Functional Data Structures

Gerth Stølting Brodal
Max-Planck-Institut für Informatik
Saarbrücken, Germany

May 5-12, 1998

1 Lecture 1

1.1 Introduction

In this sequence of lectures I will talk about **Functional Data Structures**, i.e., data structures suitable for functional programming languages.

- Purely functional languages do not allow destructive updates, **no assignments**,
- **lists** and **tuples** are the basic tools for building data structures,
- data structures can be viewed as **graphs** with nodes of out-degree $O(1)$. Integers, chars, etc. are stored at nodes of degree zero, and
- **nodes cannot be modified**.

We start with giving two natural examples illustrating the limitations of the built-in list representation of the functional language **Hugs**.

Most of the work presented has been done by Cris Okasaki from Carnegie Mellon University.

1.1.1 Example: List reversion

In a functional programming language it usually takes *linear time* to catenate two lists (linear in the length of the first list). We consider a natural implementation of a function *reversing* a list which is based on list catenation. It turns out that the running time of the list reversing procedure is quadratic - for a problem which obviously can be solved in linear time.

Two purposes: Introduce some **syntax**, and illustrate aspects of **lists** in the **functional language Haskell**.

The following is an example taken from the book:

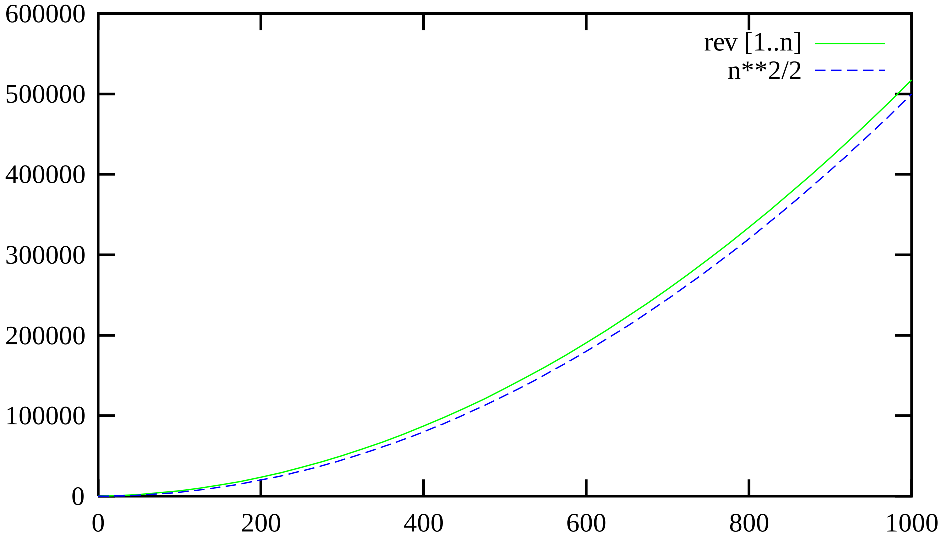
> "Haskell: The craft of functional programming," by S. Thompson, page 140.

List reversing program:
— Reverse.hs —

```
rev :: [t] -> [t]
rev [] = []
rev (e:l) = (rev l) ++ [e]
```

Explanation:

- line 1 = **type definition**: Function rev takes a **list** of elements of type `t` and returns a list of elements of type `t`
- lines 2-3 = **implementation**: Two cases, captured by **patterns**: the reverse of an empty list is the empty list, and the reverse of a nonempty list with **first-element** `e` and **tail** `l` is recursively defined. `++` denotes **list-catenation**.



[[SLIDE]] Experiments show that list reversion requires time $\Theta(n^2)$.

