CO202 - Algorithms II

8th October 2019

Introduction

Note that this course is taught in Haskell, and in the style of Dijkstra (structure of algorithms), instead of Knuth (analysis and complexity).

List Insertion

An algorithm to insert elements in a sorted list;

```
1  insert :: Int -> [Int] -> [Int]
2  insert x [] = [x]
3  insert x (y:ys)
4    | x <= y    = x:y:ys
5    | otherwise = y:insert x ys
```

In Haskell, we do this by case analysis, first looking at the base case (line 2) - where the list is empty. The second case (line 3) considers the non-empty list. The evaluation is as follows, for a simple example;

$$
\begin{array}{lll}
\texttt{insert 4 [1,3,6,7,9]} & & \\
\rightsquigarrow \texttt{1:insert 4 [3,6,7,9]} & & \text{definition of } \texttt{insert} \\
\rightsquigarrow \texttt{1:3:insert 4 [6,7,9]} & & \text{definition of } \texttt{insert} \\
\rightsquigarrow \texttt{1:3:4:6:[7,9]} & & \text{definition of } \texttt{insert}
\end{array}
$$

To give a cost, we will measure the number of steps, which approximates time - the number of steps is essentially each transition from the LHS of `=` to the RHS. The measure of input will be $n = \texttt{length xs}$. We write a recurrence relationship that ties together n with the algorithm;

$$
\begin{array}{ll}
T(0) = 1 & \text{1 transition} \\
T(n) = 1 + T(n-1) & \text{looking at worst case, line 5}
\end{array}
$$

The structure of the complexity should follow the structure of the algorithm itself. However, we are interested in a closed form for $T(n)$, where we can directly obtain the value without evaluating recursively. The easiest way to do this is to unroll the definition, and look for patterns;

$$
\begin{aligned}
T(n) &= 1 + T(n-1) \\
&= 1 + (1 + T(n-2)) \\
&= 1 + (1 + \cdots + T(n-n)) \\
&= 1 + n
\end{aligned}
$$

Insertion Sort

The previous algorithm can be used as the basis for insertion sort. For each element in the unsorted list, we insert it into the sorted list (which is initially empty).

```
1  isort :: [Int] -> [Int]
2  isort [] = []
3  isort (x:xs) = insert x (isort xs)
```

We assume that `insert`, and `isort` both give us a sorted list, assuming the input lists were also sorted. An example of this on a small list is as follows;

$$
\begin{array}{lll}
\texttt{isort [3,1,2]} & & \\
\rightsquigarrow \texttt{insert 3 (isort [1,2])} & & \text{definition of } \texttt{isort}
\end{array}
$$

$\leadsto$ `insert 3 (insert 1 (isort [2]))`	definition of `isort`
$\leadsto$ `insert 3 (insert 1 (insert 2 (isort [])))`	definition of `isort`
$\leadsto$ `insert 3 (insert 1 (insert 2 []))`	definition of `isort`
$\leadsto$ `insert 3 (insert 1 [2])`	definition of `insert`
$\leadsto$ `insert 3 (1:2:[])`	definition of `insert`
$\leadsto$ `1:insert 3 (2:[])`	definition of `insert`
$\leadsto$ `1:2:(insert 3 [])`	definition of `insert`
$\leadsto$ `1:2:[3]`	definition of `insert`

This cost 9 steps to evaluate. The recurrence relation generalises this (similarly $n = $ `length xs`);

$$T_{\mathsf{isort}}(0) = 1$$
$$T_{\mathsf{isort}}(n) = 1 + T_{\mathsf{insert}}(n-1) + T_{\mathsf{isort}}(n-1)$$

However, we want to find this in closed form;

$$
\begin{aligned}
T_{\mathsf{isort}}(n) &= 1 + n + T_{\mathsf{isort}}(n-1) \\
&= 1 + n + (1 + n - 1 + T_{\mathsf{isort}}(n-2)) \\
&= \dots \\
&= \frac{n(n+1)}{2} + 1 + n
\end{aligned}
$$

A more thorough analysis will teach us about;

- evaluation strategies and cost
- counting carefully and crudely
- abstract interfaces
- data structures

11th October 2019

Laziness

In the last lecture, we saw `isort` sorts in approximately n^2 steps.

```
1  minimum :: [Int] -> Int
2  minimum = head . isort
```

The evaluation of `minimum` takes n steps, when given a sorted list;

```
    minimum [1,2,3]
 ⤳ head (sort [1,2,3])
 ⤳ ...
 ⤳ head (insert 1 (insert 2 (insert 3 [])))
 ⤳ head (insert 1 (insert 2 [3]))
 ⤳ head (insert 1 (2:[3]))
 ⤳ head 1:2:[3]
 ⤳ 1
```

The worst case is a reversed list, as follows;

```
    minimum [3,2,1]
```

```
⤳ ...
⤳ head (insert 3 (insert 2 (insert 1 [])))
⤳ head (insert 3 (insert 2 [1]))
⤳ head (insert 3 (1:insert 2 []))
⤳ head (1:insert 3 (insert 2 []))
```

The important part is to note that the minimum value, 1, is floated to the left, for a total of n steps. Therefore, this still takes linear time. This evaluation relies on laziness, hence we can build the large computation on the RHS of the :.

Normal Forms

There are three normal forms that values can take;

- **normal form (NF)**

  This is fully evaluated, and there is no more work to be done - an expression is in NF if it is;

  - a constructor applied to arguments in NF
  - a λ-abstraction (function) whose body is in NF

- **head normal form (HNF)**

  An expression is in HNF if it is;

  - a constructor applied to arguments in any form
  - a λ-abstraction (function) whose body is in HNF

- **weak head normal form (WHNF)**

  An expression is in WHNF if it is;

  - a constructor applied to arguments in any form
  - a λ-abstraction (function) whose body is in any form

Looking at the last line in the previous evaluation, we have two constructors; cons (:) and the empty list ([]). The LHS of : is in normal form, but the RHS isn't, and therefore it cannot be in normal form.

Evaluation Order

There are two main evaluation strategies;

- **applicative order** (eager / strict evaluation)   goes to normal form

  Evaluates as much as possible, until it ends up in normal form. It evaluates the left-most, inner-most expression first. For example, in the final step `head (1:insert 3 (insert 2 []))`, it would first evaluate 2, then [], and then `insert 2 []`, and so on.

- **normal order** (lazy evaluation)   goes to weak head normal form

  This evaluates the left-most, outer-most expression first.

Counting Carefully

Here we are concerned at counting the steps mechanically in strict evaluation. This is done for a simplified language, containing constants, variables, functions, conditionals, and pattern matching. We will write e^T to denote the number of steps it takes to reduce e. Additionally, if f is a primitive function, then $f^T\ e_1\ \ldots\ e_n = 0$, otherwise $f\ e_1\ \ldots\ e_n = e$, and $f^T\ e_1\ \ldots\ e_n = 1 + e^T$.

$$k^T = 0 \qquad\qquad \text{constants}$$

$$x^T = 0 \qquad\qquad \text{evaluated variables}$$

$$(f\ e_1\ \ldots\ e_n)^T = (f^T\ e_1\ \ldots\ e_n) + e_1^T + \cdots + e_n^T \qquad\qquad \text{function with arguments}$$

$$(\texttt{if}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2)^T = p^T + (\texttt{if}\ p\ \texttt{then}\ e_1^T\ \texttt{else}\ e_2^T) \qquad\qquad \text{conditional}$$

$$\left(\texttt{case}\ e\ \texttt{of}\ \begin{cases} p_1 & \to e_1 \\ & \vdots \\ p_n & \to e_n \end{cases}\right)^T = e^T + \left(\texttt{case}\ e\ \texttt{of}\ \begin{cases} p_1 & \to e_1^T \\ & \vdots \\ p_n & \to e_n^T \end{cases}\right) \qquad\qquad \text{pattern matching}$$

This is very involved for tiny examples, and becomes much more complex for lazy evaluation.

Counting Crudely

We mainly use asymptotic notation to achieve this. Certain functions dominate others when given enough time - as the input increases.

L-functions are the smallest class of one-valued functions on real variables $n \in \mathbb{R}$, containing constants, the variable n, and are closed under arithmetic, exponentiation, and logarithms. They tend to be monotonic after a given time, and tend to a value.

Consider $f(n) = 2n$, and $g(n) = \frac{n^2}{4}$ - at $n = 1$, $f(1) > g(1)$, however at some point on the number line, g begins to dominate. Comparing functions can be achieved by studying their ratios (with well-behaved functions, like L-functions, the ratio will tend to 0, infinity, or a constant);

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

Any L-function is ultimately continuous of constant sign, monotonic, and approaches 0, ∞, or some definite limit as $n \to \infty$. Furthermore, $\frac{f}{g}$ is an L-function if both f and g are. We can now introduce notation compare function;

$$f \prec g \triangleq \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \qquad\qquad \text{also written as } f \in o(g(n))$$

$$f \preceq g \triangleq \limsup_{n \to \infty} \frac{f(n)}{g(n)} < \infty \qquad\qquad \text{also written as } f \in O(g(n))$$

$$f \asymp g \triangleq f \in (O(g(n)) \cap \Omega(g(n))) \qquad\qquad \text{also written as } f \in \Theta(g(n))$$

$$f \succeq g \triangleq \limsup_{n \to \infty} \left|\frac{f(n)}{g(n)}\right| > 0 \qquad\qquad \text{also written as } f \in \Omega(g(n))$$

$$f \succ g \triangleq \lim_{n \to \infty} \left|\frac{f(n)}{g(n)}\right| = \infty \qquad\qquad \text{also written as } f \in \omega(g(n))$$

Visually, we can represent this in the following three graphs. Note that $\delta, \delta_1, \delta_2$ are just constant multipliers. The first plot shows that as n gets larger $f(n)$ will exist within the shaded region bounded above by $\delta g(n)$, and similarly (on the other extreme) the third plot shows that as n gets larger, $f(n)$ will exist within the region bounded below by $\delta g(n)$. If f is constrained (as time progresses) within the region bounded by $\delta_1 g(n)$ and $\delta_2 g(n)$, then we have the second plot.
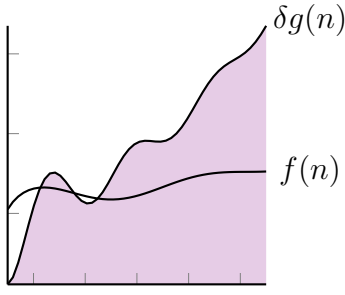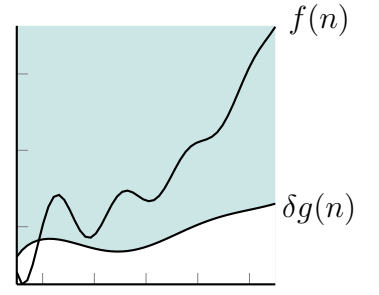
$$f(n) \in O(g(n))$$
$$f \preceq g$$
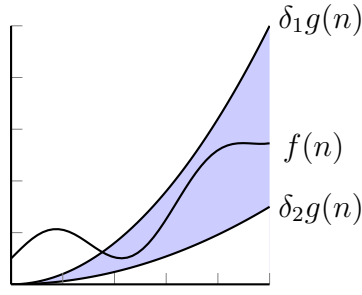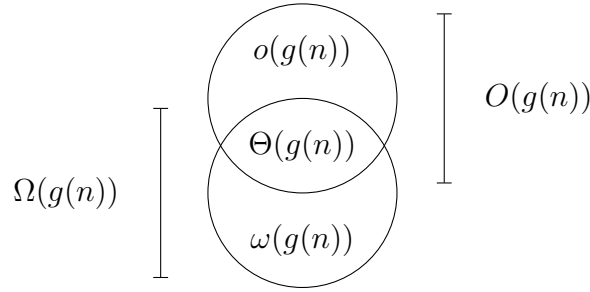
$$f(n) \in \Theta(g(n))$$
$$f \asymp g$$

$$f(n) \in \Omega(g(n))$$
$$f \succeq g$$

If f and g are L-functions, then either;

$$f \in o(g), \ f \in \Theta(g), \text{ or } f \in \Omega(g)$$

Another method of visualising this is as a Venn diagram, with the upper circle being $O(g(n))$, and the lower circle being $\Omega(g(n))$;

