50001 - Algorithm Analysis and Design - Lecture 1

Oliver Killane

12/11/21

Lecture Recording

Lecture recording is available here

Introduction

An algorithm is a method of computing a result for a given problem, at its core in a systematic/-mathematical means.

This course extensively uses haskell instead of pesudocode to express problems, though its lessons still apply to other languages.

Fundamentals

Insertion Problem

Given an integer x and a sorted list ys, produce a list containing $x : ys$ that is ordered.

Note that this can be solved by simply using $sort(x : ys)$ however this is considered wasteful as it does not exploit the fact that ys is already sorted.

An example algorithm would be to traverse ys until we find a suitable place for x

```haskell
insert :: Int -> [Int] -> [Int]
insert x [] = [x]
insert x yss@(y:ys)
    | x <= y      = x:yss
    | otherwise = y : insert x ys
```

Call Steps

In order to determine the complexity of the function, we use a **cost model** and determine what steps must be taken.

For example for $insert\ 4\ [1, 3, 6, 7]$

$insert\ 4\ [1, 3, 6, 7]$
$\rightsquigarrow$ { definition of $insert$ }
$1 : insert\ 4\ [3, 6, 7]$
$\rightsquigarrow$ { definition of $insert$ }
$1 : 3 : insert\ 4\ [6, 7]$
$\rightsquigarrow$ { definition of $insert$ }
$1 : 3 :: 4 : [6, 7]$

Hence this requires 3 call steps.

We can use recurrence relations to get a generalised formula for the worst case (maximum number of calls):

$$\begin{aligned} T_{insert} 0 &= 1 \\ T_{insert} 1 &= 1 + T_{insert}(n - 1) \end{aligned}$$

We can solve the recurrence:

$$\begin{aligned}
T_{insert}(n) &= 1 + T_{insert}(n-1) \\
T_{insert}(n) &= 1 + 1 + T_{insert}(n-2) \\
T_{insert}(n) &= 1 + 1 + \cdots + 1 + T_{insert}(n-n) \\
T_{insert}(n) &= n + T_{insert}(0) \\
T_{insert}(n) &= n + 1
\end{aligned}$$

More complex algorithms

```haskell
isort :: [Int] -> [Int]
isort []     = []
isort (x:xs) = insert x (isort xs)
```

$$\begin{aligned}
T_{isort} 0 &= 1 \\
T_{isort} n &= 1 + T_{insert}(n-1) + T_{isort}(n-1)
\end{aligned}$$

Hence by using our previous formula for *insert*

$$T_{isort} n \quad = 1 + n + T_{isort}(n-1)$$

And by recurrence:

$$\begin{aligned}
T_{isort}(n) &= 1 + n + (1 + n - 1) + T_{isort}(n-2) \\
T_{isort}(n) &= 1 + n + (1 + n - 1) + (1 + n - 2) + \cdots + T_{isort}(n-n) \\
T_{isort}(n) &= 1 + n + (1 + n - 1) + (1 + n - 2) + \cdots + T_{isort}(0) \\
T_{isort}(n) &= n + n + (n-1) + (n-2) + \cdots + (n-n) + 1 \\
T_{isort}(n) &= 1 + n + \textstyle\sum_{i=0}^{n} i \\
T_{isort}(n) &= \textstyle\sum_{i=0}^{n+1} i \\
T_{isort}(n) &= \frac{(n+1) \times (n+1)}{2}
\end{aligned}$$

50001 - Algorithm Analysis and Design - Lecture 2

Oliver Killane

12/11/21

Lecture Recording

Lecture recording is available here

Evaluation & Cost Models

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

When analysing the cost of **minimum** we must consider how the function is evaluated.

For example we could shortcut the once **isort** has determined the first element (the minimum) of the list.

Cost Model

A model to determine the time taken to execute a program.

The model assigns cost to different operations (e.g comparisons, calls, memory reads/writes)

A very generalised cost model assigns cost based on the number of **reductions** required to evaluate a program.

Small While Language

We can define a small language of expressions as follows:

$$e ::= x \mid k \mid f\ e_1 \dots e_n \mid \text{if } e \text{ then } e_1 \text{ else } e_2$$

where k means constant and x is the variable form.
Infix functions such as $+, -, \times$ are written normally, and are also expressions as they can be used in the form $(+)\ e_1\ e_2$.

There are also several primitive constants: $True, False, 0, 1, 2, \dots$
List constants and operations are also primitive: $[], (:), \mathit{null}, \mathit{head}, \mathit{tail}$

Evaluation Order

- **Applicative Order** Strict evaluation
  The leftmost, innermost reducible expression is evaluated first.

e.g for $fst(3 \times 2, 1+2)$
$fst(3 \times 2, 1+2)$
$\leadsto$ { Definition of $\times$ }
$fst(6, 1+2)$
$\leadsto$ { Definition of $+$ }
$fst(6, 3)$
$\leadsto$ { Definition of fst }
6

- **Normal Order** Lazy evaluation
  The leftmost outer reducible is evaluated first. Efectively evaluating the function before its arguments.
  e.g for $fst(0, 1+2)$
  $fst(3 \times 2, 1+2)$
  $\leadsto$ { Definition of fst }
  3×2
  $\leadsto$ { Definition of $\times$ }
  6

For a given program, if **applicative** and **normal** terminate, then they produce the same value in normal form.

However there are some programs where **normal** evaluation terminates, but **applicative** will not.

Applicative	**Normal**
$fst(0, \text{crazy nonesense})$	$fst(0, \text{crazy nonesense})$
$\leadsto$ { By lack of definition for crazy nonesense }	$\leadsto$ { Definition of fst }
$CRASH!$	0

The program may by syntactically correct, but have an error such as zero-division which will not be evaluated and hence not result in improper termination under **normal** order.

$$\text{Applicative Terminates} \Rightarrow \text{Normal Terminates}$$

Cost Model for Small While

We can evaluate a cost model for the small while language by creating a function T to assign cost to expressions.

Type	Function	Explanation
non-primitive function	$f\ a_1\ \dots\ a_n = e$ $T(f)\ a_1\ \dots\ a_n = T(e) + 1$	Given we have already computed all argument, the cost of the function is the cost of the expression it produces, and a single call.
primitive function	$T(f)\ x \dots\ x_n = 0$	Primitive functions are assumed to be free.
Variable	$T(x) = 0$	accessing variables is free.
Application	$T(f\ e_1\ \dots\ e_n) = T(f)\ e_1\ \dots\ e_n + T(e_1) + \cdots + T(e_n)$	When applying a function we must consider both its cost, and the cost of all argument expressions.
Conditional	$T(\text{if } p \text{ then } e_1 \text{ else } e_2) = T(p) + \text{if } p \text{ then } T(e_1) \text{ else } T(e_2)$	Cost of condition and of the resulting expression.

Cost Model Example

Given the function:

$$mul\ m\ n = \text{if } m = 0 \text{ then } 0 \text{ else } n + mul\ (m-1)\ n$$

Evaluate $T(mul\ 3\ 100)$

(1)	$mul\ 3\ 100$	
(2)	$T(\text{if } 3 = 0 \text{ then } 0 \text{ else } 100 + mul\ (3-1)\ 100) + 1$	By Rule for non-primitive functions
(3)	$T(3 = 0) + T(100 + mul\ (3-1)\ 100) + 1$	By rule for conditionals
(4)	$0 + T(100 + mul\ (3-1)\ 100) + 1$	By primitive functions
(5)	$T(+)(100\ mul\ (3-1)\ 100) + T(100) + T(mul\ (3-1)\ 100) + 1$	By application rule
(6)	$0 + T(100) + T(mul\ (3-1)\ 100) + 1$	By rule for primitive functions
(7)	$0 + T(mul\ (3-1)\ 100) + 1$	By rule for constants
(8)	$T(mul)\ (3-1)\ 100 + T(3-1) + T(100) + 1$	By application rule
(9)	$T(mul)\ (3-1)\ 100 + T(-)3\ 1 + T(100) + 1$	By application rule
(10)	$T(mul)\ 2\ 100 + 1$	By application rule
(11)	$T(\text{if } 2 = 0 \text{ then } 0 \text{ else } 100 + mul\ (2-1)\ 100) + 1 + 1$	By Rule for non-primitive functions
(12)	$T(2 = 0) + T(100 + mul\ (2-1)\ 100) + 2$	By rule for conditionals
(13)	$0 + T(100 + mul\ (2-1)\ 100) + 2$	By primitive functions
(14)	$T(+)(100\ mul\ (2-1)\ 100) + T(100) + T(mul\ (2-1)\ 100) + 2$	By application rule
(15)	$0 + T(100) + T(mul\ (2-1)\ 100) + 2$	By rule for primitive functions
(16)	$0 + T(mul\ (2-1)\ 100) + 2$	By rule for constants
(17)	$T(mul)\ (2-1)\ 100 + T(2-1) + T(100) + 2$	By application rule
(18)	$T(mul)\ (2-1)\ 100 + T(-)2\ 1 + T(100) + 2$	By application rule
(19)	$T(mul)\ 1\ 100 + 2$	By application rule
(20)	$T(\text{if } 1 = 0 \text{ then } 0 \text{ else } 100 + mul\ (1-1)\ 100) + 2 + 1$	By Rule for non-primitive functions
(21)	$T(2 = 0) + T(100 + mul\ (1-1)\ 100) + 3$	By rule for conditionals
(22)	$0 + T(100 + mul\ (1-1)\ 100) + 3$	By primitive functions
(23)	$T(+)(100\ mul\ (1-1)\ 100) + T(100) + T(mul\ (1-1)\ 100) + 3$	By application rule
(24)	$0 + T(100) + T(mul\ (1-1)\ 100) + 3$	By rule for primitive functions
(25)	$0 + T(mul\ (1-1)\ 100) + 3$	By rule for constants
(26)	$T(mul)\ (1-1)\ 100 + T(1-1) + T(100) + 3$	By application rule
(27)	$T(mul)\ (1-1)\ 100 + T(-)2\ 1 + T(100) + 3$	By application rule
(28)	$T(mul)\ 0\ 100 + 3$	By application rule
(29)	$T(\text{if } 0 = 0 \text{ then } 0 \text{ else } 100 + mul\ (1-1)\ 100) + 3 + 1$	By Rule for non-primitive functions
(30)	$T(0 = 0) + T(0) + 4$	By rule for conditionals
(31)	$0 + T(0) + 4$	By rule for primitive functions
(32)	$0 + 4$	By rule for variables
(33)	4	

50001 - Algorithm Analysis and Design - Lecture 3

Oliver Killane

12/11/21

Lecture Recording

Lecture recording is available here

Asymptotics

L-Function

A **Logarithmico-exponential** function f is:

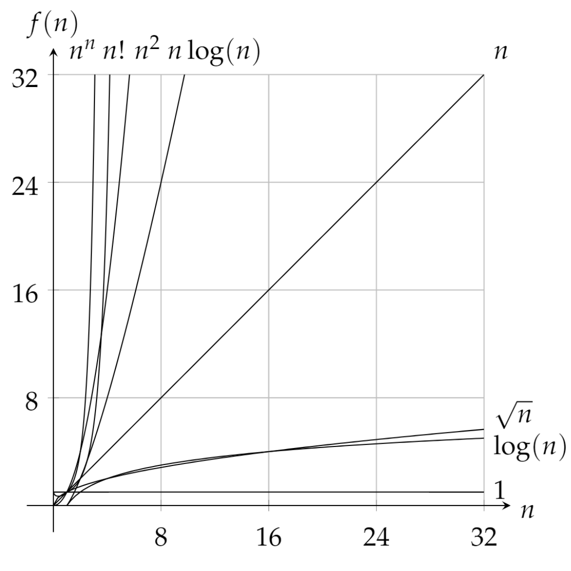
- real: $f \in X \to Y$ where $X, Y \subset \mathbb{R}$
- positive: $\forall x \in X.[f(x) \leq 0]$
- monotonic: $\forall x_1, x_2 \in X.[x_1 < x_2 \Leftrightarrow f(x_1) < f(x_2)]$ (positive monotonic) or $\forall x_1, x_2 \in X.[x_1 < x_2 \Leftrightarrow f(x_1) > f(x_2)]$ (negative monotonic)
- one valued: $\forall x \in X, y_1, y_2 \in Y.[f(x) = y_1 \wedge f(x) = y_2 \Rightarrow y_1 = y_2]$
- on a real variable defined for all values greater than some definite value: $X \equiv \{x | x > \text{definite limit} \wedge x \in \mathbb{R}\}$

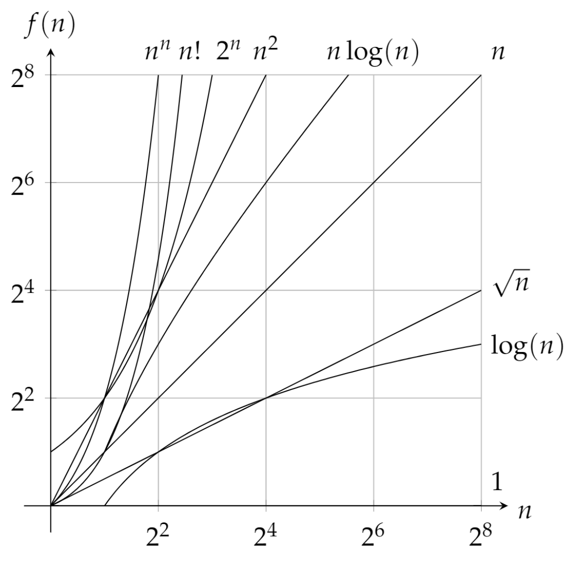
L-Functions are continuous, of constant sign and as $n \to \infty$ the value $f(n)$ tends to $0, \infty$ or some other positive definite limit.

