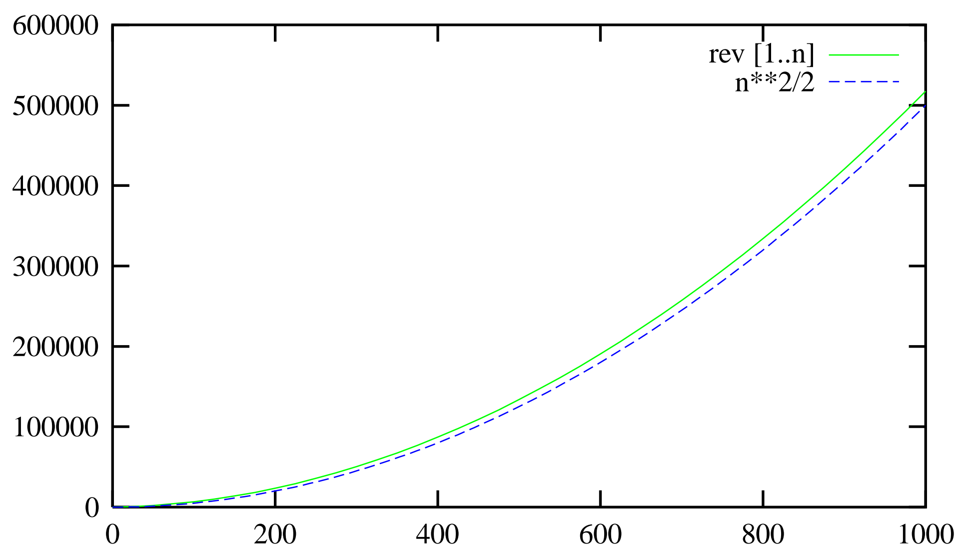
Reversing Lists

“Haskell: The craft of functional programming,” by Simon Thompson, page 140

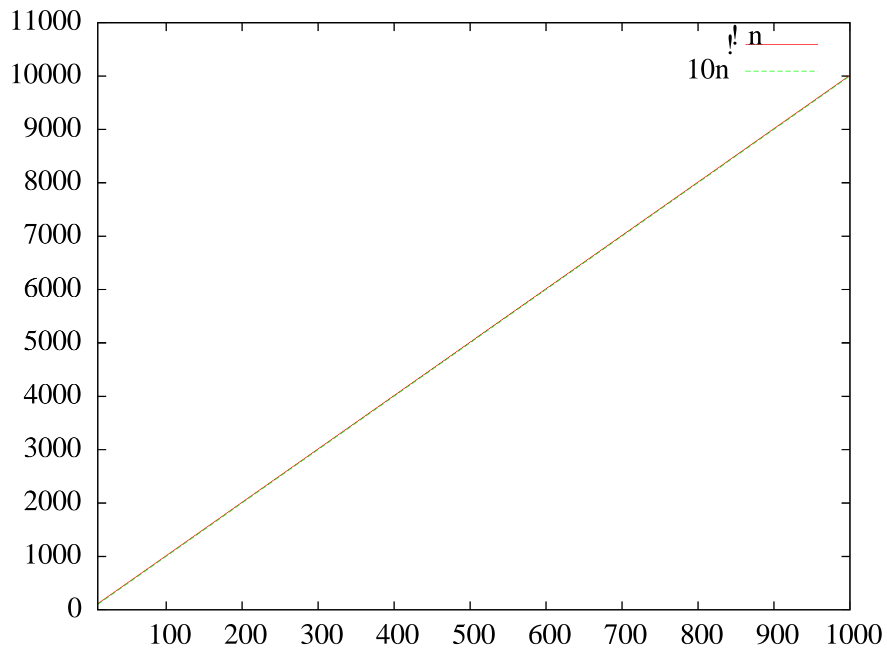
— Reverse.hs —

```
rev :: [t] -> [t]

rev [] = []
rev (e:l) = (rev l) ++ [e]
```