Finally, this can also be defined by the following;

$$o(g(n)) = \{f \mid \forall \delta > 0. \ \exists n_0 > 0. \ \forall n > n_0. \ |f(n)| < \delta g(n)\}$$
$$O(g(n)) = \{f \mid \exists \delta > 0. \ \exists n_0 > 0. \ \forall n > n_0. \ |f(n)| \leq \delta g(n)\}$$
$$\Theta(g(n)) = \left\{f \ \middle| \ \begin{array}{c} (\exists \delta > 0. \ \exists n_0 > 0. \ \forall n > n_0. \ |f(n)| \leq \delta g(n)) \\ \wedge \\ (\exists \delta > 0. \ \forall n_0 > 0. \ \exists n > n_0. \ f(n) \geq \delta g(n)) \end{array} \right\}$$
$$= O(g(n)) \cap \Omega(g(n))$$
$$\Omega(g(n)) = \{f \mid \exists \delta > 0. \ \forall n_0 > 0. \ \exists n > n_0. \ f(n) \geq \delta g(n)\}$$
$$\omega(g(n)) = \{f \mid \forall \delta > 0. \ \forall n_0 > 0. \ \exists n > n_0. \ f(n) > \delta g(n)\}$$

15th October 2019

Basic Lists

In Haskell, lists are given by **two** constructors;
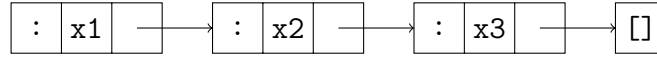
```
1  data [a] = [] | (:) a [a]
```

This creates two values, called constructors (RHS of data declaration, hence we can pattern match on these unlike other functions);

```
1  [] :: [a]
2  (:) :: a -> [a] -> [a]
```

In Haskell, data structures are persistent by default.

$$[x1,x2,x3] = x1:x2:x3:[]$$

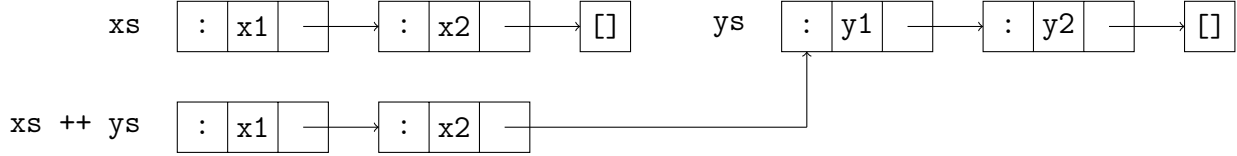Visually, we can look at this as "cells" with their constructors and arguments;

Appending lists together is achieved with (++);

```
1  (++) :: [a] -> [a] -> [a]
2  [] ++ ys = ys
3  (x:xs) ++ ys = x:(xs ++ ys)
```

When we do xs ++ ys, the final structure points to ys. The trade-off here is that we didn't have to modify ys, but we had to create a new x1, and x2;

Therefore, the complexity is linear, but only depends on xs, and not ys, hence we can say;

$$T_{(++)} \in O(n), \text{ where } n = \texttt{length xs in xs ++ ys}$$

Folding

The structure of lists is completed reduced by the foldr function;

```
1  foldr :: (a -> b -> b) -> b -> [a] -> b
2  foldr f k [] = k
3  foldr f k (x:xs) = f x (foldr f k xs)
```
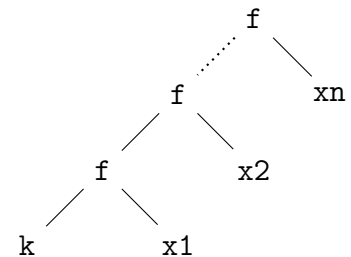
This replaces (:) with f, and [] with k;

Note that doing foldr (:) [] is the same as id, but with more work, and also xs ++ ys is equivalent to foldr (:) ys xs. Similarly, we have foldl;

```
1  foldl :: (b -> a -> b) -> b -> [a] -> b
2  foldl f k [] = k
3  foldl f k (x:xs) = foldl f (f k s) xs
```


Consider a binary operator (⋄);

$$x \diamond (y \diamond z) = (x \diamond y) \diamond z \qquad\qquad \diamond - \text{associative}$$
$$\epsilon \diamond y = y \qquad\qquad \epsilon - \text{left-unit}$$
$$x \diamond \epsilon = x \qquad\qquad \epsilon - \text{right-unit}$$

When folding with ⋄ and ϵ, foldr and foldl coincide.

List Concatenation

An application of this is list concatenation, consider `concat`;

```
1  concat :: [[a]] -> [a]
2  concat [] = []
3  concat (xs:xss) = xs ++ concat xss
```

This can be written as a `foldr`;

```
1  rconcat :: [[a]] -> [a]
2  rconcat = foldr (++) []
```

Since `(++)` is associate with a neutral element `[]`, we can write a `foldl` version;

```
1  lconcat :: [[a]] -> [a]
2  lconcat = foldl (++) []
```

While `lconcat = rconcat`, this only represents extensional (what values are produced) equality. They are not intensionally (how those values are produced) equal.

18th October 2019

Concatenation and Associativity

The complexity of `xs ++ ys` is; $T_{(++)}(m, n) \in O(m)$, where $m = $ `length xs`, and $n = $ `length ys` - note that it doesn't consider n as expected. We want to consider the complexity of;

$$
\overbrace{((\underbrace{(\overbrace{\underbrace{(xs_1 ++ xs_2)}_{n} ++ xs_3}^{2n}) ++ ...)}_{3n} ++ xs_{m+1})}^{mn}
$$

In lieu of having individual variables, we can approximate all the lists `xs1` to `xs`$_{m+1}$ as having the same length n. This means that if $m = $ `length xss`, then `concat xss` has complexity;

$$
n + 2n + 3n + \cdots + mn = n(1 + 2 + 3 + \cdots + m) \in O(nm^2)
$$

If we now look at the right associative version;

$$
xs_1 ++ (xs_2 ++ (xs_3 ++ (... ++ (xs_m ++ xs_{m+1}))))
$$

Note that each of the `++` operations cost n, and we have m of them, under the same assumption. Therefore, we have $O(mn)$.

Note that adding to the right of a list is similar to doing the first version, which is left associative, and also more expensive. Preferably, we'd add to the left of the list, however this isn't always possible, as we'd also like to be able to add to the right of the list.

Function Composition

We do not want to suffer the consequences of poor associativity. To solve this, we notice that function composition pays no associativity penalty. First note that function composition is as follows;

$$
(f . g) x = f (g x)
$$

We can also show that function composition is both extensionally and intensionally equal, by unrolling the definition;

$$
((f . g) . h) x = f (g (h x))
$$
$$
vs
$$
$$
(f . (g . h)) x = f (g (h x))
$$

For our lists, we will replace with;

$$((xs_1\ ++)\ .\ \ (xs_2\ ++)\ .\ \ ...\ \ .\ \ (xs_{m+1}\ ++))\ []$$

Note that each of $(xs_i\ ++)$ is a partially applied function that takes a list, and adds xs_i to the start. This is equivalent to the right associative version we had before, which has a better time complexity.

Difference List (DList)

The idea is to replace lists with functions that take in a list and give a list back. Previously, we'd consider xs_1 a list, but we now use $(xs_1\ ++)$, which needs to be applied to the empty list. A value of type `a` in the list is now a function of type `[a] -> [a]`. We can create a new datatype;

```
1  data DList a = DList ([a] -> [a])
```

We need ways of making lists from `DList`s, and vice versa;

```
1  -- technically called the abstraction function
2  toList :: DList a -> [a]
3  toList (DList fxs) = fxs []
4
5  fromList :: [a] -> DList a
6  fromList xs = DList (xs ++)
7  -- or        DList (\ys -> xs ++ ys)
```

We also need a way of adding `DList`s together (note the notes use ⋄, but I can't really get that with the listing environment, so I will use `<>`);

```
8  (<>) :: DList a -> DList a -> DList a
9  DList fxs <> DList fys = DList (fxs . fys)
10 -- or                    DList (\zs -> fxs (fys zs))
```

Note that taking the head or tail of this list now requires constructing the entire list, which will take linear time.

List Anatomy
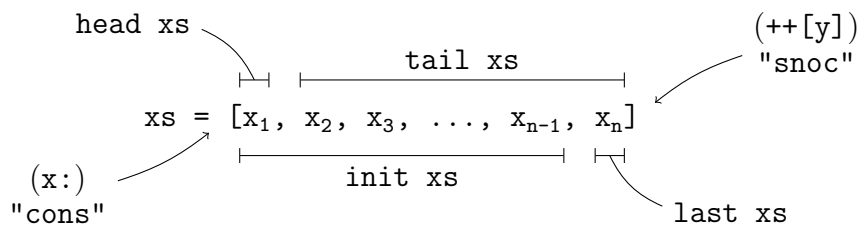
List Interface

Rather than dealing with lists directly, we want to work with a specification instead. For this, we are creating a type class - an abstraction mechanism that has a class of objects, which have certain operations - and depending on what is given the operations can operate slightly differently. Note that `List` is the name of the type class, and `list` is the variable of abstraction.

```
1  class List list where
2    empty :: list a
3    cons :: a -> list a -> list a
4    snoc :: a -> list a -> list a
5    head :: list a -> a
6    tail :: list a -> list a
7    init :: list a -> list a
8    last :: list a -> a
```

```
9     length :: list a -> Int
10    (++) :: list a -> list a -> list a
11    null :: list a -> Bool
12    single :: list a -> Bool
13
14    toList :: list a -> [a]
15    fromList :: [a] -> list a
```

Note that `toList` and `fromList` must behave nicely when composed;

$$id = toList \ . \ \ fromList$$

However, composing the functions in reverse does not necessarily yield the same result. Consider the case with our DList - it would be perfectly valid to have (`DList reverse`); however, since the space of functions is much greater than the space of lists, we cannot reasonably convert `revert` into a list, and then hope to get `reverse` back out.

However, we can say that if we had something representing a list, converting it to a list and then back to our implementation of a list, should yield something similar.

$$normalise = fromList \ . \ \ toList$$

Exercises

1. Prove formally that $(n+1)^2 \in \Theta(n^2)$ by exhibiting the necessary constants.

   We set $f(n) = (n+1)^2$, and $g(n) = n^2$. By our definition of Θ, we need to satisfy both the following;

   $$O(g(n)) = \{f \mid \exists \delta > 0. \ \exists n_0 > 0. \ \forall n > n_0. \ |f(n)| \leq \delta g(n)\}$$
   $$\Omega(g(n)) = \{f \mid \exists \delta > 0. \ \forall n_0 > 0. \ \exists n > n_0. \ f(n) \geq \delta g(n)\}$$

   The first case, we want to find δ such that;

   $$\begin{aligned}
   (n+1)^2 &\leq \delta n^2 & &\Leftrightarrow \\
   n^2 + 2n + 1 &\leq \delta n^2 & &\Leftrightarrow \\
   0 &\leq (\delta - 1)n^2 - 2n - 1 & \text{assuming } \delta - 1 = 3, &\Rightarrow \\
   0 &\leq (3n + 1)(n - 1)
   \end{aligned}$$

   Hence we have $\delta = 4$. The second case, Ω, is trivial to prove with $\delta = 1$.

