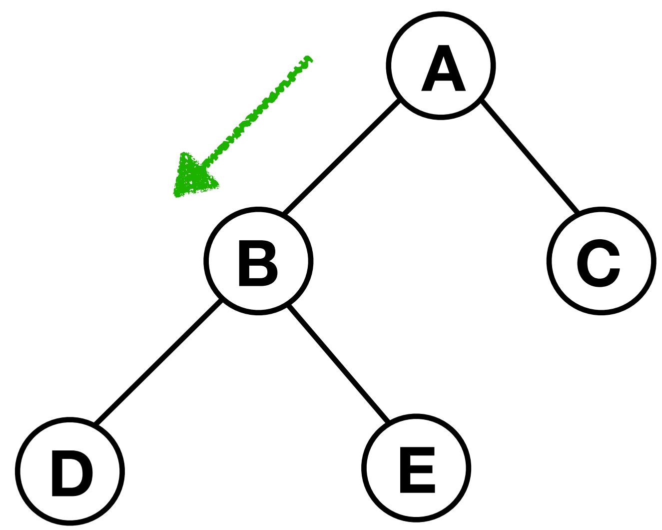
Example:

```
> let promote = prefix "Lady ";
let promote : string -> string = <fun>

> prefix "Ms. " "Smith";;
- : string = "Ms. Smith"

> promote  "Johnson";;
- : string = "Lady Smith"
```

Warm-Up



Pre-order ?

A - B - D - E - C

In-order ?

D - B - E - A - C

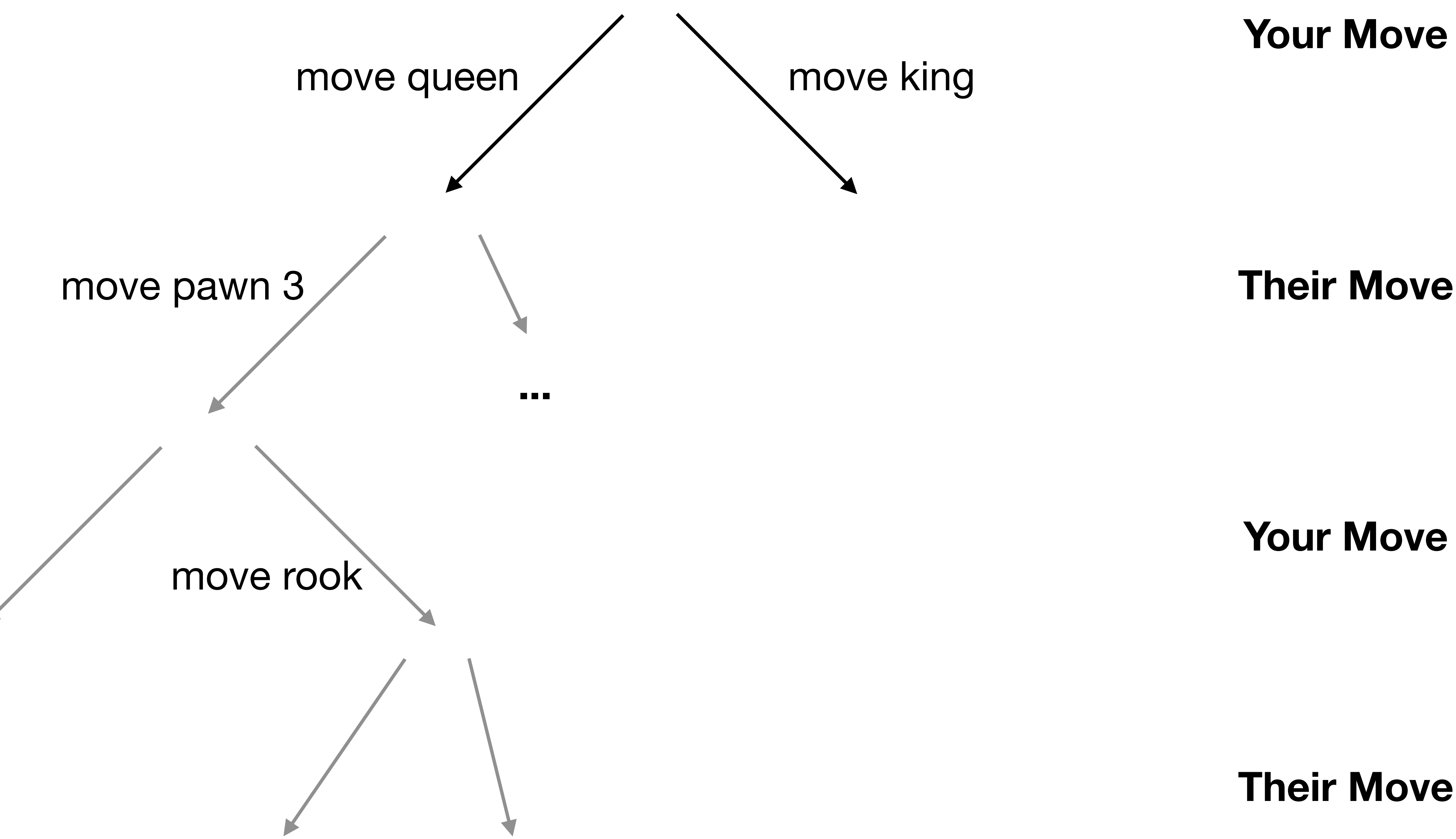
Post-order ?