Array Indexing

!! is included in the Prelude of `Hugs`

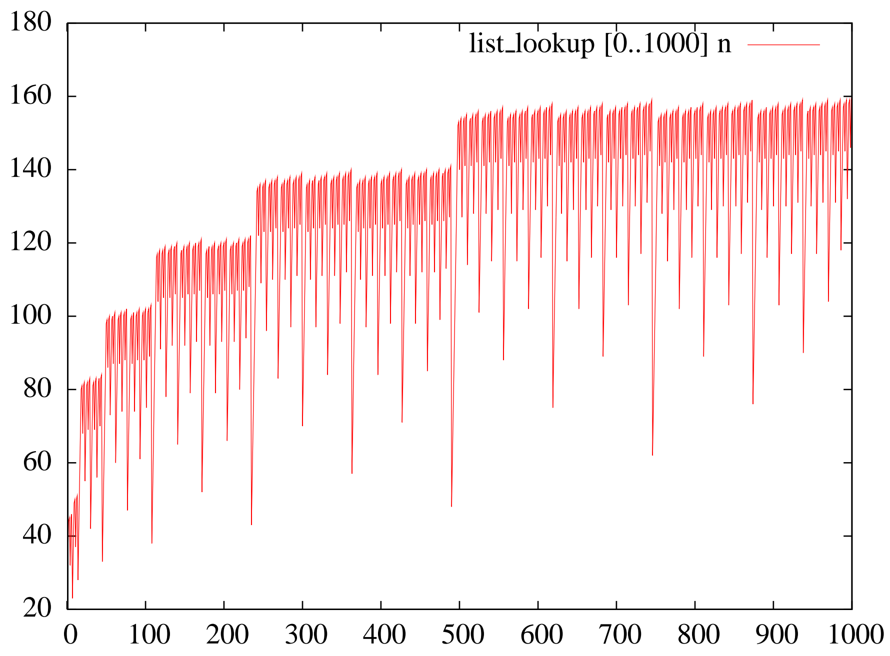


Functional Arrays

— FunctionalArray.hs —

```
module FunctionalArray where

data Tree t = Leaf t | Node t (Tree t) (Tree t)

type Func_array t = [(Int,Tree t)]

list_empty   :: Func_array t
list_isempty :: Func_array t -> Bool
list_head    :: Func_array t -> t
list_tail    :: Func_array t -> Func_array t
list_cons    :: t -> Func_array t -> Func_array t
list_lookup  :: Func_array t -> Int -> t
list_update  :: Func_array t -> Int -> t -> Func_array t

tree_lookup  :: Int -> Tree t -> Int -> t
tree_update  :: Int -> Tree t -> Int -> t -> Tree t

tree_lookup size (Leaf e) 0 = e
tree_lookup size (Node e t1 t2) 0 = e
tree_lookup size (Node e t1 t2) i
    | i<=size'  = tree_lookup size' t1 (i-1)
    | otherwise = tree_lookup size' t2 (i-1-size')
    where size' = div size 2

tree_update size (Leaf e) 0 v = Leaf v
tree_update size (Node e t1 t2) 0 v = Node v t1 t2
tree_update size (Node e t1 t2) i v
    | i<=size'  = Node e (tree_update size' t1 (i-1) v) t2
    | otherwise = Node e t1 (tree_update size' t2 (i-1-size') v)
    where size' = div size 2

list_empty = []

list_isempty [] = True
list_isempty _  = False

list_head ((_,Leaf e):_) = e
list_head ((_,Node e _ _):_) = e

list_tail ((_,Leaf e):l) = l
list_tail ((size,Node e t1 t2):l) = ((size',t1):(size',t2):l)
    where size' = div size 2

list_cons e ((size1,t1):(size2,t2):l)
    | size1==size2 = ((1+2*size1),Node e t1 t2):l
    | otherwise    = ((1,Leaf e):(size1,t1):(size2,t2):l)
list_cons e l = ((1,Leaf e):l)

list_lookup ((size,t):l) i
    | i<size = tree_lookup size t i
    | otherwise = list_lookup l (i-size)

list_update ((size,t):l) i v
    | i<size = ((size,tree_update size t i v):l)
    | otherwise = ((size,t):list_update l (i-size) v)
```