2. Give examples of expressions that are;

   1. in head normal form, but not in normal form   `5:take 2 [1,2,3]`
   2. in weak head normal form, but not in head normal form   `\x -> take 2 [1,2,3]`
   3. in no normal form of any kind   `take 2 [1,2,3]`
         it's not a constructor, and there isn't a λ-abstraction
   4. in normal form, but not in weak head normal form   not possible

3. Using the definition of $f \in O(g(n))$ as a set, derive the definition of $f \in \omega(g(n))$ as a set.

   Recall that $f \in \omega(g(n)) \Leftrightarrow f \notin O(g(n))$, and the following definition;

   $$O(g(n)) = \{f \mid \exists \delta > 0. \ \exists n_0 > 0. \ \forall n > n_0. \ |f(n)| \leq \delta g(n)\}$$

Using equivalences for first-order logic;

$$\neg \exists \delta > 0.\ \exists n_0 > 0.\ \forall n > n_0.\ |f(n)| \leq \delta g(n)$$
$$\equiv \forall \delta > 0.\ \neg \exists n_0 > 0.\ \forall n > n_0.\ |f(n)| \leq \delta g(n)$$
$$\equiv \forall \delta > 0.\ \forall n_0 > 0.\ \neg \forall n > n_0.\ |f(n)| \leq \delta g(n)$$
$$\equiv \forall \delta > 0.\ \forall n_0 > 0.\ \exists n > n_0.\ \neg(|f(n)| \leq \delta g(n))$$
$$\equiv \forall \delta > 0.\ \forall n_0 > 0.\ \exists n > n_0.\ f(n) > \delta g(n)$$

Hence we can write ω as the following;

$$\omega(g(n)) = \{f \mid \forall \delta > 0.\ \forall n_0 > 0.\ \exists n > n_0.\ f(n) > \delta g(n)\}$$

4. The composition rule gives the time complexity of two functions f and g when they are composed. By using the definition of e^T, derive the accurate cost of $f\ (g\ x)$ using strict evaluation.

$$
\begin{aligned}
((f \circ g)\ x)^T &= 1 + (f(g(x)))^T \\
&= 1 + f^T(g(x)) + (g(x))^T \\
&= 1 + f^T(g(x)) + g^T x + x^T \\
&= 1 + f^T(g(x)) + g^T x
\end{aligned}
$$

5. Justify whether each of the following is true or false;

1. $2n^2 + 3n \in \Theta(n^2)$ — true
2. $2n^2 + 3n \in O(n^3)$ — true
3. $n \log n \in O(n\sqrt{n})$ — ($\log n$ grows slower than $\sqrt{n}$) true
4. $n + \sqrt{n} \in O(n \log n)$ — false
5. $2^{\log n} \in O(n)$ — (use laws of logarithms) true

22nd October 2020

Double-Ended Queues (Deque)

These are sometimes called symmetric lists. A normal list has the following complexities;

- `cons x xs` $O(1)$
- `head xs` $O(1)$
- `tail xs` $O(1)$
- `snoc x xs` $O(n)$
- `last xs` $O(n)$
- `init xs` $O(n)$

We define a `Deque` as two lists, where the first list is in normal order, and the second list, containing the remainder of the whole list, is in reverse order (see the `toList` function)

```
1  data Deque a = Deque [a] [a]
2
3  toList :: Deque a -> [a]
4  toList (Deque xs ys) = xs ++ reverse ys
```

To add to the end of the list, we add to the front of the second list, hence we have constant time `snoc`, as well as constant time `cons`.

To get desirable asymptotic complexities, there are two invariants that we require;

- `null xs` $\Rightarrow$ `null ys` $\vee$ `single ys`
- `null ys` $\Rightarrow$ `null xs` $\vee$ `single xs`

To make a `Deque` from a list, some symmetry is preferable.

```
5  fromList :: [a] -> Deque [a]
6  fromList xs = Deque us (reverse vs)
7    where
8      (us, vs) = splitAt (div n 2) xs
9      n = length xs
```

Some operations are easy to analyse and define;

```
10  snoc :: a -> Deque a -> Deque a
11  snoc y (Deque [] ys) = Deque ys [y]
12  snoc y (Deque xs ys) = Deque xs y:ys
```

This has $\Theta(1)$ complexity. It's important to note that we can do line 2 due to our invariant - if we know xs is empty, then ys must also be empty, or be a singleton list - and therefore it can be placed as the first list without a reversal.

```
21  last :: Deque a -> a
22  last (Deque xs []) = head xs -- or last xs
23  last (Deque xs y:ys) = y
```

This is also $O(1)$ complexity. Similarly, we know that xs is either empty (which would cause an error anyways), or it is singleton - thus the last item is the only item in the Deque.

```
1  empty :: Deque a
2  empty = Deque [] []
3
4  tail :: Deque a -> Deque a
5  tail (Deque [] ys) = empty
6  tail (Deque [x] ys) = Deque (reverse vs) us
7    where
8      (us, vs) = splitAt (div n 2) ys
9      n = length ys
10  tail (Deque xs ys) = Deque (tail xs) ys
```

Here we define the empty Deque, which is used in the first case (and doesn't cause an error). In the third case, we know that there is more than one element in xs (since it didn't match the second case), and therefore we simply take the tail (since we won't have an empty list, unlike the second case). In the second case, we discard the single x. Symmetry is maintained by using the same split as fromList. Note that ys is reversed, hence we can write ys = [y_n, y_{n-1}, ..., y_0], and therefore (us, vs) = ([y_n, y_{n-1}, ..., y_j], [y_{j-1}, y_{j-2}, ..., y_0]), therefore vs needs to be reversed and put first. Both the first, and third cases are $O(1)$, but the worst case complexity is the second case, which has the reversal, and is $O(n)$.

Amortised Analysis

In the worst case, the cost is $O(n)$, where $n =$ length ys. However, we rarely encounter this case.

The idea of amortised analysis is that while it may have a high complexity in a single instance of the worst case, but it may not cost that amount in a sequence of operations. The cost of some functions is better expressed in terms of how it performs in a wider context.

Consider repeated applications of tail, in a chain (such that we apply tail to the result of the previous application) - each successive call to tail will cost less;

$$xs_0 \overset{\text{tail}}{\rightsquigarrow} xs_1 \overset{\text{tail}}{\rightsquigarrow} xs_2 \overset{\text{tail}}{\rightsquigarrow} \ldots \overset{\text{tail}}{\rightsquigarrow} xs_n$$

We can use amortised analysis to work out the true cost.

$$xs_0 \overset{op_0}{\rightsquigarrow} xs_1 \overset{op_1}{\rightsquigarrow} xs_2 \overset{op_2}{\rightsquigarrow} \ldots \overset{op_n}{\rightsquigarrow} xs_{n+1}$$

To perform amortised analysis, we need to define the following;

- **cost**   assign a real cost $C_{\mathtt{op_i}}(\mathtt{xs_i})$ for each operation $\mathtt{op_i}$ on data $\mathtt{xs_i}$
- **amortised cost**   assign an amortised cost $A_{\mathtt{op_i}}(\mathtt{xs_i})$ for each operation $\mathtt{op_i}$ on data $\mathtt{xs_i}$
- **size**   define $S(\mathtt{xs})$ that calculates the size of the data $\mathtt{xs}$

We use these to establish the following;

$$\underbrace{\sum_{i=0}^{n-1} C_{\mathtt{op_i}}(\mathtt{xs_i})}_{\text{total actual cost}} \leq \underbrace{\sum_{i=0}^{n-1} A_{\mathtt{op_i}}(\mathtt{xs_i})}_{\text{total amortised cost}}$$

For example, if we choose $A_{\mathtt{op_i}}(\mathtt{xs_i}) = 1$, then the total cost is $O(n)$. To prove this inequality (let it be equation (1)), we use the size function (physicist's approach);

$$C_{\mathtt{op_i}}(\mathtt{xs_i}) \leq A_{\mathtt{op_i}}(\mathtt{xs_i}) + \overbrace{\underbrace{S(\mathtt{xs_i})}_{\text{input size}} - \underbrace{S(\mathtt{xs_{i+1}})}_{\text{output size}}}^{\text{dif between data structure}}$$

We can sum over this as follows;

$$\sum_{i=0}^{n-1} C_{\mathtt{op_i}}(\mathtt{xs_i}) \leq \sum_{i=0}^{n-1} A_{\mathtt{op_i}}(\mathtt{xs_i}) + S(\mathtt{xs_0}) - S(\mathtt{xs_n})$$

Let this be equation 2.

25th October 2019

Size Function

One way of thinking about the size function is that it is similar to a bank; when a cheap operation is performed, we can put money in, and when an expensive operation is performed, we take money out.

If we show equation 2, we have shown equation 1 - given that $S(\mathtt{xs_0}) = 0$.

Example on Deques

First, we can assign costs as follows;

$$C_{\text{cons}}(\mathtt{xs}) = 1$$
$$C_{\text{snoc}}(\mathtt{xs}) = 1$$
$$C_{\text{head}}(\mathtt{xs}) = 1$$
$$C_{\text{last}}(\mathtt{xs}) = 1$$
$$C_{\text{tail}}(\mathtt{Deque\ xs\ ys}) = k \qquad\qquad \text{where } k = \mathtt{length\ ys}$$

Now, we can assign amortised costs. Regardless of the operation, we can say the charge is 2, hence we are overcharging by 1 for each of the constant time operations, in the hope that when we get to $\mathtt{tail}$, we have enough "saved up";

$$A_{\text{op}}(\mathtt{Deque\ xs\ ys}) = 2$$

Finally, we can define the size function to be the difference in size of $\mathtt{xs}$ and $\mathtt{ys}$. The expensive operation of doing $\mathtt{tail}$ happens when one of the two lists is empty, and reversal has to be done. Once this "distance" increases to a certain amount, we have to charge the expensive operation.

$$S(\mathtt{Deque\ xs\ ys}) = |\mathtt{length\ xs} - \mathtt{length\ ys}|$$

We can now attempt to prove equation 2. Given `Deque xs' ys' = tail (Deque xs ys)`, we have the following sizes (in the worst case, where `xs` is a singleton list, and `ys` is of length k, and after the operation the difference is at most 1, since the list is (approximately) symmetric);

$$S(\texttt{Deque xs ys}) = k - 1$$
$$S(\texttt{Deque xs' ys'}) = 1$$

Substituting gives the following;

$$\underbrace{C_{\texttt{tail}}(\texttt{Deque xs ys})}_{=k} \leq \underbrace{A_{\texttt{tail}}(\texttt{Deque xs ys})}_{=2} + \underbrace{S(\texttt{Deque xs ys})}_{=k-1} - \underbrace{S(\texttt{Deque xs' ys'})}_{=1}$$
$$\Leftrightarrow$$
$$k \leq 2 + (k-1) - 1 = k \text{ (which is true)}$$

Note that if we overcharged (had a number higher than 2), we'd still have a constant cost.

Counting

Once again, we encounter Peano numbers (see **CO142** from last year);

```
1  data Peano = Zero | Succ Peano
2
3  inc :: Peano -> Peano
4  inc n = Succ n
5
6  -- notice this errors for Zero
7  dec :: Peano -> Peano
8  dec (Succ n) = n
9
10 add :: Peano -> Peano -> Peano
11 add Zero n = n
12 add (Succ m) n = Succ (add m n)
```

This has a duality with lists;

```
1  data [a] = [] | (:) a [a]
2
3  -- mirrors inc
4  cons :: a -> [a] -> [a]
5  cons x xs = x:xs
6
7  -- mirrors dec (notice the error case)
8  tail :: [a] -> [a]
9  tail (x:xs) = xs
10
11 -- mirrors add
12 (++) :: [a] -> [a] -> [a]
13 [] ++ ys = ys
14 (x:xs) ++ ys = x:(xs ++ ys)
```

Counting in binary is as follows (our representation has the least significant bit first);

```
1  type Binary = [Bit] -- [] is decimal 0
2  data Bit = 0 | 1
3
4  inc :: Binary -> Binary
5  inc [] = [1]
6  inc (0:bs) = 1:bs
7  inc (1:bs) = 0:inc bs
```

Notice that, in the worst case, `inc` has complexity n, where n is the number of bits. Once again, doing amortised analysis, we define the following;

$$C_{\texttt{inc}}(\texttt{bs}) = t + 1 \qquad \text{where } t = \texttt{length (takeWhile (== 1) bs)}$$
$$A_{\texttt{inc}}(\texttt{bs}) = 2 \qquad \text{overcharge for amortised cost}$$
$$S(\texttt{bs}) = \texttt{length (filter (== 1) bs)}$$

Size function is some kind of measure of how much "money" we have in the "bank". We're "paying" when we have a lot of 1s, therefore for any 1 we put down, we have to pay for it later - hence we define the size function as the number of 1s in the bit string. Additionally, recall that having $S(\texttt{xs}_0) = 0$ allows us to show equation 1, by showing equation 2 (which can be done with the inequality, without the summation).