Functions that aren't **L-Functions** are called **Wild Functions**.

In asymptotics we use **L-Functions** to describe the growth of time used by algorithms as the size of the input to an algorithm grows.

Common functions are shown below:





Du Bois-Reymond Theorem

Defines inequalities for the rate of increase of functions.

Where $lim = \lim_{n \to \infty} \dfrac{f(n)}{g(n)}$

$(<)$	$f \prec g \Leftrightarrow lim = 0$	g grows much faster than f
$(\leq)$	$f \preccurlyeq g \Leftrightarrow lim < \infty$	g grows much faster than f or some multiple of f
$(=)$	$f \asymp g \Leftrightarrow 0 < lim < \infty$	$g(n)$ grows towards some constant times $f(n)$
$(\geq)$	$f \succcurlyeq g \Leftrightarrow lim > 0$	f grows much faster than g or some multiple of g
$(>)$	$f \succ g \Leftrightarrow lim = \infty$	f grows much faster than g

These operators form a trichotomy such that one of the below will always hold:

$$f \prec g \quad f \asymp g \quad f \succ g$$

Further the operators $\succ$ and $\prec$ are converse:

$$f \succ g \Leftrightarrow g \prec f$$

And transitive:

$$f \prec g \land g \prec h \Rightarrow f \prec h$$
$$f \preccurlyeq g \land g \preccurlyeq h \Rightarrow f \preccurlyeq h$$

We can place the common **L-Functions** in order:

$$1 \prec \log n \prec \sqrt{n} \prec n \prec n \log n \prec n^2 \prec n^3 \prec n! \prec n^n$$

Bachman-Landau Notation

Comparison with Bois-Reymond	Set definition
$f \in o(g(n)) \Leftrightarrow f \prec g$	$o(g(n)) = \{f \mid \forall\delta > 0.\exists n_0 > 0.\forall n > n_0[f(n) < \delta g(n)]\}$
$f \in O(g(n)) \Leftrightarrow f \preccurlyeq g$	$O(g(n)) = \{f \mid \exists\delta > 0.\exists n_0 > 0.\forall n > n_0[f(n) \leq \delta g(n)]\}$
$f \in \Theta(g(n)) \Leftrightarrow f \asymp g$	$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$
$f \in \Omega(g(n)) \Leftrightarrow f \succcurlyeq g$	$\Omega(g(n)) = \{f \mid \exists\delta > 0.\exists n_0 > 0.\forall n > n_0[f(n) \geq \delta g(n)]\}$
$f \in \omega(g(n)) \Leftrightarrow f \succ g$	$\omega(g(n)) = \{f \mid \forall\delta > 0.\exists n_0 > 0.\forall n > n_0[f(n) > \delta g(n)]\}$

50001 - Algorithm Analysis and Design - Lecture 4

Oliver Killane

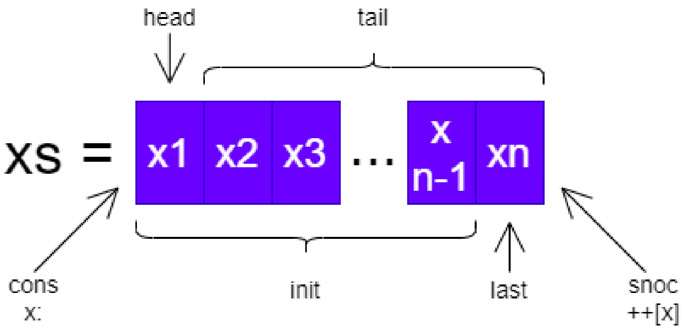
12/11/21

Lecture Recording

Lecture recording is available here

Lists

```haskell
data List [a] = [] | (:) a [a]

-- or...
data List a where
    Empty :: List a
    Cons :: a -> List a -> List a
```

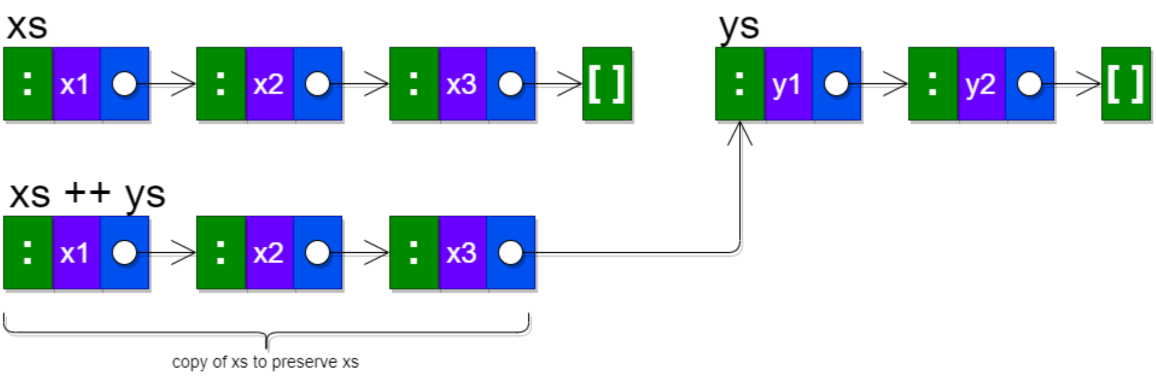


Lists in **Haskell** are a persistent data structure, meaning that when operations are applied to lists the original list is maintained (not mutated).

Append

We can append lists, by traversing over the first list, copying values (this ensures both argument lists are preserved).

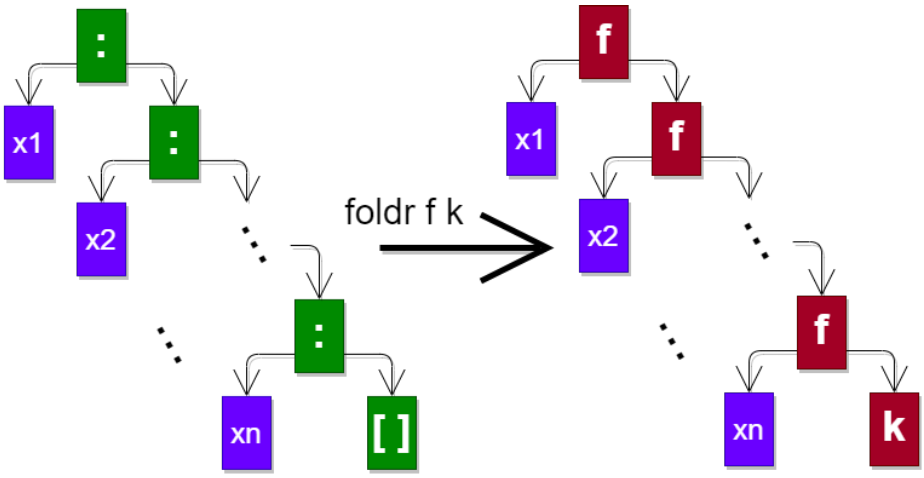
```haskell
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
```



As the entire first list must be traversed, the cost of $xs{+}{+}ys$ is $T_{(++)}(n) \in O(n)$ where $n = length\ xs$

Foldr

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f k []     = k
foldr f k (x:xs) = f x (foldr f k xs)
```



As you can see, $foldr\ (:)\ [] \equiv id$.
Foldr can also be expressed through bracketing

$$foldr\ f\ k\ [x_1, x_2, \ldots, x_n] \equiv f\ x_1\ (f\ x_2\ (\ldots(f\ x_n\ k)\ldots))$$

Associativity

Associativity determines how operations are grouped in the absence of brackets.

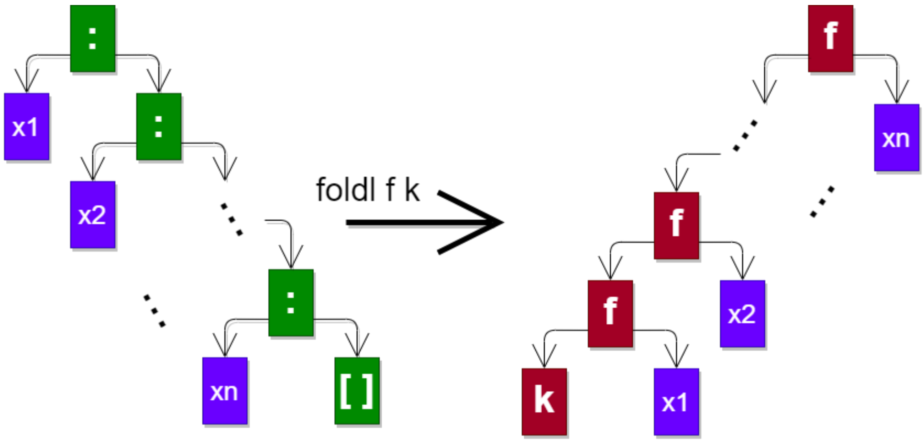
$a \spadesuit b \spadesuit c$	unbracketed statement
$((a) \spadesuit b) \spadesuit c$	$\spadesuit$ is left associative
$a \spadesuit (b \spadesuit (c))$	$\spadesuit$ is right associative

If $\spadesuit$ is associative, then the right & left associative versions are equivalent.

foldr applies functions in a right-associative scheme.

Foldl

```haskell
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f k []     = k
foldl f k (x:xs) = foldl f (f k x) xs
```



As you can see $foldl\ (snoc)\ []\equiv id$.
Foldl can be expressed through bracketing

$$foldl\ f\ k\ [x_1, x_2, \ldots, x_n] \equiv f\ (\ldots(f\ (f\ k\ x_1)x_2)\ldots x_n)$$

Monoids

Consider the case when for some $\bigstar$ and ϵ: $foldr\ \bigstar\ \epsilon \equiv foldl\ \bigstar\ \epsilon$. For this to be possible for $\bigstar\ ::\ a \to a \to a$ and $\epsilon\ ::\ a$.

$$\begin{array}{rl} \bigstar \text{ must be associative} & x \bigstar (y \bigstar z) \equiv (x \bigstar y) \bigstar z \\ \epsilon \text{ must have no effect} & \epsilon \bigstar n = n \end{array}$$

These properties for a **monoid** $(a, \bigstar, \epsilon)$.
Other example include:

$$\begin{array}{cccc} (lists, ++, []) & (\mathbb{N}, +, 0) & (\mathbb{N}, \times, 1) & (bool, \wedge, true) \\ (bool, \vee, false) & (\mathbb{R}, max, \infty) & (\mathbb{R}, min, -\infty) & (Universal\ set, \cup, \emptyset) \end{array}$$

We can also find monoids of functions:

$$(a \to a, (.), id)$$

as $(id\ .\ g)x \equiv id(g\ x)$ and $((f\ .\ g)\ .h)x = f(g(h\ x))$

Concat

We can easily define concat recursively as:

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

We can also notice that $([[a]], (++), [])$ is a monoid, so we can use **foldr** or **foldl**

```
concatr :: [[a]] -> [a]
concatr = foldr (++) []
```

```
concatr :: [[a]] -> [a]
concatr = foldl (++) []
```

as (++) makes a copy of the first argument (to ensure persistent data), if we apply is in a left associative bracketing scheme we will have to make larger & larger copies.

$$(\dots((((([\,]{+\!+}_0\ xs_1){+\!+}_m\ xs_2){+\!+}_{2m}\ xs_3){+\!+}_{3m}\ xs_4\dots){+\!+}_{mn}\ xs_n$$

Hence where $n = \mathit{length}\ \mathit{xss}$ and $m = \mathit{length}\ xs_1 = \mathit{length}\ xs_2 = \dots = \mathit{length}\ xs_n$.

$$T_{concatl}(m,n) \in O(n^2m)$$
$$T_{concatr}(m,n) \in O(nm)$$

DLists

Instead of storing a list, we store a composition of functions that build up a list.

$$\begin{gathered} xs_1{+\!+}xs_2{+\!+}xs_3{+\!+}\dots{+\!+}xs_n \\ \Downarrow \\ f\ xs_1 \bullet f\ xs_2 \bullet f\ xs_3 \bullet \dots \bullet f\ xs_n \\ \Downarrow \\ (xs_1\ {+\!+}) \bullet (xs_2\ {+\!+}) \bullet (xs_3\ {+\!+}) \bullet \dots \bullet (xs_n\ {+\!+}) \end{gathered}$$

We can then apply this function on the empty list $[\,]$ to get the resulting list.