D - E - B - C - A

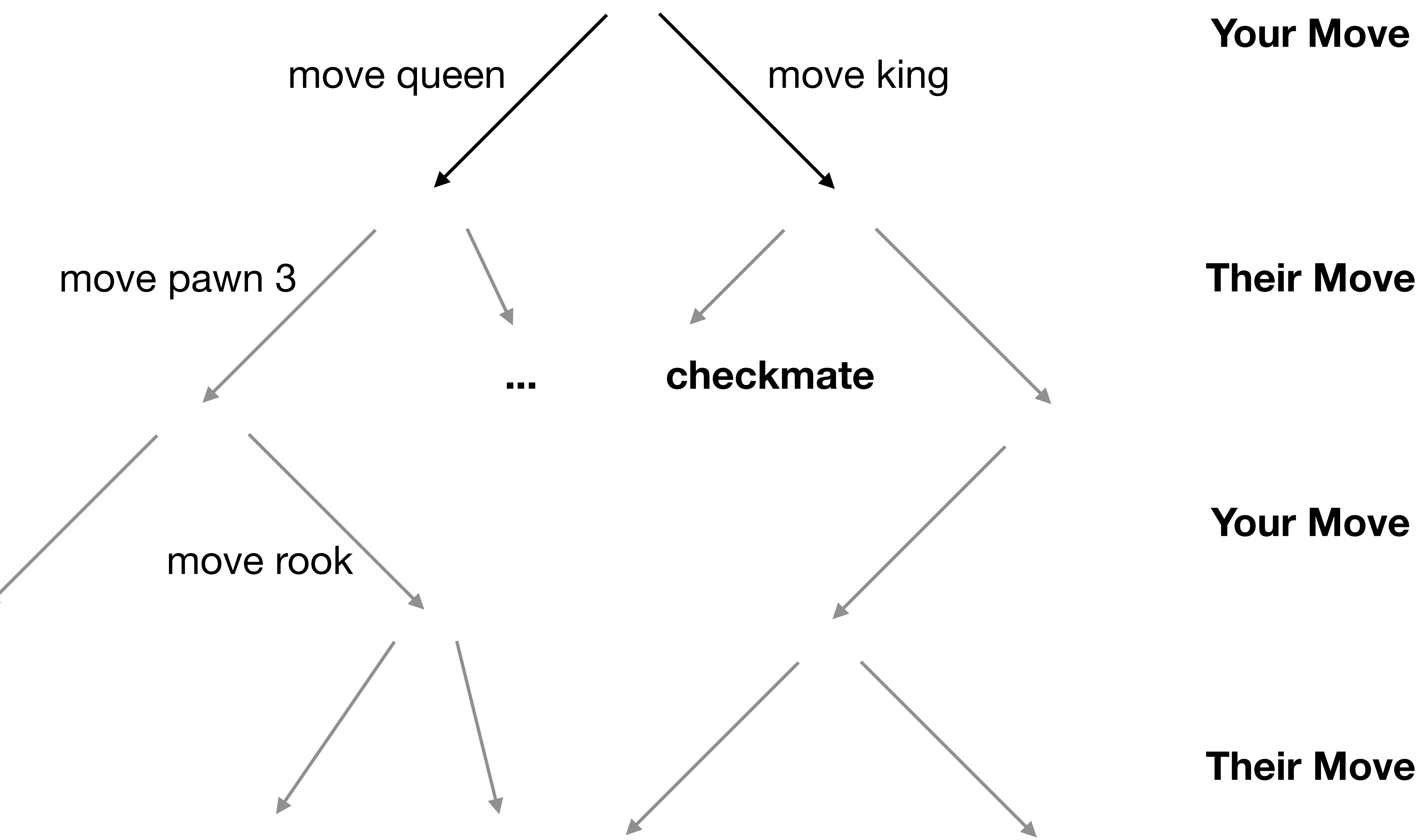
What kind of traversal is this?