Given `bs' = inc bs`, we want to verify the following;

$$\underbrace{C_{\texttt{inc}}(\texttt{bs})}_{=t+1} \leq \underbrace{A_{\texttt{inc}}(\texttt{bs})}_{=2} + \underbrace{S(\texttt{bs})}_{=b} - \underbrace{S(\texttt{bs'})}_{=b'}$$

Note that we can relate b and b' as $b' = b - t + 1$, since we flip all the leading 1s to 0s (hence the $-t$), and then flip a 0 to a 1 (hence the $+1$). By doing those substitutions, we can conclude the following;

$$t + 1 \leq 2 + b - (b - t + 1) = t + 1 \text{ (which is true)}$$

Note that we want A to be as tight of a bound as possible - while the inequality would still hold true if we set A to be the number of bits, we'd end up with a poor estimate.

Relation with Lists

Comparing binary numbers to Peano numbers, we are going up in powers of two, and therefore we can communicate larger numbers with less space. Relating this back to lists, searching for an item by traversing the entire list mirrors Peano numbers.

Binary numbers are in the form;

$$b_0 b_1 b_2 \ldots b_n \text{ meaning } 2^0 b_0 + 2^1 b_1 + 2^2 b_2 + \cdots + 2^n b_n$$

We will work with a list representation that has $1, 2, 4, \ldots, 2^n$ elements. Therefore, for a list;

$$[\texttt{t}_0, \ \texttt{t}_1, \ \ldots, \ \texttt{t}_n]$$

We will store 1 element in the $\texttt{t}_0$ position, 2 elements in the $\texttt{t}_1$ position, and 2^n elements in the $\texttt{t}_n$ position. Depending on how many elements we want, we will either store or not store in these positions - consider having 6 elements; nothing will be stored in the $\texttt{t}_0$ position, but we will store 2 elements in $\texttt{t}_1$, and 4 elements in $\texttt{t}_2$. To do this, we will be using trees, which also gives us logarithmic access.

Looking up in a list is expensive (takes $\Theta(k)$ instructions);

```
1  (!!) :: [a] -> Int -> a
2  (x:xs) !! 0 = x
3  (x:xs) !! k = xs !! (k - 1)
```

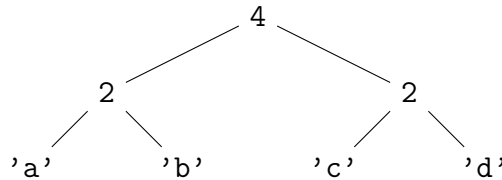To improve upon this, we seek $O(\log n)$ instead.

Trees

This can be done with trees (note that the `Int` in the `Node` constructor is the size of the tree);

```
1  data Tree a = Leaf a | Node Int (Tree a) (Tree a)
```



```
2  size :: Tree a -> Int
3  size (Leaf x) = 1
4  size (Node n l r) = n
```

In lieu of taking the sum of the subtrees, we can simply maintain an invariant that has `n` be the size of the tree. This can be done with a **smart constructor**;

```
5   node :: Tree a -> Tree a -> Tree a
6   node l r = Node (size l + size r) l r
```

We can imagine that this has an instance of the `List` class;

```
7   instance List Tree where
8     toList :: Tree a -> [a]
9     toList (Leaf x) = [x]
10    toList (Node n l r) = toList l ++ toList r -- we can do better here
11    ...
12    (!!) :: Tree a -> Int -> a
13    Leaf x !! 0 = x
14    Node n l r !! k
15      | k < m     = l !! k
16      | otherwise = r !! k - m
17      where
18        m = size l
```

Random Access Lists (RALists)

Lists can be represented using binary numbers as follows;

```
1   type RAList a = [Maybe (Tree a)]
2
3   instance List RAList where -- can't actually do this on type synonyms
4     toList :: RAList a -> [a]
5     toList = concat . map to
6       where
7         to :: Maybe (Tree a) -> [a]
8         to Nothing = []
9         to (Just t) = toList t -- note this is toList :: Tree a -> [a]
10    (!!) :: RAList a -> Int -> a
11    (Nothing : ts) !! k = ts !! k
12    (Just t : ts) !! k
13      | k < m     = t !! k -- (!!) for Tree (overloading)
14      | otherwise = ts !! k - m
15      where
16        m = size t
```

We have the following extensional equality;

$$\text{toList txs !! k = txs !! k where tks :: RAList a}$$

However, the RHS performs better, as the LHS constructs the list, and then performs a walk down it. The complexity of `(!!)` (on `RAList`) is $O(\log k)$.

To implement `cons`, we need to mirror `inc` on binary numbers;

```
17  cons :: a -> RAList a -> RAList a
18  cons x xs = consT (Leaf x) xs
19    where
20      consT :: Tree a -> RAList a -> RAList a
21      consT t [] = [Just t]
22      consT t (Nothing:ts) = (Just t):ts
23      consT t ((Just t'):ts) = Nothing:consT (node t t') ts
```

29th October 2019

Divide and Conquer, Merge Sort

This strategy has the following three parts;

1. divide a problem into subproblems

2. apply the strategy to subproblems to make subsolutions

3. combine subsolutions to make a solution

A classic example is merge sort;

```
1   msort :: [Int] -> [Int]
2   msort [] = []
3   msort [x] = [x]
4   msort xs = merge (msort us) (msort vs)
5     where
6       (us, vs) = splitAt (div n 2) xs
7       n = length xs
8
9   merge :: [Int] -> [Int] -> [Int]
10  merge [] ys = ys
11  merge xs [] = xs
12  merge (x:xs) (y:ys)
13    | x <= y    = x:merge xs (y:ys)
14    | otherwise = y:merge (x:xs) ys
```

We can perform the following recurrence relation to obtain the complexity;

$$
\begin{aligned}
T_{\texttt{msort}}(0) &= 1 \\
T_{\texttt{msort}}(1) &= 1 \\
T_{\texttt{msort}}(n) &= \underbrace{T_{\texttt{length}}(n)}_{O(n)} + \underbrace{T_{\texttt{splitAt}}\left(\frac{n}{2}\right)}_{O(n)} + \underbrace{T_{\texttt{merge}}\left(\frac{n}{2}\right)}_{O(n)} + 2 \cdot T_{\texttt{msort}}\left(\frac{n}{2}\right)
\end{aligned}
$$

For this, we can draw a recursion tree to represent the structure (note the initial $c \cdot n$ represents the three $O(n)$ costs combined, and so on);
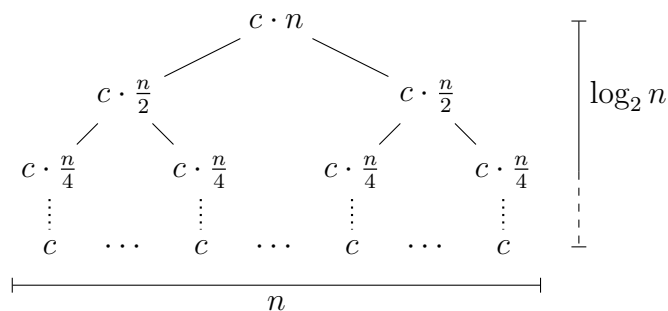
Therefore, we can conclude the following; $\qquad\qquad\qquad\qquad\qquad\qquad T_{\texttt{msort}}(n) \in \Theta(n \log n)$

Quicksort

For this implementation, we will use the head of the list as the pivot - using an arbitrary element is better.

```
1   qsort :: [Int] -> [Int]
2   qsort [] = []
3   qsort [x] = [x]
```

```
4  qsort (x:xs) = qsort us ++ [x] ++ qsort vs
5    where
6      (us, vs) = partition (< x) xs
7
8  partition :: (a -> Bool) -> [a] -> ([a], [a])
9  partition p xs = (filter p xs, filter (not . p) xs)
```

In the best case, the cost is the same as `msort`, since we are splitting the list in the middle. However, in the worst case we have the following;

$$
\begin{aligned}
T_{\text{qsort}}(n) &= \underbrace{T_{\text{partition}}(n)}_{O(n)} + \underbrace{T_{\text{(++)}}(n-1)}_{O(n)} + T_{\text{qsort}}(1) + T_{\text{qsort}}(n-1) \\
&= c \cdot n + T_{\text{qsort}}(n-1) \\
&= \underbrace{c \cdot n + c \cdot n + \cdots + c \cdot n}_{n} + 1 \\
&= c \cdot n^2 + 1
\end{aligned}
$$

Hence $T_{\text{qsort}} \in O(n^2)$.

Binary Search Trees / AVL Trees

We start with a tree which mirrors `qsort`;

```
1  data QTree a = QNil | Node (QTree a) a (QTree a)
```

The idea is that, for the diagram on the right, elements in l are less than x and elements in r are greater than x;
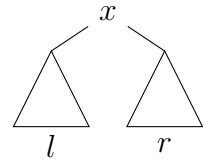

```
2  fromList :: [Int] -> QTree Int
3  fromList [] = QNil
4  fromList (x:xs) = QNode (fromList us) x (fromList vs)
5    where
6      (us, vs) = partition (< x) xs
```

Notice that this essentially encapsulates the call structure of `qsort` in memory. Flattening this tree will give us a sorted list, equivalent to `qsort`. The `fromList` function has the same complexity as `qsort`, which is $O(n \log n)$ in the best case, and $O(n^2)$ in the worst case.

The goal is to change the data structure so that it takes $O(n \log n)$ on average. To achieve this performance, we want to keep the tree balanced - hence we want to store the height of the tree (in lieu of traversing it, since we will use this operation frequently). A binary search tree has this representation (with the `Int` representing height, and a smart constructor `bnode` to maintain this property);

```
1  data BTree a = BNil | BNode Int (BTree a) a (BTree a)
2
3  height :: BTree a -> Int
4  height BNil = 0
5  height (BNode h l x r) = h
6
7  bnode :: BTree a -> a -> BTree a -> BTree a
8  bnode l x r = BNode h l x r
9    where
10     h = 1 + max (height l) (height r)
```

The algorithm will consist of repeated inserts into the tree - the `fromList` function will take a list of elements and `foldr` over it. The `insert` function must maintain the balance of the tree.

```
11  fromList :: [Int] -> BTree Int
12  fromList xs = foldr insert BNil
13
14  insert :: Int -> BTree Int -> BTree Int
15  insert x BNil = bnode BNil x BNil
16  insert x (BNode h l y r)
17    | x < y  = balance (insert x l) y r
18    | x == y = BNode h l y r -- discarding if equal
19    | x > y  = balance x y (insert x r)
```

Note that `balance` is another smart constructor.