Functional Arrays: Performance



Stacks

— Stack.hs —

```
module Stack where

type Stack t = [t]

push   :: Stack t -> t -> Stack t
pop    :: Stack t -> (t,Stack t)
empty  :: Stack t

push l e = (e:l)

pop (e:l) = (e,l)

empty = []
```

Queues

— Queue1.hs —

```
module Queue where

type Queue t = [t]

inject :: Queue t -> t -> Queue t
pop    :: Queue t -> (t,Queue t)
empty  :: Queue t

inject l e = l ++ [e]

pop (e:l) = (e,l)

empty = []
```

— Queue2.hs —

```
module Queue where

type Queue t = ([t],[t])

inject :: Queue t -> t -> Queue t
pop    :: Queue t -> (t,Queue t)
empty  :: Queue t

inject (l,r) e = (l,(e:r))

pop ((e:l),r) = (e,(l,r))
pop ([],(e:r)) = pop (reverse (e:r),[])

empty = ([],[])
```

Queues: Real-time

— Queue3.hs —

```
module Queue where

type Queue t = ([t],[t],Work t)

inject    :: Queue t -> t -> Queue t
pop       :: Queue t -> (t,Queue t)
empty     :: Queue t
progress  :: Queue t -> Queue t
progress' :: Queue t -> Queue t

data Work t = Nil | Rev [t] [t] | Cat Int [t] [t] [t]

inject (l,r,w) e = progress(progress(progress (l,e:r,w)))

pop (e:l,r,w) = (e, progress(progress(progress' (l,r,w))))

empty = ([],[],Nil)

progress (l,[],Nil)               = (l,[],Nil)
progress (l,r,Nil)                = progress (l,[],Rev [] r)
progress (l,r,Rev l1 (e:r1))      = (l,r,Rev (e:l1) r1)
progress (l,r,Rev l1 [])          = progress (l,r,Cat 0 [] l l1)
progress (l,r,Cat s r1 (e:l1) l2) = (l,r,Cat (s+1) (e:r1) l1 l2)
progress (l,r,Cat 0 r1 [] l2)     = (l2,r,Nil)
progress (l,r,Cat 1 (e:r1) [] l2) = ((e:l2),r,Nil)
progress (l,r,Cat s (e:r1) [] l2) = (l,r,Cat (s-1) r1 [] (e:l2))

progress' (l,r,Cat s r1 l1 l2) = progress (l,r,Cat (s-1) r1 l1 l2)
progress' w = progress w
```

Double Ended Queues

— Deque.hs —

```
module Deque where

type Queue t = [t]

push   :: Queue t -> t -> Queue t
pop    :: Queue t -> (t,Queue t)
inject :: Queue t -> t -> Queue t
eject  :: Queue t -> (t,Queue t)
empty  :: Queue t

push l e = [e] ++ l
pop (e:l) = (e,l)
inject l e = l ++ [e]
eject [e] = (e,[])
eject (e:l) = (e',e:l')
              where (e',l') = eject l
empty = []
```

Deques: Amortized Constant

— Deque2.hs —

```
module Deque where

type Queue t = ([t],[t])

push   :: Queue t -> t -> Queue t
pop    :: Queue t -> (t,Queue t)
inject :: Queue t -> t -> Queue t
eject  :: Queue t -> (t,Queue t)
empty  :: Queue t

push (l,r) e = (e:l,r)

pop (e:l,r) = (e,(l,r))
pop ([],r) = pop (l',r')
             where l'=reverse (drop s r)
                   r'=take s r
                   s=div (length r) 2

inject (l,r) e = (l,e:r)

eject (l,e:r) = (e,(l,r))
eject (l,[]) = eject (l',r')
               where l'=take s l
                     r'=reverse (drop s l)
                     s=div (length l) 2

empty = ([],[])
```

