1 Binary Search Trees

1.1 Searching a BST

2 OCaml Notes

2.1 Mutual Recursion

2.2 Trace

3 Matching and Extracting

4 Summary

Objectives

By the end of class you’ll:

- know what a BST is, and the runtime for looking something up in one
- how to declare mutually recursive functions in OCaml
- how to use tracing in OCaml
- have a better idea of how to approach the ‘Any’ case in extract

1 Binary Search Trees

A Binary Search Tree (BST) is a special type of binary tree. The data in all of its nodes are comparable, and as such, the tree is organized to take full advantage of this.

We can use the following definition of binary trees in OCaml:

```ocaml
type 'a btree =
  | Leaf
  | Node of 'a btree * 'a * 'a btree
```

For example,

```ocaml
Leaf
Node (Leaf, 17, Leaf)
Node (Leaf, 17, Node (Leaf, 18, Leaf))
Node (Node (Leaf, 17, Leaf), 22, Node (Leaf, 18, Leaf))
```
If a binary tree is a leaf, then it is automatically a BST. If it is a node, then the following left and right invariants, respectively, must hold:

- Every value in a node’s left subtree must be less than the node’s value; and
- Every value in a node’s right subtree must be greater than the node’s value.

Figure 1 depicts two examples of binary search trees. Specifically, they store the set \{1, 3, 4, 5, 6, 7, 11\}. The representation of a collection of data by a BST is not unique.

**Question:** Can we swap 4 and the 6 in the tree on the left?

**Answer:** No, that would violate the invariants. 6 is greater than 5, so it cannot be in 5’s left subtree. Similarly, 4 is less than 5, so it cannot be in 5’s right subtree.

### 1.1 Searching a BST

Searching for something in a BST for a datum can be described, as follows:

1. If the tree is a leaf, return false.
2. Otherwise, when the tree is a node,
   (a) If the datum equals the node’s value, return true.
   (b) If the datum is less than the node’s value, continue your search in the left subtree.
   (c) If the datum is greater than the node’s value, continue your search in the right subtree.

This description translates directly into the following is_member procedure:

```scheme
(* Input : a datum, datum, and a btree
 * Output : a boolean indicating whether datum is in the input btree *)
let rec is_member (datum : 'a) : 'a btree -> bool =
| Leaf      -> false
| Node (left, x, right) when (datum < x) -> is_member datum left
| Node (left, x, right) when (datum > x) -> is_member datum right
| Node _    -> true
```
Let’s analyze this procedure. Observe that it visits at most one element at each level of the BST. Hence, its run time is $O(\text{depth})$. But what is the depth? Let $n$ be the number of elements in the BST. In the worst-case, the depth is precisely $n$, making the run time $O(n \mapsto n)$.

In practice, however, searching a BST tends to be much faster. We can define a **balanced** BST (informally) as one in which every leaf node resides at approximately the same depth. The depth of a balanced BST containing $n$ elements is $\log n$, making the runtime of the search procedure only $O(n \mapsto \log n)$.

# 2 OCaml Notes

## 2.1 Mutual Recursion

Writing two recursive functions that referenced each other in Racket was pretty straightforward. You wrote both functions like normal, ran your program, and it would work. However, things are different in OCaml.

Let’s consider two functions, called `my-odd` and `my-even`:

```ocaml
let rec my-odd (n:int) : bool = function
| 0 -> false
| n when (n > 0) -> my-even (n-1)
| _ -> failwith "not a natural number"

let rec my-even (n:int) : bool = function
| 0 -> true
| n when (n > 0) -> my-odd (n-1)
| _ -> failwith "not a natural number"
```

Together, these functions will return `true` when $n$ is even, and `false` if $n$ is odd. Or rather, that’s what they *should* do.

The functions are not visible to each other in their environments, so when `my-odd` tries to call `my-even`, OCaml won’t know what to do, and will subsequently crash. So, if we want to use mutually recursive functions in OCaml, we have to use the following syntax:

```ocaml
let rec
my_odd (n:int):bool = match n with
| 0 -> false
| n when (n > 0) -> my_even (n-1)
| _ -> failwith "not a natural number"
and
my_even (n:int):bool = match n with
| 0 -> true
| n when (n > 0) -> my_odd (n-1)
| _ -> failwith "not a natural number";;
```

By using the keyword `and`, we can ensure that both bindings are placed in an environment where both are visible.
2.2 Trace

If you want to see the input and output for every time a specific function in your program is called, OCaml can help you out with that! Before the code where you want to trace the input/output, write:

```ocaml
#trace <function-name>;;
```

Where `<function-name>` is the function that you want to trace.

To stop tracing the function, write:

```ocaml
#untrace <function-name>;;
```

Where `<function-name>` is the function that you were tracing.

3 Matching and Extracting

Spike walked the class through an example of a matching function, called `compare`, and expanded it into a simpler version of `extract`, called `matches`. For more information, see the slides!

4 Summary

Ideas

- Binary search trees (BSTs) are sorted binary trees. When they’re balanced, the runtime to search for something is in $O(n \mapsto \log(n))$.

Skills

- Use the keyword `and` to define mutually recursive functions
- You can trace a function to see the input/output of whenever that function is called. It’s helpful for debugging recursive functions

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs017/feedback](http://cs.brown.edu/courses/cs017/feedback)