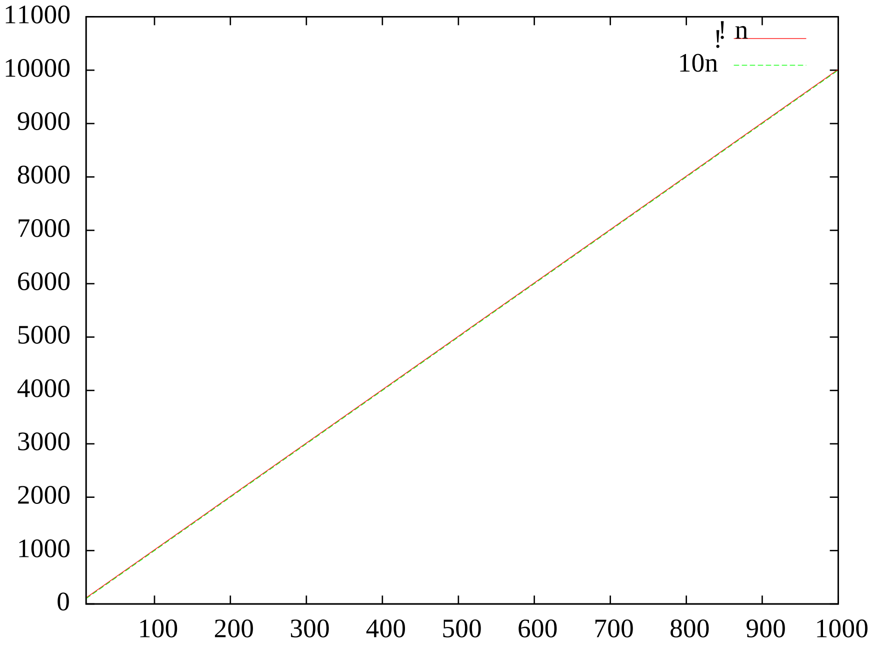
The reason: **List catenation takes time linear in the length of the first list**.

Blackboard example: [1,2,3]++[4,5].

A more efficient representation of lists supporting constant time catenation will be given later in this course (3. lecture).

1.1.2 Example: Array indexing

The !! operator selects the ith index of a list. We results of testing !! from the prelude of Hugs were:



Conclusion: It takes time $O(n)$ to select the nth element of a list.

[[SLIDE]]

A more efficient list representation allowing faster indexing will be given next.

1.2 Random Access Lists – Okasaki '95

Would like to extend the usual lists with fast random accesses:

	lists	balanced trees	random access lists
Push (cons)	1	$\log n$	1
Head/Tail	1	$\log n$	1
Lookup/Update	n	$\log n$	$\log n$

By storing a list as a **balanced tree** all operations can be done in logarithmic time. To support the standard operations Head/Tail/Cons at the head of a list in constant time we need to do something different, i.e., combine the two solutions.

1.2.1 Skew binary numbers – Meyers '83

A **skew binary** number is a number

$$d_\ell, \ldots, d_2, d_1$$

where

- $d_i \in \{0, 1, 2\}$
- $d_j = 2 \Rightarrow \forall i < j : d_i = 0$

Example: 11001200

The number represented by $d_\ell, \ldots, d_2, d_1$ is

$$\sum_{i=1}^{\ell} d_i \cdot (2^i - 1)$$

FACT: Every number has a **unique** skew binary representation.

FACT: $2^{i+1} - 1 = 2 \cdot (2^i - 1) + 1$

Increment:

- Let j be given such that $d_j \neq 0, \forall i < j : d_i = 0$
- If $d_j = 1$ then $d_1 := d_1 + 1$, else $d_j = 0$ and $d_{j+1} := d_{j+1} + 1$

Decrement:

- Let j be given such that $d_j \neq 0, \forall i < j : d_i = 0$
- $d_j := d_j - 1$. If $j > 1$ then $d_{j-1} := 2$

List representation:

- Let a list be represented by a sequence of **complete binary trees** of size $2^i - 1$
- The sequence of trees correspond to the skew binary representation of the length of the list
- A preorder traversal of the trees reveals the stored list

Example: [1,2,3,4,5,6,7,8,9,10,11,12,13] =
Size $13_{10} = 120$ (skew-binary)

```
  1 -- 4 ------ 7
 2 3  5 6    8     9
          10 11 12 13
```

Tail:

```
2 - 3 -- 4 ------ 7
        5 6    8     9
            10 11 12 13
```