Binomial Queues

— Binomial.hs —

```
module BinomialQ where

data Tree t = Node t Int [Tree t]

type BinomialQ t = [Tree t]

empty      :: Ord t => BinomialQ t
is_empty   :: Ord t => BinomialQ t -> Bool
insert     :: Ord t => BinomialQ t -> t -> BinomialQ t
meld       :: Ord t => BinomialQ t -> BinomialQ t -> BinomialQ t
find_min   :: Ord t => BinomialQ t -> t
delete_min :: Ord t => BinomialQ t -> BinomialQ t

link (Node e1 r1 c1) (Node e2 r2 c2)
  | e1<e2 = Node e1 (r1 +1) ((Node e2 r2 c2):c1)
  | e1>=e2 = Node e2 (r2 +1) ((Node e1 r1 c1):c2)

ins [] v = [v]
ins ((Node e1 r1 c1):l) (Node e2 r2 c2)
  | r2<r1  = ((Node e2 r2 c2):(Node e1 r1 c1):l)
  | r1==r2 = ins l (link (Node e1 r1 c1) (Node e2 r2 c2))

empty = []

is_empty q = null q

insert q e = ins q (Node e 0 [])

meld [] q = q
meld q [] = q
meld ((Node e1 r1 c1):l1) ((Node e2 r2 c2):l2)
  | r1<r2  = ((Node e1 r1 c1):(meld l1 ((Node e2 r2 c2):l2)))
  | r1>r2  = ((Node e2 r2 c2):(meld l2 ((Node e1 r1 c1):l1)))
  | r1==r2 = ins (meld l1 l2) (link (Node e1 r1 c1) (Node e2 r2 c2))

find_min [(Node e r c)] = e
find_min ((Node e r c):l) = min e (find_min l)

delete_min q = meld  l (reverse c)
  where ((Node e r c),l) = get_min q
        get_min [(Node e r c)] = ((Node e r c),[])
        get_min ((Node e r c):l)
          | e<e1  = ((Node e r c),l)
          | e>=e1 = ((Node e1 r1 c1),((Node e r c):l1))
          where (Node e1 r1 c1,l1) = get_min l
```

Skew Binomial Queues

— SkewBQ.hs —

```
module SkewBQ where

data Tree t = Node t Int [Tree t] [t]

type SkewBQ t = [Tree t]

empty      :: Ord t => SkewBQ t
is_empty   :: Ord t => SkewBQ t -> Bool
insert     :: Ord t => t -> SkewBQ t -> SkewBQ t
meld       :: Ord t => SkewBQ t -> SkewBQ t -> SkewBQ t
find_min   :: Ord t => SkewBQ t -> t
delete_min :: Ord t => SkewBQ t -> SkewBQ t

link       :: Ord t => Tree t -> Tree t -> Tree t
skew_link  :: Ord t => t -> Tree t -> Tree t -> Tree t

rank (Node _ r _ _) = r

element (Node e _ _ _) = e

link (Node e1 r1 c1 z1) (Node e2 r2 c2 z2)
   | e1<=e2 = Node e1 (r1 + 1) ((Node e2 r2 c2 z2):c1) z1
   | e1>e2  = Node e2 (r2 + 1) ((Node e1 r1 c1 z1):c2) z2

skew_link e v1 v2
   | e3<=e = Node e3 r3 c3 (e:z3)
   | e3>e  = Node e r3 c3 (e3:z3)
  where (Node e3 r3 c3 z3) = link v1 v2

empty = []

is_empty q = null q

insert e (v1:v2:l)
   | rank v1==rank v2 = ((skew_link e v1 v2):l)
   | otherwise = ((Node e 0 [] []):v1:v2:l)
insert e l = ((Node e 0 [] []):l)

ins [] v = [v]
ins (v1:l) v2
   | rank v1> rank v2 = (v2:v1:l)
   | rank v1==rank v2 = ins l (link v1 v2)

uniqify [] = []
uniqify (v:l) = ins l v

meld_unique [] l = l
meld_unique l [] = l
meld_unique (v1:l1) (v2:l2)
   | rank v1<rank v2 = (v1:(meld_unique l1 (v2:l2)))
   | rank v1>rank v2 = (v2:(meld_unique l2 (v1:l1)))
   | otherwise       = ins (meld_unique l1 l2) (link v1 v2)

meld l1 l2 = meld_unique (uniqify l1) (uniqify l2)

find_min [v] = element v
find_min (v:l) = min (element v) (find_min l)

delete_min q = foldr insert (meld l (reverse c)) z
  where ((Node e r c z),l) = get_min q
        get_min [v] = (v,[])
        get_min (v:l)
           | element v< element v1 = (v,l)
           | element v>=element v1 = (v1,v:l1)
          where (v1,l1) = get_min l
```