```haskell
newtype DList a = DList ([a] -> [a])

instance List DList where
    toList :: DList a -> [a]
    toList (DList fxs) = fxs []

    fromList :: [a] -> DList a
    fromList xs = DList (xs++)

    (++) :: DList a -> DList a -> DList a
    DList fxs ++ DList fys = DList(fxs . fys)
```

We can form a **monoid** of (DList,++,$\mathit{DList}\ \mathit{id}$).

50001 - Algorithm Analysis and Design - Lecture 5

Oliver Killane

12/11/21

Lecture Recording

Lecture recording is available here

DLists Continued...

Monoids (again)

A **monoid** is a triple $(M, \diamond, \epsilon)$ where $\diamond$ is associative and of type $M \to M \to M$, and $x \diamond \epsilon \equiv x$.

```haskell
class Monoid m where
    (<>) :: m -> m -> m
    mempty :: m
```

A haskell typeclass can then be instantiated for many other data types. For example the **monoid** $(\mathbb{Z}, +, 0)$ (note that we cannot enforce **monoid** properties through haskell, unlike languages such as **agda**).

```haskell
-- declaring newtype so that many monoid instance on Int do not conflict
newtype PlusInt = PlusInt Int

instance Monoid PlusInt where
    (<>) :: PlusInt -> PlusInt -> PlusInt
    (<>) = (+)

    mempty :: PlusInt
    mempty = 0
```

Likewise we can abstract Lists to a class (which we can instantiate for DLists).

List Class

```haskell
class List list where
    empty :: list a
    single :: a -> list a

    (:) :: a -> lis a -> list a
    snoc :: list a -> list a -> list a

    head :: list a -> a
    tail :: list a -> list a

    last :: list a -> a
    init :: list a -> list a

    (++) :: list a -> list a -> list a

    length :: list a -> Int

    fromList :: [a] -> list a
    toList :: list a -> [a]
```

$[a]$ is out abstract list type, and $list a$ is our concrete type.

It is critical to ensure that $toList \bullet fromList \equiv id$

But in general $fromList \bullet toList \not\equiv id$ (this is as the internal representation may change and much information about the internal representaion cannot be preserved by toList, for example an unbalanced tree changed to a list maybe be balanced when converted back to a tree).

We also included $normalise :: fromList \bullet toList$ as a useful tool to reset the internal structure (for example to rebalanced the tree representation of a list)

Haskell Implementation

To prevent conflicts due to Prelude functions already being defined we can use:

```haskell
import Prelude hiding(head, tail, (++), etc...)
import qualified Prelude
```

To help ensure correctness we can use **Quickcheck** to check properties

```
cabal install --lib QuickCheck
```

Then can use quickcheck to define properties we want to test:

```haskell
import Test.QuickCheck

-- code to test written here ...

prop_propertyname :: InputTypes -> Bool
prop_propertyname = test code

-- example for normalise (takes a list type, that has equality defined for it)
prop_normalise (Eq a, Eq (list a), List list) => list a -> Bool
prop_normalise xs = (toList . fromList) xs == xs

-- Can return properties (requires show) using the triple-equals
prop_assoc :: (Eq (list a), Show (list a), List list)
    => list a -> list a -> list a -> Property
prop_assoc xs ys zs = (xs ++ ys) ++ zs === xs ++ (ys ++ zs)
```

```
ghci file_to_check.hs
*file_to_check> quickCheck (prop_normalise :: [Int] -> Bool)
+++ OK, passed 100 tests.
*file_to_check> quickCheck (prop_normalise :: [Bool] -> Bool)
+++ OK, passed 100 tests.
*file_to_check> verboseCheck (prop_normalise :: [Bool] -> Bool)
Passed:
...

Passed:
...

etc...

+++ OK, passed 100 tests.
```

50001 - Algorithm Analysis and Design - Lecture 6

Oliver Killane

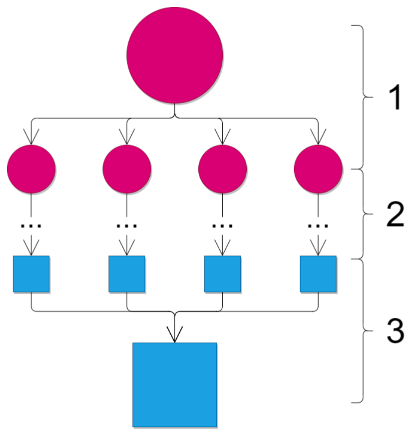
12/11/21

Lecture Recording

Lecture recording is available here

Divide & Conquer

1. Divide a problem into subproblems
2. Divide and conquer subproblems into subsolutions
3. Conquer subsolutions into a solution



Merge Sort

```
msort :: Ord a => [a] -> [a]
msort [] = []
msort [x] = [x]
msort xs = merge (msort us) (msort vs)
  where (us,vs) = splitAt (length xs `div` 2) xs

merge :: Ord a => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge xss@(x:xs) yss@(y:ys)
  | x <= y    = x : merge xs yss
  | otherwise = y : merge xss ys
```

SplitAt divides, and **merge** Conquers. We can calculate the time complexity for the recurrence relations below (based on recursive structure of **msort**):

$$T_{msort}(0) = 1$$
$$T_{msort}(1) = 1$$
$$T_{msort}(n) = T_{length}(n) + T_{splitAt}(\frac{n}{2}) + T_{merge}(\frac{n}{2}) + 2 \times T_{msort}(\frac{n}{2})$$

We can simplify the complexity of **msort**

$$
\begin{aligned}
T_{msort}(n) &= T_{length}(n) + T_{splitAt}(\frac{n}{2}) + T_{merge}(\frac{n}{2}) + 2 \times T_{msort}(\frac{n}{2}) \\
&= n + \frac{n}{2} + \frac{n}{2} + \frac{n}{2} + 2 \times T_{msort}(\frac{n}{2}) \\
&= \frac{5}{2} \times n + 2 \times T_{msort}(\frac{n}{2})
\end{aligned}
$$

Master Theorem

For an algorithm *algo* such that:

$$T_{algo}(n) = a \times T_{algo}(\frac{n}{b}) + f(n) + \text{base cases}$$

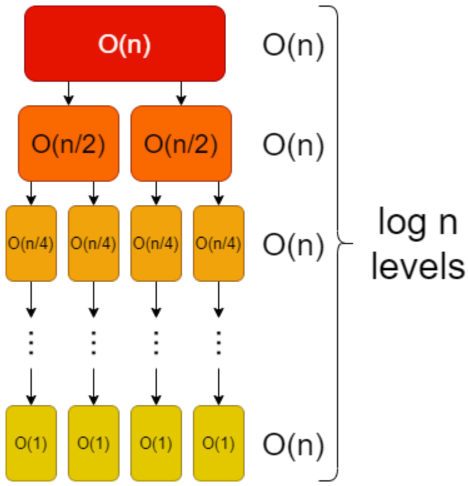
The work at recursion level $\log_b n$ is $\Theta(a^{\log_b n})$ To calculate the order of the time complexity:

1. Get the recurrence relation in the form above.
2. Get the critical exponent E by formula: $E = \log_b a = \dfrac{\log a}{\log b}$.
3. Given $f(n) = n^c$ we can express the work as a geometric sum $\sum_{i=0}^{\log n} ar^i$ where $r = \dfrac{a}{b^c}$.

$$r > 1 \Leftrightarrow a > b^c \Leftrightarrow \log_b a > c \Leftrightarrow E > c$$

$$
\begin{array}{lr}
E < c & T_{algo}(n) \in \Theta(f(n)) \\
E = c & T_{algo}(n) \in \Theta(f(n)\log_b n) = \Theta(f(n)\log n) \\
E > c & T_{algo}(n) \in \Theta(a^{\log_b n}) = \Theta(n^{\log_b a}) = \Theta(n^E)
\end{array}
$$

By master theorem we can easily see $T_{msort}(n) \in \Theta(n \log n)$. We can also calculate it using a graph:



Quicksort

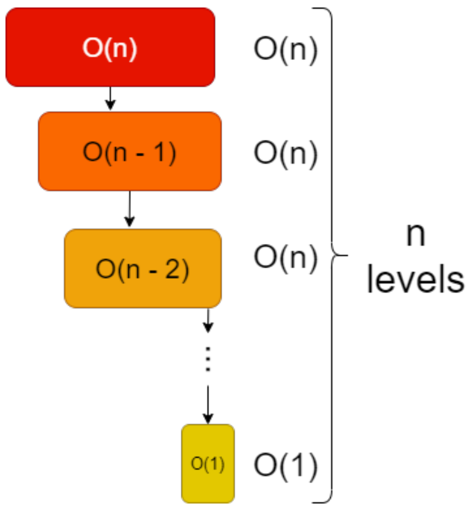
```haskell
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort [x] = [x]
qsort (x:xs) = qsort us ++ x:qsort vs
  where (us, vs) = partition (<x) xs

partition :: (a -> Bool) -> [a] -> ([a],[a])
partition p xs = (filter p xs, filter (not . p) xs)

dxgfdg
```

Note for simplicity, we assume the lists are split into equal parts.

$$\begin{aligned} T_{qsort}(0) &= 1 \\ T_{qsort}(1) &= 1 \\ T_{qsort}(n) &= T_{partition}(n-1) + T_{++}(\frac{n}{2}) + 2 \times T_{qsort}(\frac{n}{2}) \end{aligned}$$

In the worst case, the partition splits xs into $(xs, [])$, we have complexity:

$$\begin{aligned} T_{qsort}(n) &= T_{partition}(n-1) + T_{++}(n-1) + T_{qsort}(0) + T_{qsort}(n-1) \\ &= 2(n-1) + n + 1 + T_{qsort}(n-1) \end{aligned}$$



We can once again use master theorem, or a diagram such as below to see the complexity: Hence in the worst case $T_{qsort}(n) \in O(n^2)$ (same as insertion sort).

50001 - Algorithm Analysis and Design - Lecture 7

Oliver Killane

13/11/21

Lecture Recording

Lecture recording is available here

Dynamic Programming

A technique to efficiently calculate solutions to certain recursive problems.

1. Describe an inefficient recursive algorithm.
2. Reduce inefficiency by storing intermediate shared results.

Fibonacci Sequence

Fully Recursive

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

$T_{fib}(0) \quad = 1$
$T_{fib}(1) \quad = 1$
$T_{fib}(n) \quad = 1 + T_{fib}(n-2) + T_{fib}(n-1)$
The complexity of this algorithm is $T_{fib}(n) \in O(2^n)$.

Saving Intermediate Results

We can use a helper function which takes the remaining number of additions, and the two previous values.

```haskell
fib :: Int -> Integer
fib n = fibHelper n 0 1
  where
    fibHelper :: Int -> Integer -> Integer -> Integer
    fibHelper 0 x y = x
    fibHelper n x y = fibHelper (n-1) y (x + y)
```

$T_{fib}(0) \quad = 1$
$T_{fib}(1) \quad = 1$
$T_{fib}(n) \quad = 1 + T_{fib}(n-1)$
The complexity of this algorithm is $T_{fib}(n) \in O(n)$.

This way every value is calculated only once for each call. However values are not saved between calls.

Memoisation

```haskell
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

fib :: Int -> Integer
fib n = fibs !! n
```

This creates a large list, we must only index on the list to get the value. By using an array we can reduce the time taken to get to the nth element.

Array Based Memoisation

```haskell
-- Array Data Type
import Data.Array ( Ix(range), Array, array )

-- Tabulate function
tabulate :: Ix i => (i,i) -> (i -> e) -> Array i e
tabulate (a,b) f = array (a,b) [(i,f i) | i <- range (a,b)]

-- Ix (class of all indexes)

-- T(!) is in O(1)
(!) :: Ix i => Array i e -> i -> e

-- Range creates a lits of indexes in a range
range :: Ix i => (i,i) -> [i]

-- Array function creates an array from a range of indexes & values
array :: Ix i => (i,i) -> [(i,e)] -> Array i e
```

Hence we can make our algorithm:

```haskell
import Data.Array ( Array, (!) )

fib :: Int -> Integer
fib n = table ! n
  where
    table :: Array Int Integer
    table = tabulate (0,n) memo

    memo :: Int -> Integer
    memo 0 = 0
    memo 1 = 1
    memo n = table ! (n-2) + table ! (n-1)
```

Here we can do constant time lookups for values in the table. If a value is not present, it is lazily evaluated using other elements in the table.

In this way we only calculate each fibonacci number once, and only when we need it. Further it is saved for any subsequent calls to fib.

Edit-Distance

The **Edit-Distance** Problem is concered with calculating the **Levenshtein** distance between two strings.

Levenshtein Distance

The number of insertions, deletions & updates required to convert one string into another.

$$toil \to_{+1} oil \to_{+1} il \to_{+1} ill$$

```haskell
dist :: String -> String -> Int
dist xs [] = length xs
dist [] ys = length ys
dist xxs@(x:xs) yys@(y:ys)
  = minimum
    [dist xxs ys + 1,
     dist xs yys + 1,
     dist xs ys + if x == y then 0 else 1]
```

This problem becomes of order $O(3^n)$ as it recurs 3 ways for each call.

We can reuse results for two substrings through memoisation, first we make a new recursive version that uses the index we are checking in each string:

```haskell
dist :: String -> String -> Int
dist xs ys = dist' xs ys (length xs) (length ys)

dist' :: String -> String -> Int -> Int -> Int
dist' xs ys i 0 = i
dist' xs ys 0 j = j
dist' xs ys i j
  = minimum [dist' xs ys i (j-1) + 1,
             dist' xs ys (i-1) j + 1,
             dist' xs ys (i-1) (j-1) + if x == y then 0 else 1]
  where
    m = length xs
    n = length ys
    x = xs !! (m-i)
    y = ys !! (n-j)
```

We can then use **tabulate** to create a memoised version.