depth-first

Breadth-First v Depth-First Tree Traversal



Breadth-First v Depth-First Tree Traversal



Breadth-First v Depth-First Tree Traversal

binary trees as *decision trees*

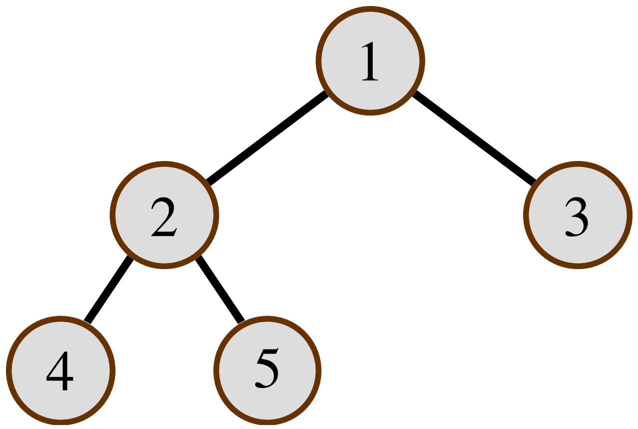
Look for *solution nodes*

- *Depth-first*: search one subtree in full before moving on
- *Breadth-first:* search all nodes at level k before moving to $k + 1$

  Finds *all* solutions — nearest first!

Reminder: type `tree`

```
type 'a tree = Lf
             | Br of 'a * 'a tree * 'a tree
```




```
Br(1, Br(2, Br(4, Lf, Lf),
            Br(5, Lf, Lf)),
      Br(3, Lf, Lf))
```

Breadth-First Tree Traversal — Using Append

```
let rec nbreadth = function
  | [] -> []
  | Lf :: ts -> nbreadth ts
  | Br (v, t, u) :: ts ->
      v :: nbreadth (ts @ [t; u])
```

queue

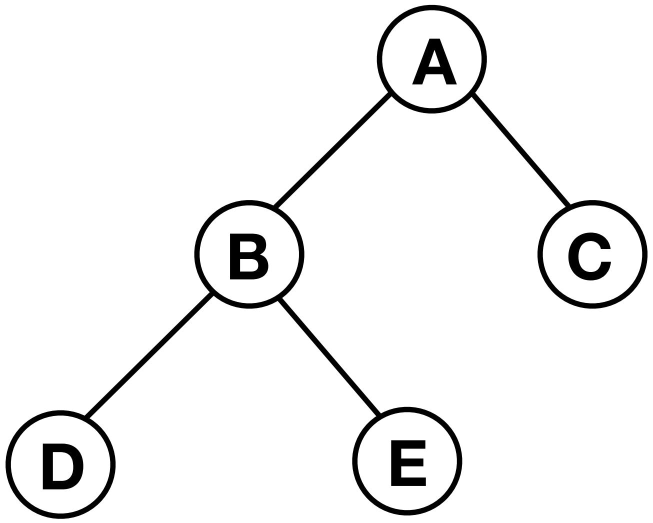
Keeps an *enormous queue* of nodes of search

Wasteful use of *append*

25 SECS to search depth 12 binary tree (4095 labels)

* careful: assumes depth starts at 1

Breadth-First Tree Traversal — Using Append

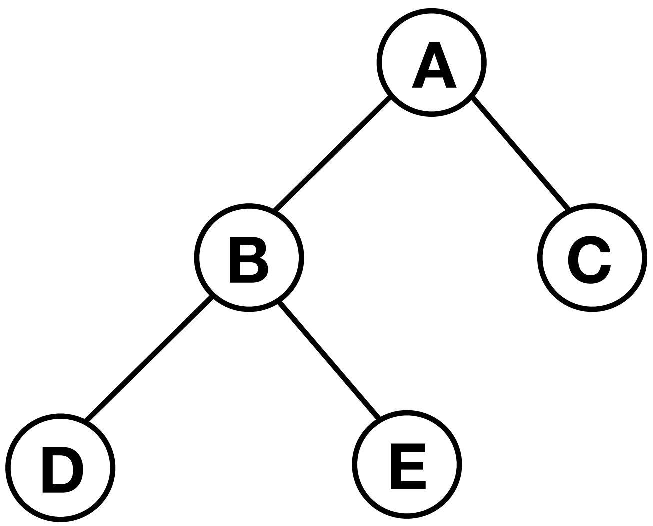


Notation in this example:

$Br(v_A, t_B, t_C)$ *is a tree* t_A *with root value* v_A *and subtrees* t_B, t_C

$nbreadth([t_A])$	(* *ts* is empty *)
$v_A :: nbreadth([] @ [t_B; t_C])$	(* put root value into list *)
$v_A :: nbreadth([t_B; t_C])$	(* execute append *)
$v_A :: v_B :: nbreadth([t_C] @ [t_D; t_E])$	(* append new subtrees *)
$v_A :: v_B :: nbreadth([t_C; t_D; t_E])$	
$v_A :: v_B :: v_C :: nbreadth([t_D; t_E] @ [Lf; Lf])$	
$v_A :: v_B :: v_C :: nbreadth([t_D; t_E; Lf; Lf])$	
...	

Breadth-First Tree Traversal — Using Append



Notation in this example:

$Br(v_A, t_B, t_C)$ is a tree t_A with root value v_A and subtrees t_B, t_C

$nbreadth([t_A])$	(* *ts* is empty *)
$v_A :: nbreadth([]\ @\ [t_B; t_C])$	(* put root value into list *)
$v_A :: nbreadth([t_B; t_C])$	(* execute append *)
$v_A :: v_B :: nbreadth([t_C]\ @\ [t_D; {}_E])$	(* append new subtrees *)
$v_A :: v_B :: nbreadth([t_C; t_D; t_E])$	
$v_A :: v_B :: v_C :: nbreadth([t_D; t_E]\ @\ [Lf; Lf])$	
$v_A :: v_B :: v_C :: nbreadth([t_D; t_E; Lf; Lf])$	
…	

first arg of append grows!

Breadth-First Tree Traversal — Using Append

Two key operations in `nbreadth` example:

[tree_1; tree_2; ...; tree_N]

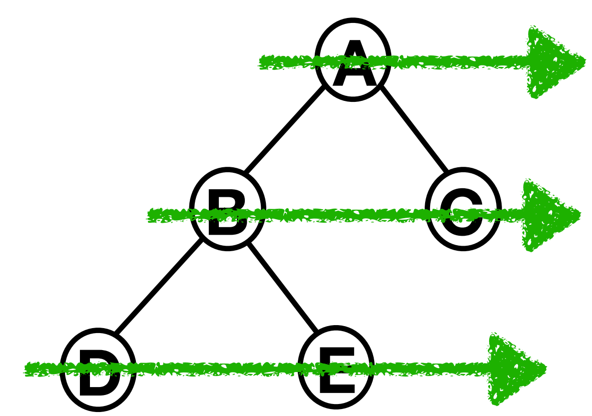
remove tree from head

add new subtrees to tail

The **order** matters:

Process what we first put into list *first*,

before we process its descendants.



-> find a better data-structure than ordinary list

An Abstract Data Type: Queues

We want: efficient FIFO data-structure

- `qempty` is the *empty queue*
- `qnull` *tests* whether a queue is empty
- `qhd` *returns* the element at the *head* of a queue
- `deq` *discards* the element at the *head* of a queue
- `enq` **adds** an element at the **end** of a queue

qhd

deq [tree_1; tree_2; ...; tree_N] ← enq

Efficient Functional Queues: Idea

Goal: avoid q@[x] since O(length(q))

Key idea: reverse back half of list!

Represent the queue $x_1 \; x_2 \; \ldots \; x_m \; \underline{y_n \; \ldots \; y_1}$

by **a pair of lists**

$$([x_1, x_2, \ldots, x_m], \; [y_1, y_2, \ldots, y_n])$$

Add new items to *rear list*

Remove items from *front list*; if empty move *rear* to *front*

careful! (reversed)

Amortized time per operation is $O(1)$

Efficient Functional Queues: Idea

Goal: deq `[1; 2; 3; 4; 5; 6]` $\xleftarrow{\text{enq}}$ `7`

Functional queue: `([1; 2; 3], [6; 5; 4])`

pattern-match and discard   cons 7

`1 :: [2; 3]`   `7 :: [6; 5; 4]`

Result: `([2; 3], [7; 6; 5; 4])`

Rationale of amortized cost, for a queue of length *n*:

- *n* `enq`, *n* `deq` operations
- *2n* cons operations for queue of length *n*
- O(1) cost per operation

