Recap: Binary Search Trees (BST)

- A BST is a binary tree with additional invariants:
  - Empty is a BST
  - 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

- A BST is a way of representing a set of integers
- The invariant enables efficient lookup
  - Only need to search one path from the root (not the whole tree)
- Last time: lookup, is_bst, insert, started delete

Announcements

- Homework 2 is due tonight at 11:59:59pm.
- Homework 3 will be available on the web soon.
  - Due next Wednesday
  - Practice with BSTs, generic functions, abstract datatypes
Deletion – No Children: (delete t 3)

3 < 5

Deletion – No Children: (delete t 3)

5

3 > 1

Deletion – One Child: (delete t 7)

7 > 5

Deletion – One Child: (delete t 7)

If the node to be deleted has no children, simply replace it by the Empty tree.
Subtleties of the Two-Child Case

• Suppose Node(lt, x, rt) is to be deleted and lt and rt are both themselves nonempty BSTs.
  • Then:
    – There exists a maximum element, m, of lt (why?)
    – m is smaller than every element of rt (why?)

• To promote m we replace the deleted node by:
  Node(delete lt m, m, rt)
  – i.e. we recursively delete m from lt
  – Note the resulting tree satisfies the BST invariants

• Question: will this always work?

Tree_max: A partial function

let rec tree_max (t:tree) : int =
begin match t with
  | Empty -> ????
  | Node(lt, x, rt) -> ...
end

• Problem: tree_max isn’t defined for all binary trees.
  – In particular, it isn’t defined for the empty binary tree
  – Technically, tree_max is a partial function
• What to do?
**Solutions to Partiality: Option 1**

- Return a *default or error value*
  - e.g. define `tree_max Empty` to be `-1`
  - Error codes used often in C programs; null used often in Java
- But...
  - What if -1 (or whatever default you choose) really *is* the maximum value?
  - Can lead to many bugs if the default or error value isn’t handled properly by the callers.

- Defaults should be avoided if possible

---

**BST Invariants and tree_max**

- For delete, 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.
- So: we can write `tree_max` *assuming* that the input tree is a nonempty BST:

```ocaml
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
```

- Note: BST invariant is used because it guarantees that the maximum valued node is farthest to the right

---

**Solutions to Partiality: Option 2**

- Abort the program:
  - In OCaml: `failwith "an error message"`
- Whenever it is called, `failwith` aborts the program and reports the error message it is given.
- This solution to partiality is appropriate whenever you *know* that a certain case is impossible.
  - Often happens when there is an invariant on a datastructure
  - The compiler isn’t smart enough to figure out that the case is impossible...
  - `failwith` is also useful to “stump out” unimplemented parts of your program.

*There are a few other ways to deal with partiality (using datatypes or exceptions) that we’ll see later in the course*

---

**BST Delete**

```ocaml
(* Deletes n from the BST tree t *)
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
```
Balanced Trees

- Lookup, insert, and delete take time at worst proportional to the height of the tree.
  - How to minimize the height?
  - A balanced tree is one in which all leaves have exactly distance k or distance k+1 from the root.
- Our insert function can yield very unbalanced trees:
  - See CIS121 for ways of improving search trees to maintain balance...

Demo: Binary Search Trees

Generic Functions and Data

Structurally Identical Functions

- Observe: many functions on lists, trees, and other datatypes don’t depend on the contents, only on the structure.
- Compare: length for “int list” vs. “string list”

```ocaml
let rec length1 (l:int list) : int = begin match l with
    | [] -> 0
    | _::tl -> 1 + (length1 tl)
end
```

```ocaml
let rec length2 (l:string list) : int = begin match l with
    | [] -> 0
    | _::tl -> 1 + (length2 tl)
end
```

The functions are identical, except for the type annotation for l.
Notation for Generic Types

- OCaml provides syntax for generic function types
  - lets us share the common definition:
    ```ocaml
    let rec length (l:'a list) : int = begin
      match l with
      | [] -> 0
      | _::tl -> 1 + (length tl)
    end
    ```

- Notation: ‘a is a type variable; it indicates that the function length can be used on a list for any type t.

Examples:
- length [1;2;3] use length on an int list
- length ["a";"b";"c"] use length on a string list

Generic List Append

- The return type can also be generic — in this case the result is of the same type as the inputs.
- The return type can also be generic — in this case the result is of the same type as the inputs.

```ocaml
let rec append (l1:'a list) (l2:'a list) : 'a list = begin
  match l1 with
  | [] -> l2
  | h::tl -> h::(append tl l2)
end
```

Generic Zip

- Functions can operate over multiple generic types.
- Pattern matching works over generic types.

```ocaml
let rec zip (l1:'a list) (l2:'b list) : ('a*'b) list = begin
  match (l1,l2) with
  | (h1::t1, h2::t2) -> (h1,h2)::(zip t1 t2)
  | _ -> []
end
```

- Distinct type variables can be instantiated differently:
  - zip [1;2;3] ["a";"b";"c"]
  - Here, ‘a instantiated to int, ‘b to string
  - Result is the (int * string) list:
    - [(1,"a");(2,"b");(3,"c")]

User-defined Generic Datatypes

- Recall our integer tree type:
  ```ocaml
type tree =
  | Empty
  | Node of tree * int * tree
  ```

- We can define a generic version by adding a type parameter, like this:
  ```ocaml
type 'a tree =
  | Empty
  | Node of ('a tree) * 'a * ('a tree)
  ```

- Parameter ‘a used here
- Note that the recursive uses also mention ‘a
Function Types

- In OCaml, the type of functions from input $t$ to output $u$ is written:
  
  $$ t \rightarrow u $$

- Functions with multiple arguments use multiple arrows
- Here are some examples we have already seen:

  ```
  size : tree -> int
  hamming_distance : helix -> helix -> int
  acids_of_helix : helix -> acids list
  length : 'a list -> int
  zip : 'a list -> 'b list -> ('a*'b) list
  lookup : tree -> int -> bool
  ```

A Design Problem

- Suppose that at Step 2 (“Formalize the Interface”) we decide that we need to process a set of values.
  - Examples: set of students in a class, set of coordinates in a graph, set of answers to a survey, set of samples from Gulf oil spill probes, ...

- Given what you know of OCaml, what would you do?

Abstract Datatypes

- Example:
  - concrete ‘set’ representation – the implementation – is a list or a tree
  - abstract interface defines the operations in terms of ‘set’

- The interface restricts how other parts of the program can interact with the data.
  - The other parts of the program can’t break any invariants that are being maintained behind the interface.
  - It is possible to change the implementation of the abstract datatype without changing the rest of the program.

BIG IDEA: Hide the concrete representation of a datatype behind an abstract interface.
Example Module Interface: set.mli

```ocaml
type 'a set
val empty : 'a set
val is_empty : 'a set -> bool
val member : 'a -> 'a set -> bool
val insert : 'a -> 'a set -> 'a set
val union : 'a set -> 'a set -> 'a set
val remove : 'a -> 'a set -> 'a set
val filter : ('a -> bool) -> 'a set -> 'a set
val to_list : 'a set -> 'a list
```

The keyword 'val' indicates values that must be defined.

- OCaml lets you write interfaces (as above) in a .mli file
- The corresponding implementation goes in the .ml file

Module Implementation: set.ml

```ocaml
type 'a tree = ...
type 'a set = 'a tree

let empty : 'a set = Empty
let is_empty (s:'a set) : bool =
  begin match s with
  | Empty -> true
  | _ -> false
  end
...
```

- The implementation has to include all of the values
  - It can contain more functions and type definitions (e.g. auxiliary functions)
  - The types of the provided implementations must match the interface