# Programming in Haskell
# Aug–Nov 2015

## LECTURE 20

## OCTOBER 29, 2015

S P Suresh
Chennai Mathematical Institute

# Priority queues

- **Priority queue**: a queue, with each element having a **priority**
- Elements exit the queue by priority, not in the order they entered
- Think of me in the snack queue!
- Each element in a priority queue is a pair **(p,v)**, where **p** is the priority and **v** is the value
- Assume that priorities are integers

# Priority queues

- **Operations on priority queues**: `insert` and `delmax`
- `insert :: PriorityQueue a -> a -> PriorityQueue a`
- `delmax :: PriorityQueue a -> (a,PriorityQueue a)`

# Priority queue implementations

- **Unsorted lists**
  - `insert` – O(1) time, `delmax` – O(N) time
- **Sorted lists** – descending order of priority
  - `insert` – O(N) time, `delmax` – O(1) time
- **Balanced binary search trees**
  - `insert` – O(log N) time, `delmax` – O(log N) time (just go down the rightmost path till the end, and remove the node)

# Heaps

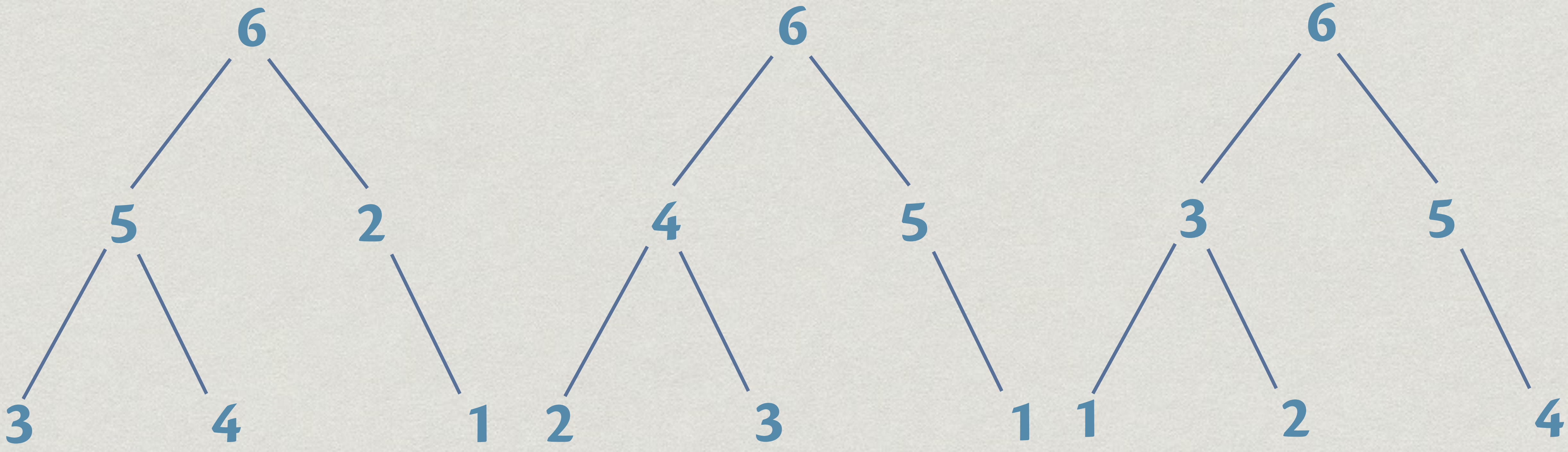
- A **heap** is another way to implement priority queues
- To determine the maximum, it is not necessary that all elements be sorted
- We need to keep track of the maximum
- Also the possible second maximum, to be installed as the new maximum after `delmax`
- The next maximum ...

# Heaps

* A **heap** is a binary tree satisfying the **heap** property
* `data Heap a = HNil | HNode a (Heap a) (Heap a)`
* **The heap property**: The value at a node is larger than the value at its two children
* **Heap**: A tree where every node satisfies the heap property

