CS3110 Spring 2016 Lecture 18: Binary Search Trees

Robert Constable

Note: Yaron Minsky from Jane Street and co-author of *Real World OCaml* will give a guest lecture on Tuesday, May 3.

Topics

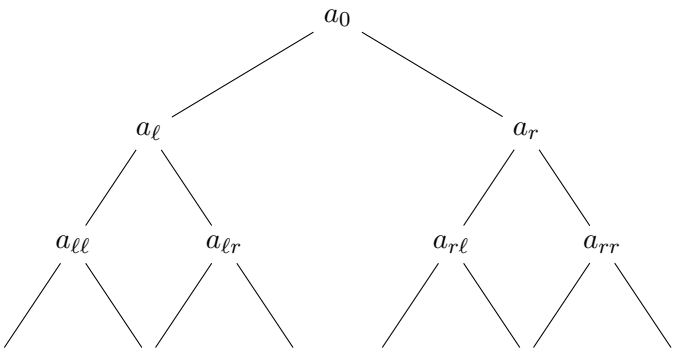
1. Remarks on "coordinate-free" convex hull algorithm.
   - Can have an n^2 algorithm, could we do better?
   - Donald Knuth wrote a short monograph on convex hull algorithms and coordinate free computational geometry: *Axioms and Hulls*, Springer-Verlag 1992.
2. Elements of Binary Search Trees (BST's).
   - Definition.
   - Use as key value store.
   - Insert function.
3. Discussion of *delete operation* in BST's.
   - The idea, examples.
   - Code.
   - Why is it correct?
   - Nuprl proof idea.
4. Splay trees, Red-Black trees, AVL trees and adjusting to the data statistics – brief remarks.

Elements of Binary Search Trees

OCaml definition:

`type` α `bst = Null | Leaf of` α `| Node of (`α `bst * `α `* `α `bst)`

Example:



This is an example of a *recursive* or *inductive* type. It is not defined in terms of pointers, and the algorithms to process operations on BST's are simply recursive functions.

We have already seen lists defined as recursive types. That experience generalizes to trees. We compute on trees using recursive functions. We prove properties using induction.

Remarks on proof assistants

The "pointer-free" approach to recursive types is motivated by *type theory* and the particular theories proposed by Nuprl and Coq in the early 1980's, building on ideas from Russell and Whitehead's book (3 volumes), *Principia Mathematica*. [2, 1]

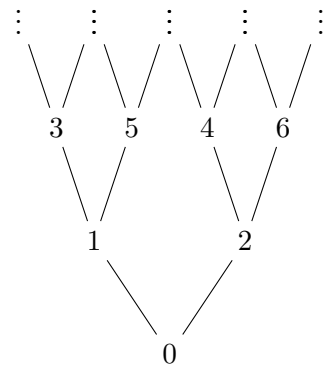
These mathematical theories examined "strange types" such as $T = \mathbb{N} + T \to T$, in OCaml notation: `t = L int | R (t` $\to$ `t)`.

The proof assistants also introduced *co-inductive* types such as *streams*:

α `strm =` α `* (`α `strm).`

An int stream could be this: (2, 3, 5, 7, 11, ...), *unending*.

Co-inductive trees are sometimes called *spreads*. Spreads look like this, possibly *unbounded*, upward:



Binary Search Tree "mechanics"

The typical use of BST's is to store values and provide rapid access to them. A common paradigm is a "key-value store," something like a dictionary, with every word as the *key*. We store *definitions* at the key.

In these lecture notes we use the example provided by Mark Bickford, storing pairs of an integer and a real; elements of the type `(int * real)`. The integer is a key for looking up the value, e.g. (3, cosine(3)), (4, sine(4)), ...

Mark uses a *comparison operator* `cmp : E → E → int` where

`cmp a b < 0` means $a < b$
`cmp a b > 0` means $a > b$
`cmp a b = 0` means a is equivalent to b.

To say `cmp a b = 0` means that we consider a and b to be "equivalent"

Proving properties of a binary search tree

It turns out to be quite difficult to find a clear readable and rigorous account of why the common operations on BST's, *insert*, *look-up*, and *delete* are correct for functional languages. Marck Bickford, whom you have met, has proved these operations correct using Nuprl. We have produced the notes so that you can understand such a proof. These notes are especially important for *BST delete*, a subtle operation.

