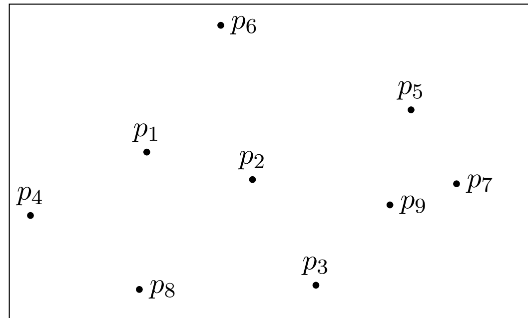
CS3110 Spring 2016 Lecture 19: Binary Search Tree Theory

Robert Constable

Topics

1. Remarks on "implementing" coordinate free convex hulls.

   We approach this topic as a result in *Euclidean geometry*. To frame a solvable task, we need to include the given points in a bounded region of the plane, e.g.

   Why is that? We need to *create line segments* and their extensions. We need to *sweep* lines parallel to a constructed segment, and detect when and where they touch a point. We need to know when to stop.

2. Comments on Splay-trees, Red/Black trees, AVL-trees, 2-3 trees,...

3. Comments on "OCaml logic." Note, when we are using types for logic, we assume they are total types rather than *partial types*, which all OCaml types are. Is α list $\rightarrow \alpha$ provable?

4. Developing a type based theory of trees using *recursive types*, also called *inductive types* in the setting of *total types*, e.g. Coq. This gives us a view of a possible future for CS education.

Trees

Among the important variants of binary search trees are Red/Black trees, AVL trees, 2-3 trees, and Splay trees. These variants are designed to ensure that the trees are of $\log(n)$ depth to store n elements. All of these are discussed in Cormen [1].

- Red/Black trees: on all paths the number of black nodes is the same, children of a red node must be black, each insert is red, and then the colors are adjusted.
- AVL trees: height of left subtree differs by at most one from the height of the right, is maintained as an invariant.
- 2-3 trees (Hopcroft 1970): every internal node has 2 or 3 children.
- Splay trees: amortized time per opertion is $\log(n)$.

Binary Search Tree theory

There are some results about binary search trees in Coq, but they do not fit well with the course. Mark Bickford was able to formalize exactly the results we need, using Nuprl. His account also covers ideas from type theory that I think are very important.

Here is his definition of a BST, with values of type E.

Definition: Binary tree type

```
bs_tree = null
   | leaf(E)
   | node bs_tree(E) E bs_tree
```

Mark uses this relation, $x \in T$, for t a bs_tree. Here is his definition:

Definition: Ordered tree

$x \in \text{null} = \text{false}$
$x \in \text{leaf}(v) = (x = v)$
$x \in \text{node}(a, v, b) = (x \in a) \text{ or } (x = v) \text{ or } (x \in b)$

Fact. If a tree t is ordered and $x : E$, then *there is at most one element* $v \in t$ *such that cmp(x,v)=0.*

As mentioned before, it is possible to start with a more general notion of tree, *infinite trees* also called *co-trees* or spreads. Here is the definition:

```
treeco(E) == corec(x. l:Atom x if l = a "leaf" then E
                                if l = a "node"
                                then left: x*x
                                else Void fi)
```

We can measure the size of such a tree.

```
fun size, p = let l,x = p in
            if l = a "leaf" then 0
            if l = a "node"
            then let left,right = x
            in (1 + size left + size right)
            else 0
```

Definition: `treeco_size(p) = fix(fun size, p_body(size,p))`

Where `body(size,p)` is the function body defined above.

Definition: `tree(E) =` {`p:treeco(E) | treeco_size(p)`↓}

Fix operator

The *fix* operator is used to build recursive functions in Nuprl's type theory. Here is how they work.

```
fact n = if n = 1 then 1 else n * fact(n-1)
```

Instead start with the functional

$$\lambda(f.\lambda(n.\ \text{if } n = 1 \text{ then } 1 \text{ else } n * f(n-1)))$$

then "take the least fixed point" $\text{fix}(\lambda(f.\lambda(n.\ \text{body}(f,n))))$.

