Recitation 10 — Balanced Trees

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

2 November 2011

Today's Agenda:

- Announcements
- Operations on Trees
- Analysis of QUICKSORT, round 2

1 Announcements

- Exam 2 will be Thursday, 10 November at 7pm, in Doherty 2210.
- Exam 2 review will be Wednesday, 9 November at 7pm, in HH B103
- Questions about homework or class?
- Homework 7 is a practice exam and is out now. It's due on 8 November at 23:59 EST. You may not use any late days for this assignment because we'll be going over parts of it in recitation and review on 9 November.

  There are many more questions than there will be on the exam, and a lot of them are harder or more open ended than we would ask on the exam, but they're all relevant material. This time, we're asking that you write up solutions to all the questions on the practice and submit them as usual.

  We don't expect your submission for HW7 to have the same level of detail as you usually put into a homework assignment. If we're convinced that you spent some time understanding each problem and have produced the key insights towards its solution, you'll get full credit.

2 Operations on Trees

As in lecture, we'll be talking about simple binary trees as values of type

```
datatype 'd BST = Leaf
                | Node of ('d BST * 'd BST * key * 'd)
```

Since we need to know an ordering on the keys, the type `key` comes from an ordered structure that also provides that ordering, so it doesn't appear as a parameter in the type. The data stored at each node, however, can be of any type, so BSTs are parametric in that type.

We use values of type `'d BST` to represent totally ordered finite sets of pairs of type $(key *' d)$ by the following mapping:

1. *Leaf* represents $\emptyset$
2. If T_1 represents a set S_1 and T_2 represents a set S_2, then *Node*$(T_1, (k, v), T_2)$ represents the set $S_1 \cup S_2 \cup \{(k, v)\}$

The tree structure directly encodes the ordering of the set: a left child is less than its parent; a right child greater.

2.1 Join

We mentioned *join* in class as the dual to *split* to form a basis for operations on BSTs, but we didn't discuss its implementation.

Recall the spec for *join*:

[$join(L, m, R) : BST \times (key \times data)\ option \times BST \to BST$]
This takes a left BST L, an optional middle key-data pair m, and a right BST R. It requires that all keys in L are less than all keys in R. Furthermore if the optional middle element is supplied, then its key must be larger than any n L and less than any in R. It creates a new BST which is the union of L, R and the optional m.

2.1.1 Implementation

```
fun join (T1 : 'd bst, m : (key * 'd) option, T2 : 'd bst) : 'd bst =
    case m
     of SOME (k,v) => Node(T1,T2, k, v)
      | NONE =>
        case T2
         of Leaf => T1
          | Node(L,R,k,v) => Node(join(T1, NONE,L), R, k, v)
```

- We inspect the optional value
  - If it's `SOME(k,v)`, then we make it the value of the new root and terminate. This is correct because of the spec we have on `join` is actually quite strong
  - If it's `NONE`, then we need to figure out how to make a new tree from the two trees without breaking the BST invariants.
    - If `T2` contains no keys, we return `T1`
    - Otherwise: the left sub-tree is the result of joining `T1` and the left sub-tree of `T2`; the right sub-tree is the right sub-tree of `T2`; and the key-value pair is the key-value pair of `T2`.
      This preserves the BST invariants because we've promoted the right sub-tree of the right child. That means that every element in the new right child is greater than everything in the left child by assumption and the spec of join.

2.1.2 Cost

The analysis of this depends on the relative sizes of `T1` and `T2`, which is a balance criterion, so we'll analyize this more carefully when we talk about specific balancing trees. `split` remains largely unchanged for the various trees, but `join` changes more substantially.

Informally, we recurr down `T1` and not `T2`, so the work and span should be proportional only to the depth of `T1`. If it's balanced, this will be logarithmic.

2.2 Union

If A and B are sets, their union is

$$A \cup B := \{x : x \in A \vee x \in B\}$$

as in Figure 1. If we're using trees to represent sets, recall that we can compute the intersection as follows:

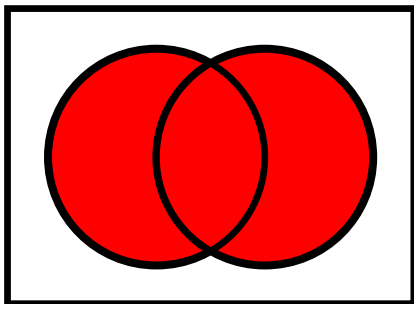

Figure 1: Union of Two Sets

```
fun union (T1 : 'd bst, T2 : 'd bst) : 'd bst =
    case T1
     of Leaf => T2
      | Node(L1, R1, k1, v1) =>
        let
          val (L2, _, R2) = split (T2, k1)
        in
          join (union(L1, L2), SOME (k1,v1), union(R1,R2))
        end
```