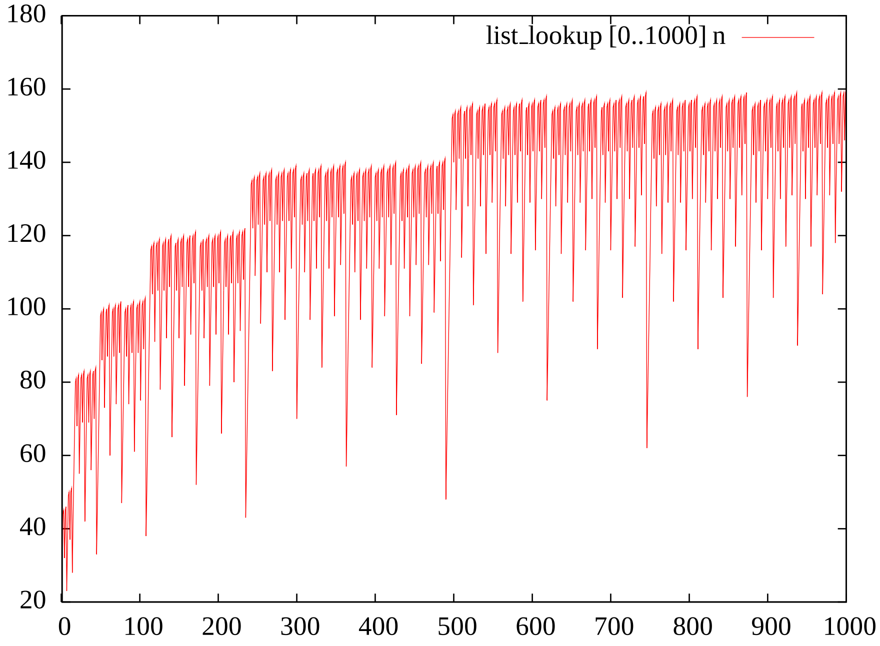
or Cons(0,.)

```
    0 -------- 7
  1   4      8     9
 2 3 5 6  10 11 12 13
```

— 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)
```



The graph shows the time for performing lookups to a list with $1001 = 511 + 255 + 127 + 63 + 31 + 2 * 7$ elements

Cris Okasaki has done some experimental work in Standard ML, and observed that in random access lists never perform worse than a factor of **two** compared to standard lists.

Note: **Finger search trees** could also be used to obtain the same asymptotic bounds, but it is non-trivial to adopt well-known finger search trees to a functional setting.

Note: Skew binary numbers will be used in the implementation of efficient priority queues.

1.3 Stacks

Stacks can be implemented as

— 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 = []
```

1.4 Queues

The semantics can be described by the following Haskell code:

— 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 = []
```

Bottleneck: Inject requires linear time, because of the linear time ++ operator!

A standard implementation which is efficient in the amortized sense is given by the code. The amortized time for each operation is $O(1)$. A queue is represented by a pair of lists (l, r) such that $l + +(\mathrm{reverse} r)$ is equal to the queue as a list.

— 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 = ([],[])
```

Notice that `pop` requires linear time when `reverse` is performed.

Example: [1,2,3,4,5,6,7] could be represented by the pair ([1,2,3],[7,6,5,4])

Amortized analysis:

$\Phi(Q) = |r|,$ if $Q = (l, r)$

$\Rightarrow$ `pop` and `inject` take amortized constant time.

In general a list is divided into a left and right part, and is stored as a pair consisting of the left part plus the reverse of the right part. The reverse of the right part allows efficient `inject` operations.

Unfortunately functional data structures are **persistent**, i.e., all old data structures are remembered. This implies that an expensive pop operation causing a list reversion can be repeated over and over again without doing any insertions.

To overcome this problem the expensive operation is spread over a sequence of operations by doing the list reversion and catenation **incrementally in advance** $\Rightarrow$ **Real time queue** (all operations take worst-case $O(1)$ 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
```

Approach: (l,r,incremental-work)

```
 (l,r,Nil)
 (l,[],Rev [] r)
 (l,[],Rev l1 [])       ,  l1=reverse r
 (l,[],Cat 0 [] l l1)
 (l,[],Cat s r1 [] l1) ,  r1=reverse l, s=|r1|
 (l,[],Cat 0 [] [] l2) ,  l2=(reverse r1)++l1=l++l1
 (l2,[],Nil)
```

It can be shown that applying progress 3 times is sufficient to guarantee $Q \neq \emptyset \Rightarrow l \neq \emptyset$.

1.5 Double ended queues - deques

— 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 = []
```

Notice that inject and eject take linear time!