```haskell
import Data.Array ( Array, (!) )

dist :: String -> String -> Int
dist xs ys = table ! (m,n)
  where
    table = tabulate ((0,0),(m,n)) (uncurry memo)

    memo :: Int -> Int -> Int
    memo i 0 = i
    memo 0 j = j
    memo i j
      = minimum [table ! (i, j - 1) + 1,
                 table ! (i-1,j) + 1,
                 table ! (i - 1,j - 1) + if x == y then 0 else 1]
      where
        x = ays ! (m - i)
        y = ays ! (n - j)

    m = length xs
    n = length ys
```

```
    axs,ays :: Array Int Char
    axs = fromList xs
    ays = fromList ys
```

As there are at most $m \times n$ entires in the table, and each are calculated at most once, and the lookup time is constant (using arrays), the complexity is $O(mn)$.

50001 - Algorithm Analysis and Design - Lecture 8

Oliver Killane

17/11/21

Lecture Recording

Lecture recording is available here

Amortized Analysis

So far we have studied complexity of a single, isolated run of an algorithm. **Amortizsed Analysis** is about understanding cost in a wider context (e.g averaged over many calls to a routine).

Dequeues

Dequeue

A Dequeue is a double ended queue. An abstract datatype that generalises a queue. Elements can be added or removed from either end.

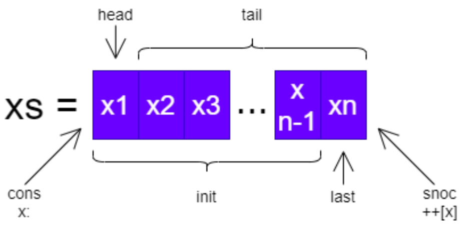
Common associated funtions are:

snoc	Insert element at the back of the queue.
cons	Insert element at the front of the queue.
eject	Remove last element.
pop	remove fist element.
peek	Examine but do not remove first element.

Dequeues are also called head-tail linked lists or symmetric lists.

we use a dequeue when we want to reduce the time taken to perform certain operations.

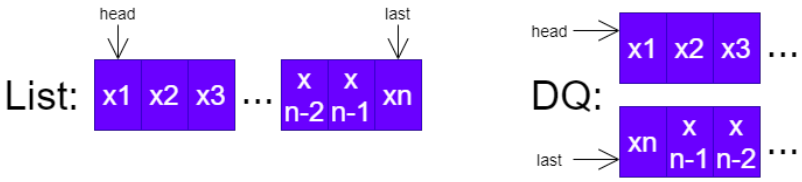
List Operation Complexity



cons	O(1)	**snoc**	O(1)
head	O(1)	**last**	O(1)
tail	O(1)	**init**	O(1)

Dequeue Structure

To achieve $O(1)$ complexity in the **snoc**, **init** and **last** we use two lists.



One list starts contains the start of the list, and the other the end (reversed).

We keep two invariants for *Deqeue us sv*:

$$null\ us \Rightarrow null\ sv \vee single\ sv$$

$$null\ sv \Rightarrow null\ us \vee single\ us$$

In other words, if one list is empty, the other can contain at most 1 element.

An example implementation in haskell is below:

```
-- can ignore certain patterns due to invariant
{-# OPTIONS_GHC -Wno-incomplete-patterns #-}

data Deqeue a = Deqeue [a] [a]

instance List Deqeue where
  toList :: Deqeue a -> [a]
  toList (Deqeue us sv) = us ++ reverse sv

  fromList :: [a] -> Deqeue a
  fromList xs = Deqeue us (reverse vs)
    where (us,vs) = splitAt (length xs `div` 2) xs

  -- use the invariant, if [] sv then sv = [x] or []

  -- O(1)
  cons :: a -> Deqeue a -> Deqeue a
  cons x (Deqeue us []) = Deqeue [x] us
  cons x (Deqeue us sv) = Deqeue (x:us) sv

  -- O(1)
  snoc :: Deqeue a -> a -> Deqeue a
  snoc (Deqeue [] sv) x = Deqeue sv [x]
  snoc (Deqeue us sv) x = Deqeue us (x:sv)

  -- O(1)
  last :: Deqeue a -> a
  last (Deqeue _ (s:_)) = s
  last (Deqeue [u] _) = u
  last (Deqeue [] []) = error "Nothing in the deqeue"

  -- O(1)
  head :: Deqeue a -> a
  head (Deqeue (u:_) _) = u
  head (Deqeue [] [v]) = v
  head (Deqeue [] []) = error "Nothing in the deqeue"

  -- O(1)
  tail :: Deqeue a -> Deqeue a
```

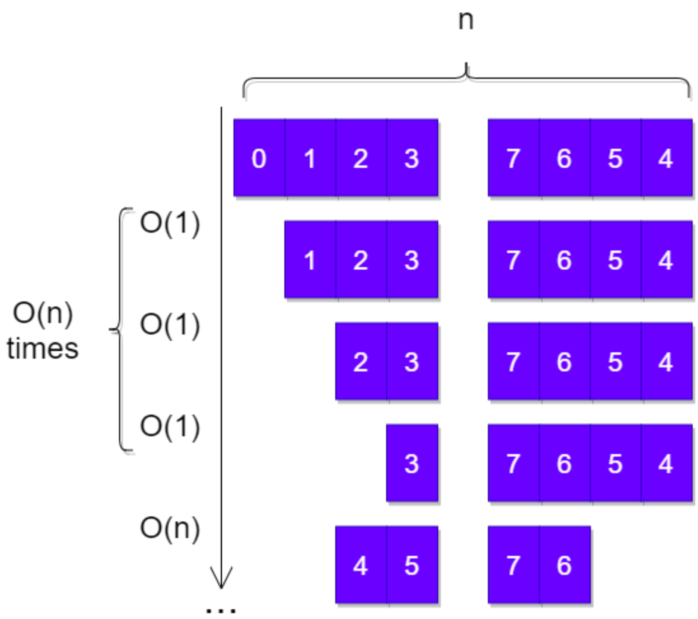
```haskell
 tail (Dequeue [] []) = error "Nothing in the dequeue"
 tail (Dequeue [] _) = empty
 tail (Dequeue _ []) = empty
 tail (Dequeue [u] sv) = Dequeue (reverse su)
   where
     n = length sv
     (su, sv') = splitAt (n `div` 2) sv
     -- note: could also do a fromList (reverse sv) but less efficient

 tail (Dequeue (u:us) sv) = Dequeue us sv

 -- O(1)
 init :: Dequeue a -> Dequeue a
 init (Dequeue [] []) = error "Nothing in the dequeue"
 init (Dequeue [] _) = empty
 init (Dequeue _ []) = empty
 init (Dequeue us [s]) = fromList us
 init (Dequeue us (s:sv)) = Dequeue us sv

 isEmpty :: Dequeue a -> Bool
 isEmpty (Dequeue [] []) = True
 isEmpty _ = False

 empty :: Dequeue a
 empty = Dequeue [] []
```

When considering the cost of **tail** and **init** we must consider that there are two possibilities:

High Cost	$init(Dequeue\ us\ [s])$ $tail(Dequeue\ [u]\ sv)$	This operation is $O(n)$ complexity due to the **spitAt** and **reverse** operation done on half of a list.
Low Cost	$init(Dequeue\ us\ (s:sv))$ $tail(Dequeue\ (u:us)\ sv)$	Low cost $O(1)$ operation as it requires only a pattern match on the first element.

As both of these operations rebalance the **Dequeue** to to be balanced (half the queue on each list), we these operations can have an amortized cost of $O(1)$.

We know this as the average cost is of order $O(1)$. The $O(n)$ cost is incurred every $n/2$ calls to **tail**/**init**.


n
O(n)
times
O(1)
O(1)
O(1)
O(n)
0 1 2 3
7 6 5 4
1 2 3
2 3
3
4 5
7 6
...

50001 - Algorithm Analysis and Design - Lecture 9

Oliver Killane

17/11/21

Lecture Recording

Lecture recording is available here

Amortization

The complexity of **tail** is an example of **Amortized analysis**, where operation's wider context are considered when calculating the complexity.

$$xs_0 \overset{op_0}{\rightsquigarrow} xs_1 \overset{op_1}{\rightsquigarrow} xs_2 \overset{op_2}{\rightsquigarrow} xs_3 \overset{op_3}{\rightsquigarrow} \dots \overset{op_{n-1}}{\rightsquigarrow} xs_n$$

We defined 3 parts:

1. **Cost Function**
   $C_{op_i}(xs_i)$ determines the cost of operation op_i on data xs_i. Estimating how many steps it takes for each operation to execute.

2. **Amortized Cost Function**
   $A_{op_i}(xs_i)$ for each operation op_i on data xs_i.

3. **Size Function**
   $S(xs)$ that calculates the size of data xs

We define these functions with the goal to show that:

$$C_{op_i}(xs_i) \leq A_{op_i}(xs_i) + S(xs_i) - S(xs_{i+1})$$

The cost of the operation is smaller than the amortized cost, plus the difference in size of the data structure before and after the operation.

Once this is shown, we can infer that:

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

Furthermore when $S(xs_0) = 0$ this implies:

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

This means the cost of the operations is less than the sum of the amortized costs.

For example, if $A_{op_i}(xs) = 1$ then the total cost will be bounded by $O(n)$.

Tail example

$$C_{cons}(xs) = 1 \quad C_{snoc}(xs) = 1 \quad C_{head}(xs) = 1 \quad C_{last}(xs) = 1$$

For tail we can do the following:

(1)	$C_{tail}(Dequeue\ us\ sv) = length\ sv$	(Create a cost function of **tail**.)
(2)	$A_{op}(xs) = 2$	(Create an arbitrary cost function.)
(3)	$S(Dequeue\ us\ sv) = \|length\ us - length\ sv\|$	(Create a size function for **dequeue**.)
(4)	$Dequeue\ us\ sv$ where $length\ sv = k$	(Worst case where us is a singleton)
(5)	$S(Dequeue\ us\ sv) = k - 1$ $S(Dequeue\ us'\ sv') = 1$	(Size of the next data structure can be at most 1.)
(6)	$C_{tail}(Dequeue\ us\ sv) = k$	(Calculate the worst case cost of **tail**.)
(7)	$k \leq 2 + (k - 1) - 1 = k + 2$	(As this inequality holds, the time complexity of all n instructi

As the time complexity of all n instructions together is $O(n)$, the amortized cost of a single instruction is $O(1)$.

About the size function

We want to balance the size function such that:

- The size function is 0 to start with.
- The size between operations is large enough to prove the inequality.

The size function is arbitrary, if you cannot choose a size function that satisfied the goal inequality, then you're probably making a mistake

Peano Numbers

```
data Peano = Zero | Succ Peano

-- analogous to (:) Cons
incr :: Peano -> Peano
incr = Succ

-- analogous to tail
decr :: Peano -> Peano
decr (Succ n) = n
decr Zero = error "Cannot decrement zero"

-- analogous to (++) concatenate
add :: Peano -> Peano -> Peano
add a Zero = a
add a (Succ b) = Succ (add a b)

-- tail recursive version for extra goodness!
add a b = add (incr a) (decr b)
```

This shows how similar operations of similarly structured data can be.

Binary Numbers

```
data Bit = I | O

-- [LSB,...,MSB]
type Binary = [Bit]

```

```haskell
incr :: Binary -> Binary
incr [] = [I]
incr (O:bs) = I:bs
incr (I:bs) = O:incr bs
```

we can do amortized analysis on incr:

(1)	$C_{incr}(bs) = t + 1$ where $t = length\ (takeWhile\ (== I)\ bs)$	(Create a cost function.)
(2)	$A_{incr}(bs) = 2$	(Create Amortized Cost)
(3)	$S(bs) = length.filter\ (== I)\ \$\ bs$	(Create size function.)
(4)	Given $bs' = incr\ bs$ we show $C_{incr}(bs) \leq A_{incr}bs + S(bs) - S(bs')$	(Setup inequality)
(5)	$t + 1 \leq 2 + S(bs) - (S(bs) - t + 1)$	(Subsitiute in inequality)
(6)	$t + 1 \leq 1 + t$	(Hence inequality holds)
(7)	$S(start) = 0$	(Start size is zero.)

Hence The sum of C is smaller than the sum of A, as this is over n operations and $\sum A = 2n$, $C_{incr}(bs) \in O(1)$.

50001 - Algorithm Analysis and Design - Lecture 10

Oliver Killane

18/11/21

Lecture Recording

Lecture recording is available here

List lookup

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

As you ca see **!!** costs $O(n)$ as it may traverse the entire list.

If we want this access to be faster, we can use trees:

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

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

instance List Tree where
  toList :: Tree a -> [a]
  toList Tip = []
  toList (Leaf n) = [n]
  toList (Fork n l r) = toList l ++ toList r

  -- Invariant: size Node n a b = n = size a + size b
  length :: Tree a -> Int
  length Tip = 0
  length (Leaf _) = 1;
  length (Node n _ _) = n;

  (!!) :: Tree a -> Int -> a
  (Leaf x) !! 0 = x
  (Node n l r) !! k
    | k < m = l!!k
    | otherwise = r!! (k-m)
    where m = length l

  -- case for Tip !! n or Leaf !! >0
  _ !! _ = error "(!!): Cannot get list index"
```