Mark's proof can be found here (www.nuprl.org/wip/Standard2/tree_1/index). This is a directory of tree related theories. Proofs related to binary search trees come after 'Comment: tree_1_end'. The delete proof can be found at 'Definition: bs_tree_delete'. As can be seen from this page, there are many components - definitions, lemmas, supporting proofs - that are needed before we are able to prove the delete function.

These lecture notes include expository material that Mark helped prepare, both for this lecture and for help reading the Nuprl proof.

Reasoning about BST operators

We define the notion of an ordered tree recursively by cases. We take the tree to be mathematically defined as

```
bst(E) = null | leaf(E) | node bst(E) E bst(E).
```

Define $x \in T$ for t a bst.

$x \in$ null	is false
$x \in$ leaf(v)	iff $x = v$
$x \in$ node(a, v, b)	iff $(x \in a)$ or $(x = v)$ or $(x = b)$

We say that a tree t is *ordered* by the comparison operator `cmp`, iff

ordered(cmp, null)	is true
ordered(cmp, leaf(v))	is true
ordered(cmp, node(a, v, b))	ordered(cmp,a) & ordered(cmp, b) and for all $x \in a$, cmp$(x, v) < 0$ and for all $x \in b$, cmp$(x, v) > 0$

To define the delete operation precisely enough to prove that it is correctly implemented we need to know that the *rightmost* element of the tree is the largest. (See the diagram.)

Examples of deletion done in lecture:

D1. Delete a leaf, e.g. 49, 69, 74.

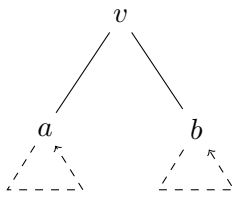
D2. Delete a one-child node, e.g. 68, 58.

D3. Delete a two-child node, e.g. 57, 52.

We use the strategy of the *maximum of the left subtree* in Mark's proof/algorithm. One could also use the minimum of the right subtree.

Here is the delete operation using the maximum of the left subtree. *We are deleting x from tree t, using comparison cmp.*

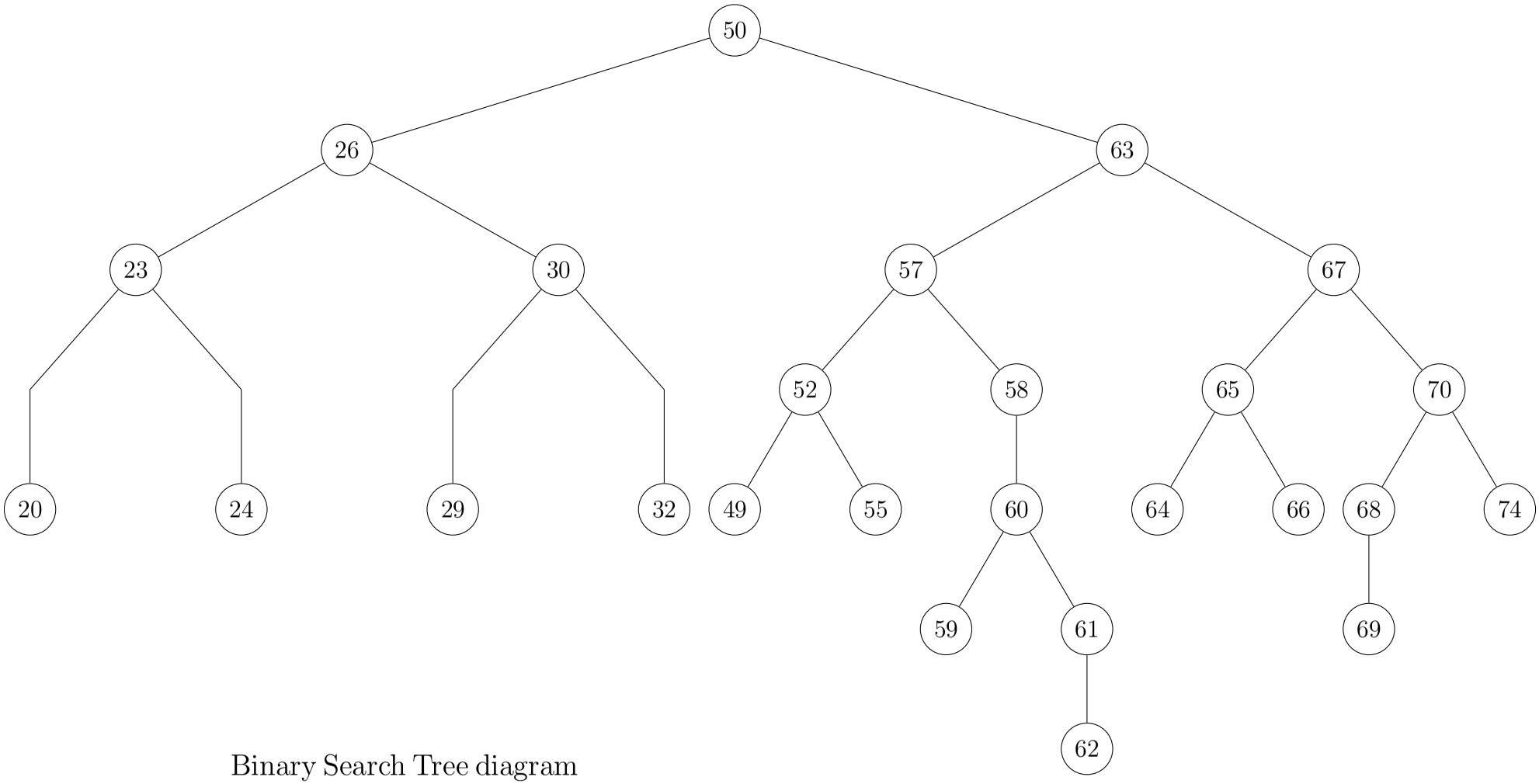
```
let rec delete cmp x t
  match t with
  Null ->  Null
  | Leaf(v) -> cmp x v = 0 then Null else Leaf(v)
  | node(a,v,b) ->
     if cmp x v < 0 then Node(delete cmp x a, v, b)     (*delete in left subtree*)
     else if cmp x > 0 then Node(a, v, delete cmp x b)  (*delete in right subtree*)
     else if isnull a then b
     else let m, a' = treemax v a in Node(a', m, b)
```