1st November 2019

BTrees (Continued)

The intuition is that the tree given to `insert` is roughly balanced, such that the difference in heights of the left and right subtrees are at most 1. We define a tree to be **biased** if its subtrees differ in height;

```
20  bias :: BTree a -> Int
21  bias BNil = 0
22  bias (BNode h l x r) = height l - height r
```

Therefore, we are off by at most 2 after performing the insertion. The `balance` smart construction ensures the trees are balanced when they are originally off by at most 2;

```
23  balance :: BTree Int -> Int -> BTree Int -> BTree Int
```

Given `balance l x r`, we have the following cases;

1. trees differ by at most 1;

    (a) the tree is balanced
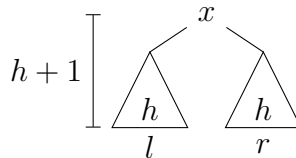
    (b) the tree is off by 1 (showing only left tree being larger, without loss of generality)
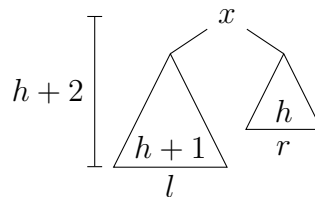
    Regardless, this is fine, hence `balance l x r = bnode l x r`

2. the trees are biased by 2;

    (a) assume that `height l > height r`, and also `height height ll ≥ height rl` (where `rl` is the right subtree of the left subtree)
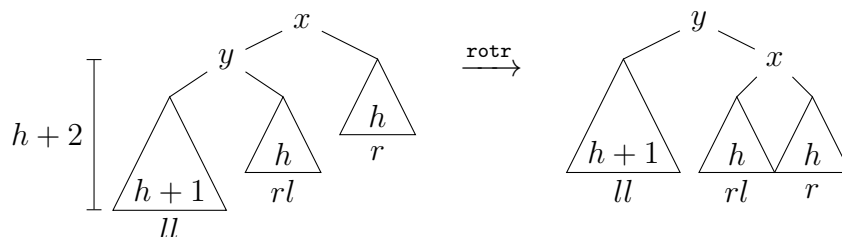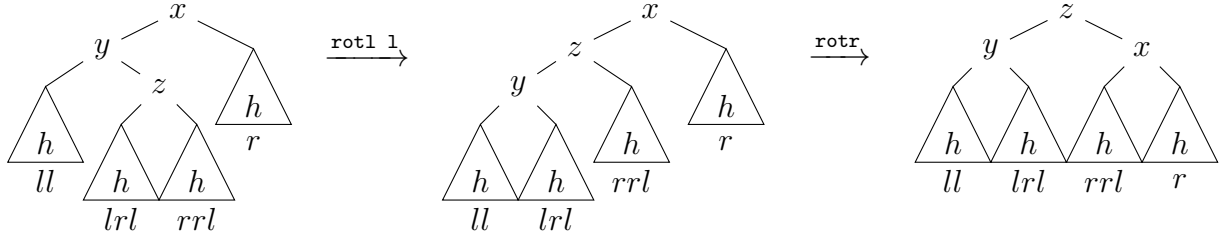
```
1  balance l x r = rotr (node l x r)
2
3  rotr :: BTree a -> BTree a
4  rotr (BNode _ (BNode _ ll y rl) x r)
5    = bnode ll y (bnode rl x r)
```

(b) assume that `height l > height r`, and also `height height ll < height rl`;


```
1  balance l x r = rotr (node (rotl l) x r)
2
3  rotl :: BTree a -> BTree a
4  rotl (BNode _ ll y (BNode _ lrl z rrl))
5    = bnode (bnode ll y lrl) z rrl
```

Putting all of this together;

```
23  balance :: BTree a -> a -> BTree a -> BTree a
24  balance l x r
25    | abs b <= 1 = bnode l x r                          -- (case 1a, 1b)
26    | b == 2     =
27        if 0 <= bias l then rotr (node l x r)           -- (case 2a)
28                       else rotr (node (rotl l) x r)    -- (case 2b)
29    | b == -2    =
30        if bias r <= 0 then rotl (node l x r)           -- (symmetry of 2a)
31                       else rotl (node l x (rotr r))    -- (symmetry of 2b)
```

Since `balance`, `rotr`, and `rotl` have no recursion, and are straight pattern matches, the process of balancing a tree is constant time.

However, we have recursion in `insert`. Since we have balanced the tree, we can therefore say that the size of the left and right subtrees are roughly equal, hence the height of the tree is approximately $\log_2 n$; and since we are doing a constant time operation at each level;

$$T_{\texttt{insert}}(n) \in O(\log n)$$

Similarly, since it is a balanced tree, looking up values is also $O(\log n)$.

By our definition of `fromList`, we are folding over a list of size n, and performing a $\log n$ operation for each, hence we have $O(n \log n)$.

Sorting with BTrees

Since the tree is ordered, we a natural sorting algorithm;

$$\texttt{sort = toList . fromList}$$

Since we already know `fromList` is $O(n \log n)$, `toList` is the limiting factor - if we can get it to $O(n \log n)$ or better, the overall cost of the sort is $O(n \log n)$.

```
32  toList :: BTree a -> [a]
33  toList BNil = []
34  toList (Node _ l x r) = toList l ++ [x] ++ toList r
```

Assuming we made `toList` use $O(n)$ time (the implementation above takes $O(n^2)$, but we could've used a `DList`), we cannot get a better `fromList`, since it would lower the overall cost of `sort` to $O(n)$, which isn't possible due to a lower bound on sorting algorithms.

Recall that `DList` uses function composition, which gives preferable associativity. Therefore, we want the following;

$$\text{toList l ++ [x] ++ toList r}$$
$$\downarrow$$
$$\text{(toList' l . (x:) . toList' r) []}$$

Note that the function `toList'` takes a `BTree a`, and gives a **function** that takes a `[a]` and gives back `[a]`;

```
1  toList' :: BTree a -> ([a] -> [a])
2  toList' BNil xs = xs
3  toList' (BNode _ l x r) xs = (toList' l . (x:) . toList' r) xs
```

Therefore, we can write the following;

```
1  toList t = toList' t []
```

Hence our `toList` is $O(n)$.

November 5th 2019

Dynamic Programming

The idea behind dynamic programming is that we start with a recursive function, and repeated calls do not get executed again. We are trading space for speed. An example is calculating Fibonacci numbers (note that `Int` is bounded by the CPU, and `Integer` is larger and bounded by RAM);

```
1  fib :: Int -> Integer
2  fib 0 = 0
3  fib 1 = 1
4  fib n = fib (n - 1) + fib (n - 2)
```
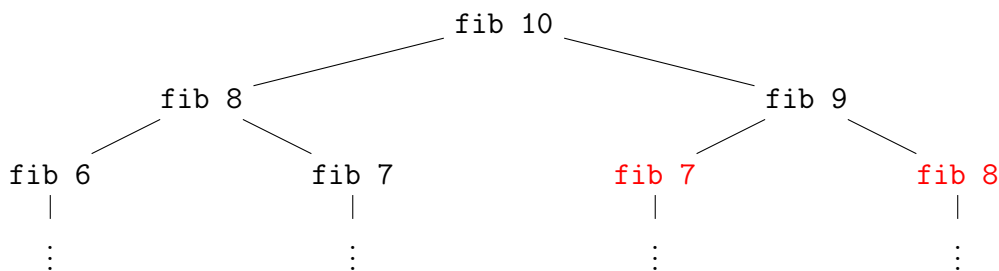

However, notice that some values are recomputed, which wastes time. Ideally, we'd want to store the values we compute in a table, giving us constant time lookup.

```
1  fib' :: Int -> Integer
2  fib' n = table ! n
3    where
4      table :: Array Int Integer
5      table = tabulate (0, n) mfib
6
7      mfib 0 = 0
8      mfib 1 = 1
9      mfib n = table ! (n - 2) + table ! (n - 1)
```

Note that (!) is $O(1)$, and has the following type;

$$(!) \ :: \ \text{Ix i} \Rightarrow \text{Array i e} \rightarrow \text{i} \rightarrow \text{e}$$

Memoisation is idea that if an argument has been computed the result can be retrieved immediately, but if it isn't, it needs to be computed and stored. Tabulation is a way to implement memoisation.

To construct an array we use the `array` function, which is used by `tabulate` (which also requires the `range` function);

```
1  range :: Ix i => (i, i) -> [i]
2  array :: Ix i => (i, i) -> [(i, e)] -> Array i e
3
4  tabulate :: Ix i => (i, i) -> (i -> e) -> Array i e
5  tabulate (a, b) f = array (a, b) [(i, f i) | i <- range (a, b)]
```

Edit-Distance Problem

This works out how many times we need to update, delete, or insert a character to get from one string to another (the *Levenshtein* distance between two strings). We can better formalise this as a decision tree, where each node has 3 possible options. We can remove the first character from the first string, which costs 1, "remove" the first characters of both strings, which costs 0 if they are equal, and 1 otherwise, or remove the first character from the second string, which costs 1. At the end, we will have an empty string compared with something else - which has a cost of however long the other string is. The edit-distance is the **minimum** cost of all the paths to the root.
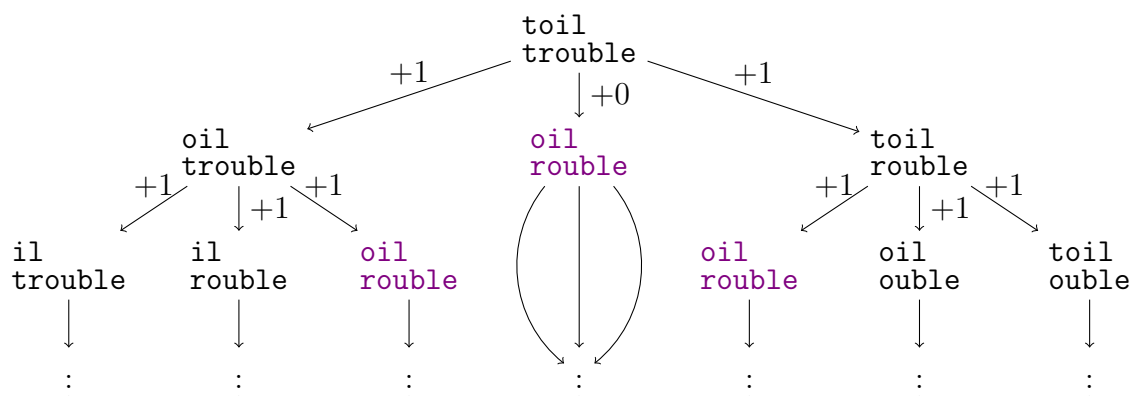
However, notice that the comparison between `oil` and `rouble` is done multiple times. Formally, we can write it as a recursive function;

```
1  dist :: String -> String -> Int
2  dist xs [] = length xs
3  dist [] ys = length ys
4  dist (x:xs) (y:ys) = minimum
5    [dist xs (y:ys) + 1
6    ,dist xs ys + if x == y then 0 else 1
7    ,dist (x:xs) ys + 1]
```

However, notice that our current function uses two strings - and in order to construct a table indexing on a pair of strings, we'd need to enumerate an extremely large number of values. Since we are decomposing the values systematically, we can instead consider an integer.

```
1  dist' :: String -> String -> Int -> Int -> Int
2  dist' xs ys 0 j = j
3  dist' xs ys i 0 = i
4  dist' xs ys i j = minimum
5    [dist' xs ys (i - 1) j + 1
6    ,dist' xs ys (i - 1) (j - 1) + if x == y then 0 else 1
```

```
7       ,dist' xs ys i (j - 1) + 1]
8     where
9       x = xs !! i
10      y = ys !! j
```

Note that we are now checking the strings in reverse. This can be tabulated with a pair (Int, Int);

```
1  dist'' :: String -> String -> Int
2  dist'' xs ys = table ! (m, n)
3    where
4      m = length xs
5      n = length ys
6
7      table = tabulate ((0, 0), (m, n)) mdist
8
9      mdist 0 j = j
10     mdist i 0 = i
11     mdist i j = minimum
12       [table ! ((i - 1), j) + 1
13       ,table ! ((i - 1), (j - 1)) + if  x == y then 0 else 1
14       ,table ! (i, (j - 1)) + 1]
15       where
16         x = xs !! i
17         y = ys !! j
```

While x = xs !! i and y = ys !! j isn't constant time, we assume we can simply tabulate it to obtain constant time access, and ignore it. The original version was $O(3^{m+n})$, and this has been reduced to $O(mn)$.

November 8th 2019

Note that this lecture is 3 hours long, with the first hour being in **CO220**. It starts with a remark that the examples of dynamic programming we've looked at so far only gives us the answer, and not how the answer was reached.

