Lecture 23: Currying, Big-O, Records, and BST

10:00 AM, Oct 28, 2019

Contents

Objectives

By the end of class you'll:

- know what a BST is, and the runtime for looking something up in one
- Prove some Big-O lemmas
- know what a record is and learn currying

1 Multi-parameter Procedures

We've written two procedures (one anonymous and one user-defined), each of which adds 17 to its parameter. Similarly, we could write a procedure (or two) that adds 18 to its parameter. Or smelling abstraction, we could write a more general procedure that consumes *two* parameters and produces their sum. Opting for anonymity, here's how that would look:

```
fun (x, y) => x + y
```

Now for the tricky question. What is the type of this anonymous procedure? This question is tricky because this procedure appears to consume two parameters, but Reason procedures can only consume one parameter!

> **Question:** Is the type of this anonymous procedure `int * int ->int`?
>
> **Answer:** I'm afraid not. That would be the type of a procedure that consumes a 2-element tuple as its sole input:
>
> ```
> fun (x, y) -> x + y
> ```

What's going on here is that

```
fun x y -> x + y
```

is actually syntactic sugar for

```
fun x -> (fun y -> x + y)
```

As you can, see, we have one procedure that consumes an `int` and produces a procedure. The produced procedure in turn consumes an `int` and produces an `int`.

This means of defining multi-parameter procedures is called **c**urrying . [1]

With it, we nest one-parameter procedures, one for each parameter we need. That way, all are bound within the body of the innermost procedure just as they would be bound in the body of a multi-parameter procedure in Racket.

The type of a curried procedure follows from its definition. For example, the type of our anonymous procedure is:

```
int -> (int -> int)
```

As the definition of `add` consists of two nested **`fun`** expressions, the type of `add` consists of two nested -> type-expressions. This correspondence is the key to understanding the types of curried procedures, which is why it is important to remember the way multi-parameter **`fun`** expressions desugar.

Let's see how function application works:

```
(((fun x -> (fun y -> (fun z -> x + y + z))) 1) 2) 3
=> ((fun y -> (fun z -> 1 + y + z)) 2) 3
=> (fun z -> 1 + 2 + z) 3
=> 1 + 2 + 3
=> 6
```

Here is our same adding procedure from before, applied to three arguments. As you can see, each application eliminates one of our nested procedures until we just have 1 + 2 + 3 left, which evaluates normally.

Writing all those parentheses seems a bit verbose, right? Even in Racket you don't need to introduce a new pair of parentheses for each and every argument. Well fortunately on associates some things so that we need far less parentheses in practice.

The -> is right associative: i.e., it binds most tightly to the right, so that
`fun` `x ->` **`fun`** `y ->x + y` means
`fun` `x ->(` **`fun`** `y ->x + y)` and
`int ->int ->int` means
`int ->(int ->int)`.
On the other hand, procedure application is left associative: i.e., it binds most tightly to the left,

[1] Currying is named for the logician Haskell Curry, who studied under David Hilbert. Curry is one of very few mathematicians to be remembered by naming something for both his first (the functional programming language, Haskell) and last name.

> so that
> `add` 17 1 means
> (`add` 17)1.
>
> We can now rewrite our example from above as:
>
> ```
> (fun x -> fun y -> fun z -> x + y + z) 1 2 3
> => (fun y -> fun z -> 1 + y + z) 2 3
> => (fun z -> 1 + 2 + z) 3
> => 1 + 2 + 3
> => 6
> ```
>
> Much less cluttered!

It turns out that just as Reason will curry our procedure expression for us, it also will curry our procedure definitions. How convenient! For example, `add` defined to be our procedure from before,

```
let add (x : int) (y : int) : int = x + y
```

is syntactic sugar for

```
let add : int -> int -> int = fun x -> (fun y -> x + y)
```

Here are two more examples, where we drop the parentheses for readability.

This definition:

```
let foo (x : int) (y : int) (z : int) : int = (x + y) * z
```

is syntactic sugar for this one:

```
let foo : int -> int -> int -> int =
  fun x -> fun y -> fun z -> (x + y) * z
```

And this definition:

```
let bar (a : float) (b : float) (c : float) (d : float) : float =
  (a -. (b /. c)) ** d
```

is syntactic sugar for this one:

```
let bar : float -> float -> float -> float -> float =
  fun a -> fun b -> fun c -> fun d -> (a -. (b /. c)) ** d
```

Since Reason has no notion of a multi-parameter procedure, all multi-parameter procedures are either curried, or (more rarely) written with a single-parameter procedure that consumes a tuple.