# Example heaps

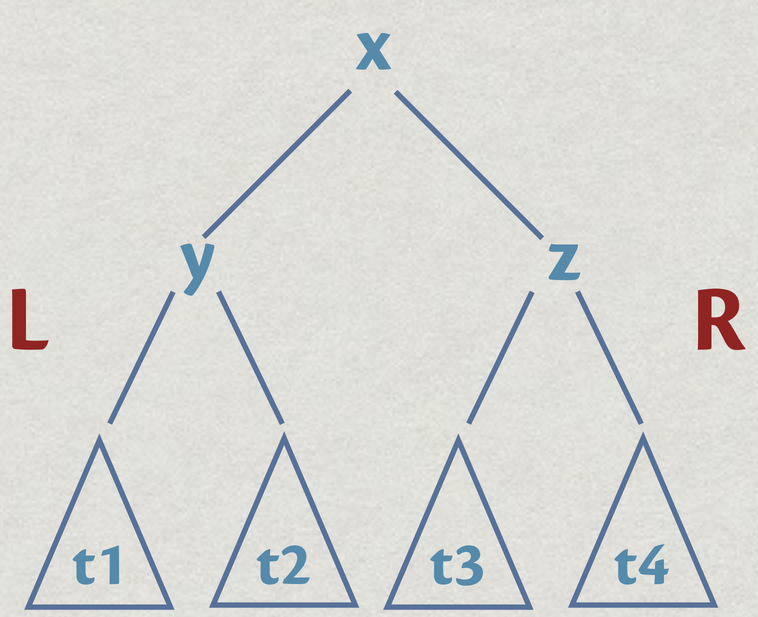
Three heaps with elements [1..6]




In a heap, the largest element is always at the root

# Repairing heaps

- Assume that **L** and **R** below, with roots **y** and **z**, are already heaps
- How do we ensure that the tree rooted at **x** is a heap




- If **x >= max y z**, all is okay, else swap **x** with **max y z**, say **z**
- Now heap property holds at the root
- **L** is undisturbed, but **R** might fail to be a heap
- Recursively repair **R**
- This process is called **sifting**

# Sifting

- ```haskell
  sift :: Ord a => Heap a -> Heap a
  sift HNil = HNil
  sift t@(HNode x HNil HNil) = t
  sift t@(HNode x (HNode y t1 t2) HNil)
    | x >= y     = t
    | otherwise = HNode y (sift (HNode x t1 t2) HNil
  sift t@(HNode x HNil (HNode z t3 t4))
    | x >= z     = t
    | otherwise = HNode z HNil (sift (HNode x t3 t4)
  sift (HNode x tl@(HNode y t1 t2) tr@(HNode z t3 t4))
    | x >= max y z = HNode x tl tr
    | y >= max x z = HNode y (sift (HNode x t1 t2)) tr
    | z >= max x y = HNode z tl (sift (HNode x t3 t4))
  ```

- Note the **as-patterns**
  ```

# Form a heap

- If the tree is balanced, there are at most log N recursive calls needed to sift, and the resulting heap is still balanced
- Start with a balanced tree, recursively **heapify** both subtrees, and then sift
- Procedure taking time $T(N) = 2T(N/2) + c.\log N$

# Form a heap

- $T(N) = c.\log N + 2T(N/2)$

- Letting $N = 2^m$,

$$
\begin{aligned}
T(N) &= cm + 2T(2^{m-1}) \\
&= cm + 2[c(m-1) + 2T(2^{m-2})] \\
&= cm + 2c(m-1) + 2^2[c(m-2) + 2T(2^{m-3})] \\
&= \dots \\
&= c[\ m + 2(m-1) + 2^2(m-2) + 2^3(m-3) + \dots + 2^{m-1}(m-m+1)] \\
2T(N) &= c[\qquad 2m \qquad + 2^2(m-1) + 2^3(m-2) + \dots + 2^{m-1}(m-m+2) + 2^m(m-m+1)] \\
T(N) &= c[-m + 2 \qquad + 2^2 \qquad + 2^3 \qquad + \dots + 2^{m-1} \qquad + 2^m] \\
&= c[2N - \log N] \\
&= O(N)
\end{aligned}
$$

# Form a heap

- ```
  heapify :: Ord a => AVLTree a -> Heap a
  heapify HNil                = HNil
  heapify (HNode tl x h tr) = sift (HNode x
                                      (heapify tl)
                                      (heapify tr))
  ```

- ```
  listToHeap :: Ord a => [a] -> Heap a
  listToHeap = heapify . mkAVLTree
  ```

- `listToHeap` works in O(N) time

- `mkAVLTree` will only produce a **balanced tree**, **not a balanced search tree**, since the input need not be sorted

# List a heap in sorted order

- ```
  horder :: (HTree a) -> [a]
  horder HNil = []
  horder (HTree x h1 h2) = x:(merge (horder h1)
                                    (horder h2))
  ```

- The output is in descending order – apply `reverse` to the output

- `horder` takes O(N log N) time – cannot do better

- `heapsort = horder . heapify . mkAVLTree`
  ```

# Insert and deletemax

- Efficient computation of union of two heaps of size $M$ and $N$ in time $O(\log M + \log N)$
- Use **heap union** to **insert** and **delete maximum**
- `insert t x = union t (HNode x HNil HNil)`
- `delmax (HNode x t1 t2) = (x, union t1 t2)`
- Both `insert` and `delmax` run in $O(\log N)$ time