The Travelling Salesman Problem

The travelling salesman problem is notoriously difficult to solve efficiently. Given $p_0, \ldots, p_n$, the goal is to find the shortest path in a graph that visits all these nodes and returns to the start.
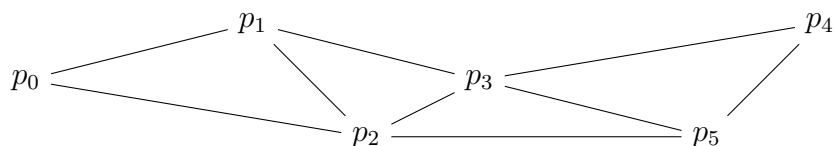
Bitonic Travelling Salesman

The bitonic version, abbreviated to BTSP, has nodes on a plane instead. We then make the following assumptions;

1. every node is connected to every other node, and the distance (cost) is simply the Euclidean distance

2. no two points exist on top of each other (such that no two points have the same x co-ordinate, if we looked at the xy plane)

3. we have a bitonic path - this means that it starts from the left-most node and goes to the right-most node, and then back again (see the examples below)

p_1 ... p_0 ... p_3 ... p_2 — valid
p_1 ... p_0 ... p_3 ... p_2 — valid
p_1 ... p_0 ... p_3 ... p_2 — invalid
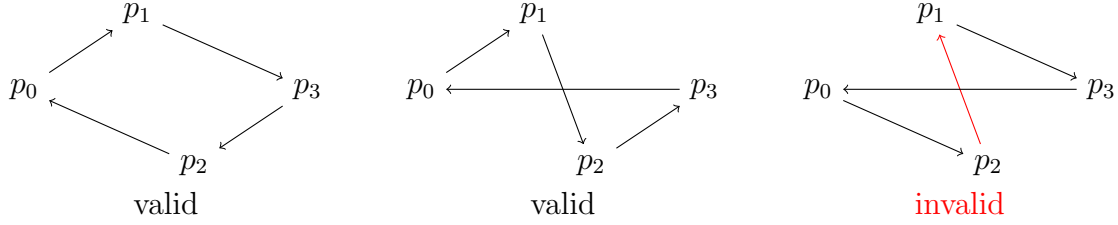
Note that the left-most node must be connected to the second left-most node, and the right-most node must be connected to the second right-most node.

This can be solved using dynamic programming in $O(m^2)$ time, for m nodes. Note that for the cases below, `bitonic i` means the shortest path up to `i+1`, and we write $\overline{p_i p_j}$ to denote the (Euclidean) distance between p_i and p_j;

- **base case** $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ `bitonic 0` $= 2 \cdot \overline{p_0 p_1}$

- **recursive case**

  Let `bitonic i` be the shortest bitonic path for some value i. We can also assume that we've calculated `bitonic i` for all $i \leq n$ (strong induction).
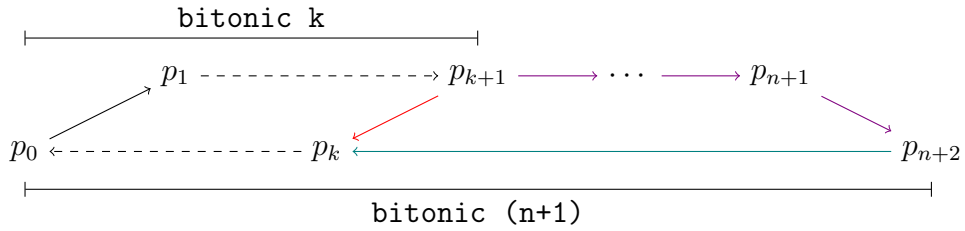
  p_{n+2} must connect to p_{n+1}, and also some p_k. Bitonicity means that $p_{k+1} \to \cdots \to p_{n+1}$ must be connected. Therefore, we can conclude;

  $$\texttt{bitonic (n+1)} = \texttt{bitonic k} + \overline{p_{k+1} \ldots p_{n+2}} - \overline{p_{k+1} p_k} + \overline{p_{n+2} p_k}$$

  As a side note, if there was a better path that uses some point after p_{k+1}, that would be picked up by taking the minimum value for all `k`s.

We must now find this k, but it must be minimal (assume `d :: Int -> Int -> Double` gives us the distance between two points, such that `d i j` $= \overline{p_i p_j}$);

```haskell
import Data.Array
import Data.List

type Point = (Double, Double)

ps :: Array Int Point
ps = array (0, 3) (zip [0 .. ] [(0,0), (1,2), (2,3), (3,3)])

dist :: Array Int Point -> Int -> Int -> Double
dist ps i j = sqrt (x*x + y*y)
  where
    (xi, yi) = ps ! i
    (xj, yj) = ps ! j
```

```
14      x = xi - xj
15      y = yi - yj
16
17  bitonic :: (Int -> Int -> Double) -> Int -> Double
18  bitonic d 0 = 2 * d 0 1
19  bitonic d n =
20    minimum [ bitonic d k - d k (k + 1) + d k (n + 1)
21             + sum [ d (i) (i + 1) | i <- [k + 1 .. n]]]
22             | k <- [0 .. n - 1]]
23
24  -- returns edges as pairs of points
25  bitonic' :: (Int -> Int -> Double) -> Int -> (Double, [(Int, Int)]
26  bitonic' d 0 = (2 * d 0 1, [(0, 1)])
27  bitonic' d n =
28    minimum [ bitonic' d k ~- (k, k + 1) ~+ (k, n + 1)
29             ~+~ ssum [(d i (i + 1), [(i, (i + 1)])) | i <- [k + 1 .. n]]
30             | k <- [0 .. n - 1]]
31    where
32     (~-) :: (Double, [(Int, Int)]) -> (Int, Int) -> (Double [(Int, Int)])
33     (x, ps) ~- (i, j) = (x - d i j, ps \\ [(i, j), (j, i)])
34
35     (~+) :: (Double, [(Int, Int)]) -> (Int, Int) -> (Double [(Int, Int)])
36     (x, ps) ~+ (i, j) = (x + d i j, (i, j):ps)
37
38     (~+~) :: (Double, [(Int, Int)]) -> (Double, [(Int, Int)]) -> (Double, [(Int,
            Int)])
39     (x, ps) ~+~ (y, qs) = (x + y, ps ++ qs)
40
41     ssum :: [(Double, [(Int, Int)])] -> (Double, [(Int, Int)])
42     ssum = foldr1 (~+~)
```

Note that the "strange" operators are used to perform operations on the pairs.

12th November 2019

Introduction to Red-Black Trees

These trees have the same asymptotic complexity as binary search (AVL) trees, which is $O(\log n)$ for access, and insertion. In terms of pragmatics, binary trees had balancing operations after every insertion, which was constant, but still a large overhead. Red-Black trees have a cheaper rebalancing, but the trees become more lopsided (but has more expensive access). In addition, we had to keep the height of the tree with binary trees (an integer), but Red-Black trees have a single bit.

Assuming an empty node is a black node, RB trees have the following invariants;

1. every red node has a black parent

2. every path from the root to a leaf must have the same number of black nodes

For invariant 2, if we had no red nodes, we will have a perfectly balanced tree. Due to invariant 1, the most lopsided a tree can become is if we alternated black, red, black, red, and so on, as it must still have the same number of black nodes.
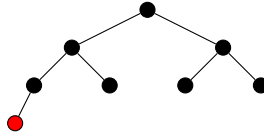
Some examples are as follows;

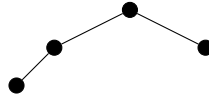- **valid RB tree**   perfectly balanced tree, with no red nodes
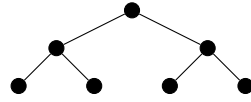
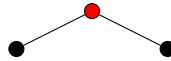- **valid RB tree**   same number of black nodes in each path
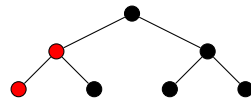
- **invalid RB tree**   invariant 2 does not hold

- **invalid RB tree**   invariant 1 does not hold

- **invalid RB tree**   neither invariants hold

Similarly, we are also ensuring that elements in the left subtree are smaller, and elements in the right subtree are larger. The implementation is as follows;

```haskell
data Colour = R | B                          -- (R)ed | (B)lack
data RBTree a = E | N Colour (RBTree a) a (RBTree a) -- (E)mpty | N(ode)

member :: Ord a => a -> RBTree a -> Bool
member x E = False
member x (N _ l y r)
  | x < y  = member x l
  | x == y = True
  | x > y  = member x r

insert :: Ord a => a -> RBTree a -> RBTree a
insert x t = blacken (insx t)
  where
    insx E = N R E x E
    insx (N c l y r)
      | x < y  = balance c (insx l) y r
      | x == y = N c l y r
      | x > y  = balance c l y (insx r)

-- ensures invariant 1 holds
blacken :: RBTree a -> RBTree a
blacken E = E
blacken (N _ l y r) = N B l y r
```

For `balance`, there are 4 possible cases (we can assume that the subtrees, a, b, c, d, have already been fixed due to recursion);
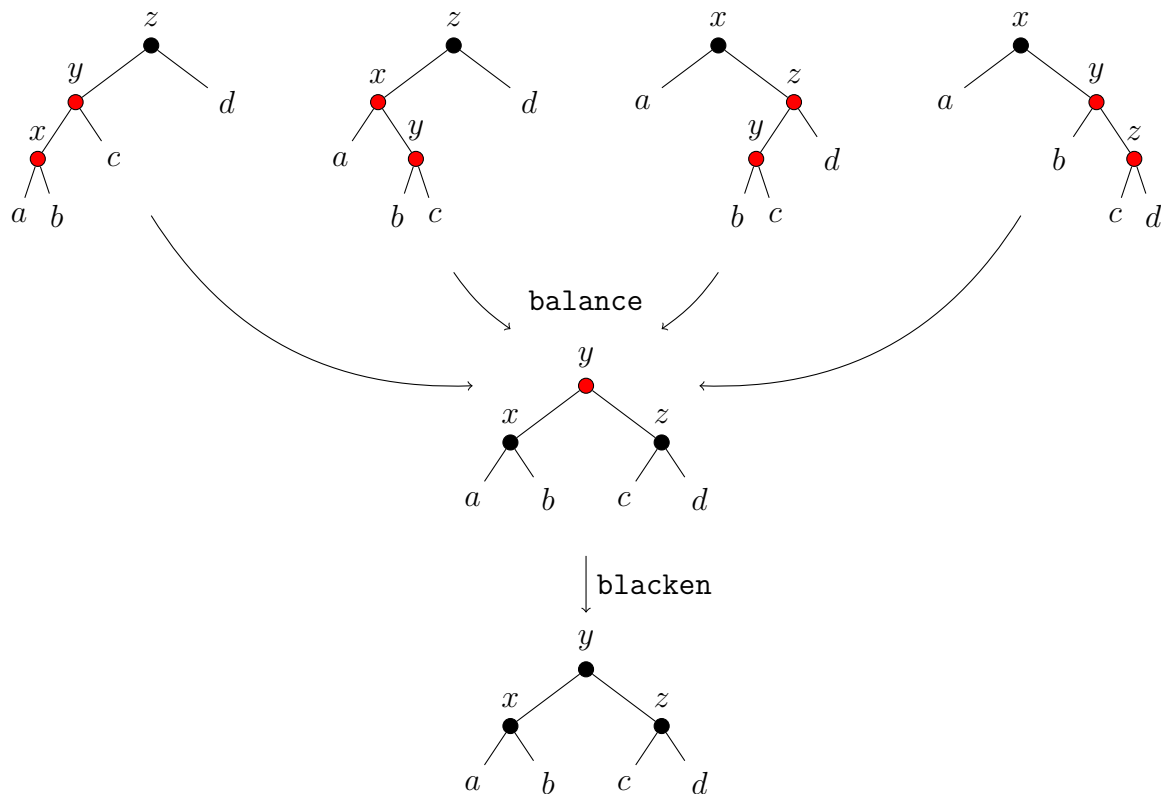
As shown above, `balance` is a simple pattern match (each B case corresponds to a case above);

```
24  balance :: Colour -> RBTree a -> a -> RBTree a -> RBTree a
25  balance B (N R (N R a x b) y c) z d = t
26  balance B (N R a x (N R b y c)) z d = t
27  balance B a x (N R (N R b y c) z d) = t
28  balance B a x (N R b y (N R c z d)) = t
29  -- case without a black root
30  balance c l x r = N c l x r
31    where -- not valid Haskell, can't span across multiple patterns
32      t = N R (N B a x b) y (N B c z d)
```

