Red-Black Trees

Many invariants make quick work

15-150 M21

Lecture 0716
16 July 2021

0 Sets

```
3 ∈ S1?
 >  No
5 ∈ S1?
 >  Yes
7 ∈ S1?
 >  No
S2 = insert(3,S1)
3 ∈ S2?
 >  Yes
5 ∈ S2?
 >  Yes
7 ∈ S2?
 >  No
```

S1: ?

S2: ?

SET signature

aux-library/SET.sig

```
signature SET =
sig
  structure Elt : EQ

  type Set

  val empty : Set

  exception ExistingElt
  val insert : Elt.t * Set  -> Set
  val overwrite : Elt.t * Set  -> Set
```

REPL Demonstration:
Insert, Overwrite, and Lookup

aux-library/SET.sig

```
  val remove : Elt.t * Set -> Set

  val lookup : Set -> Elt.t -> Elt.t option

  val union : Set -> Set -> Set

  val toString : (Elt.t -> string) -> Set ->
 string
end
```

Simple Version

aux-library/Set.sml

```
functor ListSet (Elt : EQ):>SET
```

aux-library/Set.sml

```
  (* INVARIANT: S : Set contains no duplicates
  *  (according to Elt.equal)    *)
  type Set = Elt.t list

  val empty = []

  fun lookup [] y = NONE
    | lookup (x::xs) y =
        if (Elt.equal x y)
        then SOME(x)
        else lookup xs y
```

Towards a better implementation: Adding a sortedness invariant

aux-library/SET.sig

```
signature ORD =
sig
  type t

  (* INVARIANT: equal is a comparison function *)
  val cmp : t * t -> order
end
```

Recall `cmpEqual : ORD -> EQ`

```
functor cmpEqual (K : ORD):EQ =
struct
  type t = K.t
  fun equal x y = K.cmp(x,y) = EQUAL
end
```

aux-library/Set.sml

```
functor OrdListSet (EltOrd : ORD):> SET
```

aux-library/Set.sml

```
  structure Elt = cmpEqual(EltOrd)

  (* INVARIANT: S : Set contains no duplicates
   *   (according to Elt.equal) and is sorted
   *   (according to EltOrd.cmp)
 *)
  type Set = Elt.t list
```

aux-library/Set.sml

```
   fun lookup [] y = NONE
     | lookup (x::xs) y =
         case EltOrd.cmp(y,x) of
           LESS => NONE
         | EQUAL => SOME(x)
         | GREATER => lookup xs y
```

```
3 ∈ S1?
 >  No
 5 ∈ S1?
 >  Yes
 7 ∈ S1?
 >  No
 S2 = insert(3,S1)
 3 ∈ S2?
 >  Yes
 5 ∈ S2?
 >  Yes
 7 ∈ S2?
 >  No
```

S1:
[1,2,4,5,6,10,11]

S2:
[1,2,3,4,5,6,10,11]

One note about the code

- Want `Elt.t` to be transparent
- Want `Set` to be opaque

aux-library/Set.sml

```
functor OrdListSet (EltOrd : ORD):> SET
where type Elt.t = EltOrd.t=
struct
```