References

[1] C. A. R. Hoare. Notes on data structuring. In *Structured Programming*. Academic Press, New York, 1972.

[2] A.N. Whitehead and B. Russell. *Principia Mathematica*, volume 1, 2, 3. Cambridge University Press, 2nd edition, 1925–27.



Binary Search Tree diagram

Appendix: Notes and code from Mark Bickford

CS3110 Spring 2016: Binary Search Tree (BST) 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 $E \to E \to int$ such that $cmp\ a\ b < 0$ means that a is "less than" b and $cmp\ a\ b > 0$ means that a is "greater than" b while $cmp\ a\ b = 0$ 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 "rightmost" 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:

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

OCaml code

```
type 'a bstree  = Null | Leaf of 'a | Node of ('a bstree) * 'a * ('a bstree)

let isnull t = match t with
                Null -> true
                | Leaf(v) -> false
                | Node(a,v,b) -> false
(* lookup x in tree t using comparison cmp: return option *)
let rec lookup cmp x t =
  match t with
  Null -> None
  | Leaf(v) -> if cmp x v = 0 then Some(v) else None
  | Node(a,v,b) -> let c = cmp x v in
                   if c < 0 then lookup cmp x a
                   else if c > 0 then lookup cmp x b
                   else Some(v)

let rec treemax d t =
  match t with
  Null -> d, Null
  | Leaf(v) -> v, Null
  | Node(a,v,b) -> if isnull b then v,a else
                     let m,b' = treemax d b in
                     m, Node(a, v, b')

let rec delete cmp x t =
  match t with
  Null -> Null
  | Leaf(v) -> if cmp x v = 0 then Null else Leaf(v)
  | Node(a,v,b) ->
      if cmp x v < 0 then Node(delete cmp x a, v, b)
      else if cmp x v > 0 then Node(a, v, delete cmp x b)
      else if isnull a then b
      else let m,a' = treemax v a in
           Node(a', m, b )
```

```
let rec insert cmp x t =
  match t with
  Null -> Leaf(x)
  | Leaf(v) -> let c = cmp x v in
               if c < 0 then Node(Leaf(x),v, Null)
               else if c > 0 then Node(Null,v,Leaf(x))
               else Leaf(x)
  | Node(a,v,b) ->
      let c = cmp x v in
      if c < 0 then Node(insert cmp x a, v, b)
      else if c > 0 then Node(a, v, insert cmp x b)
      else Node(a,x,b)

(* For (int bstree) we can use the comparison (fun x y -> x-y) *)
let insert_int = insert (fun x y -> x-y)
let delete_int = delete (fun x y -> x-y)
let lookup_int = lookup (fun x y -> x-y)
```