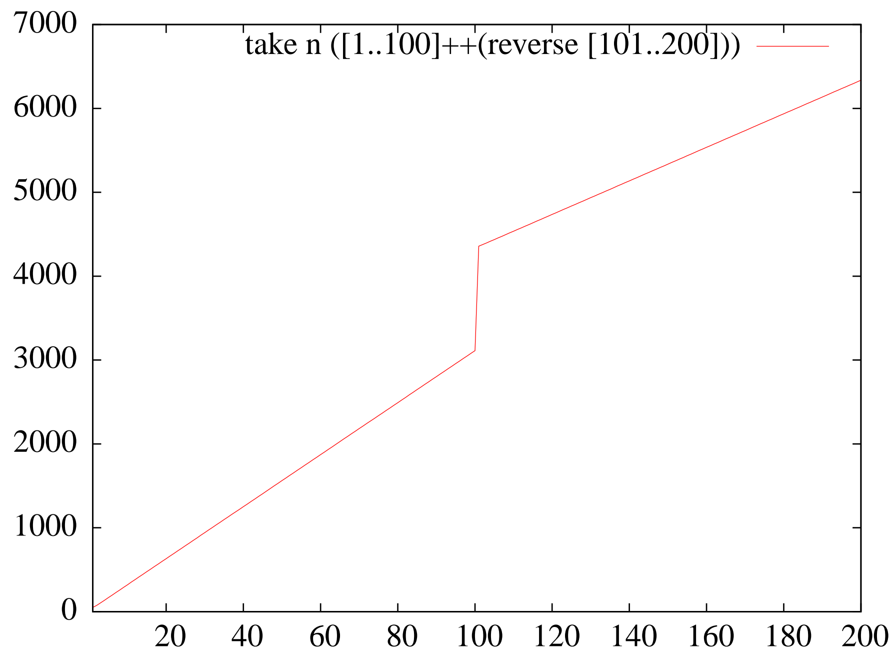
Data Structural Bootstrapping: Constant Time Meld

— BootSkew.hs —

```
module SkewRoot where

import SkewBQ

data BootSkew t = Empty | Nonempty (Elm t)

data Elm t = Element t (SkewBQ (Elm t))

instance Eq t => Eq (Elm t) where
  (Element e1 q1) == (Element e2 q2) = e1 == e2

instance Ord t => Ord (Elm t) where
  (Element e1 q1) <= (Element e2 q2) = e1 <= e2

empty'      :: Ord t => BootSkew t
is_empty'   :: Ord t => BootSkew t -> Bool
insert'     :: Ord t => t -> BootSkew t -> BootSkew t
meld'       :: Ord t => BootSkew t -> BootSkew t -> BootSkew t
find_min'   :: Ord t => BootSkew t -> t
delete_min' :: Ord t => BootSkew t -> BootSkew t

empty' = Empty

is_empty' Empty = True
is_empty' (Nonempty _) = False

insert' e q = meld' (Nonempty (Element e empty)) q

meld' Empty q = q
meld' q Empty = q
meld' (Nonempty (Element e1 q1)) (Nonempty (Element e2 q2))
  | e1 <= e2 = Nonempty (Element e1 (insert (Element e2 q2) q1))
  | e1 >  e2 = Nonempty (Element e2 (insert (Element e1 q1) q2))

find_min' (Nonempty (Element e _)) = e

delete_min' (Nonempty (Element _ q))
  | is_empty q = Empty
  | otherwise  = Nonempty (Element e1 (meld q1 q2))
  where Element e1 q1 = find_min q
        q2 = delete_min q
```

