Warmup

# type 'a tree =
| Br of 'a * 'a tree * 'a tree
| ??

What's the missing definition here to make a binary tree?
Warmup

```ocaml
# type 'a tree =
| Br of 'a * 'a tree * 'a tree
| Lf
```
What is the term when 'a is present in the type definition?
Warmup

What is the term when 'a is present in the type definition?

polymorphic
Warmup

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

What is the term when 'a is present in the type definition? *polymorphic*

What is the term when the type definition refers to itself?
Warmup

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

What is the term when 'a is present in the type definition?

*polymorphic*

What is the term when the type definition refers to itself?

*recursive*
Warmup

```ocaml
# type 'a option =
| None
| Some of 'a

# exception Not_found
```

Why use option types vs raising exceptions?
Warmup

# type 'a option =
| None
| Some of 'a

# exception Not_found

Why use option types vs raising exceptions?

```
val change_exn : int list -> int -> int list
val change_opt : int list -> int -> int list option
```

Every call to change_opt must check the option

```
match change_opt with
| None -> ... (* error case *)
| Some ch -> ... (* success case *)
```
Dictionaries

- A dictionary attaches **values** to identifiers (known as **keys**).
- Define the **operations** we want over the dictionary:
  - **lookup**: find an item in the dictionary
  - **update** / **insert**: replace / store an item in the dictionary
  - **delete**: remove an item from the dictionary
  - **empty**: the null dictionary with no keys
  - **Missing**: exception for errors in lookup and delete
Implementing a dictionary

- Simplest representation for a dictionary is an association list (a list of key/value tuples).

```ocaml
# exception Missing
exception Missing
```
Implementing a dictionary

- Simplest representation for a dictionary is an association list (a list of key/value tuples).

```ocaml
# exception Missing
exception Missing

# let rec lookup = function
| [], a -> raise Missing
| (x, y) :: pairs, a ->
  if a = x then
    y
  else
    lookup (pairs, a)
val lookup : ('a * 'b) list * 'a -> 'b = <fun>
```
Implementing a dictionary

- Simplest representation for a dictionary is an association list (a list of key/value tuples).

```ocaml
# exception Missing
exception Missing

# let rec lookup = function
  | [], a -> raise Missing
  | (x, y) :: pairs, a ->
    if a = x then
      y
    else
      lookup (pairs, a)

val lookup : ('a * 'b) list * 'a -> 'b = <fun>

# let update (l, b, y) = (b, y) :: l
val update : ('a * 'b) list * 'a * 'b -> ('a * 'b) list = <fun>
```
Implementing a dictionary

• Simplest representation for a dictionary is an **association list** (a list of key/value tuples).

```ocaml
# exception Missing
exception Missing

# let rec lookup = function
| [], a -> raise Missing
| (x, y) :: pairs, a ->
  if a = x then
    y
  else
    lookup (pairs, a)
val lookup : ('a * 'b) list * 'a -> 'b = <fun>

# let update (l, b, y) = (b, y) :: l
val update : ('a * 'b) list * 'a * 'b -> ('a * 'b) list = <fun>
```

Lookup is O(n)
Implementing a dictionary

• Simplest representation for a dictionary is an association list (a list of key/value tuples).

```ocaml
# exception Missing
exception Missing

# let rec lookup = function
  | [], a -> raise Missing
  | (x, y) :: pairs, a ->
    if a = x then
      y
    else
      lookup (pairs, a)
val lookup : ('a * 'b) list * 'a -> 'b = <fun>

# let update (l, b, y) = (b, y) :: l
val update : ('a * 'b) list * 'a * 'b -> ('a * 'b) list = <fun>
```

Lookup is $O(n)$

Update is $O(1)$
Implementing a dictionary

- Simplest representation for a dictionary is an **association list** (a list of key/value tuples).

```ocaml
# exception Missing
exception Missing

# let rec lookup = function
| [], a -> raise Missing
| (x, y) :: pairs, a ->
  if a = x then
    y
  else
    lookup (pairs, a)
val lookup : ('a * 'b) list * 'a -> 'b = <fun>

# let update (l, b, y) = (b, y) :: l
val update : ('a * 'b) list * 'a * 'b -> ('a * 'b) list = <fun>
```

Lookup is **O(n)**

Update is **O(1)**

But what is the space usage?
Binary Search Trees

- Use **binary trees** as a more efficient representation than a list to get a better lookup complexity.
Binary Search Trees

- Use **binary trees** as a more efficient representation than a list to get a better lookup complexity.
Binary Search Trees

• Use **binary trees** as a more efficient representation than a list to get a better lookup complexity.

