Programming Languages and Techniques (CIS120)

Lecture 8
February 1, 2016
BST Delete
Generics
(Chapters 7 & 8)
Announcements

• Read Chapters 7 & 8 (BSTs, generics) of lecture notes
• Read Chapter 9 of lecture notes (Higher-order functions)
• HW2 due tomorrow at midnight

• My office hours: Today 3:30 – 5:00
Big idea: find things faster by searching less
Trees as Containers

• Like lists, binary trees aggregate data

```haskell
type tree =
  | Empty
  | Node of tree * int * tree
```

• As we did for lists, we can write pure functions for working with this container
  – lookup: determine whether the tree contains a particular element
  – insert: return a new tree containing a particular element
  – delete: return a new tree with a particular element removed (if present)
Binary Search Trees

• 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.
Constructing BSTs from other BSTs

Deleting an element

delete :: tree -> int -> tree
Deletion – No Children: (delete t 3)
Deletion – No Children: (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)
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: (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: A partial* function

```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
```

- We never call tree_max on an empty tree
  - This is a consequence of the BST invariants and the case analysis done by the delete function
- BST invariant guarantees that the maximum-value node is farthest to the right

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

trees.ml
Deleting From a BST

```ml
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
```
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 is in the tree
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?

1. yes
2. no

Answer: no, what if we delete the root?
Wow, implementing BSTs took quite a bit of typing...
Do we have to repeat it all again if we want to use BSTs containing strings, or characters, or floats?

or

*How not to repeat yourself, Part I.*
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`”

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

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

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

• OCaml provides syntax for functions with *generic* types

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

• Notation: `'a` is a *type variable*; the function `length` can be used on a `t 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`
let rec append (l1:'a list) (l2:'a list) : 'a list =
begin
match l1 with
| [] -> l2
| h::tl -> h::(append tl l2)
end

Note that the two input lists must have the same type of elements.

The return type is the same as the inputs.

Pattern matching works over generic types!

In the body of the branch:
  h has type 'a
  tl has type 'a list
Distinct type variables can be instantiated differently:

```
zip [1;2;3] ["a";"b";"c"]
```

Here, 'a' is instantiated to int, 'b' to string

Result is

```
[(1,"a");(2,"b");(3,"c")]
```
of type (int * string) list
• 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`
User-Defined Generic Datatypes

- BST operations can be generic too; only change is to the type annotation

(* Insert \( n \) into the BST \( t \) *)

```ocaml
let rec insert (t: 'a tree) (n: 'a) : 'a 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
```

Equality and comparison are generic — they work for any type of data too.
Does the following function typecheck?

```
let f (l : 'a list) : 'b list =
begin
  match l with
  | []    -> true::l
  | _::rest -> 1::l
end
```

1. yes
2. no

Answer: no, even though the return type is generic, the two branches must agree (so that ‘b can be consistently instantiated).
Does the following function typecheck?

```plaintext
let f (x : 'a) : 'a =
  x + 1

;; print_endline (f "hello")
```

1. yes
2. no

Answer: no, the type annotations and uses of f aren’t consistent
First-class Functions

Higher-order Programs
or
How not to repeat yourself, Part II.