Complexity

n is the number of elements in the set

	Work	**Span**
`insert/overwrite`	$O(n)$	$O(n)$
`lookup`	$O(n)$	$O(n)$

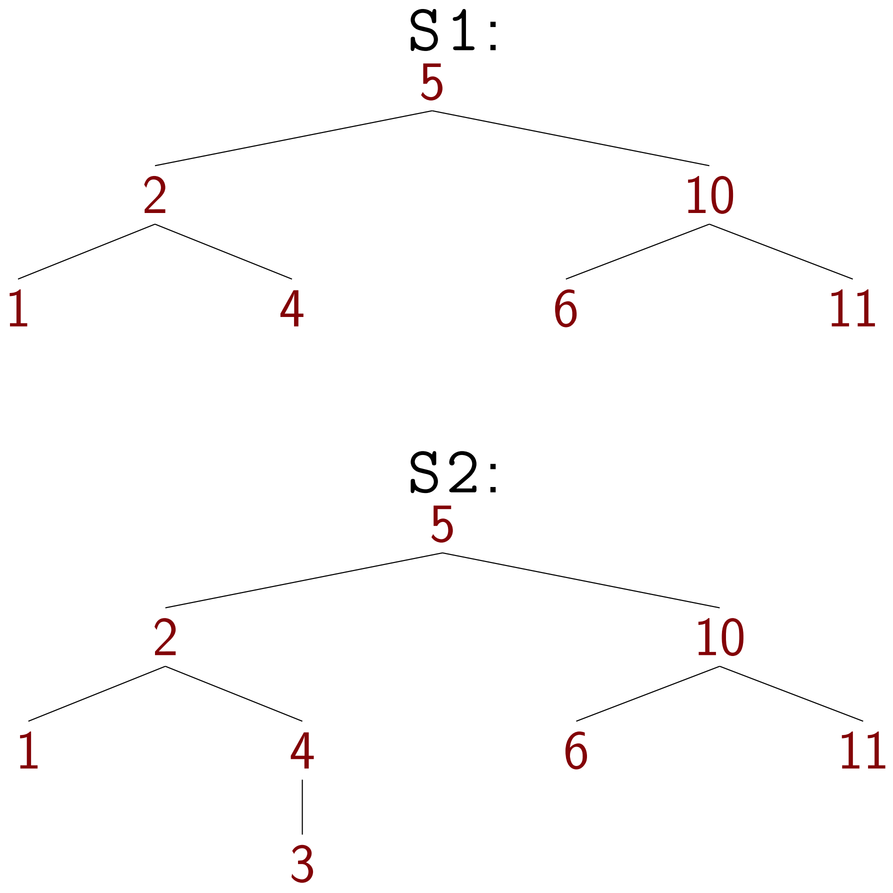
OrdTreeSet

aux-library/Set.sml

```
functor OrdTreeSet (EltOrd : ORD):> SET
where type Elt.t = EltOrd.t =
struct
  structure Elt = cmpEqual(EltOrd)

  (* INVARIANT: S : Set contains no duplicates
   *   (according to Elt.equal) and is sorted
   *   (according to EltOrd.cmp)
 *)
  type Set = Elt.t Tree.tree
```

```
3 ∈ S1?
> No
5 ∈ S1?
> Yes
7 ∈ S1?
> No
S2 = insert(3,S1)
3 ∈ S2?
> Yes
5 ∈ S2?
> Yes
7 ∈ S2?
> No
```



Complexity

n is the number of elements in the set. Assume trees are balanced.

		Work	Span
`OrdListSet`	`insert/overwrite`	$O(n)$	$O(n)$
	`lookup`	$O(n)$	$O(n)$
`OrdTreeSet`	`insert/overwrite`	$O(\log n)$	$O(\log n)$
	`lookup`	$O(\log n)$	$O(\log n)$

1 Representation Independence

Claim:

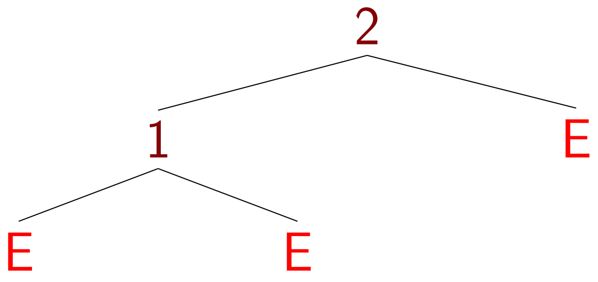
A user won't be able to tell the difference between `OrdListSet` and `OrdTreeSet`

Consider the expression

`insert(2,insert(1,empty)) : Set`

encoding the set $\{1, 2\}$.

Depending on whether we use the `ListSet`, `OrdListSet`, or `OrdTreeSet` implementation, we might get different values (of different types):

`ListSet`	`OrdListSet`	`OrdTreeSet`
`[2,1]`	`[1,2]`	

Key Idea:

These represent the same value, in their respective implementations of `Set`

To prove the **representation independence** result between `OrdListSet` and `OrdTreeSet`, take an arbitrary `EltOrd : ORD` and let

```
structure OL = OrdListSet(EltOrd)
structure OT = OrdTreeSet(EltOrd)
```

And define a binary relation $\mathcal{R}$ between the values of `OL.Set` and `OT.Set`, such that