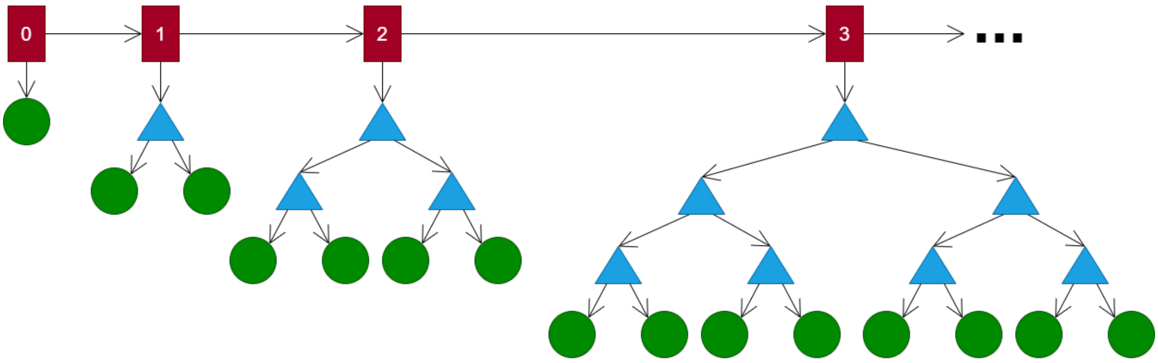
This costs $O(\log n)$ as each recursive call acts on half of the remaining list.

However we have difficulty with insertion:

Insert Quickly	$n : t = node(Leaf n)t$	Effectively becomes a linked list, $\log n$ search time ruined.
Insert Slowly	Rebalance tree (e.g **AVL** tree)	Complex and no longer $O(1)$ insert.

Random Access Lists

A list containing elements that are either nothing, or a perfect tree with size the same as 2^{index}.



The empty tree can be represented by a *Tip* value (from the notes), or using type $Maybe(Tree)$ (from the lecture) where $Tree\ a = Leaf\ x \mid Node\ n\ l\ r$.

When we add to a tree, we add to the first element of the **RAList**, if the invariant is breached (no longer perfect tree of size 2^0) it can be combined with the next list over (if empty, place, else combine and repeat).

This way while the worst case insert is $O(n)$, our amortized complexity is $O(1)$ much as with increment.

```haskell
data Tree a = Tip | Leaf a | Node Int (Tree a) (Tree a)

type RAList a = [Tree a]

instance List RAList where
  toList :: RAList a -> [a]
  toList (RaList ls) = concatMap toList ls

  fromList :: [a] -> RAList a
  fromList = foldr (:) empty

  empty :: RAList a
  empty = []

  (:) :: a -> RAList a -> RAList a
  n : [] = [Leaf n]
  n : (RAList ls) = RAList (insertTree (Leaf n) ls)
    where
      insertTree :: Tree a -> [Tree a] -> [Tree a]
      insertTree t ([])      = [t]
      insertTree t (Tip:ls) = t:ls
      insertTree t (t':ts)  = Tip: insertTree (node t t') ts

  length :: RAList a -> Int
  length (RAList ls) = foldr ((+) . length) 0 ls

  (!!) :: RAList a -> Int -> a
  (RAList []) !! _ = error "(!!): empty list"
  (RAList (x:xs)) !! n
    | isEmpty x = (RAList ts) !! k
    | n < m     = x !! n
    | otherwise = (RAList xs) !! (n-m)
    where m = length x
```

50001 - Algorithm Analysis and Design - Lecture 11

Oliver Killane

18/11/21

Lecture Recording

Lecture recording is available here

Equality

```haskell
class Eq a where
  (==) :: a -> a -> Bool
```

Eq is the typeclass for equality, any instance of this class should ensure the equality satisfied the laws:

$$\begin{aligned} \text{reflexivity} \quad & x == x \\ \text{transitivity} \quad & x == y \land y == z \Rightarrow x == z \\ \text{symmetry} \quad & x == y \Rightarrow y == x \end{aligned}$$

We also expect the idescernability of identicals (Leibniz Law):

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

For a set-like interface we have a member function:

```haskell
(in) :: Eq a => a -> Set a -> Bool
```

If we assume only that Eq holds, the complexity is $O(n)$ as we must potentially check all members of the set. To get around this, we use ordering.

Orderings

The Ord typeclass allows us to check for inequalities:

```haskell
class Eq a => Ord a where
    (<=) :: a -> a -> Bool
    (<)  :: a -> a -> Bool
    (>=) :: a -> a -> Bool
    (>)  :: a -> a -> Bool
```

We must try to ensure certain properties hold, for example for to have a partial order we require:

$$\begin{aligned} \text{reflexivity} \quad & x \leq x \\ \text{transitivity} \quad & x \leq y \land y \leq z \Rightarrow x \leq z \\ \text{antisymmetry} \quad & x \leq y \land y \leq x \Rightarrow x == y \end{aligned}$$

There are also total orders (all elements in the set are ordered compared to all others), for which we add the constraint:

$$\text{connexity} \quad x \leq y \lor y \leq x$$

Ordered Sets

```
class OrdSet ordset where
  empty    :: ordset n
  insert   :: Ord a => a -> ordset a -> ordset a
  member   :: Ord a => a -> ordset a -> Bool
  fromList :: Ord a => [a] -> ordset a
  toList   :: Ord a => ordset a -> [a]
```

We can implement this class for Trees:

```
data Tree a = Tip | Node (Tree a) a (Tree a)

instance OrdSet Tree where
  empty :: Tree n
  empty = Tip

  insert :: Ord a => a -> Tree a -> Tree a
  insert x Tip = Node Tip x Tip
  insert x (Node l y r)
    | x == y    = t
    | x < y     = Node (insert x l) y r
    | otherwise = Node l y (insert x r)

  member :: Ord a => a -> Tree a -> Bool
  member x Tip = False
  member x (Node l y r)
    | x == y    = True
    | x < y     = member x l
    | otherwise = member x r

  fromList :: Ord a => [a] -> Tree a
  fromList = foldr insert empty

  toList :: Ord a => Tree a -> [a]
  toList Tip          = []
  toList (Node l y r) = toList l ++ y:toList r
```

However the worst case here is still $O(n)$ as we do not balance the tree as more members are inserted. If the members are added in order, the tree devolves to a linked list.

We need a way to create a tree that self balances.

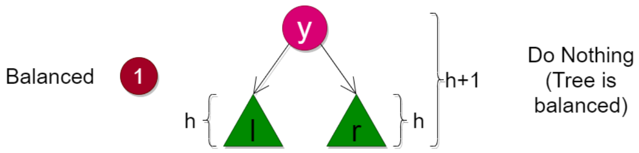
Binary Search Trees (AVL Trees)

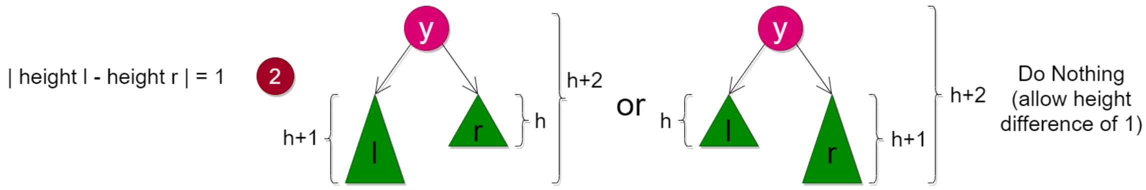
```
-- H for height! The int stored at a node is the height of that tree.
data HTree a = HTip | HNode Int (HTree a) a (HTree a)

height :: HTree a -> Int
height HTip = 0
height (HNode h _ _ _) = h

hnode :: HTree a -> a -> HTree a -> HTree a
hnode l x r = HNode h l x r
  where h = max (height l) (height r) + 1
```

When inserting into the tree we must keep the tree balanced such that no subtree's left is more than one higher than its' right.

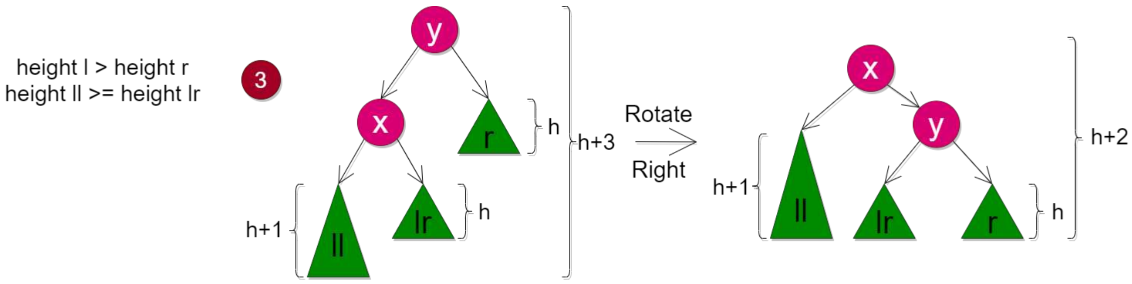
```haskell
insert :: Ord a => a -> HTree a -> HTree a
insert x HTip = hnode Tip x Tip
insert x t@(HNode _ l y r)
  | x == y    = t
  | x < y     = balance (insert x l) y r
  | otherwise = balance l y (insert x r)
```

We must rebalance the tree after insertion, this must consider the following cases:



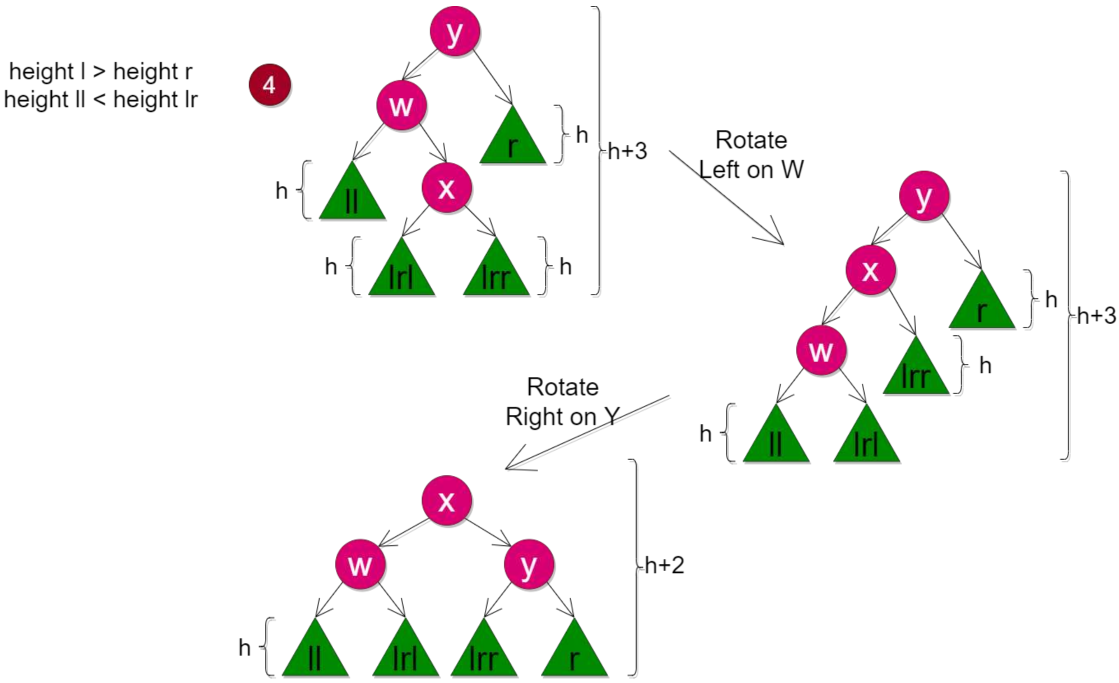


When the tree's balanced invariant has been broken, we must follow these cases:




```haskell
rotr :: HTree a -> HTree a
rotr (HNode _ (HNode _ ll x lr) y r) = hnode ll x (hnode lr y r)
```

When the right subtree's right subtree is higher:




```
rotl :: HTree a -> HTree a
rotl (HNode _ ll w (HNode _ lrl x lrr))
  = hnode (hnode ll w lrl) x lrr

-- so for the example: HNode _ wt y r -> rotr ( hnode (rotl wt) y r)
```

We can use rotate left and rotate right to for the two cases where the right subtree is 2 or more higher than the left.

```
balance :: Ord a => HTree a -> a -> HTree a -> HTree
balance l x r
  | lh == lr || abs (lr - lr) == 1 = t
  | lh > lr    = rotr(hnode (if height ll < height lr then rotl l else l) x r)
  | otherwise = rotl(hnode l x (if height rl > height rr then rotr r else r))
  where
    lh = height l
    rl = height r
    (HNode _ ll _ lr) = l
    (HNode _ rl _ rr) = r
```

50001 - Algorithm Analysis and Design - Lecture 12

Oliver Killane

18/11/21

Lecture Recording

Lecture recording is available here

Red-Black Trees

AVL trees worked by storing an extra integer (height) to use in rebalancing, **red-black trees** use an extra bit to determine if a node is red or black.

