Foundations of Computer Science

Dictionaries and binary search trees

Dr. Robert Harle & Dr. Jeremy Yallop

2020–2021

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

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

```
In[1]: exception Missing

Out[1]: exception Missing
```

Lookup is $O(n)$

```
In[2]: let rec lookup = function
         | [], a -> raise Missing
         | (x, y) :: pairs, a -> if a = x then y
                                 else lookup (pairs, a)

Out[2]: val lookup : ('a * 'b) list * 'a -> 'b = <fun>
```

Update is $O(1)$

```
In[3]: let update (l, b, y) = (b, y) :: l

Out[3]: val update : ('a * 'b) list * 'a * 'b -> ('a * 'b) list
                   = <fun>
```

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

```
In[1]: exception Missing

Out[1]: exception Missing
```

Lookup is $O(n)$

```
In[2]: let rec lookup = function
         | [], a -> raise Missing
         | (x, y) :: pairs, a -> if a = x then y
                                 else lookup (pairs, a)

Out[2]: val lookup : ('a * 'b) list * 'a -> 'b = <fun>
```

Update is $O(1)$

```
In[3]: let update (l, b, y) = (b, y) :: l

Out[3]: val update : ('a * 'b) list * 'a * 'b -> ('a * 'b) list
                   = <fun>
```

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

```
In[1]: exception Missing
Out[1]: exception Missing
```

Lookup is $O(n)$

```
In[2]: let rec lookup = function
         | [], a -> raise Missing
         | (x, y) :: pairs, a -> if a = x then y
                                 else lookup (pairs, a)
Out[2]: val lookup : ('a * 'b) list * 'a -> 'b = <fun>
```

Update is $O(1)$

```
In[3]: let update (l, b, y) = (b, y) :: l
Out[3]: val update : ('a * 'b) list * 'a * 'b -> ('a * 'b) list
          = <fun>
```

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

```
In[1]: exception Missing

Out[1]: exception Missing
```

Lookup is $O(n)$

```
In[2]: let rec lookup = function
       | [], a -> raise Missing
       | (x, y) :: pairs, a -> if a = x then y
                               else lookup (pairs, a)

Out[2]: val lookup : ('a * 'b) list * 'a -> 'b = <fun>
```

Update is $O(1)$

```
In[3]: let update (l, b, y) = (b, y) :: l

Out[3]: val update : ('a * 'b) list * 'a * 'b -> ('a * 'b) list
                   = <fun>
```

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

```
In[1]: exception Missing

Out[1]: exception Missing
```

Lookup is $O(n)$

```
In[2]: let rec lookup = function
       | [], a -> raise Missing
       | (x, y) :: pairs, a -> if a = x then y
                               else lookup (pairs, a)

Out[2]: val lookup : ('a * 'b) list * 'a -> 'b = <fun>
```

Update is $O(1)$

```
In[3]: let update (l, b, y) = (b, y) :: l

Out[3]: val update : ('a * 'b) list * 'a * 'b -> ('a * 'b) list
                   = <fun>
```

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

```
In[1]: exception Missing

Out[1]: exception Missing
```

Lookup is $O(n)$

```
In[2]: let rec lookup = function
       | [], a -> raise Missing
       | (x, y) :: pairs, a -> if a = x then y
                               else lookup (pairs, a)

Out[2]: val lookup : ('a * 'b) list * 'a -> 'b = <fun>
```

Update is $O(1)$

```
In[3]: let update (l, b, y) = (b, y) :: l

Out[3]: val update : ('a * 'b) list * 'a * 'b -> ('a * 'b) list
                   = <fun>
```

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

```
In[1]: exception Missing

Out[1]: exception Missing
```

Lookup is $O(n)$

```
In[2]: let rec lookup = function
       | [], a -> raise Missing
       | (x, y) :: pairs, a -> if a = x then y
                               else lookup (pairs, a)

Out[2]: val lookup : ('a * 'b) list * 'a -> 'b = <fun>
```

Update is $O(1)$

```
In[3]: let update (l, b, y) = (b, y) :: l

Out[3]: val update : ('a * 'b) list * 'a * 'b -> ('a * 'b) list
                   = <fun>
```

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

```
In[1]: exception Missing

Out[1]: exception Missing
```

Lookup is $O(n)$

```
In[2]: let rec lookup = function
       | [], a -> raise Missing
       | (x, y) :: pairs, a -> if a = x then y
                               else lookup (pairs, a)

Out[2]: val lookup : ('a * 'b) list * 'a -> 'b = <fun>
```

Update is $O(1)$

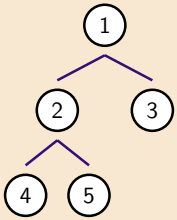
```
In[3]: let update (l, b, y) = (b, y) :: l

Out[3]: val update : ('a * 'b) list * 'a * 'b -> ('a * 'b) list
                   = <fun>
```

But what is the **space usage**?

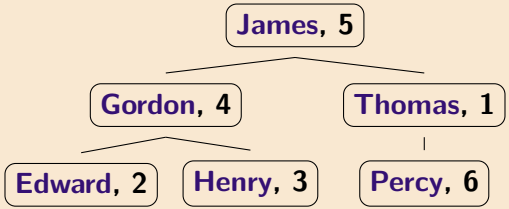
Use binary trees as a more efficient representation to get a better lookup complexity.