List Catenation v.s. Lazy Evaluation

```
take n ([1..100]++(reverse [101..200])
```



Search Trees v.s. Lazy Evaluation

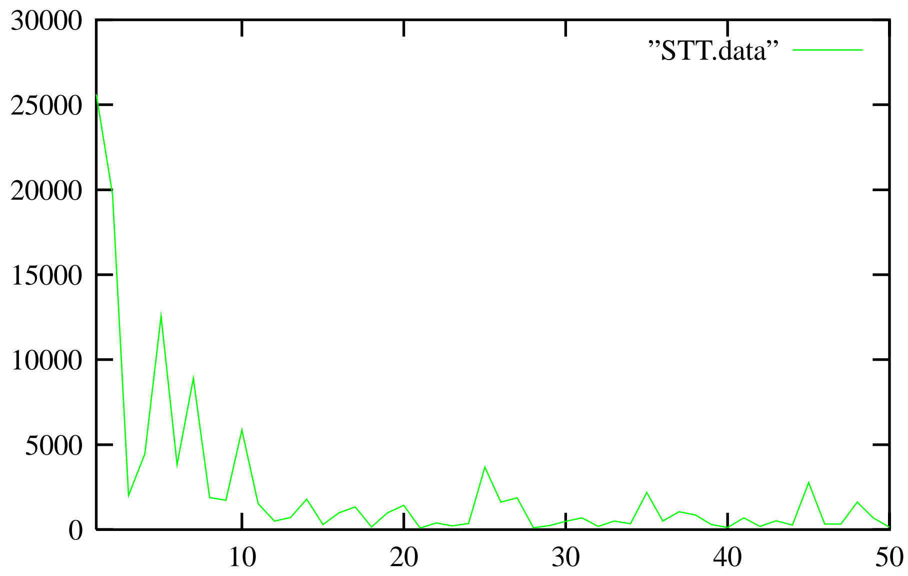
— SearchTree.hs —

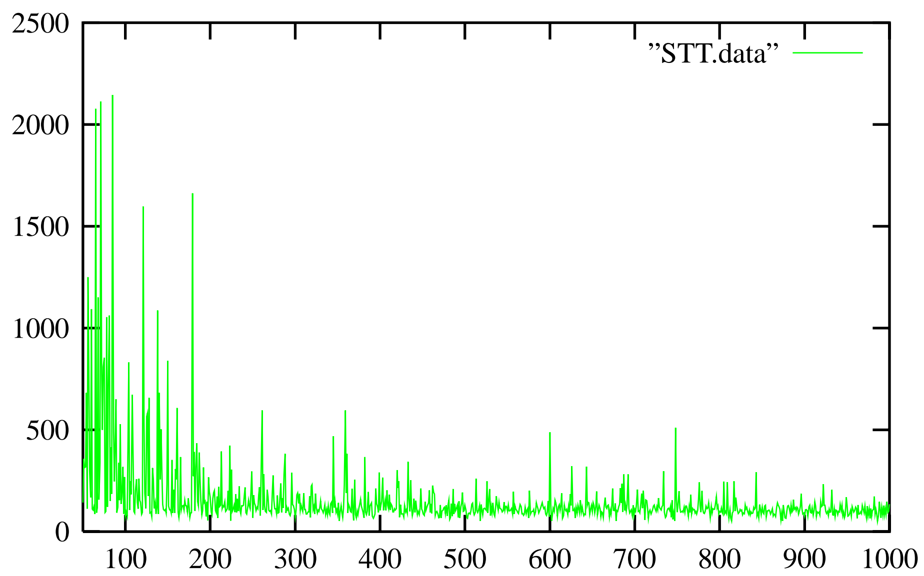
```
module SearchTree where

data Tree t = Empty | Node t (Tree t) (Tree t)

member     :: Ord t => t -> Tree t -> Bool
listToTree :: Ord t => [t] -> Tree t

member x Empty        = False
member x (Node e l r)
   | x==e = True
   | x<e  = member x l
   | x>e  = member x r

listToTree [] = Empty
listToTree l  = Node e (listToTree left) (listToTree right)
                where (e:l1) = l
                      left   = [ x | x<-l1, x<=e ]
                      right  = [ x | x<-l1, x>e  ]
```