• Each node holds a (key, value) with a total ordering for the keys

• The *left* subtree holds smaller keys and the *right* subtree holds larger keys
Binary Search Trees

- Use **binary trees** as a more efficient representation than a list to get a better lookup complexity.

- If *balanced* then lookup is $O(\log n)$

- If *unbalanced* then lookup can be $O(n)$
Binary Search Trees

```ocaml
# exception Missing of string

exception Missing of string

# let rec lookup = function
  | Br ((a, x), t1, t2), b ->
    if b < a then
      lookup (t1, b)
    else if a < b then
      lookup (t2, b)
    else
      x
  | Lf, b -> raise (Missing b)

val lookup : (string * 'a) tree * string -> 'a = <fun>
```
Binary Search Trees

# exception Missing of string
exception Missing of string

# let rec lookup = function
  | Br ((a, x), t1, t2), b ->
    if b < a then
      lookup (t1, b)
    else if a < b then
      lookup (t2, b)
    else
      x
  | Lf, b -> raise (Missing b)

val lookup : (string * 'a) tree * string -> 'a = <fun>
Binary Search Trees

# exception Missing of string
exception Missing of string

# let rec lookup = function
  | Br ((a, x), t1, t2), b ->
    if b < a then
      lookup (t1, b)
    else if a < b then
      lookup (t2, b)
    else
      x
  | Lf, b -> raise (Missing b)
val lookup : (string * 'a) tree * string -> 'a = <fun>

O(log n) if the tree is balanced
# let rec update k v = function
  | Lf -> Br ((k, v), Lf, Lf)
  | Br ((a, x), t1, t2) ->
    if k < a then
      Br ((a, x), update k v t1, t2)
    else if a < k then
      Br ((a, x), t1, update k v t2)
    else (* a = k *)
      Br ((a, v), t1, t2)

val update : 'a -> 'b -> ('a * 'b) tree -> ('a * 'b) tree = <fun>
Binary Search Trees

```ocaml
# let rec update k v = function
  | Lf -> Br ((k, v), Lf, Lf)
  | Br ((a, x), t1, t2) ->
    if k < a then
      Br ((a, x), update k v t1, t2)
    else if a < k then
      Br ((a, x), t1, update k v t2)
    else (* a = k *)
      Br ((a, v), t1, t2)

val update : 'a -> 'b -> ('a * 'b) tree -> ('a * 'b) tree = <fun>
```

Diagram of a binary search tree:
- The root node is James, 5.
- Gordon, 4 is on the right side, connected to the node with key 5.
- Both the left and right nodes of the root are Lf, indicating they are leaf nodes.
Binary Search Trees

# let rec update k v = function
| Lf -> Br ((k, v), Lf, Lf)
| Br ((a, x), t1, t2) ->
  if k < a then
    Br ((a, x), update k v t1, t2)
  else if a < k then
    Br ((a, x), t1, update k v t2)
  else (* a = k *)
    Br ((a, v), t1, t2)

val update : 'a -> 'b -> ('a * 'b) tree -> ('a * 'b) tree = <fun>
# let rec update k v = function
| Lf       -> Br ((k, v), Lf, Lf)
| Br ((a, x), t1, t2) ->
  if k < a then
    Br ((a, x), update k v t1, t2)
  else if a < k then
    Br ((a, x), t1, update k v t2)
  else (* a = k *)
    Br ((a, v), t1, t2)

val update : 'a -> 'b -> ('a * 'b) tree -> ('a * 'b) tree = <fun>
# let rec update k v = function  
| Lf -> Br ((k, v), Lf, Lf)  
| Br ((a, x), t1, t2) ->  
  if k < a then  
    Br ((a, x), update k v t1, t2)  
  else if a < k then  
    Br ((a, x), t1, update k v t2)  
  else (* a = k *)  
    Br ((a, v), t1, t2)

val update : 'a -> 'b -> ('a * 'b) tree -> ('a * 'b) tree = <fun>
Binary Search Trees