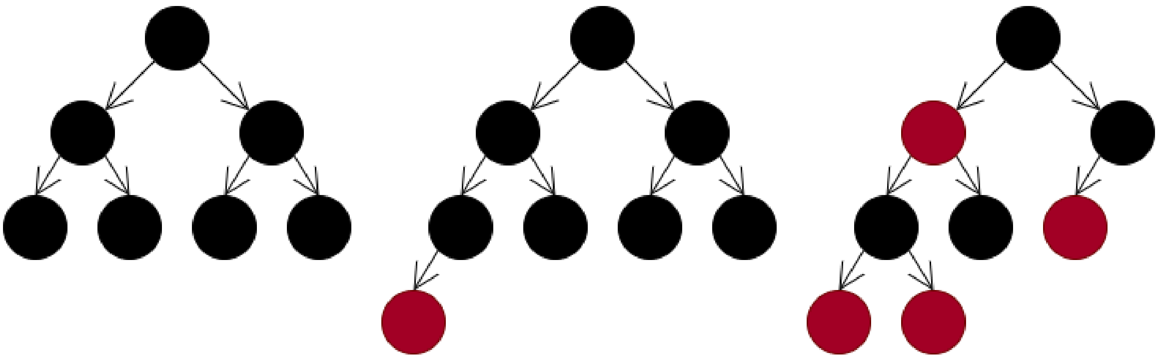
In practice they are less balanced than **AVL trees** however the insertion is faster and the data structure is a little bit smaller.

```haskell
data Colour = Red | Black

data RBTree a = Empty | Node Colour (RBTree a) a (RBTree a)
```

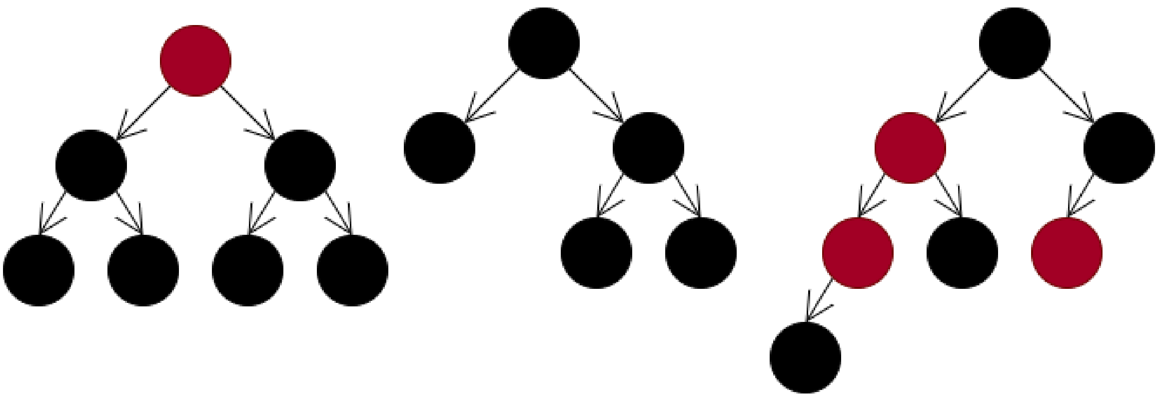
The structure relies on two invariances:

1. Every Red node must have a Black parent node.
2. Every path from the root to leaf must have the same number of black nodes.

Valid Red Black Trees

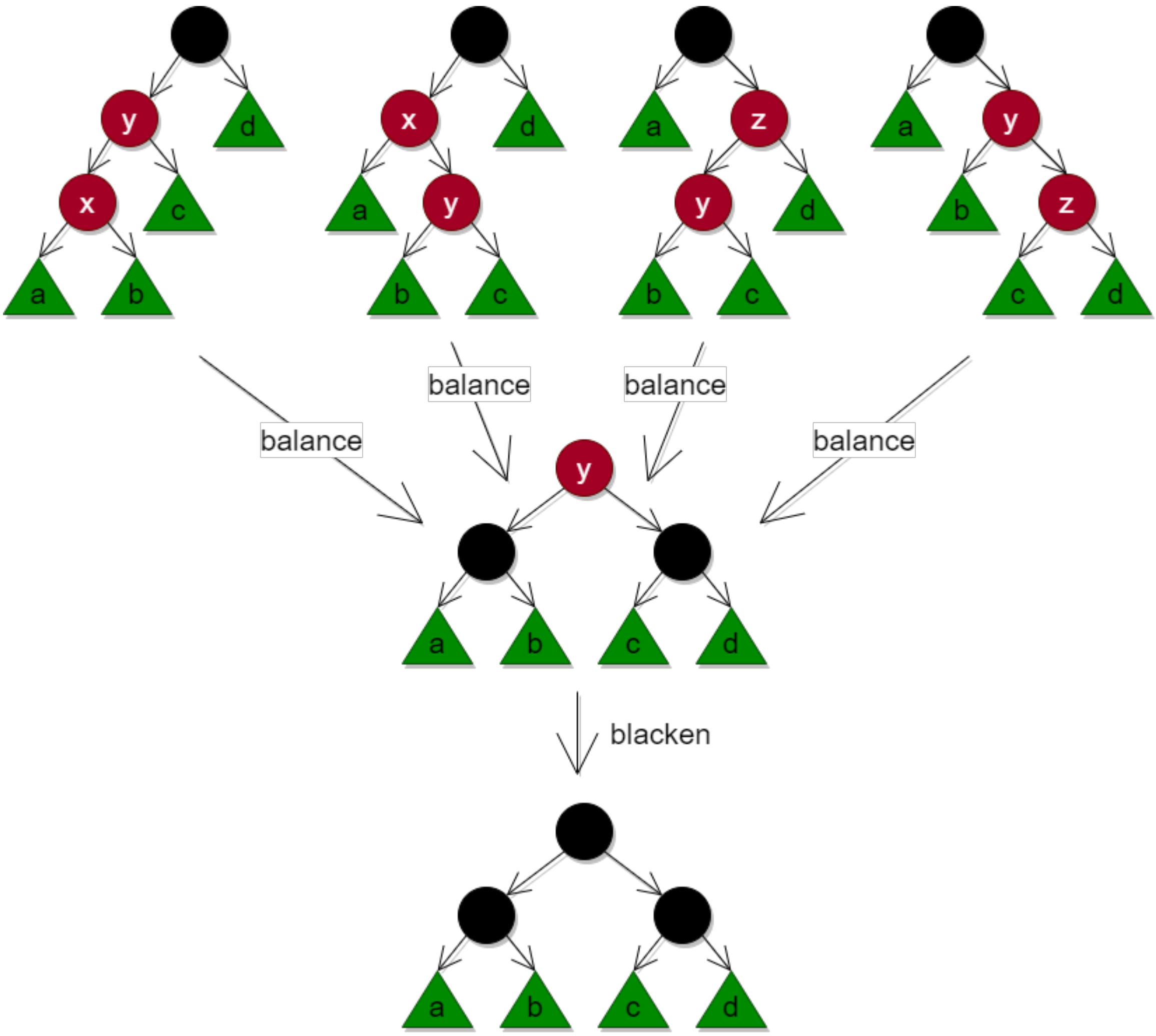


Invalid Red Black Trees



We have an insert function that needs to rebalance the tree:

```
blacken :: Ord a => RBTree a -> RBTree a
blacken (Node Red l x r) = Node Black l x r
blacken t                = t

balance :: Ord a => Colour -> RBTree a -> a -> RBTree a -> RBTree a
balance c l v r = case Node c l v r of
  Node Black (Node Red (Node Red a x b) y c) z d -> bal x y z a b c d
  Node Black (Node Red a x (Node Red b y c)) z d -> bal x y z a b c d
  Node Black a x (Node Red (Node Red b y c) z d) -> bal x y z a b c d
  Node Black a x (Node Red b y (Node Red c z d)) -> bal x y z a b c d
  t                                              -> t
  where
    bal x y z a b c d = Node Red (Node Black a x b) y (Node Black c z d)

insert :: Ord a => a -> RBTree a -> RBTree a
insert = (blacken .) . ins
  where
    ins :: Ord a => a -> RBTree a -> RBTree a
    ins x Empty = Node Red Empty x Empty
    ins x t@(Node c l y r)
      | x < y     = balance c (ins x l) y r
      | x == y    = t
      | otherwise = balance c l y (ins x r)
```



Counting

We can exploit the analogy we used with counting and trees for **RALists** here, with a difference.

Imagine a counting system that lacks zeros. We can count to 10 as:

Normal:	1	2	...	9	10	...	11	12	...	19	20	...	101	102	...	110	111
Special:	1	2	...	9	X	...	11	12	...	19	1X	...	X1	X2	...	XX	111

We can use this with the pattern of inserting elements into a red black tree (in order) to map red black trees to an incrementing number.