# Rightist heaps

- To efficiently implement `union`, we form **rightist heaps**
- These are heaps where at every node, the right subtree has at least as many nodes as the left subtree
- As with AVL trees, we assume that we store the size along with each node of a heap

# Rightist heaps

- One can easily convert any heap to a rightist heap

- ```
  realign :: Heap a -> Heap a
  realign HNil = HNil
  realign (HNode x t1 t2)
      | size t2 < size t1 = HNode x t2 t1
      | otherwise         = HNode x t1 t2
  ```

- Recall that heaps do not impose an order between left and right subtrees

- Recall that `size` is constant time if stored in each node
  ```

# Rightist heaps

- One can easily convert any heap to a rightist heap

- ```
  mkRightist :: Heap a -> Heap a
  mkRightist HNil                = HNil
  mkRightist (HNode x t1 t2) = realign (HNode x
                                 (mkRightist t1)
                                 (mkRightist t2)
  ```

- Takes time O(N)

- The **left spine** (the leftmost path) of a rightist heap is of length at most log N
  ```

# Left spine of a rightist heap

- The **left spine** (the leftmost path) of a rightist heap is of length at most $\log N$

- Assume a heap **h** of size $N = p + q + 1$, where $p$ and $q$ are sizes of the left and right subtrees, **h1** and **h2**

- $\text{lls}(h) = 1 + \text{lls}(h1)$
  $\leq 1 + \log p$
  $= \log 2 + \log p$
  $= \log (2p)$
  $\leq \log (p + q)$ – since **h** is rightist, $p \leq q$
  $\leq \log (1+p+q)$
  $= \log N$

# Union of rightist heaps

- ```
  union :: (Heap a) -> (Heap a) -> (Heap a)
  union t HNil = t
  union HNil t = t
  union (HNode x t1 t2) (HNode y t3 t4)
      | x < y     =
            realign (HNode y (union (HNode x t1 t2) t3) t4)
      | otherwise =
            realign (HNode x (union t1 (HNode y t3 t4)) t2)
  ```
- Each step of `union` makes one step down the left spine of one of the heaps
- But the left spine is bounded by log (size(heap))
- Thus overall there are at most O(log M + log N) steps required
  ```

# Summary

- **Priority queue** data structure
- **Heaps**: forming heaps using **heapify**
- Efficient **insert** and **deletemax** using **union**
- Efficient union using **rightist heaps**

# Recursion and efficiency

- Consider the function `fib`, which computes the n-th Fibonacci number F(n)

- ```
  fib 0 = 1
  fib 1 = 1
  fib n = fib (n-1) + fib (n-2)
  ```

- Lots of recursive calls, computing the same value over and over again

- Computes F(n) in unary, in effect
  ```

# Recursion and efficiency

- Let $G(n)$ be the number of recursive calls to `fib 0` in the computation of `fib n`, for $n > 1$
- $G(2) = 1$ – one call to `fib 0`
  $G(3) = 1$ – one call to `fib 0`
- **Claim:** $G(n) = F(n-2)$
  **Proof:**
  True for $n = 2$ and $n = 3$.
  For $n > 3$, $G(n) = G(n-1) + G(n-2)$, since there is one call to `fib (n-1)` and one to `fib (n-2)`.
  But $G(n-1) = F(n-3)$ and $G(n-2) = F(n-4)$, by induction hypothesis.
  Thus $G(n) = F(n-3) + F(n-4) = F(n-2)$.

# Recursion and efficiency

- How do we fix this?
- Store the computed values (**in an array**) and use them
- In a language like C, we would have this code:

```
int fibs[n];
fibs[0] = fibs[1] = 1; i = 2;
while (i <= n) {
    fibs[i] = fibs[i-1] + fibs[i-2];
    i++;
}
return fibs[n];
```

# Recursion and efficiency

- We can simplify this even more, since only the last two elements of the `fibs` array are needed

- ```
  int prev = 1, curr = 1, i = 2;
  int temp;
  while (i <= n) {
      temp = prev;
      prev = curr;
      curr = temp + prev;
      i++;
  }
  return curr;
  ```
  ```

# Recursion and efficiency

- Linear-time Fibonacci in Haskell. **Laziness to the rescue!**

- ```
  fastfib n = fibs !! n
  fibs :: [Integer]
  fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
  ```

- ```
  1:1:zipWith (+) [1,1,...] [1,...]
  ⇛ 1:1:(1+1):zipWith (+) [1,2,...] [2,...]
  ⇛ 1:1:2:(1+2):zipWith (+) [2,3,...] [3,...]
  ⇛ 1:1:2:3:(2+3):zipWith (+) [3,5...] [5,...]
  ⇛ 1:1:2:3:5:...
  ⇛ ...
  ```

# Another example: `lcss`

- Given two strings **str1** and **str2**, find the **length** of the **longest common subsequence** of **str1** and **str2**
- ```
  lcss "agcat" "gact" = 3
             – "gat" is the subsequence
  lcss "abracadabra" "bacarrat" = 6
          – "bacara" is the subsequence
  ```
  ```