1. We inspect `T1`
   (a) If it represents the empty set, we return `T2` because $X \cup \emptyset = X$ for any set X.
   (b) If not, it must be `Node(L1, R1, k1, v1)`.
      i. We split `T2` based on `k1`, computing two sets whose union is `T1` without `k1`, by the spec of *`split`*. Call them `L2` and `R2` respectively.
      ii. By the spec of *`split`*, every key in `L2` is less than `k1` and every key in `R2` is greater. By the BST invariants, every key in `L1` is less than `k1` and every key in `R1` is greater. Therefore, the union of `L1` and `L2` and the union of `R1` and `R2` will also form a BST.
      iii. We join the BSTs produced inductively from the unions with the key. We add this key unconditionally, since union contains everything in both sets.

Analysis of cost of union is done very carefully in lecture 19 notes.

2.3 Intersection

If A and B are sets, their intersection is

$$A \cap B := \{x : x \in A \wedge x \in B\}$$

as in Figure 2. If we're using trees to represent sets, we can compute the intersection as follows:

```
fun inter (T1 : 'd bst, T2 : 'd bst) : 'd bst =
    case T1
     of Leaf => Leaf
      | Node(L1, R1, k1, v1) =>
        let
          val (L2, x, R2) = split (T2, k1)
```

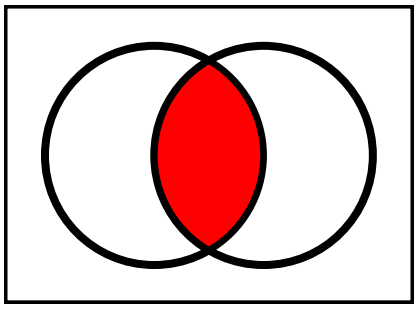

Figure 2: Intersection of Two Sets

```
  val keep = case x
              of NONE => NONE
               | _  => SOME(k1,v1)
in
  join (inter(L1, L2), keep, inter(R1,R2))
end
```

The codes proceeds by recursion on the left argument. It would be symmetric in the right argument, but either way is as good as the other.

The argument of why this code is correct is basically identical to the argument for `union` except for the new variable `keep`. Since an element is in the intersection of two sets if and only if it is in both sets, we can't just add the key-value pair immediately: instead, we only add it to the resultant set if it also appears in `T2`. By the spec if `split`, if it appears in `T2`, then it must be returned as the middle element from `split`, so this is sufficient.

Because of how similar the code is, the analysis of the cost of intersection is practically identical to the analysis of cost of union is in lecture 19 notes.

2.4 Set Difference

If A and B are sets, their difference is

$$A \backslash B := \{x : x \in A \wedge x \notin B\}$$

as in Figure 3. If we're using trees to represent sets, we can compute the difference as:

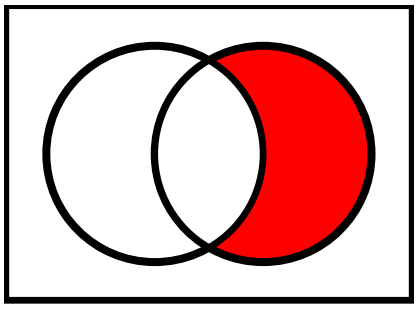

Figure 3: Difference of Two Sets

```
fun diff (T1 : 'd bst, T2 : 'd bst) : 'd bst =
    case T1
     of Leaf => Leaf
      | Node(L1, R1, k1, v1) =>
        let
          val (L2, x, R2) = split (T2, k1)
          val keep = case x
                      of NONE => SOME(k1,v1)
                       | _  => NONE
        in
          join (diff(L1, L2), keep, diff(R1,R2))
        end
```

Again, the argument of why this code is correct is basically identical to the argument for `intersection` except for the use of `keep`. Since an element is in the difference of two sets if and only if it is in first and not the second, we can't just add the key-value pair immediately: instead, we only add it to the resultant set if it fails to appear in `T2`. By the spec if `split`, if it appears in `T2`, then it must be returned as the middle element from `split`, so this is sufficient.

Because of how similar the code is, the analysis of the cost of difference is practically identical to the analysis of cost of union is in lecture 19 notes.

3 QUICKSORT Analysis

We argued in lecture yesterday that the expected runtime of randomized QUICKSORT is in $O(n \lg n)$. Tomorrow we'll argue that the likelihood that we're very far away from that expected runtime is extremely small, so we'll go over the argument again somewhat slowly.

See lecture 19 for analysis.