1.5.1 Amortized solution

Again represent the list as a pair of lists (l, r), but now when a list l or r becomes empty, split the remaining non-empty list evenly among the two new l and r lists.

— 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 = ([],[])
```

Operations take amortized constant time. Follows from the invariant

$\Phi(Q) = ||l| - |r||$, if $Q = (l, r)$

This again only holds for single threaded computations.

1.5.2 Incremental rebuilding

As for queues! Details left as an exercise!

1.6 Binomial Queues – Vuillemin ’78

— 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
```

1.7 Skew Binomail Queues – Brodal, Okasaki 1996

— 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
```

1.8 Data structural bootstrapping – Brodal, Okasaki 1996

FindMin and Meld in $O(1)$ time. Elements are pairs (item, skew-BQ), where the skew-BQ is over elements (*not* items).

— 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
```

2 Lecture 2

2.1 Lazy functional data structures

2.2 Example: Search trees

ListToTree converts a list of integers into a search tree by a recursive procedure like quicksort.

— 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  ]
```

Experiment shows that member on a subtree not yet build is expensive: 1) First random mebers are expensive, 2) latter members become cheaper.

Intuitive: The first t random queries are spread uniformly, i.e. all nodes in the top $\log t$ levels are visited, but thereafter the searches are in disjoint subtrees. Cost of creating a node is proportional to the size of the subtree rooted at the node (distributing the elements left-and-right). Total cost $O(t \log n + n \log t)$, since

Top levels: $n * \log t$

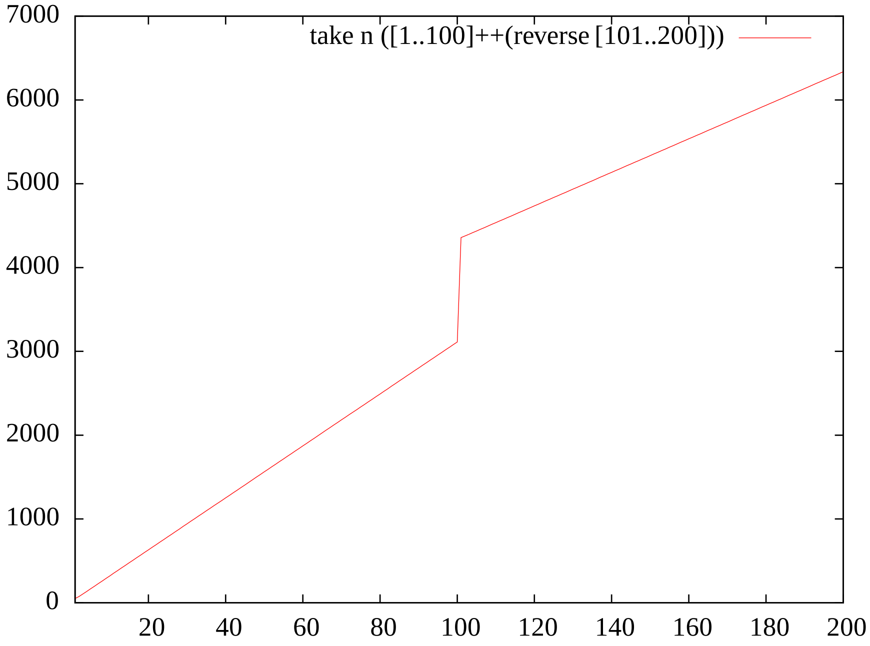
For each path: $t * n/t$

Searches: $t * \log n$

2.3 Example: Lazy list catenation

We consider the effect of lazy list catenation:

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



2.4 Amortized analysis reviewed

2.5 Queues

Consider the following code for a queue implementation based on representing a queue by a left half and a right half where the **left part always is the largest**.

— 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
```

The crucial point is that ++ is performed in a lazy fashion, i.e., the expensive reverse operation is postponed until all elements of `l` has been removed from the queue.

Theorem The above implementation supports all operations in amortized constant time.

2.6 Binomial queues

Analysis of the following code...

— 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
```

3 Lecture 3

3.1 Catenable lists

3.2 Strict implementation

3.3 Lazy implementation