# Another example: `lcss`

- ```
  lcss ""      _      = 0
  lcss _       ""     = 0
  lcss (c:cs) (d:ds)
      | c == d        = 1 + lcss cs ds
      | otherwise     = max (lcss (c:cs) ds)
                            (lcss cs (d:ds))
  ```
- `lcss cs ds` takes time $>= 2^n$, when cs and ds are of length n
- Similar problem to `fib`, same recursive call made multiple times
- **Store the computed values for efficiency**
  ```

# Linear-time sort

- Given a list of $n$ integers, each between 0 and 9999, sort the list
- Easy to do with arrays
- Count the number of occurrences of each $j \in \{0, ..., 9999\}$ in the list, storing in an array counts
- Output count[j] copies of j, j ranging from 0 to 9999

# Linear-time sort

- ```
    // Input - int arr[n];
  int counts[10000], output[n];
  for (j = 0; j < 10000; j++)
      counts[j] = 0;
  for (i = 0; i < n; i++)
      counts[arr[i]]++;
  last = 0;
  for (j = 0; j < 10000; j++)
      for (i = 0; i < counts[j]; i++)
          output[last] = j, last++;
  ```
- This works in time O(n+10000) time
  ```

# Arrays in Haskell

* Lists store a collection of elements
* Accessing the i-th element takes i steps
* Would be useful to access any element in constant time
* **Arrays** in Haskell offer this feature
* The module **Data.Array** has to be imported to use arrays

# Arrays in Haskell

* `import Data.Array`
  `myArray :: Array Int Char`

* The **indices** of the array come from `Int`
  The **values** stored in the array come from `Char`

* `myArray = listArray (0,2) ['a','b','c']`

| Index | 0 | 1 | 2 |
|---|---|---|---|
| Value | 'a' | 'b' | 'c' |

# Creating arrays: listArray

- ```
  listArray ::
          Ix i => (i,i) -> [e] -> Array i e
  ```
- `Ix` is the class of all **index** types, those that can be used as indices in arrays
  - If `Ix a`, `x` and `y` are of type `a` and `x < y`, then **the range of values between `x` and `y` is defined and finite**
- `Ix` includes `Int`, `Char`, `(Int,Int)`, `(Int,Int,Char)` etc. but not `Float` or `[Int]`
- The first argument of `listArray` specifies the smallest and largest index of the array
- The second argument is the list of values to be stored in the array
  ```

# Creating arrays: listArray

- `listArray (1,1) [100..199]`
  `array (1,1) [(1,100)]`

- `listArray ('m','p') [0,2..]`
  `array ('m','p') [('m',0),('n',2),('o',4),('p',6)]`

- `listArray ('b','a') [1..]`
  `array ('b','a') []`

- `listArray (0,4) [100..]`
  `array (0,4) [(0,100),(1,101),(2,102),(3,103),(4,104)]`

- `listArray (1,3) ['a','b']`
  `array (1,3) [(1,'a'),(2,'b'),(3,*** Exception: (Array.!): undefined array element`

# Creating arrays: listArray

- The value at index `i` of array `arr` is accessed using `arr!i` (unlike `!!` for list access)
- `arr!i` returns an exception if no value has been defined for index `i`
- `myArr = listArray (1,3) ['a','b','c']`
- ```
  myArr ! 4
  *** Exception: Ix{Integer}.index: Index (4) out of range ((1,3))
  ```
  ```

# Creating arrays: listArray

- Haskell arrays are **lazy**: the whole array need not be defined before some elements are accessed
- For example, we can fill in locations `0` and `1` of `arr`, and define `arr!i` in terms of `arr!(i-1)` and `arr!(i-2)`, for `i >= 2`
- `listArray` takes time proportional to the range of indices

# Fibonacci using arrays

- ```haskell
  import Data.Array
  fib :: Int -> Integer
  fib n = fibA!n
    where
      fibA :: Array Int Integer
      fibA = listArray (0,n) [f i | i <-[0..n]]
      f 0 = 1
      f 1 = 1
      f i = fibA!(i-1) + fibA!(i-2)
  ```

- The `fibA` array is used even before it is completely defined, thanks to Haskell's laziness

- Works in O(n) time
  ```

# Summary

* Recursive programs can sometimes be very inefficient, recomputing the same value again and again
* **Memoization** is a technique that renders this process efficient, by storing values the first time they are computed
* Haskell **arrays** provides an efficient implementation of these techniques
* Important tool to keep in our arsenal