# let rec update k v = function
  | Lf -> Br ((k, v), Lf, Lf)
  | Br ((a, x), t1, t2) ->
    if k < a then
      Br ((a, x), update k v t1, t2)
    else if a < k then
      Br ((a, x), t1, update k v t2)
    else (* a = k *)
      Br ((a, v), t1, t2)

val update : 'a -> 'b -> ('a * 'b) tree -> ('a * 'b) tree = <fun>
Binary Search Trees

```ocaml
# let rec update k v = function
  | Lf -> Br ((k, v), Lf, Lf)
  | Br ((a, x), t1, t2) ->
    if k < a then
      Br ((a, x), update k v t1, t2)
    else if a < k then
      Br ((a, x), t1, update k v t2)
    else (* a = k *)
      Br ((a, v), t1, t2)

val update : 'a -> 'b -> ('a * 'b) tree -> ('a * 'b) tree = <fun>
```

Diagram:
```
       James, 10
      /    |
     Lf    Lf
     /     /
    Lf     Lf
```

Binary Search Trees

- We reconstruct the part of the structure that has changed and return the updated version.

- OCaml shares the original structure, and values pointing to the original remain unchanged.

- This is also known as a persistent data structure.

```ocaml
# let rec update k v = function
 | Lf -> Br ((k, v), Lf, Lf)
 | Br ((a, x), t1, t2) ->
   if k < a then
     Br ((a, x), update k v t1, t2)
   else if a < k then
     Br ((a, x), t1, update k v t2)
   else (* a = k *)
     Br ((a, v), t1, t2)

val update : 'a -> 'b -> ('a * 'b) tree -> ('a * 'b) tree = <fun>
```
Traversing Trees

Tree traversal refers to visiting the nodes of each tree in a well-defined order.

- **preorder** visits the label first (ABDEFCG)
- **inorder** visits the label midway (DBEAFCG)
- **postorder** visits the label last (DEBFGCA)
Traversing Trees: preorder

- **preorder** visits the label first (ABDECFG)

```ocaml
# let rec preorder = function
| Lf -> []
| Br (v, t1, t2) ->
  [v] @ preorder t1 @ preorder t2

val preorder : 'a tree -> 'a list = <fun>
```
Traversing Trees: inorder

- **inorder** visits the label midway (DBEAFCG)

```ocaml
# let rec inorder = function
  | Lf  -> []
  | Br (v, t1, t2) ->
    inorder t1 @ [v] @ inorder t2

val inorder : 'a tree -> 'a list = <fun>
```

```
    A
   / \
  B   C
 /   /\  
D   E   F
   \  /  
    G
```
Traversing Trees: inorder

- **inorder** visits the label midway (DBEAFCG)

```ocaml
# let rec inorder = function
  | Lf -> []
  | Br (v, t1, t2) ->
    inorder t1 @ [v] @ inorder t2

val inorder : 'a tree -> 'a list = <fun>
```

For binary search trees, this order respects the sorting constraint (left key < right key)

Also imaginatively known as a **treesort**.
Traversing Trees: postorder

- **postorder** visits the label last (DEBFGCA)

```ocaml
# let rec postorder = function
   | Lf -> []
   | Br (v, t1, t2) ->
     postorder t1 @ postorder t2 @ [v]

val postorder : 'a tree -> 'a list = <fun>
```

```
       A
      / \
     /   \
    B     C
   /     /\
  D     E  F
   \    /  \  
    \  G
```
Traversing Trees: postorder

• postorder visits the label last (DEBFGCA)

```ocaml
# let rec postorder = function
| Lf -> []
| Br (v, t1, t2) ->
  postorder t1 @ postorder t2 @ [v]

val postorder : 'a tree -> 'a list = <fun>
```