Currying gives Reason a certain theoretical elegance. Whereas other languages have two different type errors: applying the wrong number of arguments to a function, and applying arguments to a non-function, Currying means the former becomes the latter. More generally, currying allows Reason to be a smaller and simpler language than it would be otherwise.

Infix Tricks For every arithmetic infix procedure application in Reason, there is corresponding prefix procedure application. In particular, for an infix procedure ⟨*op*⟩, the corresponding prefix procedure is spelled (⟨*op*⟩). For example, the following two Reason expressions are equivalent:

```
1 + 17
=> 18

( + ) 1 17
=> 18
```

If an infix operator consumes two arguments, one of type ⟨*left-type*⟩ and the other of type ⟨*right-type*⟩, and returns a value of type ⟨*val-type*⟩, then the type of that operator is *not* written as ⟨*left-type*⟩ * ⟨*right-type*⟩ -> ⟨*val-type*⟩; rather, it is written in its curried form as ⟨*left-type*⟩ -> ⟨*right-type*⟩ -> ⟨*val-type*⟩. For example, the type of (+) is `int ->int ->int`.

Note: You must enter (+) if you want Reason to return the type of +.

The infix form is shorter to type and more closely resembles standard mathematical notation, so it is used more frequently. The prefix form is useful when you want to apply one procedure to another (i.e., you want the latter to be an actual argument of the former)—for example, if you wanted to fold addition across a list of integers you would apply the fold procedure to (+).

> **Question:** Can you use this trick to make `::` into a prefix operator?
>
> **Answer:** No, I'm afraid not. But you can do this: `let cons x y = x ::y`.

2 Records

ReasonML tuples allow us to combine a group of values of various types into a single entity. Analogous to Racket structures, ReasonML **r**ecords enable us to name these values.

> Tuples and records are sometimes called **a**nd types, or **c**onjunctive types, since values of these types compound multiple other types.

Here is an example of a record type definition, followed by an annotated instance of that type:

```
(* Name and nickname database *)
type entry = {
name: string,
nickname: string
};

(* Examples of entry *)
let pp = {name : "Porcellus Smith", nickname: "Porky Pig"};
```

Although you can select field in a record by name, pattern matching is generally preferred to selection in ReasonML. Pattern matching is the subject of our next lecture.

It is straightforward to combine record types with variant types. Here is another record type:

```
(* A point in 2 dimensions is characterized by x and y coordinates *)
type point\_2d = {x : int; y : int};

(* Examples of point\_2d *)
({x = (-1); y = (-2)} : point\_2d)
let origin2 = ({x = 0; y = 0} : point\_2d);
```

And here is a variant type of points in either 2d or 3d space:

```
(* A point is either a 2d or a 3d point *)
type point =
  | Point2d of point\_2d
  | Point3d of point\_3d;

(* Examples of point *)
Point2d origin2;
Point3d origin3;
```

Typically, in CS 17, it is sufficient to use tuples, rather than records. This is because there are usually only two or three distinct fields in the data, and what each field represents is usually clear from context. However, when the size of a tuple is much more than 3, or if it would improve readability if fields were named, you should consider storing your data as records.

3 Theorems for proving Big-O class

We will now prove some useful theorems that will help you find the Big-O category of your conjectures.

3.1 Basic Theorems

Theorem 1. *If* $f \in O(h)$ *and* $g \in O(h)$*, then* $f + g \in O(h)$

Let's try an example again,

For $n > 15$, assume $f(n) \leq 11 \times h(n)$.
For $n > 37$, assume $g(n) \leq 6 \times h(n)$.
Then for $n > max(18, 37)$, $(f + g)(n) \leq 17h(n)$.
We can observe that here, $17 = 11 + 6 = c_1 + c_2$. We can now write the general proof,

Proof. For $n > M_1$, we have $f(n) \leq c_1 h(n)$.
For $n > M_2$, we know that $g(n) \leq c_2 h(n)$.
Then, for $n > max(M_1, M_2)$, we have $f(n) \leq (c_1 + c_2)h(n)$.
Hence, $f + g \in O(h)$.
To summarize, if $[M_1, c_1]$ and $[M_2, c_2]$ show that $f \in O(h)$ and $g \in O(h)$, then $[max(M_1, M_2), c_1 + c_2]$ shows that $f + g \in O(h)$. $\diamond$

3.2 Bootstrapping lemma

We now use the following lemma, an important result which will be very useful in proving other interesting Big-O properties.