50001 - Algorithm Analysis and Design - Lecture 13

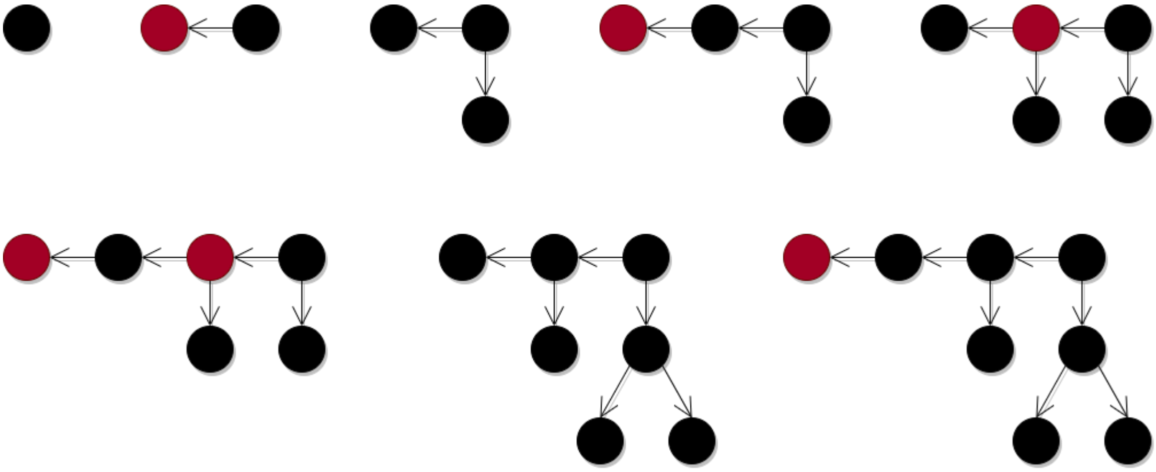
Oliver Killane

22/11/21

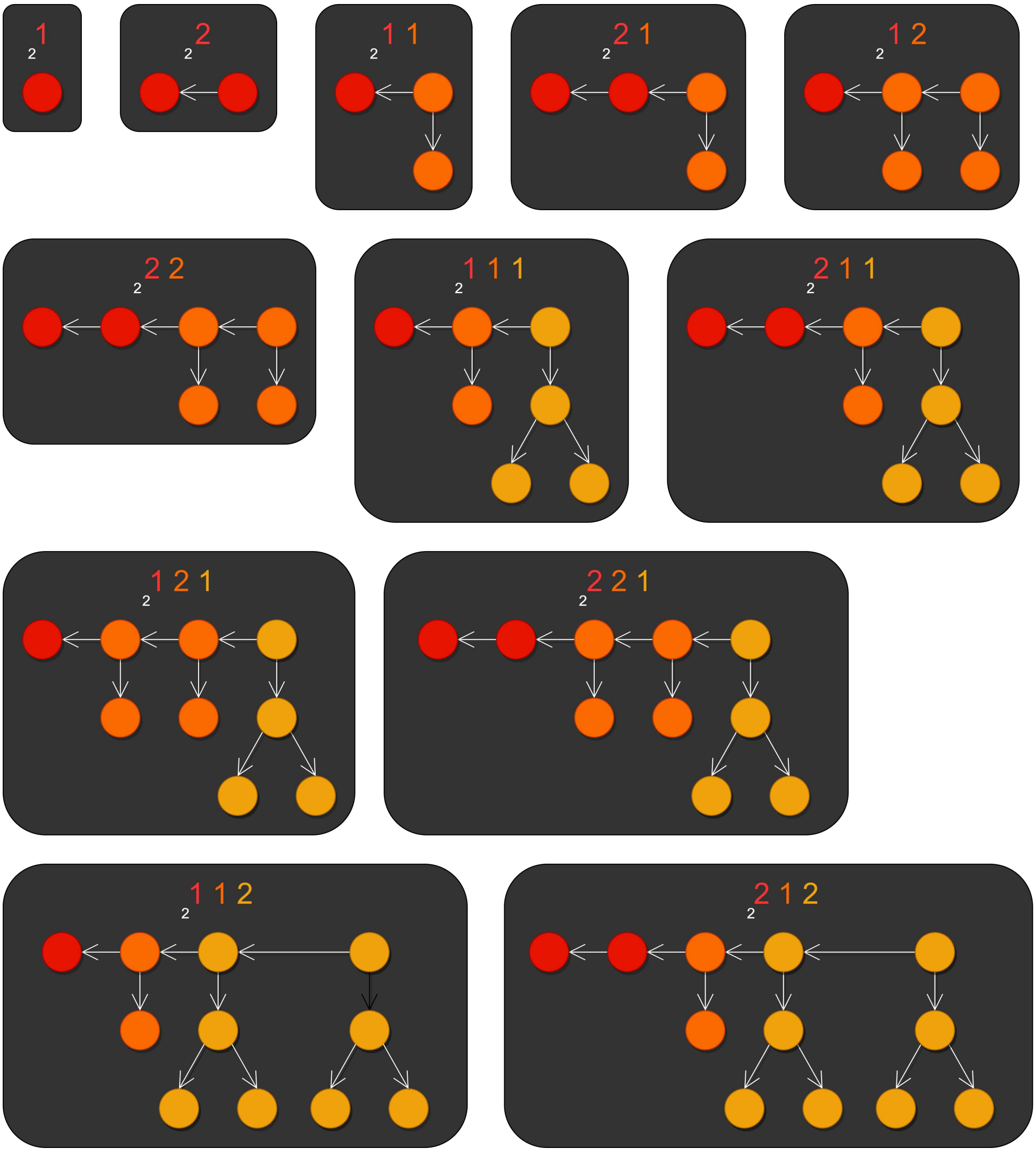
Lecture Recording
Lecture recording is available here

Red Black Trees Continued

We have a pattern with inserting elements from an ordered list into the tree.



We can encode this as a special binary number system, using 1 and 2 such that the least significant bit is a the number of trees of 2^0 nodes, and the nth is 2^n.



We can increment this by:

```
-- each element is a red black tree (+ an extra root element (a))
data Digit a = One a (RBTree a) | Two a (RBTree a) a (RBTree a)

incr :: a -> RBTree a -> [Digit a] -> [Digit a]
incr x t [] = [One x t]
incr x t ((One y u) : ds) = Two x t y u : ds
incr x t ((Two y u z v) : ds) = One x t : incr y (Node Black u z v) ds
```

We can convert a list of digits back to a red black tree by:

```
-- fold left to combine the digits together into a tree
fromList :: [a] -> RBTree a
fromList xs = foldl link Empty (foldr add xs)

link :: RBTree a -> Digit a -> RBTree a
link l (One x t) = Node Black l x t
link l (Two x t y u) = Node Black (Node Red l x t) y u
```

50001 - Algorithm Analysis and Design - Lecture 14

Oliver Killane

28/11/21

Lecture Recording

Lecture recording is available here

Randomized Algorithms

An algorithm that uses random values to produce a result.

Algorithm Type	Running time	Correct Result
Monte Carlo	Predicatable	Unpredictably
Las Vegas	Unpredictable	Predictably

Random Generation

Functions are deterministic (always map same inputs to same outputs), this is known as **Leibniz's law** or the **Law of indiscernibles**:

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

We can exhibit pesudo random behaviour using an input that varies

explicitly	(e.g Random numbers through seeds)
implicitly	(e.g Microphone or camera noise)

Inside IO Monad

We can use basic random through the IO monad like this:

```
import Control.Monad.Random (getRandom)

main :: IO ()
main = do
    x <- getRandom :: IO Int
    print (42 + x)
```

However using the **IO monad** is too specific, we may want to use random numbers in other contexts.

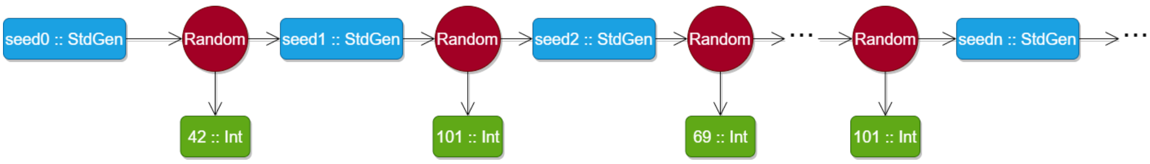
StdGen

In haskell we can use **Stdgen**.

```
import System.Random (StdGen)

-- Create a source of randomness from an integer seed
mkStdGen :: Int -> StdGen

-- Generate a random interger, and a new source of randomness
random :: StdGen -> (Int, StdGen)

-- Generate an infinite list of random numbers using an initial seed
-- (source of random).
randoms :: StdGen -> [Int]
randoms seed = x:randoms seed' where (x, seed') = random seed

-- In order to generate random value for any type, a typeclass is used
class Random a where
```

```
random   :: StdGen -> (a, StdGen)
randoms  :: StdGen -> [a]

-- Random over a (R)ange
randomR  :: (a,a)  -> StdGen -> (a, StdGen)
randomRs:: (a,a)  -> StdGen -> [a]
```

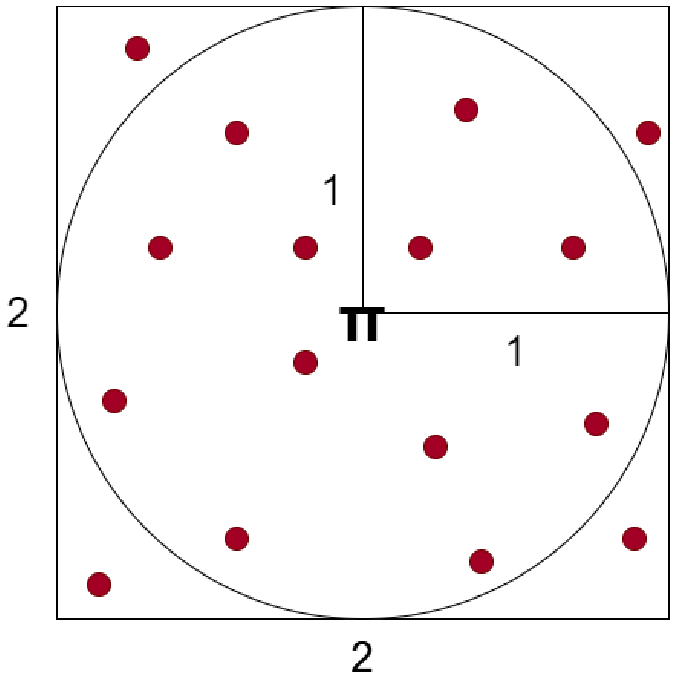
By passing the newly generated **StdGen** we can generate new values based on the original seed.



With Random Monad

Rather than passing **StdGen** seeds through the program, we can use the **MonadRandom** monad which internally uses this value.

Randomized π



(Monte Carlo Algorithm - known number of samples, known running time per sample) To estimate π, find the proportion of randomly selected spots that are within the circle.

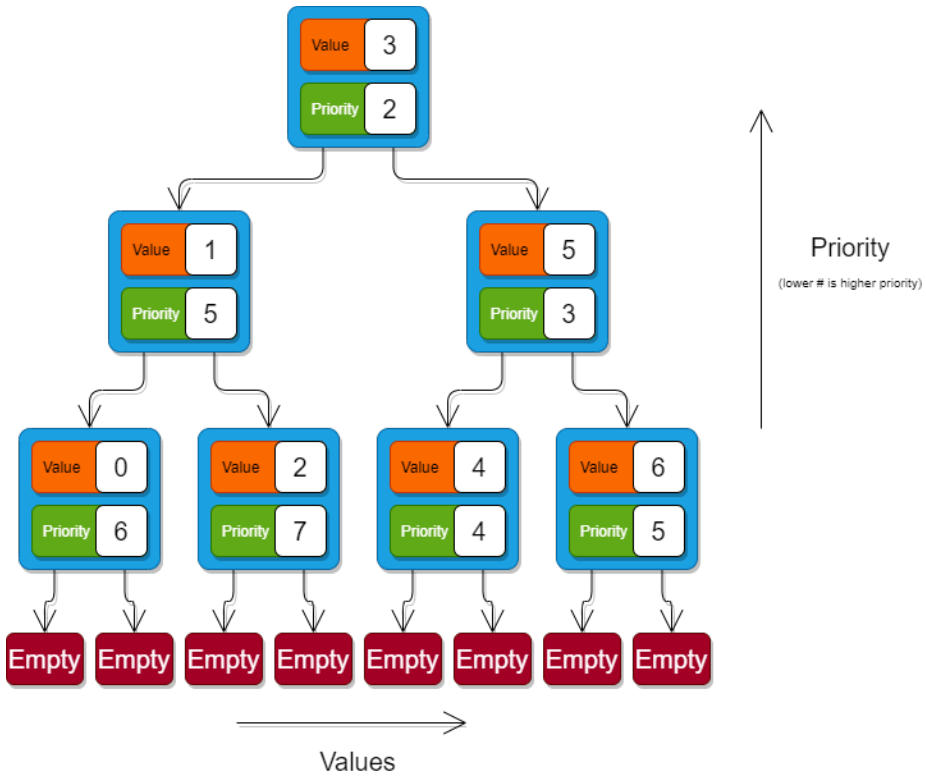
$$\begin{array}{ll} \text{Area of square} & 2 \times 2 = 4 \\ \text{Area of circle} & \pi \times 1^2 = \pi \\ \text{Probability in circle} & \frac{\pi}{4} \end{array}$$

Once we have the proportion, we can multiply by 4 to get an estimate of π.

```haskell
import Control.Monad.Random (getRandomR, randomRs, MonadRandom)
import System.Random (mkStdGen, StdGen)

-- Here we can use one quarter of the circle, hence if the distance from the
-- bottom left (0,0) to the point is within 1 then it is in the circle.
inside :: Double -> Double -> Bool
inside x y = 1 >= x * x + y * y

-- Take 1000 samples and return 4 * the proportion.
montePi :: MonadRandom m => m Double
montePi = loop samples 0
  where
    samples = 10000
    loop 0 m = return (4 * fromIntegral m / fromIntegral samples)
    loop n m = do
      x <- getRandomR (0,1)
      y <- getRandomR (0,1)
      loop (n-1) (if inside x y then m + 1 else m)




-- Using a stream of random numbers (RandomRs)

-- Get pairs of random numbers from the stream
pairs :: [a] -> [(a,a)]
pairs (x:y:ls) = (x,y):pairs ls

-- From the pairs of random numbers, get the proportion of points inside the
-- circle and use to get pi.
montePi' :: Double
montePi' = 4 * hits src / fromIntegral samples
  where
    samples = 10000
    hits    = fromIntegral .
              length .
              filter (uncurry inside) .
              take samples .
              pairs
    src     = randomRs (0, 1) (mkStdGen 42) :: [Double]
```

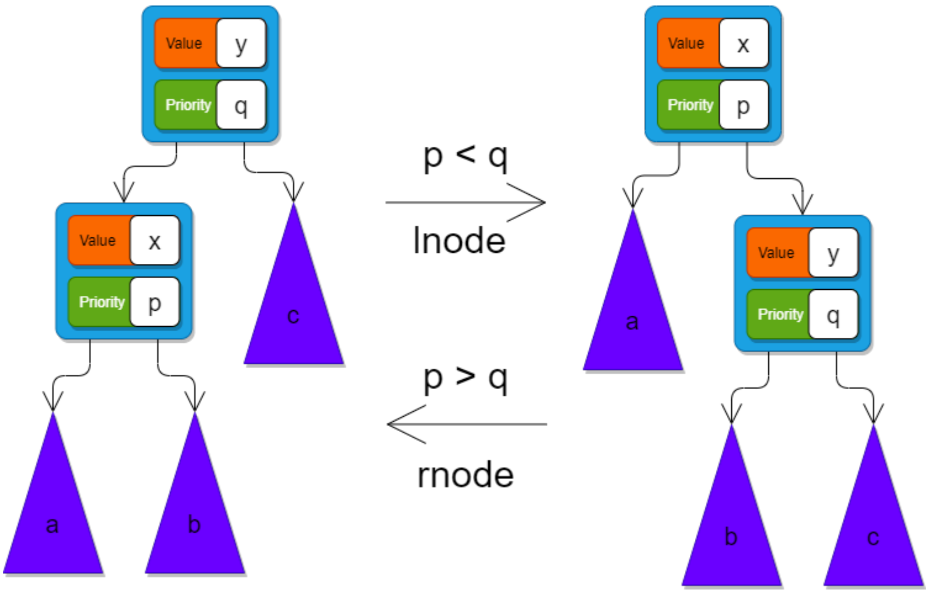
Treaps

Simultaneously a **Tree** and a **Heap**. Stores values in order, while promoting higher priority nodes to the top of the tree.




```haskell
-- Node contains child treaps, as well as value (a) and the priority (Int)
data Treap a = Empty | Node (Treap a) a Int (Treap a)

-- Normal tree search using values
member :: Ord a => a -> Treap a -> Bool
member x (Node l y _ r)
  | x == y    = True
  | x < y     = member x l
  | otherwise = member x r
member _ Empty = False

-- Priority based insert
pinsert :: Ord a => a -> Int -> Treap a -> Treap a
pinsert x p Empty = Node Empty x p Empty
pinsert x p t@(Node l y q r)
  | x == y    = t
  | x < y     = lnode (pinsert x p l) y q r
  | otherwise = rnode l y q (pinsert x p r)

-- rotate right (check left node)
lnode :: Treap a -> a -> Int -> Treap a -> Treap a
lnode Empty y q r = Node Empty y q r
lnode l@(Node a x p b) y q c
  | q > p     = Node a x p (Node b y q c)
  | otherwise = Node l y q c

-- rotate left (check right node)
rnode :: Treap a -> a -> Int -> Treap a -> Treap a
rnode l y q Empty = Node l y q Empty
```

```
rnode a x p r@(Node b y q c)
  | q < p     = Node (Node a x p b) y q c
  | otherwise = Node a x p r

-- delete node by recursively searching, then delete and merge subtrees
delete :: Ord a => a -> Treap a -> Treap a
delete x Empty = Empty
delete x (Node a y q b)
  | x == y    = merge a b
  | x < y     = Node (delete x a) y q b
  | otherwise = Node a y q (delete x b)

merge :: Treap a -> Treap a -> Treap a
merge Empty r = r
merge l Empty = l
merge l@(Node a x p b) r@(Node c y q d)
  | p < q     = Node a x p (merge b r)
  | otherwise = Node (merge l c) y q d
```