Persistent vs Mutable Data Structures

Note that with a persistent data structure, the code above will take in a "pointer" to the root of the tree, and return a **new** pointer, which is desired. This is useful in the case where we have multiple threads running, as we don't want the data structure to change unexpectedly. On the other hand, a mutable data structure modifies the tree, but has more efficient transformations (and is more complex in terms of cases).
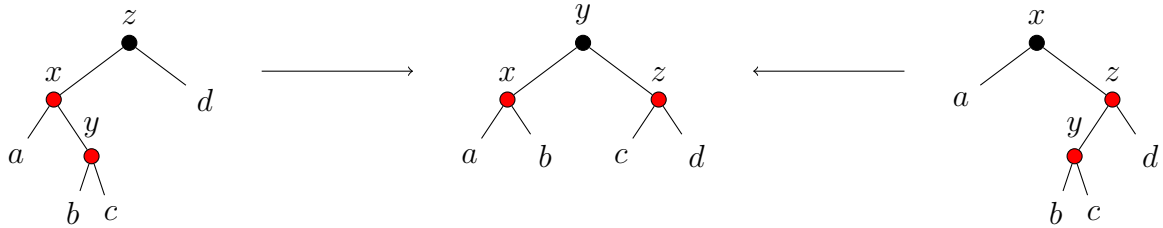
- **colour flip**
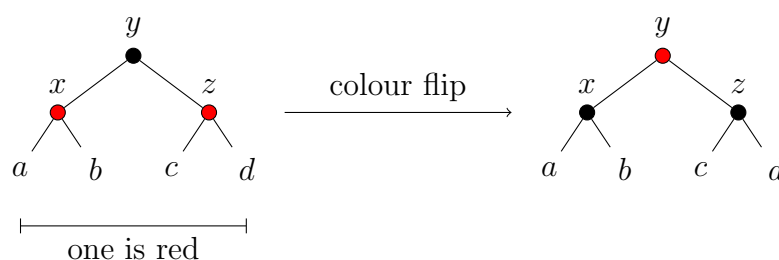
- **rotation**
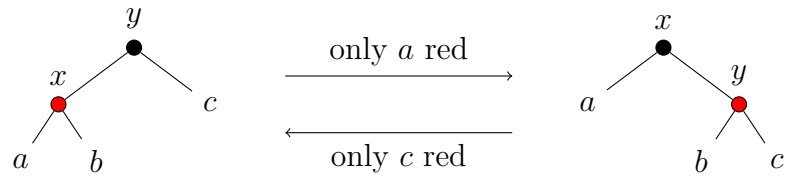
• **double rotation**

In an imperative setting, for the colour flip, only three values need to be changed (a, b, c, d no longer cause a red-red conflict, as the parents are now black). Note that the transformations can be aborted early, unlike in the persistent setting, since only parts of the tree need to be modified.

November 15th 2019

Note that this lecture has no sound.
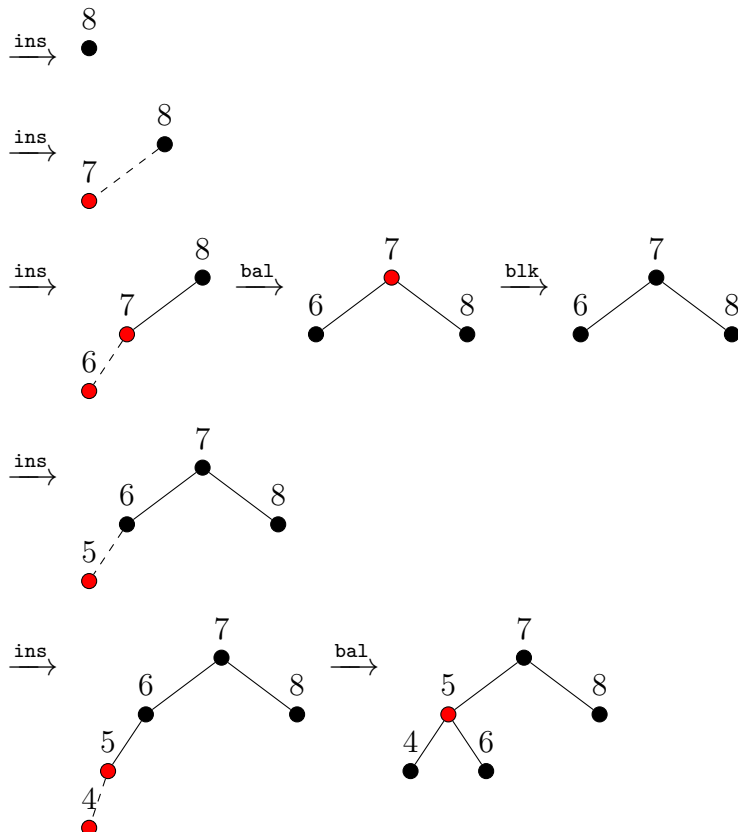
Construction of Red-Black Trees

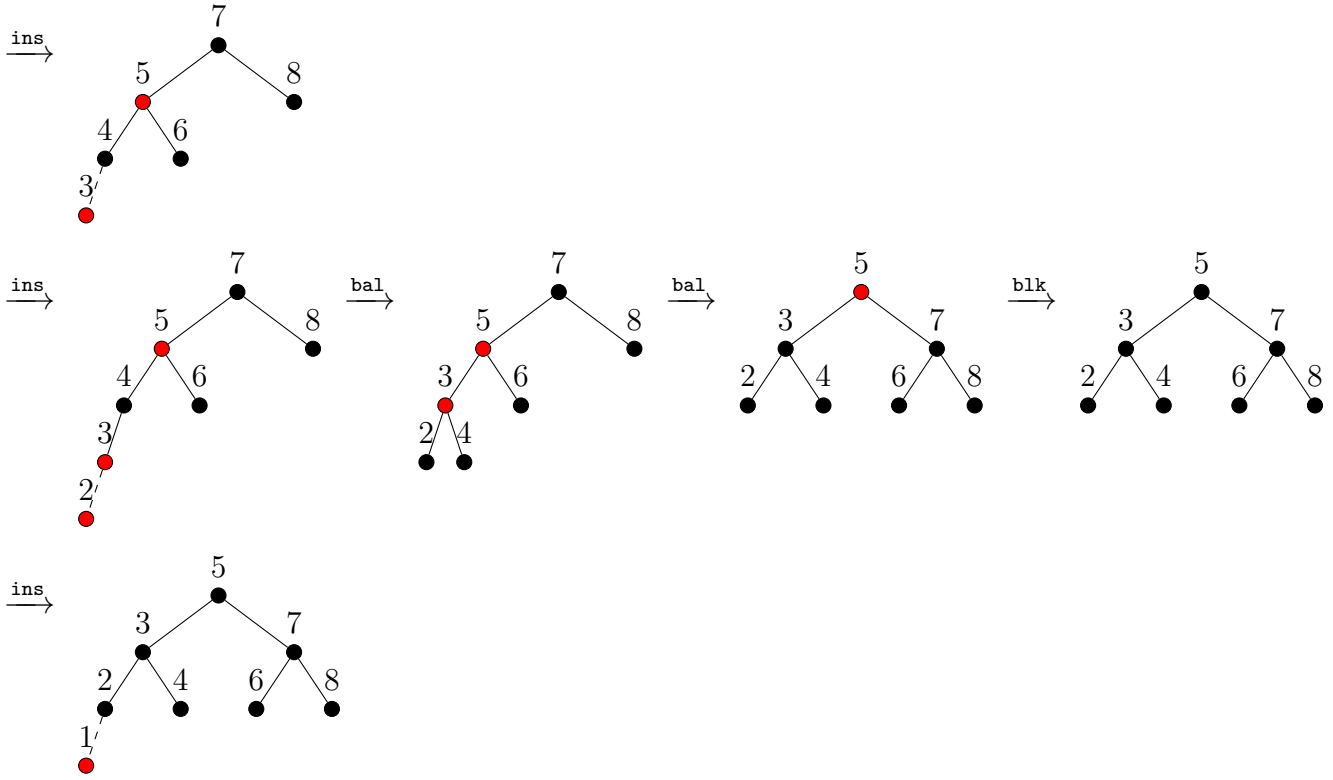Constructing this tree can be done with repeated addition, similar to before;

```
1  fromList :: Ord a => [a] -> RBTree a
2  fromList = foldr insert E
```

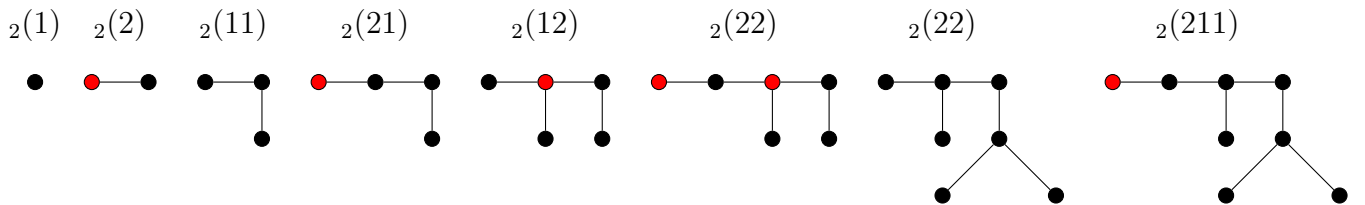Applying it to the list `[1..8]`, we get the following;

Pennant Trees

A Pennant tree of rank r is a complete binary tree of height r, with an extra node above the complete binary tree's root, which is used as the Pennant tree's root (hence it has 2^r nodes, instead of the $2^r - 1$ of the complete binary tree).

The sequence of trees can be drawn as;

This uses radix 2 base;

$$(b_{n-1} \ldots b_0)_2 =_2 (b_0 \ldots b_{n-1}) = \sum_{i=0}^{n-1} b_i 2^i$$

```
1  data Digit a = One a (RBTree a) | Two a (RBTree a) a (RBTree a)
2
3  incr :: a -> RBTree a -> [Digit a] -> [Digit a]
4  incr x t [] = [One x t]
5  incr x t ((One y u):ps) = (Two x t y u):ps
6  incr x t ((Two y u z v):ps) = (One x t):incr y (N B u z v) ps
7
8  fromList' :: [a] -> RBTree a
9  fromList' = links  . foldr add []
10
11 links = foldl link E
12
13 link l (One x t) = N B l x t
14 link l (Two x t y u) = N B (N R l x t) y u
```

The idea is to link the data structure to something simpler that we can do with low complexity, such as counting.

19th November 2019

Random Algorithms

Generally there are two approaches to random algorithms;

- **Monte Carlo**   gives answer very quickly, but low probability of being incorrect
- **Las Vegas**   gives answer in varied amounts of time, but correct

Treaps

A `Treap` is a combination of a tree and a heap. The tree is a binary search tree in symmetric order, such that elements to the left of a node are smaller, and elements to the right are larger. The heap is imposed to balance. Note that we pair the example below as (`value`, `priority`), and the priority is in rank order (hence lower values have high priority) - the priorities should be unique.
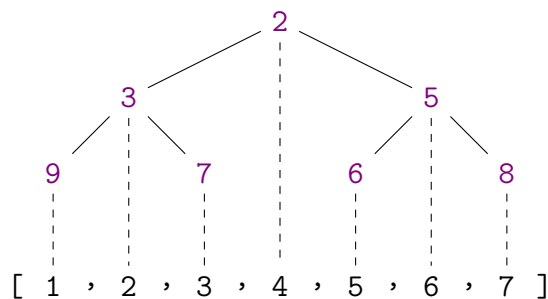
This is a heap on the priorities, but a binary search tree on the values;

```
1  data Treap a = Empty | Node (Treap a) a Int (Treap a)
2
3  member :: Ord a => a -> Treap a -> Bool
4  member x Empty = False
5  member x (Node a y p b)
6    | x < y  = member x a
7    | x == y = True
8    | x > y  = member x b
9
10 -- insertion takes in a value and a priority
11 insert :: Ord a => a -> Int -> Treap a -> Treap a
12 insert x p Empty = Node Empty x p Empty
13 insert x p (Node a y q b)
14   -- smart constructors used to maintain invariant
15   | x < y  = lnode (insert x p a) y q b
16   | x == y = Node a y q b
17   | x > y  = rnode a y q (insert x p b)
18
19 lnode :: Treap a -> a -> Int -> Treap a -> Treap a
20 lnode Empty y q c = Node Empty y q c
21 lnode l@(Node a x p b) y q c
22   | q <= p    = Node l y q c
23   | otherwise = Node a x p (Node b y q c)
24
25 rnode :: Treap a -> a -> Int -> Treap a -> Treap a
```

```
26  rnode a x p Empty = Node a x p Empty
27  rnode a x p r@(Node b y q c)
28    | p >= q    = Node a x p r
29    | otherwise = Node (Node a x p b) y q c
```

The smart constructors, `lnode` and `rnode`, ensure that the node with the highest priority (lowest value) is the parent node.