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



Use binary trees as a more efficient representation to get a better lookup complexity.

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



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

```
In[4]:  exception Missing of string

Out[4]: exception Missing of string

In[5]:  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)

Out[5]: val lookup : (string * 'a) tree * string -> 'a = <fun>
```

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

If **unbalanced** then lookup can be $O(n)$

```
In[4]:  exception Missing of string

Out[4]: exception Missing of string

In[5]:  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)

Out[5]: val lookup : (string * 'a) tree * string -> 'a = <fun>
```

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

If **unbalanced** then lookup can be $O(n)$

```
 In[4]: exception Missing of string

Out[4]: exception Missing of string

 In[5]: 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)

Out[5]: val lookup : (string * 'a) tree * string -> 'a = <fun>
```

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

If **unbalanced** then lookup can be $O(n)$

```
In[4]:  exception Missing of string

Out[4]: exception Missing of string

In[5]:  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)

Out[5]: val lookup : ('a * 'b) tree * 'a -> 'b = <fun>
```

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

If **unbalanced** then lookup can be $O(n)$

```
 In[4]: exception Missing of string

Out[4]: exception Missing of string

 In[5]: 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)

Out[5]: val lookup : (string * 'a) tree * string -> 'a = <fun>
```

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

If **unbalanced** then lookup can be $O(n)$

```
In[6]: 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)

Out[6]: val update : 'a -> 'b -> ('a * 'b) tree -> ('a * 'b) tree
                  = <fun>
```

Reconstruct the part of the structure that has changed; return the updated version.

OCaml shares the original structure; values pointing to the original remain unchanged.

This is also known as a **persistent data structure**

```
In[6]: 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)

Out[6]: val update : 'a -> 'b -> ('a * 'b) tree -> ('a * 'b) tree
                  = <fun>
```

Reconstruct the part of the structure that has changed; return the updated version.

OCaml shares the original structure; values pointing to the original remain unchanged.

This is also known as a **persistent data structure**

```
In[6]: 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)

Out[6]: val update : 'a -> 'b -> ('a * 'b) tree -> ('a * 'b) tree
                  = <fun>
```

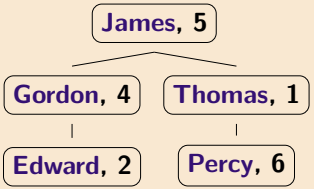
Reconstruct the part of the structure that has changed; return the updated version.

OCaml shares the original structure; values pointing to the original remain unchanged.

This is also known as a **persistent data structure**

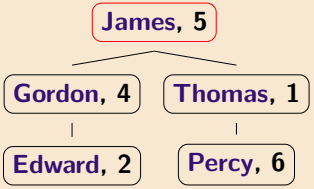
```
let rec update k v = function
| Lf -> Br ((k, v), Lf, Lf)
| Br ((a, x), t1, t2) ->
    if k < a then Br ((a, x), update k v t1, t2)
    else if a < k then Br ((a, x), t1, update k v t2)
    else Br ((a, v), t1, t2)
```

update "Henry" 3 t



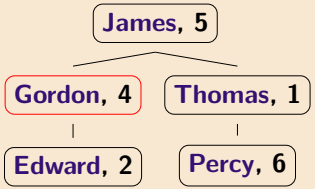
```
let rec update k v = function
| Lf -> Br ((k, v), Lf, Lf)
| Br ((a, x), t1, t2) ->
    if k < a then Br ((a, x), update k v t1, t2)
    else if a < k then Br ((a, x), t1, update k v t2)
    else Br ((a, v), t1, t2)
```

update "Henry" 3 t



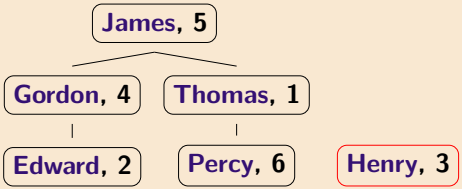
```
let rec update k v = function
| Lf -> Br ((k, v), Lf, Lf)
| Br ((a, x), t1, t2) ->
    if k < a then Br ((a, x), update k v t1, t2)
    else if a < k then Br ((a, x), t1, update k v t2)
    else Br ((a, v), t1, t2)
```

update "Henry" 3 t



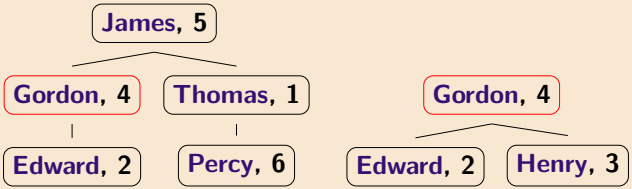
```
let rec update k v = function
| Lf -> Br ((k, v), Lf, Lf)
| Br ((a, x), t1, t2) ->
    if k < a then Br ((a, x), update k v t1, t2)
    else if a < k then Br ((a, x), t1, update k v t2)
    else Br ((a, v), t1, t2)
```

update "Henry" 3 t



```
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 Br ((a, v), t1, t2)
```

update "Henry" 3 t



```
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 Br ((a, v), t1, t2)
```

update "Henry" 3 t