FGC
Traversing Trees: postorder

- **postorder** visits the label last (DEBFGCA)

```ocaml
# let rec postorder = function
| Lf -> []
| Br (v, t1, t2) ->
    postorder t1 @ postorder t2 @ [v]

val postorder : 'a tree -> 'a list = <fun>
```

[Tree representation: A (DEB) and C (FGC)]
Traversing Trees: postorder

- postorder visits the label last (DEBFGCA)

```ocaml
# let rec postorder = function
  | Lf -> []
  | Br (v, t1, t2) ->
      postorder t1 @ postorder t2 @ [v]

val postorder : 'a tree -> 'a list = <fun>
```

![Binary tree diagram]

DEB    FGC    A
Tree traversal refers to visiting the nodes of each tree in a well-defined order.

- preorder, inorder and postorder are **depth-first** traversal algorithms.

- The other possibility is **breadth-first** by going across the levels of the tree.
Arrays

Arrays are an indexed storage area for values

• Very common data structure alongside lists and trees in most languages.

• Arrays are usually updated in-place and are imperative or mutable data structures.

• Are used in many classic algorithms such as the original Hoare in-place partition-sort.
Arrays

Arrays are an indexed storage area for values

• Elements of a list can only be reached by counting from the head of the list.

• Elements of a tree can be reached by following a path from the root.

• Elements of an array are uniformly designated by number (the "subscript").
Arrays are an indexed storage area for values

Let's first consider an immutable array

- This is known as a functional array that is a finite map from integers to data.
- Updating implies copying the array to return a new version, but pointers to old copies remain.
- Can updates be efficient?
Functional Trees

The path to element $i$ follows the **binary code** for $i$ (the "subscript")

- The numbers above are not the values, but the positions of array elements.
- Complexity of access to this is always $O(\log n)$ as the tree is always balanced.
Functional Trees

The path to element $i$ follows the **binary code** for $i$ (the "subscript")

```
# exception Subscript

# let rec sub = function
  | Lf, _ -> raise Subscript
  | Br (v, t1, t2), k ->
    if k = 1 then v
    else if k mod 2 = 0 then
      sub (t1, k / 2)
    else
      sub (t2, k / 2)
```
Functional Trees

The path to element $i$ follows the **binary code** for $i$ (the "subscript")

```ocaml
# exception Subscript

# let rec sub = function
  | Lf, _                               -> raise Subscript
  | Br (v, t1, t2), 1                   -> v
  | Br (v, t1, t2), k when k mod 2 = 0  -> sub (t1, k / 2)
  | Br (v, t1, t2), k                   -> sub (t2, k / 2)
```
Functional Trees

The path to element $i$ follows the **binary code** for $i$ (the "subscript")

```ocaml
# let rec update = function
  | Lf, k, w ->
    if k = 1 then
      Br (w, Lf, Lf)
    else
      raise Subscript (* Gap in tree *)
  | Br (v, t1, t2), k, w ->
    if k = 1 then
      Br (w, t1, t2)
    else if k mod 2 = 0 then
      Br (v, update (t1, k / 2, w), t2)
    else
      Br (v, t1, update (t2, k / 2, w))
```
Functional Trees

The path to element $i$ follows the **binary code** for $i$ (the "subscript").

```ocaml
# let rec update = function
  | Lf, k, w ->
  if k = 1 then
    Br (w, Lf, Lf)
  else
    raise Subscript (* Gap in tree *)
  | Br (v, t1, t2), k, w ->
  if k = 1 then
    Br (w, t1, t2)
  else if k mod 2 = 0 then
    Br (v, update (t1, k / 2, w), t2)
  else
    Br (v, t1, update (t2, k / 2, w))
```

$O(\log n)$ if the tree is balanced
Functional Trees

The path to element $i$ follows the **binary code** for $i$ (the "subscript")

$15 = 0b1111$

$12 = 0b1100$

$11 = 0b1011$
Complexity of Dictionary Data Structures

- **Linear search:** Most general, needing only equality on keys, but inefficient (linear time).

- **Binary search:** Needs an ordering on keys. $O(\log n)$ in the average case, binary search trees are $O(n)$ in the worst case.

- **Array subscripting:** Least general, requiring keys to be integers, but even worst-case time is $O(\log n)$. 