This evaluates to $\lambda n.\text{body}(\text{fix}(\lambda(f.\lambda(n.\ \text{body}(f,n))), n))$. We will discuss this more next week.

Theory of Binary Search Trees: Developed by Mark Bickford in Nuprl's constructive type theory.

binary tree datatype

```
bs_tree(E) = null
           | leaf(E)
           | node bs_tree(E) E bs_tree(E)
```

We can put data of any type E in our binary search tree. So, for example, E could be (int * real) which we can think of as a key-value pair where the int is the key and the real is the value. In such a case we can order the keys, but there may be no computable ordering on the values.

comparison operator To order the values of type E, we need a computable comparison operator. For example, when E is (int * real) and a = (3, cosine(3)) and b = (4, sine(7)) we see that the key of a is less than the key of b because (3-4) ¡ 0, even though we might not know easily which of the reals is smaller.

In general we suppose that there is a comparison cmp of type $E \rightarrow E \rightarrow int$ such that $cmpab < 0$ means that a is "less than" b and $cmpab > 0$ means that a is "greater than" b while $cmpab = 0$ means that a is "equivalent to" b.

The comparison cmp should satisfy these properties for all $a, b, c \in E$:

$$cmp(a,b) = -cmp(b,a) \text{ (antisymmetry)}$$

$$cmp(a,b) \geq 0 \text{ and } cmp(b,c) \geq 0 \Rightarrow cmp(a,c) \geq 0 \text{ (transitivity)}$$

$$(cmp(a,b) = 0 \Rightarrow cmp(a,c) = cmp(b,c) \text{ (congruence)}$$

For example, on type E = (int * real) the comparison

```
cmp (x,r) (y, s) = x-y
```

satisfies these three properties.

Notice that cmp (x,r) (x,s) = 0, so in general cmp(a,b)=0 does not imply that a=b. In general, cmp(a,b)=0 only means that we consider a and b equivalent for the purpose of lookup, insert, and delete in our binary search tree.

ordered tree We define $x \in t$ recursively by cases:

$$\begin{aligned} x \in null &= false \\ x \in leaf(v) &= (x = v) \\ x \in node(a,v,b) &= (x \in a) \text{ or } (x = v) \text{ or } (x \in b) \end{aligned}$$

Then we define, recursively by cases, when tree t is ordered by comparison cmp

$$\begin{aligned} \text{ordered}(cmp, null) &= true \\ \text{ordered}(cmp, leaf(v)) &= true \\ \text{ordered}(cmp, node(a,v,b)) &= \text{ordered}(cmp, a) \text{ and ordered}(cmp, b) \\ &\quad \text{and, for all } x \in a, cmp(x,v) < 0 \\ &\quad \text{and, for all } x \in b, cmp(x,v) > 0 \end{aligned}$$

Notice that if tree t is ordered, and x is in type E, then there can be at most one element $v \in t$ for which $cmp(x, v) = 0$.

max of ordered tree The maximum member of an ordered tree is the "right-most" member of the tree. If the tree is null then it does not have a maximum member, so we have to take some care. In one pass of a recursive function we can find the maximum member m of a tree t and also return the tree t' which is the result of deleting m from t. To handle the case of the null tree, we use a default value d for the maximum.