$\mathcal{R}(\texttt{lS}, \texttt{tS}) \iff$ `lS : OL.Set` represents the same set (using `OL`) that `tS : OT.Set` does (using `OT`)

aux-library/SET.sig

```
  val empty : Set

  exception ExistingElt
  val insert : Elt.t * Set   -> Set
  val overwrite : Elt.t * Set   -> Set

  val remove : Elt.t * Set -> Set

  val lookup : Set -> Elt.t -> Elt.t option

  val union : Set -> Set -> Set

  val toString : (Elt.t -> string) -> Set -> string
```

- $\mathcal{R}$(`OL.empty`, `OT.empty`)
- If $\mathcal{R}$(`lS`, `tS`), then for any `x : EltOrd.t`,

$$\mathcal{R}(\texttt{OL.overwrite(x,ls)}, \texttt{OT.overwrite(x,tS)})$$

- If $\mathcal{R}$(`lS`, `tS`), then for any `x : EltOrd.t`, either:
  - Both `OL.insert(x,ls)` and `OT.insert(x,tS)` raise their respective `ExistingElt` exceptions, or
  - $\mathcal{R}$(`OL.insert(x,ls)`, `OT.insert(x,tS)`)
- If $\mathcal{R}$(`lS`, `tS`), then for any `x : Elt.t`,

$$\texttt{OL.lookup lS x} \cong \texttt{OT.lookup tS x}$$

- **Check Your Understanding** (Analogously for `union` & `toString`)

The Point:

If all these things are true, then `OrdListSet` and `OrdTreeSet` have equivalent/indistinguishable/interchangeable behavior

OrdListSet: `type Set = Elt.t list`

OrdTreeSet: `type Set = Elt.t Tree.tree`

$$\mathcal{R}(\texttt{lS}, \texttt{tS}) \iff \texttt{lS} \cong \texttt{(inord tS)}$$

Prop. A tree `T` is sorted iff `(inord T)` is sorted.

- $\mathcal{R}$(`OL.empty`, `OT.empty`)
- If $\mathcal{R}$(`lS`, `tS`), then for any `x : EltOrd.t`,

$$\mathcal{R}(\texttt{OL.overwrite(x,ls)}, \texttt{OT.overwrite(x,tS)})$$

- If $\mathcal{R}$(`lS`, `tS`), then for any `x : EltOrd.t`, either:
  - Both `OL.insert(x,ls)` and `OT.insert(x,tS)` raise their respective `ExistingElt` exceptions, or
  - $\mathcal{R}$(`OL.insert(x,ls)`, `OT.insert(x,tS)`)
- If $\mathcal{R}$(`lS`, `tS`), then for any `x : Elt.t`,

$$\texttt{OL.lookup lS x} \quad \cong \quad \texttt{OT.lookup tS x}$$

- Analogously for `union` & `toString`

Proof Sketch: `overwrite case`

- $\mathcal{R}(\texttt{OL.empty}, \texttt{OT.empty})$
- If $\mathcal{R}(\texttt{lS}, \texttt{tS})$, then for any $\texttt{x} : \texttt{EltOrd.t}$,

$$\mathcal{R}(\texttt{OL.overwrite(x,ls)}, \texttt{OT.overwrite(x,tS)})$$

- If $\mathcal{R}(\texttt{lS}, \texttt{tS})$, then for any $\texttt{x} : \texttt{EltOrd.t}$, either:
  - Both `OL.insert(x,ls)` and `OT.insert(x,tS)` raise their respective `ExistingElt` exceptions, or
  - $\mathcal{R}(\texttt{OL.insert(x,ls)}, \texttt{OT.insert(x,tS)})$
- If $\mathcal{R}(\texttt{lS}, \texttt{tS})$, then for any $\texttt{x} : \texttt{Elt.t}$,

$$\texttt{OL.lookup lS x} \cong \texttt{OT.lookup tS x}$$

- Analogously for `union` & `toString`

5-minute break

2 Dictionaries and Red-Black Trees

0716.0 (DICT.sig)

```
signature DICT =
sig
  structure Key : EQ

  type 'a entry = Key.t * 'a
  type 'a dict

  val empty : 'a dict

  exception ExistingEntry
  val insert : 'a entry * 'a dict -> 'a dict
  val overwrite : 'a entry * 'a dict -> 'a dict
  val lookup : 'a dict -> Key.t -> 'a option
end
```