Theorem 2. *Suppose $f \in O(g)$ and we define $F(n) = nf(n), G(n) = ng(n)$, then $F \in O(G)$.*

Proof. Hint: use the same $[M, c]$ pair!
$f \in O(g)$ means for some M and c, we have for $n > M$,
$f(n) \leq C \times g(n)$.
$n \times f(n) \leq C \times n \times g(n)$.
Hence, $F(n) \leq C \times G(n)$, and our claim is proved. $\diamond$

Now we can use this lemma to prove other properties:
Let $U(n) = 1$ for all n.
Let $L(n) = n$ for all n.
Then $U \in O(L)$, which we can write as $(n \mapsto 1) \in O(n \mapsto n)$.
Bootstrapping tells us that $(n \mapsto n) \in O(n \mapsto n^2)$.
And also that $(n \mapsto n^2) \in O(n \mapsto n^3)$.
Hence, $(n \mapsto 1) \in O(n \mapsto n^3)$.
Continuing this, we see that $(n \mapsto n^k) \in O(n \mapsto n^p)$ for any natural numbers k, p with $k \leq p$.

Using the theorem that "If $f \in O(h)$ and $g \in O(h)$, then $f + g \in O(h)$", we get,
$(n \mapsto n^2 + n) \in O(n \mapsto n^2)$, because $(n \mapsto n^2) \in O(n \mapsto n^2)$ and $(n \mapsto n) \in O(n \mapsto n^2)$
In fact, for any polynomial p of degree k (where $k \in \mathbb{N}$) we have that $(n \mapsto p(n)) \in O(n \mapsto n^k)$.
Because of this, we seldom say that some procedure runs in $O(nN \mapsto n^2 + 5)$ time, but instead we say $O(n \mapsto n^2)$ time. We will mostly be dealing with polynomials in our analysis, but a function like 2^n is much worse than any of these polynomial functions, and increases in runtime extremely quickly.

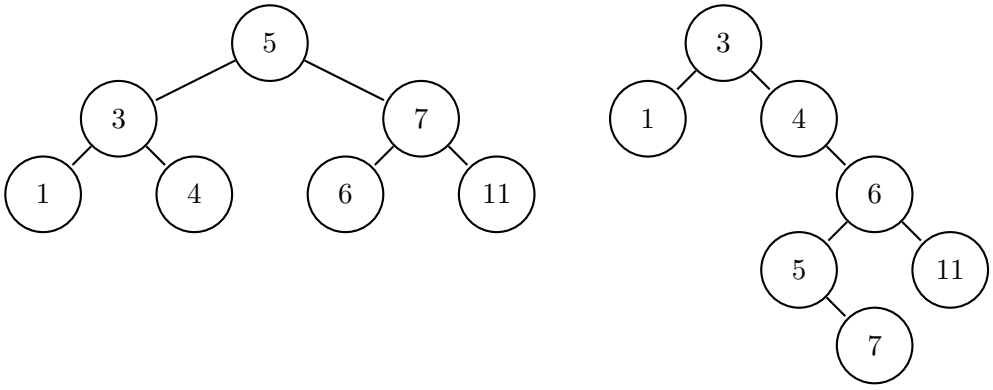


Figure 1: Two sample binary search trees that store the set $\{1, 3, 4, 5, 6, 7, 11\}$

3.3 A generalized bootstrapping lemma

Theorem 3. *Suppose* $f \in O(g)$ *and* $h \in \Omega(n \mapsto 1)$*, which means that* $h(n) \neq 0$ *for large enough* n*, we have*

$$n \mapsto f(n)h(n) \in)(n \mapsto g(n)h(n))$$

Proof. Hint: use the max of the M values and the c from $f \in O(g)$ to prove this result! $\diamond$

4 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:

```
type 'a btree =
  | Leaf
  | Node ('a btree, 'a, 'a btree) ;
```

For example,

```
Leaf
Node (Leaf, 17, Leaf)
Node (Leaf, 17, Node (Leaf, 18, Leaf))
Node (Node (Leaf, 17, Leaf), 22, Node (Leaf, 18, Leaf))
```

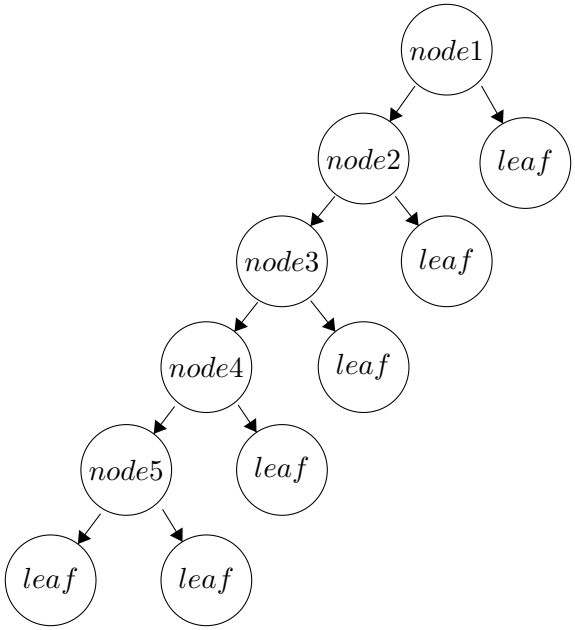
If a binary tree is a leaf, then it is automatically a BST. If it is a node, then the following **left** and **right** invariants, respectively, must hold:

- 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.