$$\begin{aligned}
\text{treemax}(d, null) &= (d, null) \\
\text{treemax}(d, leaf(v)) &= (v, null) \\
\text{treemax}(d, node(a, v, b)) &= \text{if } null(b) \text{ then } (v, a) \\
&\quad \text{else let } m, b' = \text{treemax}(d, b) \text{ in } (m, node(a, v, b'))
\end{aligned}$$

deleting from a binary search tree Deleting element x from an ordered tree t means removing from the tree any element $v \in t$ for which $cmp(x, v) = 0$ – there can be at most one such element v to remove. Here is how we do it:

$$\begin{aligned}
\text{delete}(cmp, x, null) &= null \\
\text{delete}(cmp, x, leaf(v)) &= \text{if } cmp(x, v) = 0 \text{ then } null \text{ else } leaf(v) \\
\text{delete}(cmp, x, node(a, v, b)) &= \text{if } cmp(x, v) < 0 \text{ then } node(\text{delete}(cmp, x, a), v, b) \\
&\quad \text{else if } cmp(x, v) > 0 \text{ then } node(a, v, \text{delete}(cmp, x, b) \\
&\quad \text{else if } null(a) \text{ then } b \\
&\quad \text{else let } m.a' = \text{treemax}(d, a) \text{ in } node(a', m, b)
\end{aligned}$$

Notation shortcut In the rest of this document we will write $x > v$ instead of $cmp(x, v) > 0$, $x < v$ instead of $cmp(x, v) < 0$, $x = v$ instead of $cmp(x, v) = 0$ and $x \neq v$ instead of $cmp(x, v) \neq 0$.

We can justify facts like $x = v \Rightarrow \text{not}(x < v)$ because if $x = v$ then we really have $cmp(x, v) = 0$; so by the congruence property for cmp, $cmp(x, x) = cmp(v, x)$, and by the antisymmetry property we get $cmp(x, x) = 0$. So $cmp(v, x) = 0$ and hence neither $cmp(v, x) < 0$ nor $cmp(v, x) > 0$.

Lemma 1

$$z \in \text{delete}(cmp, x, t) \Rightarrow z \in t$$

Proof. We leave this as an exercise.

Theorem 1

$$\text{ordered}(cmp, t) \Rightarrow \text{ordered}(cmp, \text{delete}(cmp, x, t))$$

proof: This is another exercise (a bit harder).

Theorem 2

$$\text{ordered}(cmp, t) \Rightarrow (z \in \text{delete}(cmp, x, t) \Leftrightarrow (z \in t \text{ and } z \neq x))$$

proof: By induction on the tree t:

t=Null: We have to prove $z \in \text{delete}(cmp, x, Null) \Leftrightarrow (z \in Null \text{ and } z \neq x)$ but this is $False \Leftrightarrow (False \text{ and } z \neq x)$ which is trivial.

t=Leaf(v): We have to prove

$$z \in \text{delete}(cmp, x, Leaf(v)) \Leftrightarrow (z \in Leaf(v) \text{ and } z \neq x)$$

When $x = v$ this is $z \in Null \Leftrightarrow (z \in Leaf(v) \text{ and } z \neq x)$ which is $False \Leftrightarrow (z = v \text{ and } z \neq x)$ which is true because $x = v$.

When $x \neq v$ this is $z \in Leaf(v) \Leftrightarrow (z \in Leaf(v) \text{ and } z \neq x)$ which is $z = v \Leftrightarrow (z = v \text{ and } z \neq x)$, and this is true because $x \neq v$.

t=Node(a,v,b): Our induction hypotheses are

$$\text{ordered}(cmp, a) \Rightarrow (z \in \text{delete}(cmp, x, a) \Leftrightarrow (z \in a \text{ and } z \neq x)) \quad (1)$$
$$\text{ordered}(cmp, b) \Rightarrow (z \in \text{delete}(cmp, x, b) \Leftrightarrow (z \in b \text{ and } z \neq x)) \quad (2)$$

We are assuming $\text{ordered}(cmp, Node(a, v, b))$, and this gives

$$\text{ordered}(cmp, a) \quad (3)$$
$$\text{ordered}(cmp, b) \quad (4)$$
$$z \in a \Rightarrow z < v \quad (5)$$
$$z \in b \Rightarrow z > v \quad (6)$$

We have to prove

$$z \in \text{delete}(cmp, x, Node(a, v, b)) \Leftrightarrow (z \in Node(a, v, b) \text{ and } z \neq x)$$

