Binary Search Trees: Inserting and Deleting
(Chapters 7 & 8)
Announcements

• Read Chapters 7 & 8
  – Binary Search Trees

• Next up: generics and first-class functions

• HW2 due *Tuesday* at midnight
Recap: Ordered Trees

Big idea: find things faster by searching less
Key Insight:

*Ordered data can be searched more quickly*

- This is why telephone books are arranged alphabetically
- Requires the ability to focus on (roughly) half of the current data
A binary search tree (BST) is a binary tree with some additional invariants:

- **Node(lt, x, rt)** is a BST if
  - *lt* and *rt* are both BSTs
  - all nodes of *lt* are < *x*
  - all nodes of *rt* are > *x*

- **Empty** is a BST

The BST invariant means that container functions can take time proportional to the **height** instead of the **size** of the tree.
Searching a BST

(* Assumes that t is a BST *)

let rec lookup (t:tree) (n:int) : bool =
begin
match t with
| Empty -> false
| Node(lt,x.rt) ->
  if x = n then true
  else if n < x then lookup lt n
  else lookup rt n
end

• The BST invariants guide the search.
• Note that lookup may return an incorrect answer if the input is not a BST!
  – This function assumes that the BST invariants hold of t.
Manipulating BSTs

Inserting an element

\[
\text{insert} : \text{tree} \rightarrow \text{int} \rightarrow \text{tree}
\]

"insert \ t\ x" returns a new tree containing \( x \) and all of the elements of \( t \)
Inserting into a BST

• Challenge: can we make sure that the result of insert really is a BST?
  – i.e., the new element needs to be in the right place!

• Payoff: we can build a BST containing any set of elements
  – Starting with Empty, apply insert repeatedly
  – If insert preserves the BST invariants, then any tree we get from it will be a BST by construction
    • No need to check!
  – Later: we can also “rebalance” the tree to make lookup efficient (NOT in CIS 120; see CIS 121)

First step: find the right place...
Inserting a new node: (insert t 4)
Inserting a new node: (insert t 4)
Inserting Into a BST

(*) Insert n into the BST t *)
let rec insert (t:tree) (n:int) : tree =
    begin match t with
    | Empty -> Node(Empty,n,Empty)
    | Node(lt,x,rt) ->
        if x = n then t
        else if n < x then Node(insert lt n, x, rt)
        else Node(lt, x, insert rt n)
    end

• Note the similarity to searching the tree.
• Assuming that t is a BST, the result is also a BST. (Why?)
• Note that the result is a new tree with (possibly) one more Node; the original tree is unchanged

Critical point!
Manipulating BSTs

Deleting an element

delete : tree -> int -> tree

"delete t x" returns a tree containing all of the elements of t except for x
Deletion – No Children: (delete t 3)
Deletion – No Children: \( (\text{delete t 3}) \)

If the node to be deleted has no children, simply replace it by the Empty tree.
Deletion – One Child: (delete t 7)
Deletion – One Child: (delete t 7)

If the node to be delete has one child, replace the deleted node by its child.
Deletion – Two Children: (delete t 5)
Deletion – Two Children: \((\text{delete } t \ 5)\)

If the node to be delete has two children, promote the maximum child of the left tree.
How to Find the Maximum Element?

What is the max element of this subtree?
How to Find the Maximum Element?

Just for fun, how do we find the max element of the whole tree?
Tree Max

let rec tree_max (t:tree) : int =
begin
match t with
| Node(_,x,Empty) -> x
| Node(_,_,rt) -> tree_max rt
| _ -> failwith "tree_max called on Empty"
end

• BST invariant guarantees that the maximum-value node is farthest to the right

• Note that `tree_max` is a partial* function
  – Fails when called with an empty tree

• Fortunately, we never need to call `tree_max` on an empty tree
  – This is a consequence of the BST invariants and the case analysis done by the delete function

* Partial, in this context, means “not defined for all inputs”.
Code for BST delete

bst.ml
Deleting From a BST

```plaintext
let rec delete (t: tree) (n: int) : tree =
begin match t with
| Empty -> Empty
| Node(lt, x, rt) ->
  if x = n then
    begin match (lt, rt) with
      | (Empty, Empty) -> Empty
      | (Node _, Empty) -> lt
      | (Empty, Node _) -> rt
      | _ -> let m = tree_max lt in
        Node(delete lt m, m, rt)
    end
  else if n < x then Node(delete lt n, x, rt)
  else Node(lt, x, delete rt n)
end
```

See bst.ml
Subtleties of the Two-Child Case

- Suppose Node(lt,x,rt) is to be deleted and lt and rt are both themselves nonempty trees.

- Then:
  1. There exists a maximum element, m, of lt (Why?)
  2. Every element of rt is greater than m (Why?)

- To promote m we replace the deleted node by:
  Node(delete lt m, m, rt)
  - I.e. we recursively delete m from lt and relabel the root node m
  - The resulting tree satisfies the BST invariants
if we insert a label $n$ into a BST and then immediately delete $n$, do we always get back a tree of exactly the same shape?

When poll is active, respond at PollEv.com/120fall18

Text **120FALL18** to **22333** once to join

yes

no
If we insert a label n into a BST and then immediately delete n, do we always get back a tree of exactly the same shape?

1. yes
2. no

Answer: no  (what if the node was in the tree to begin with?)
If we insert a value \( n \) into a BST that *does not* already contain \( n \) and then immediately delete \( n \), do we always get back a tree of exactly the same shape?

- **Yes**
- **No**
If we insert a value \( n \) into a BST that does not already contain \( n \) and then immediately delete \( n \), do we always get back a tree of exactly the same shape?

1. yes
2. no

Answer: yes
If we delete \( n \) from a BST (containing \( n \)) and then immediately insert \( n \) again, do we always get back a tree of exactly the same shape?
If we delete $n$ from a BST (containing $n$) and then immediately insert $n$ again, do we always get back a tree of exactly the same shape?

1. yes
2. no

Answer: no  (e.g., what if we delete the item at the root node?)
BST Performance

- **lookup** takes time proportional to the *height* of the tree.
  - not the *size* of the tree (as it did with *contains* for unordered trees)

- In a *balanced tree*, the lengths of the paths from the root to each leaf are (almost) *the same*.
  - no leaf is too far from the root
  - the height of the BST is minimized
  - the height of a balanced binary tree is roughly $\log_2(N)$ where $N$ is the number of nodes in the tree

![balanced tree](image1)

![unbalanced tree](image2)