0716.1 (dict.sml)

```
functor RBDict (KeyOrd : ORD) :> DICT
  where type Key.t = KeyOrd.t =
struct
```

REPL Demonstration:

`RBDict`

0716.1 (dict.sml)

```
functor RBDict (KeyOrd : ORD) :> DICT
  where type Key.t = KeyOrd.t =
struct
  structure Key = cmpEqual(KeyOrd)
  type 'a entry = Key.t * 'a
```

0716.2 (dict.sml)

```
datatype 'a dict =
    Empty
  | Red of 'a dict * 'a entry * 'a dict
  | Black of 'a dict * 'a entry * 'a dict
```

```
val empty = Empty

fun lookup d k' =
  let
    fun lk (Empty) = NONE
      | lk (Red tree) = lk' tree
      | lk (Black tree) = lk' tree
    and lk' (L, (k,v), R) =
    (case KeyOrd.cmp(k',k)
       of EQUAL => SOME(v)
        | LESS => lk L
          | GREATER => lk R)
  in
    lk d
  end
```

Defn. A value `T : t dict` is said to be a **red-black tree (RBT)** if it satisfies three conditions:

1. `T` is sorted by its keys
   (either `T=Empty`; or
   (`T=Red(L,(k,v),R)` or `T=Black(L,(k,v),R)` such that `L` and `R` are sorted by
   `k` is `KeyOrd.cmp`-greater-than-or-equal to the key `k'` of any entry `(k',v')` in `L` and
   `k` is `KeyOrd.cmp`-less-than-or-equal to any key in `R`))
2. For any `Red` node in `T`, both its children are black (`Empty` is considered black)
3. Every path from the root of `T` to its leaves contains the same number of black nodes, called the *black height* of `T`

Demonstration:
RBT-preseving insert

```
fun overwrite (entry,Empty) =
    Red(Empty,entry,Empty)
  | overwrite ((k',v'),Red(L,(k,v),R)) =
      case Key.cmp(k',k) of
        LESS => Red(overwrite((k',v'),L),(k,v),R)
      | EQUAL => Red(L,(k',v'),R)
      | GREATER => Red(L,(k,v),overwrite((k',v'),R))
  | overwrite ((k',v'),Black(L,(k,v),R)) =
      case Key.cmp(k',k) of
        LESS => Black(overwrite((k',v'),L),(k,v),R)
      | EQUAL => Black(L,(k',v'),R)
      | GREATER => Black(L,(k,v),overwrite((k',v'),R))
```

Problem

In the `Red` case: the recursive call to `overwrite((k'v'),L)` could return a `Red` node, so `overwrite` would produce a `Red` root with a `Red` child

Defn. A value `T : t dict` is said to be an **almost red-black tree (ARBT)** if it satisfies three conditions:

1. `T` is sorted by its keys
2. For any* `Red` node in `T`, both its children are black (`Empty` is considered black)

   * *except for maybe the root*
3. Every path from the root of `T` to its leaves contains the same number of black nodes, called the *black height* of `T`

.

`restoreLeft : 'a dict -> 'a dict`
REQUIRES: Either `D` is an RBT, or `Black(L,e,R)` where `L` is an ARBT and `R` is an RBT
ENSURES: `restoreLeft(D)` is an RBT with the same entries as `D` and the same black height

`restoreRight : 'a dict -> 'a dict`
REQUIRES: Either `D` is an RBT, or `Black(L,e,R)` where `L` is an RBT and `R` is an ARBT
ENSURES: `restoreLeft(D)` is an RBT with the same entries as `D` and the same black height

Code Review: `overwrite and insert`

Overwrite helper

If we have an `e : t entry` in scope, define

`helpero : t dict -> t dict`
REQUIRES: `D` is an RBT
ENSURES: `helpero D` evaluates to `D'` containing all the entries of `D`, plus `e` (replacing any existing entry in `D` whose key is `Key.equal` to the key of `e`). `D'` is an RBT if the root of `D` is `Black`, and `D'` is an ARBT if the root of `D` is `Red`.

Check Your Understanding

Write the spec of `helperi`.

Summary

- Prove equivalence of two structures ascribing to the same signature by relating values representing the same structure
- Carefully invariants and dutifully maintain them, using the opacity of the modules system to prevent the user from breaking them

Next Time

- Start *Applications* portion of course
- Parallel data structures and algorithms

Thank you!