Foundations of Computer Science

Binary trees

Dr. Robert Harle & Dr. Jeremy Yallop

2020–2021

A data structure with multiple branching is called a **tree**.

Trees are nearly as fundamental a structure as lists.

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

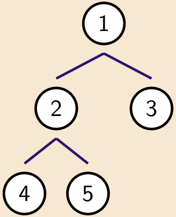
Each node is **either**:
a **leaf** (empty)
a **branch** with a label and two subtrees.

"Polymorphic" (parameterised) type

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

int tree

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



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

```
 In[1]: type 'a mylist =
        | Nil
        | Cons of 'a * 'a mylist

Out[1]: type 'a mylist = Nil | Cons of 'a * 'a mylist

 In[2]: Cons (1, Cons (2, Cons (3, Nil)))

Out[2]: - : int mylist = Cons (1, Cons (2, Cons (3, Nil)))
```

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

```
In[1]:  type 'a mylist =
        | Nil
        | Cons of 'a * 'a mylist

Out[1]: type 'a mylist = Nil | Cons of 'a * 'a mylist

In[2]:  Cons (1, Cons (2, Cons (3, Nil)))

Out[2]: - : int mylist = Cons (1, Cons (2, Cons (3, Nil)))
```

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

```
 In[1]: type 'a mylist =
        | Nil
        | Cons of 'a * 'a mylist

Out[1]: type 'a mylist = Nil | Cons of 'a * 'a mylist

 In[2]: Cons (1, Cons (2, Cons (3, Nil)))

Out[2]: - : int mylist = Cons (1, Cons (2, Cons (3, Nil)))
```

Binary Trees & Lists

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

```
 In[1]:  type 'a mylist =
         | Nil
         | Cons of 'a * 'a mylist

Out[1]:  type 'a mylist = Nil | Cons of 'a * 'a mylist

 In[2]:  Cons (1, Cons (2, Cons (3, Nil)))

Out[2]:  - : int mylist = Cons (1, Cons (2, Cons (3, Nil)))
```

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

```
 In[1]: type 'a mylist =
        | Nil
        | Cons of 'a * 'a mylist
Out[1]: type 'a mylist = Nil | Cons of 'a * 'a mylist
 In[2]: Cons (1, Cons (2, Cons (3, Nil)))
Out[2]: - : int mylist = Cons (1, Cons (2, Cons (3, Nil)))
```

Polymorphic & Recursive

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

Recursive

```
type shape =
| Null
| Join of shape * shape
```

Polymorphic

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

Simple Operations on Trees

```
In[3]:  (* number of branch nodes *)
        let rec count = function
        | Lf -> 0
        | Br (v, t1, t2) -> 1 + count t1 + count t2

Out[3]: val count : 'a tree -> int = <fun>

In[4]:  (* length of longest path *)
        let rec depth = function
        | Lf -> 0
        | Br (v, t1, t2) -> 1 + max (depth t1) (depth t2)

Out[4]: val depth : 'a tree -> int = <fun>
```

Use pattern matching to build expressions over trees

The invariant $\text{count}(t) \leq 2^{\text{depth(t)}} - 1$ holds above

Simple Operations on Trees

```
In[3]:  (* number of branch nodes *)
        let rec count = function
        | Lf -> 0
        | Br (v, t1, t2) -> 1 + count t1 + count t2

Out[3]: val count : 'a tree -> int = <fun>

In[4]:  (* length of longest path *)
        let rec depth = function
        | Lf -> 0
        | Br (v, t1, t2) -> 1 + max (depth t1) (depth t2)

Out[4]: val depth : 'a tree -> int = <fun>
```

Use pattern matching to build expressions over trees

The invariant $\text{count}(t) \leq 2^{\text{depth(t)}} - 1$ holds above

Simple Operations on Trees

```
In[3]:  (* number of branch nodes *)
        let rec count = function
        | Lf -> 0
        | Br (v, t1, t2) -> 1 + count t1 + count t2

Out[3]: val count : 'a tree -> int = <fun>

In[4]:  (* length of longest path *)
        let rec depth = function
        | Lf -> 0
        | Br (v, t1, t2) -> 1 + max (depth t1) (depth t2)

Out[4]: val depth : 'a tree -> int = <fun>
```

Use pattern matching to build expressions over trees

The invariant $\text{count}(t) \leq 2^{\text{depth(t)}} - 1$ holds above

Simple Operations on Trees

```
In[3]:  (* number of branch nodes *)
        let rec count = function
        | Lf -> 0
        | Br (v, t1, t2) -> 1 + count t1 + count t2

Out[3]: val count : 'a tree -> int = <fun>

In[4]:  (* length of longest path *)
        let rec depth = function
        | Lf -> 0
        | Br (v, t1, t2) -> 1 + max (depth t1) (depth t2)

Out[4]: val depth : 'a tree -> int = <fun>
```

Use pattern matching to build expressions over trees

The invariant $\text{count}(t) \leq 2^{\text{depth(t)}} - 1$ holds above

```
In[3]:  (* number of branch nodes *)
        let rec count = function
        | Lf -> 0
        | Br (v, t1, t2) -> 1 + count t1 + count t2

Out[3]: val count : 'a tree -> int = <fun>

In[4]:  (* length of longest path *)
        let rec depth = function
        | Lf -> 0
        | Br (v, t1, t2) -> 1 + max (depth t1) (depth t2)

Out[4]: val depth : 'a tree -> int = <fun>
```

Use pattern matching to build expressions over trees

The invariant $\text{count}(t) \leq 2^{\text{depth(t)}} - 1$ holds above