50001 - Algorithm Analysis and Design - Lecture 15

Oliver Killane

29/11/21

Lecture Recording

Lecture recording is available here

Randomized Treaps

By using a random value for priority when inserting values into the treap, we can ensure a high likelihood of balancing, without complex balancing being required.

We can use this to create a randomized quicksort.

```haskell
import System.Random (StdGen, mkStdGen, random)
-- node random :: StdGen -> (Int, StdGen)

data RTreap a = RTreap StdGen (Treap a)

insert :: Ord a => a -> RTreap a -> RTreap a
insert x (RTreap seed t) = RTreap seed' (pinsert x p t)
  where (p, seed') = random seed

-- note 42 is used for
empty :: RTreap a
empty = RTreap (mkStdGen 42) Empty

-- Build up tree, requires O(n log n)
fromList :: Ord a => [a] -> RTreap a
fromList xs = foldr insert empty xs

-- Linear time conversion (use treap tolist)
toList :: RTreap a -> [a]
toList (RTreap _ t) = tolist t

-- Randomiozed Quicksort O(n log n)
-- Effectively the random priorities are the partitions, first pivot is the
-- highest priority.
rquicksort :: Ord a => [a] -> [a]
rquicksort = toList . fromList
```

Randomized Binary Trees

We can balance a binary tree without using a treap, by inserting at the root (and rotating the tree to ensure it is ordered) with a certain probability.

```haskell
import System.Random (StdGen, mkStdGen, randomR)

data BTree a = Empty | Node (BTree a) a (BTree a)

insert :: Ord a => a -> BTree a -> BTree a
insert x Empty = Node Empty x Empty
insert x t@(Node l y r)
  | x == y = t
  | x < y = Node (insert x l) y r
  | otherwise = Node l y (insert x r)

```

```haskell
-- basic lefty/right rotations
rotr :: BTree a -> a -> BTree a -> BTree a
rotr (Node a x b) y c = Node a x (Node b y c)
rotr _ _ _= error "(rotr): left was empty"

rotl :: BTree a -> a -> BTree a -> BTree a
rotl a x (Node b y c) = Node (Node a x b) y c
rotl _ _ _= error "(rtol): right was empty"


-- Insert to the root of the tree (maintaining order)
insertRoot :: Ord a => a -> BTree a -> BTree a
insertRoot x Empty = Node Empty x Empty
insertRoot x t@(Node l y r)
  | x == y = t
  | x < y = rotr (insertRoot x l) y r
  | otherwise = rotl l y (insertRoot x r)

-- Randomized binary tree
data RBTree a = RBTree StdGen Int (BTree a)

empty :: RBTree a
empty = RBTree (mkStdGen 42) 0 Empty

-- chance of 1 / n+1 of inserting at root.
insert' :: Ord a => a -> RBTree a -> RBTree a
insert' x (RBTree seed n t) = RBTree seed' (n+1) (f x t)
  where
    f = case p of
      0 -> insertRoot
      _ -> insert
    (p, seed') = randomR (0,n) seed
```

For every insert we have chance $\frac{1}{n+1}$ of inserting at the root of the tree. Then this occurs, the contents are rotated to ensure the tree's ordering is maintained.

This means that there is a very high probability of balance being maintained, however correct results are only returned when distinct elements are inserted at most once.

50001 - Algorithm Analysis and Design - Lecture 16

Oliver Killane

18/12/21

Lecture Recording

Lecture recording is available here

Mutable Algorithms

We can use **STRef s a** to hold a mutable reference to **a** that can be created, read and modified.

```haskell

-- State Transformer (ST) takes a state s and return value a.
-- "Give me any program with any state s, and if it returns an a, so will I".
runST :: (forall s . ST s a) -> a

-- Take a value a, produces a program with some state s, that returns a
-- reference with that state and the value stored.
newSTRef :: a -> ST s (STRef s a)

-- Takes in a reference in some state s, performs a computation to get a
readSTRef :: STRef s a -> ST s a

-- Take in a reference, and a new value a, performing a computation to update
-- the referenced value (update does not return anything itself, hence ()).
writeSTRef :: STRef s a -> a -> ST s ()

-- Takes a value, returns a computation that executes in any state s to return
-- an a.
return :: a -> ST s a
```

We can use this to create a mutable version of fibonacci.

```haskell
import Data.STRef (newSTRef, readSTRef, writeSTRef)
import Control.Monad.ST (runST)

-- immutable looping fibonacci
fib :: Int -> Integer
fib n = loop n 0 1
  where
    loop :: Int -> Integer -> Integer -> Integer
    loop 0 x y = x
    loop n x y = loop (n-1) y (x+y)

-- mutable looping fibonacci
fib0 :: Int -> Integer
fib0 n = runST $ do
  rx <- newSTRef 0
  ry <- newSTRef 1
  let loop 0 = do readSTRef rx
      loop n = do {
        x <- readSTRef rx;
        y <- readSTRef ry;
        writeSTRef rx y;
        writeSTRef ry (x + y);
        loop (n - 1);}
  loop n
```

Mutable Datastructures

Array

```haskell
import Control.Monad.ST
import Data.Array.ST

-- Use indexable i, range, default to get a mutable array in some state
newArray :: Ix i => (i,i) -> a -> ST s (STArray s i a)

-- Use indexable i, a mutable array and return the value a with state s
readArray :: Ix i -> STArray s i a -> i -> ST s a

-- Use indexable i, a mutable array, and the value to place, return a
-- computation that updates the array (new state) but returns no value.
-- note:
writeArray :: Ix i => STArray s i a -> i -> a -> ST s ()
```

Each operation is assumed to take constant time.

For example, an algorithm to find the smallest natural number not in a list.

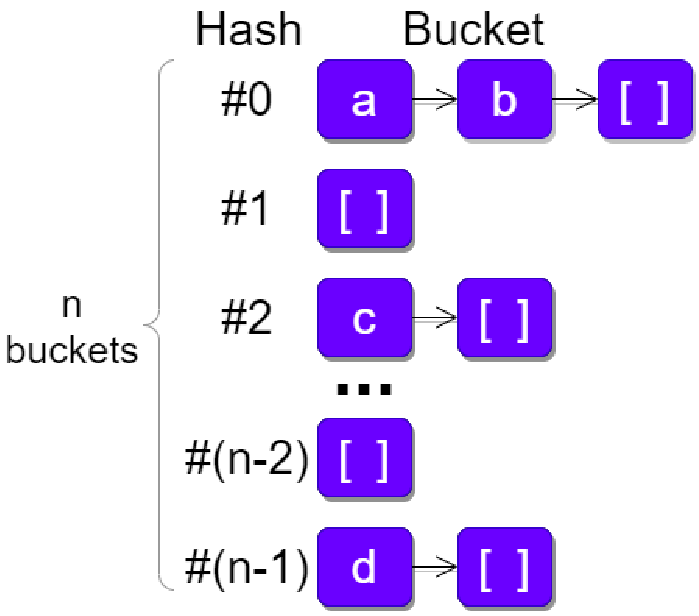
```haskell
import Data.Array.MArray (MArray(newArray))
import Data.Array.ST
import Control.Monad.ST
import Data.List ((\\))

-- immutable
minfree :: [Int] -> Int
minfree xs = head ( [0..] \\ xs )

-- effectively the same as minfree
minfree' :: [Int] -> Int
minfree' xs = head . filter(not . (`elem` xs)) $ [0..]


-- Builds up an array of which are present,
minfreeMut :: [Int] -> Int
minfreeMut = length . takeWhile id . checklist

{-
Build an array, at each index True/False for if the index is in xs
We only need to use an array of size (length xs) as we do not care about
natural numbers larger than this (if they are in the list, then a smaller
natural number was missed).

xs [0,1,2,3,6,7,8]
ys [T,T,T,T,F,F,T] ... (don't care about 7 or 8)

-}
checklist :: [Int] -> [Bool]
checklist xs  = runST $ do {
  ys <- newArray (0,l - 1) False :: ST s (STArray s Int Bool);
  sequence [writeArray ys x True | x <- xs, x < l];
  getElems ys;
  }
  where
    l = length xs
```

Hash

```haskell
class Hashable a where
    hash :: a -> Int
```

A hash generates an integer from some data. Typically range restricted (e.g hashmap can hold a finite number of entries), and the hash function should be designed to reduce collisions (two distinct data having the same hash).

Below is an example of a bucket based hash map, using linked list buckets.



50001 - Algorithm Analysis and Design - Lecture 17

Oliver Killane

18/12/21

Lecture Recording

Lecture recording is available here

Mutable Nub

This can be useful for the **nub** function (removes duplicates from a list) by using a bucket based hashmap of items from the list to determine duplication.

```haskell
import Control.Monad (when)
import Data.Array.ST
    ( getElems, newListArray, readArray, writeArray, STArray )
import Control.Monad.ST ( ST, runST )


-- immutable version:
nub :: Eq a => [a] -> [a]
nub = reverse . foldl nubHelper []
  where
    nubHelper :: Eq a => [a] -> a -> [a]
    nubHelper ns c
      | c `elem` ns = ns
      | otherwise   = c:ns

-- mutable version, create a hash table of characters to track which have
-- already been seen.
nubMut :: (Hashable a, Eq a) => [a] -> [a]
nubMut xs = concat $ runST $ do
  axss <- newListArray (0, n - 1) (replicate 256 [ ]) :: ST s (STArray s Int [a])
  sequence [do {
    let hx = hash x `mod` (n - 1)
    ys <- readArray axss hx
    unless (x `elem` ys) $ do writeArray axss hx (x : ys)}
  | x <- xs]
  getElems axss
  where
    -- number of buckets in the hash table
    n = 256
```

Quicksort

We can also implement quicksort, which can be done in place in an array (saved memory and time as accesses are constant time).

By using mutable data structures we can swap elements when reordering by reading and writing from the array.

```haskell
import Data.Array.ST ( readArray, writeArray, STArray, getElems )
import Control.Monad.ST ( ST )

-- swap elements over (classic store in temp & swap over other before writeback)
swap :: STArray s Int a -> Int -> Int -> ST s ()
swap axs i j = do
  temp <- readArray axs i
```

```
  readArray axs j >>= writeArray axs i
  writeArray axs j temp

qsort :: Ord a => [a] -> [a]
qsort xs = runST $ do
  axs <- newListArray (0,n) xs
  aqsort axs 0 n
  getElems axs
  where
    n = length xs - 1

{-
Partition around a pivot (k) (all smaller to left, larger to right
(unsorted)), then recur on these partitions
Index:     0       (k-1) k (k+1)    n
Contents: [ ..<= x .. ][x][ .. >x ..]
-}
aqsort :: Ord a => STArray s Int a -> Int -> Int -> ST s ()
aqsort axs i j
  | i >= j = return ()
  | otherwise = do
      k <- apartition axs i j
      aqsort axs i (k - 1)
      aqsort axs (k + 1) j

apartition :: Ord a => STArray s Int a -> Int -> Int -> ST s Int
apartition asx p q = do
  x <- readArray axs p
  let loop i j
    | i > j = do
      swap axs p j
      return j
    | otherwise = do
      u <- readArray axs i
      if u < x
        then do loop (i + 1) j
        else do
          swap axs i j
          loop i (j - 1)
  loop (p+1) q
```