1) Apply `listToTree` to a list with 1000 random numbers

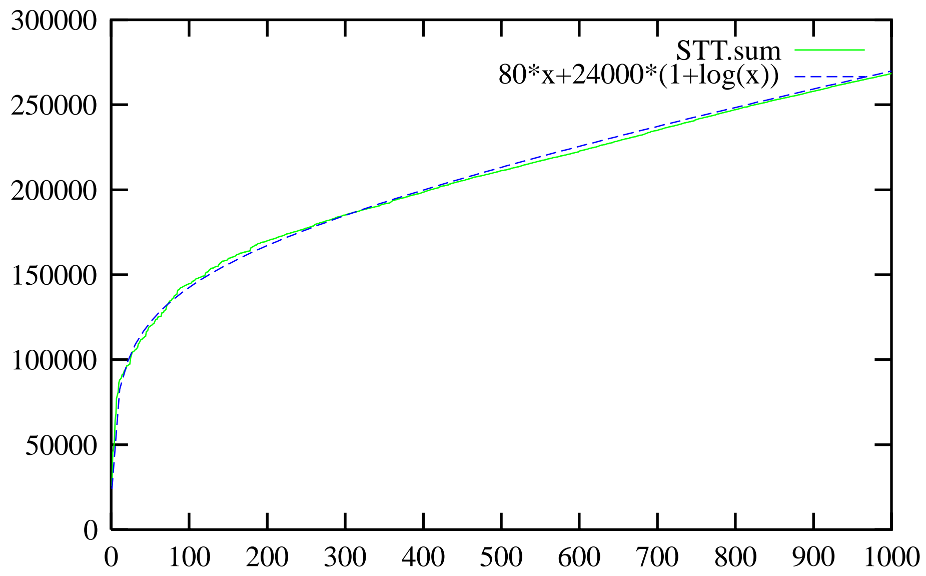
2) Apply `member` with 1000 random numbers

Search Trees v.s. Lazy Evaluation





Search Trees v.s. Lazy Evaluation



Queues v.s. Lazy Evaluation

— Queue4.hs —

```
module Queue where

type Queue t = (Int,[t],Int,[t])

inject :: Queue t -> t -> Queue t
pop    :: Queue t -> (t,Queue t)
adjust :: Queue t -> Queue t
empty  :: Queue t
size   :: Queue t -> Int

inject (sl,l,sr,r) e = adjust (sl,l,sr+1,e:r)

pop (sl,e:l,sr,r) = (e,adjust (sl-1,l,sr,r))

adjust (sl,l,sr,r)
     | sl>=sr     = (sl,l,sr,r)
     | otherwise  = (sl+sr,l++(reverse r),0,[])

empty = (0,[],0,[])

size (sl,_,sr,_) = sl + sr
```

Lists with Lazy Catenation

— CatList.hs —

```
module CatList where

import Queue

data List t = Empty | Non_empty (Node t)
data Node t = Node t (Queue (Node t))

makelist :: t -> List t
hd       :: List t -> t
tl       :: List t -> List t
catenate :: List t -> List t -> List t

link          :: Node t -> Node t -> Node t
link_children :: Queue (Node t) -> Node t

makelist e = Non_empty (Node e empty)

hd (Non_empty (Node e _)) = e

tl (Non_empty (Node _ c))
  | size c == 0  = Empty
  | otherwise    = Non_empty (link_children c)

catenate Empty l = l
catenate l Empty = l
catenate (Non_empty v1) (Non_empty v2) = Non_empty (link v1 v2)

link (Node e c) v = Node e (inject c v)

link_children c
  | size c == 1  = chd
  | otherwise    = link chd (link_children ctl)
                   where (chd,ctl) = pop c
```

Reversing Lists

— Reverse.hs —

```
rev :: [t] -> [t]

rev [] = []
rev (e:l) = (rev l) ++ [e]
```