Efficient Functional Queues: Code

```
type 'a queue = Q of 'a list * 'a list

let norm = function
| Q ([], tls) -> Q (List.rev tls, [])
| q -> q

let qnull q = (q = Q ([], []))

let enq (Q (hds, tls)) x =
  norm (Q (hds, x::tls))

exception Empty

let deq = function
| Q (x::hds, tls) -> norm (Q (hds, tls))
| _ -> raise Empty
```

Breadth-First Tree Traversal — Using Queues

```
let rec breadth q =
  if qnull q then []
  else
    match qhd q with
    | Lf -> breadth (deq q)
    | Br (v, t, u) ->
        v :: breadth (enq (enq (deq q) t) u)
```

removing first subtree

enq. its children

0.14 secs to search depth 12 binary tree (4095 labels)

200 times faster!

* careful: assumes depth starts at 1

Iterative Deepening: Another Exhaustive Search

Breadth-first search examines $O(b^d)$ nodes:

General formula:

$$1 + b + \cdots + b^d = \frac{b^{d+1} - 1}{b - 1}$$

$b =$ branching factor

$d =$ depth

For binary tree: $2^{d+1} - 1$

Space and time complexity: $O(b^d)$

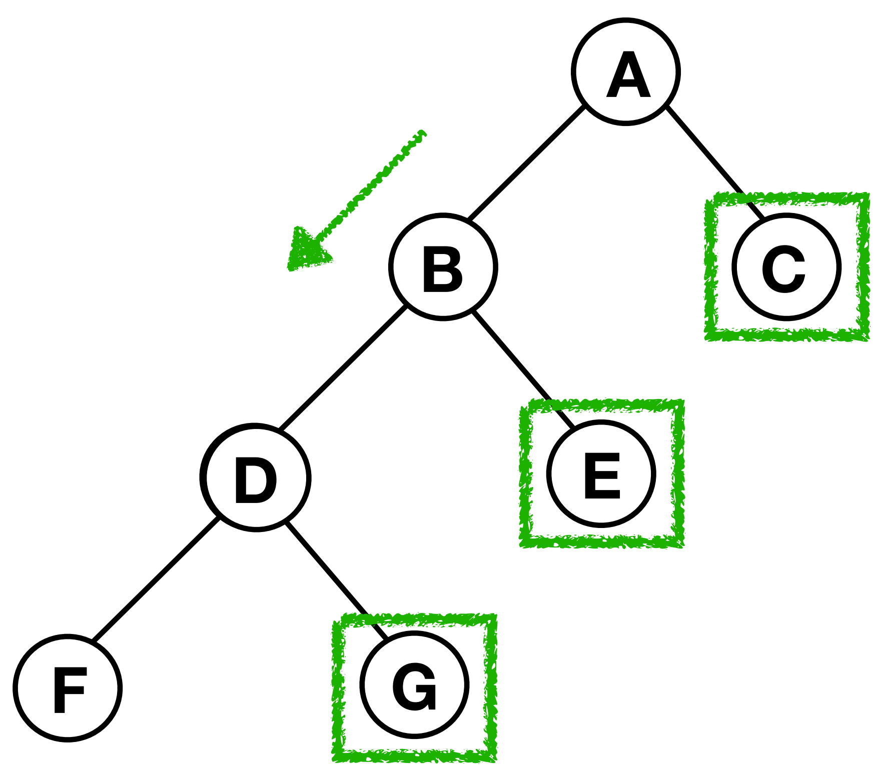
* careful: assumes depth starts at 0

Iterative Deepening: Another Exhaustive Search

Idea behind iterative deepening:

- Use **DFS** to get benefits of BFS
- Recompute nodes at depth *d* instead of storing them
- Complexity: $b/(b-1)$ times that for BFS (if b>1)
- Space requirement at depth $\boldsymbol{d}$ drops from $\boldsymbol{b^d}$ to $\boldsymbol{d}$

Recall depth-first search:



Space complexity: $O(d)$

Another Abstract Data Type: Stacks

- `empty` is the *empty stack*
- `null` *tests* whether a stack is empty
- `top` *returns* the element at the *top* of a stack
- `pop` *discards* the element at the *top* of a stack
- `push` *adds* an element at the *top* of a stack

A Survey of Search Methods

1. **Depth-first**: use a *stack* (efficient but incomplete)
2. **Breadth-first**: use a *queue* (uses too much space!)
3. **Iterative deepening**: use (1) to get benefits of (2)
   (trades time for space)
4. **Best-first**: use a *priority queue* (heuristic search)

The data structure determines the search!