.

case $x < v$

$$\begin{aligned}
z \in \text{delete}(cmp, x, Node(a, v, b)) &\Leftrightarrow \\
z \in Node(\text{delete}(cmp, x, a), v, b) &\Leftrightarrow \\
(z \in \text{delete}(cmp, x, a)) \text{ or } (z = v) \text{ or } (z \in b) &\Leftrightarrow \\
(z \in a \text{ and } z \neq x) \text{ or } (z = v) \text{ or } (z \in b) &\Leftrightarrow \\
((z \in a) \text{ or } (z = v) \text{ or } (z \in b)) \text{ and } (z \neq x) &\Leftrightarrow \\
(z \in Node(a, v, b)) \text{ and } (z \neq x) &
\end{aligned}$$

In the third step, to go from $z \in \text{delete}(cmp, x, a)$ to $(z \in a \text{ and } z \neq x)$, we use the induction hypothesis (1) and hypothesis (3). In the next step, we used the hypothesis (6) to get that since $x < v$, if either $z = v$ or $z \in b$ then $z \neq x$.

case $x > v$

$$\begin{aligned} z \in \text{delete}(cmp, x, Node(a, v, b)) &\Leftrightarrow \\ z \in Node(a, v, \text{delete}(cmp, x, b)) &\Leftrightarrow \\ (z \in a) \text{ or } (z = v) \text{ or } (z \in \text{delete}(cmp, x, b)) &\Leftrightarrow \\ (z \in a) \text{ or } (z = v) \text{ or } (z \in b \text{ and } z \neq x) &\Leftrightarrow \\ ((z \in a) \text{ or } (z = v) \text{ or } (z \in b)) \text{ and } (z \neq x) &\Leftrightarrow \\ (z \in Node(a, v, b)) \text{ and } (z \neq x) & \end{aligned}$$

In the third step, to go from $z \in \text{delete}(cmp, x, b)$ to ($z \in b$ and $z \neq x$), we use the induction hypothesis (2) and hypothesis (4). In the next step, we used the hypothesis (5) to get that since $x > v$, if either $z = v$ or $z \in a$ then $z \neq x$.

case $x = v$, $a = Null$

$$\begin{aligned} z \in \text{delete}(cmp, x, Node(a, v, b)) &\Leftrightarrow \\ z \in b &\Leftrightarrow \\ (False \text{ or } (z = v) \text{ or } (z \in b)) \text{ and } (z \neq x) &\Leftrightarrow \\ ((z \in a) \text{ or } (z = v) \text{ or } (z \in b)) \text{ and } (z \neq x) &\Leftrightarrow \\ (z \in Node(a, v, b)) \text{ and } (z \neq x) & \end{aligned}$$

This time, in the third step, we used the hypothesis (6) to see that when $z \in b$ then $z \neq x$.

case $x = v$, $a \neq Null$ We let $m, a' = \text{treemax}(d, a)$.

$$\begin{aligned} z \in \text{delete}(cmp, x, Node(a, v, b)) &\Leftrightarrow \\ z \in Node(a', m, b) &\Leftrightarrow \\ (z \in a' \text{ or } (z = m) \text{ or } (z \in b)) &\Leftrightarrow \\ (z \in a' \text{ or } (z = m) \text{ or } (z \in b)) \text{ and } (z \neq x) &\Leftrightarrow \\ (z \in a \text{ or } (z \in b)) \text{ and } (z \neq x) &\Leftrightarrow \\ ((z \in a) \text{ or } (z = v) \text{ or } (z \in b)) \text{ and } (z \neq x) &\Leftrightarrow \\ (z \in Node(a, v, b)) \text{ and } (z \neq x) & \end{aligned}$$

In the third step we used the ordering assumptions to see that if ($z \in a'$ or $(z = m)$ or $(z \in b)$) then $z \neq x$. Then, in the forth step, we used a lemma (that we have to prove!) about the members of $\text{treemax}(d, a)$.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.