5 Depth of Binary Trees

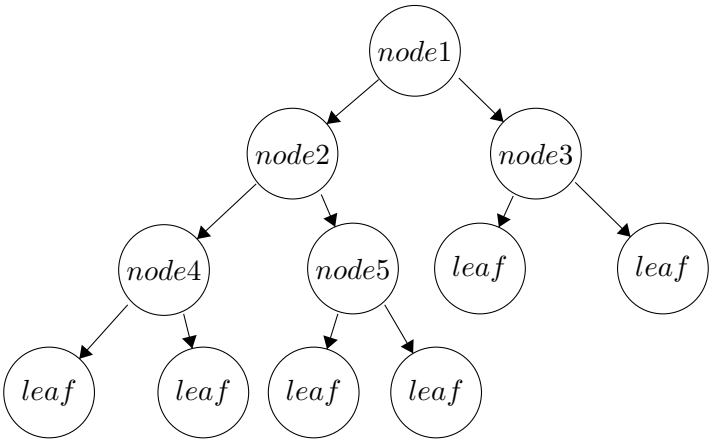
Informally, the depth of a binary tree is the number of steps needed to get to the bottom of the tree. Now let's consider, for a binary tree with n nodes, what could be the largest possible depth?

Take the example of a tree with 5 nodes. Such a tree would look like:



Thus the largest depth a binary tree with 5 nodes can have is 5.

Now consider, what could be the least possible depth of a bianry tree with n nodes? The shallowest tree with 5 nodes would look like:

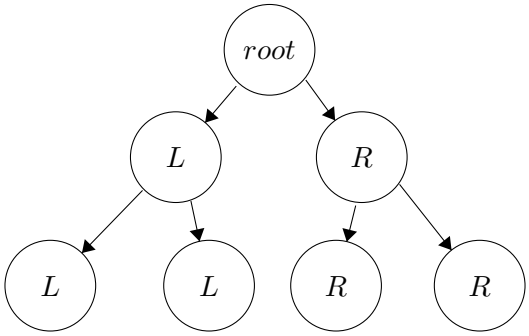


This tree has depth 3. It is not possible to have a binary tree of 5 nodes with only depth 2 becuase there would only be room for 3 nodes.

Now, if a binary tree has depth k, what's the largest number of nodes it can have? Below is a table with the number of nodes for values of k from 0 to 6:

k	**nodes**
0	0
1	1
2	3
3	7
4	15
5	31
6	63

We can think of the number of nodes in a tree as being the root, plus number of nodes in the left subtree, plus the number in the right subtree. The left and right subtrees are equivalent to a tree with depth 1 lesser than our tree, and have the same number of nodes.



So, let $Q(k)$ be the largest number of nodes in a tree of depth k.
We have,
$Q(0) = 0$
$Q(k) = 1 + 2Q(k-1)$, becuase in a depth k tree, the left and right subtrees can each contain at most $Q(k-1)$ nodes, and their parent (the root) accouts for one more.

We can now use plug and chug on this recursive formula we have for largest number of nodes (like the table above) to get $Q(k) \leq 2^k - 1$.

Our next question is, if we want to put n nodes in a tree, how deep must it be?
Well, we can put $\frac{n-1}{2}$ nodes in each subtree.
So $D(n)$, the maximum depth of an n node tree satisfies $D(n) \geq D(\frac{n-1}{2} + 1)$

This is basically asking the question: how many times do I need to divide n by 2 till I get to a number less than or equal to 1?
This is (informally) the definition of $log_2(n)$! Or rather, $\lceil log_2(n) \rceil$, since we are taking the ceiling of this number. More formally, a mathematical definition of log would be

$$log_2(n) = x \text{ if } 2^x = n$$

However, for us, the intuitive understanding that the number of times you need to divide a number by 2 to reach 1 is $log_2(n)$ is very effective in helping us analyze procedures!

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: `http://cs.brown.edu/courses/csci0170/feedback`.