Note that in general, when writing these algorithms, base it off the data structure (instead of considering the conditions). Deletion is a combination of searching and merging (we do not need a priority);

```
30  delete :: Ord a => a -> Treap a -> Treap a
31  delete x Empty = Empty
32  delete x (Node a y p b)
33    | x < y  = Node (delete x a) y p b
34    | x == y = merge a b
35    | x > y  = Node a y p (delete x b)
36
37  merge :: Treap a -> Treap a -> Treap a
38  merge Empty r = r
39  merge l Empty = l
40  merge l@(Node a x p b) r@(Node c y q d)
41    | p < q    = Node a x p (merge b r)
42    | otherwise = Node (merge l c) y q d
```

Turing a `Treap` into a list is as follows (since we can use it as a sorting algorithm);

```
43  toList :: Treap a -> [a]
44  toList Empty = []
45  toList (Node a x p b) = toList a ++ [x] ++ toList b
```

Making a `Treap` requires values and priorities;

```
46  fromList :: Ord a => [(a, Int)] -> Treap a
47  fromList xs = foldr (uncurry insert) Empty xs
48
49  -- recall uncurry :: (a -> b -> c) -> ((a, b) -> c)
```

Randomised Quicksort

Leibniz Law says the following;

$$x = y \Rightarrow f\ x = f\ y$$

Random algorithms violate this, therefore we aim to have a deterministic algorithm that depends on something non-deterministic (by "feeding in" randomness). Passing in `StdGen` means that we do not violate Leibniz Law, as we are passing in the randomness as a parameter.

```
1  rquicksort :: Ord a => StdGen -> [a] -> [a]
2  rquicksort g xs = toList (fromList (zip xs ps))
3    where
4      ps = randoms g
```

22nd November 2019

Note on Balancing Treaps and Leibniz Law

The `merge` function does not take into account the heights of the trees, unlike previous tree structures where more work had to be done to rebalance them at each stage. Instead, we hope that the tree structure is balanced due to the heap, and that it "probably" will be with the randomised priorities.

This allows us to randomly choose pivot points for quicksort - thus the `Treap` generated represents the decision tree for quicksort, with all the pivots being randomly chosen.

There are two techniques we can use to satisfy Leibniz Law, given that `f ::  a -> b`, we can modify it as follows;

1. **explicit parameter** `f' ::  StdGen -> a -> b`
2. **implicit parameter** `f'' ::  MonadRandom m => a -> m b`

Treaps (Continued)

Recall the insert function on `Treap`s is as follows;

```
1  insert :: Ord a => a -> Int -> Treap a -> Treap a
```

When we are designing a data structure that invokes randomness, it's useful to keep the random seed within the data structure, as it allows the end users to produce the same results. Rather than expecting a priority to be passed in we can put it into the data structure, which also gives us an insertion function of the desired type;

```
1  data RTreap a = RTreap StdGen (Treap a)
2
3  insert' :: Ord a => a -> RTreap a -> RTreap a
4  insert' x (RTreap seed t) = RTreap seed' (insert x p t)
5    where
6      (p, seed') = next seed
```

An `RTreap` must be instantiated, and this can be done when creating an empty `RTreap`;

```
1  empty :: RTreap a
2  empty = RTreap (mkStdGen 42) Empty
```

Randomness can be implemented using the same methods;

1. explicitly;

   ```
   1  empty' :: StdGen -> RTreap a
   2  empty' seed = RTreap seed Empty
   ```

2. implicitly

   ```
   1  empty'' :: MonadRandom m => m (RTreap a)
   2  empty'' = do seed <- newStdGen
   3               return (RTreap seed Empty)
   ```

Randomness enters the program through the main function;

```
1  main :: IO ()
2  main = do t <- empty''
3            ... -- t :: RTreap a
```

We have the following functions for generating random values;

- `random ::  Random a => StdGen -> (a, StdGen)` any possible value for `a`
- `randomR ::  Random a => (a, a) -> StdGen -> (a, StdGen)` any possible value in range
- `randoms ::  Random a => StdGen -> [a]` produces a list of random values
- `randomRs ::  Random a => (a.  a) -> StdGen -> [a]` similar

Monte Carlo Pi

Calculation of π can be done with a Monte Carlo randomised algorithm, to some degree of accuracy.
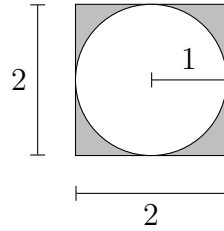
The area of the whole square is 4, and the area of the circle is π. If we randomly choose points inside the circle, the probability of it landing inside the circle is $\frac{\pi}{4}$.

```haskell
pi :: MonadRandom m => m Double
-- first 0 is the number of circle hits
-- second 0 is the number of "throws" so far
pi = loop 1000 0 0
  where
    loop 0 m n = return (4 * fromIntegral m / fromIntegral n)
    loop s m n = do
      x <- getRandomR (0, 1)
      y <- getRandomR (0, 1)
      let m' = if inside (x, y) then m + 1 else m
          n' = n + 1
          s' = s - 1
      loop s' m' n'

inside :: (Double, Double) -> Bool
inside (x, y) = x*x + y*y <= 1
```

A more functional version is as follows (this forms a clean separation from the randomness in lines 2 and 3, with the purely mathematical code below);

```haskell
pi' :: MonadRandom m => m Double
pi' = do xs <- getRandomRs (0, 1 :: Double)
         ys <- getRandomRs (0, 1 :: Double)
         let xys = take 1000 (zip xs ys)
             bs = map inside xys
             m = length (filter id bs)
             return (4 * fromIntegral m / 1000)
```

Randomised BTrees

Previously, we would need to do an "expensive" rebalancing (rotation) on every insertion with our AVL trees. The randomised version simply inserts blindly, without rebalancing, however there is a small probability on every insertion that the tree is "re-rooted" with the newly inserted value, with a probability of $\frac{1}{n+1}$, where n is the size of the tree.

```haskell
import System.Random

data BTree a = BNil | BNode (BTree a) a (BTree a)
  deriving Show

insert :: Ord a => a -> BTree a -> BTree a
insert x BNil = BNode BNil x BNil
```

```haskell
8   insert x (BNode l y r)
9     | x < y  = BNode (insert x l) y r
10    | x == y = BNode l y r
11    | x > y  = BNode l y (insert x r)
12
13  -- this inserts at the root, and then performs rotations (same cost as AVL)
14  insertRoot :: Ord a => a -> BTree a -> BTree a
15  insertRoot x BNil = BNode BNil x BNil
16  insertRoot x (BNode l y r)
17    | x < y  = rotr (insertRoot x l) y r
18    | x == y = BNode l y r
19    | x > y  = rotl l y (insertRoot x r)
```
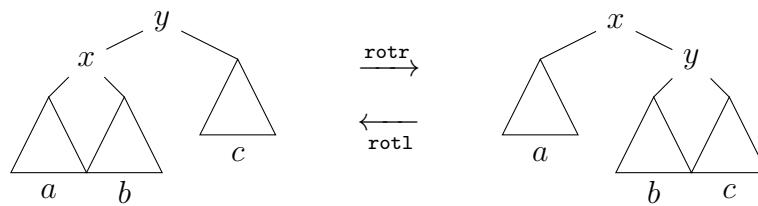
The rotations, `rotr` and `rotl`, can be captured with the following diagram;


```haskell
20  rotr :: BTree a -> a -> BTree a -> BTree a
21  rotr (BNode a x b) y c = BNode a x (BNode b y c)
22
23  rotl :: BTree a -> a -> BTree a -> BTree a
24  rotl a x (BNode b y c) = BNode (BNode a x b) y c
25
26  -- this is a randomised BTree, not a Red Black Tree
27  data RBTree a = RBTree StdGen Int (BTree a)
28    deriving Show
29
30  empty :: RBTree a
31  empty = RBTree (mkStdGen 42) 0 BNil
32
33  insert' :: Ord a => a -> RBTree a -> RBTree a
34  insert' x (RBTree seed n t)
35    | p == 0    = RBTree seed' (n+1) (insertRoot x t)
36    | otherwise = RBTree seed' (n+1) (insert x t)
37    where
38      (p, seed') = randomR (0, n) seed
```

Pennant Trees (Recap?)

It actually cuts out at the end when he starts to go over it, but there is a useful paper on it;

   https://pdfs.semanticscholar.org/b7eb/ce70900c26125240537ba722aeec2cf44a2e.pdf

Referring back to the "left-spine" view of the trees from the previous lecture - there is a pattern. If r is the rank of the right-most pennant, the black condition implies a pennant of rank i appears either once or twice, for all $0 \leq i \leq r$. Since the size of rank i pennant is 2^i, it corresponds to the 1-2 system. It abandons 0, therefore each natural number has a unique representation.

November 26th 2019

Simple PageRank

A simple implementation of PageRank is the sum of the SimplePageRanks of the pages pointing to a particular node divided by the number of nodes it's pointing to.
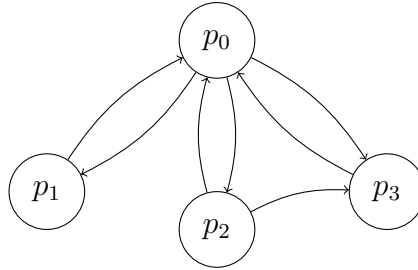
$$SPR(p_0) = \frac{SPR(p_1)}{1} + \frac{SPR(p_2)}{2} + \frac{SPR(p_3)}{1}$$
$$SPR(p_1) = \frac{SPR(p_0)}{3}$$
$$SPR(p_2) = \frac{SPR(p_0)}{3}$$
$$SPR(p_3) = \frac{SPR(p_0)}{3} + \frac{SPR(p_1)}{2}$$

Let $S(p_i)$ be the sources of p_i (what links to it), and $T(p_i)$ be the targets of p_i (where it links to);

$$SPR(p_i) = \sum_{p_j \in S(p_i)} \frac{SPR(p_j)}{|T(p_j)|}$$

PageRank

In reality, there needs to be some damping factor, as a user may stop clicking at some point. Additionally, a user may reach a page from elsewhere, and not just from clicking links.

$$PR(p_i) = \frac{1-d}{n} + d \sum_{p_j \in S(p_i)} \frac{PR(p_j)}{|T(p_j)|}$$

Note that n is the total number of pages in consideration, and we assume that all pages have outgoing links. Since PageRank is a probability distribution, all pages start with $PR(p_i) = \frac{1}{n}$, and this probability gets moved around. To avoid making the assumption, we can say that the page rank is distributed if it has no outgoing links;

$$PR(p_i) = \frac{1-d}{n} + d \sum_{p_j \in S(p_i)} \frac{PR(p_j)}{L(p_j)} \text{ where } L(p_j) = \begin{cases} |T(p_j)| & \text{if } |T(p_j)| > 0 \\ \frac{1}{n} & \text{otherwise} \end{cases}$$

29th November 2019

Ford-Fulkerson Max-Flow Method

In the graph below, we have the maximum compacity on each edge. There is a source node s, and a target node t.
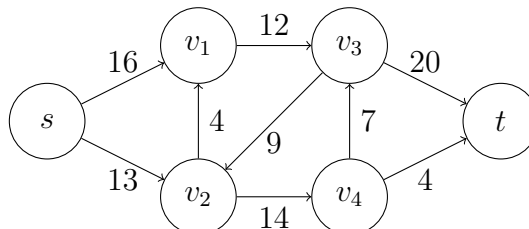
This starts by picking an augmenting path, which is any path from s to t. Once we pick the path, we add the minimum capacity of all the capacities on the path to the "flow used". We keep doing this until we cannot pick more paths - since the next path that is picked isn't exact, this is a method and not an algorithm.

An implementation (not working) of this in Haskell is as follows;

```
untilNothing :: (a -> Maybe b) -> (b -> a -> a) -> a -> a
-- (a -> Maybe a) takes a graph, and may produce a path
-- (b -> a -> a)  takes a path, a graph, and gives back a graph
-- a              is the initial graph
-- a              is the final graph
```