Lecture
Data structures (DAT037)

Nils Anders Danielsson

2014-11-14

Today

- Binary trees.
- Priority queues.
    - Binary heaps.
    - Leftist heaps.

Binary trees

Binary trees

- A binary tree is either empty or a node.
- A node may contain a value.
- A node has two subtrees (possibly empty):
  A left one and a right one.
- Terminology:
  - Parent, child, sibling, grandchild etc.
  - Root, leaf.

Binary trees

One representation:

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

Example:

```
tree :: Tree Integer
tree = Node (Node Empty 2 Empty)
            1
            (Node Empty 3 (Node Empty 5 Empty))
```

Binary trees

Another representation:

```
class Tree<A> {
  class TreeNode {
    A        contents;
    TreeNode left;   // null if left child is missing.
    TreeNode right;  // null if right child is missing.
  }

  TreeNode root;  // null if tree is empty.
}
```

Binary trees

Height:

- Empty trees have height -1.
- Otherwise:
  Number of steps from root to deepest leaf.

Binary trees

Height:

- Empty trees have height -1.
- Otherwise:
  Number of steps from root to deepest leaf.

```
height :: Tree a -> Integer
height Empty        = -1
height (Node l _ r) = 1 + max (height l) (height r)
```

Binary trees

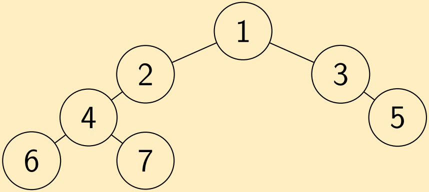
Height:

- Empty trees have height -1.
- Otherwise:
  Number of steps from root to deepest leaf.

```
int height(TreeNode n) {
  if (n == null) {
    return -1;
  } else {
    return 1 + Math.max(height(n.left),
                        height(n.right));
  }
}
```

```
int s(TreeNode n) {
  if (n == null) {
    return 0;
  } else {
    return 1 + s(n.left) + s(n.right);
  }
}
```

What is the result of applying s to the root of the following tree?



Priority queues

Priority queues

Queues where every element has a certain priority.

Interface (example):

- Constructor for empty queue.
- `insert`: Inserts element.
- `find-min`: Returns minimum element.
- `delete-min`: Deletes minimum element.
- `decrease-key`: Decreases priority.
- `merge`: Merges two queues.

Priority queues

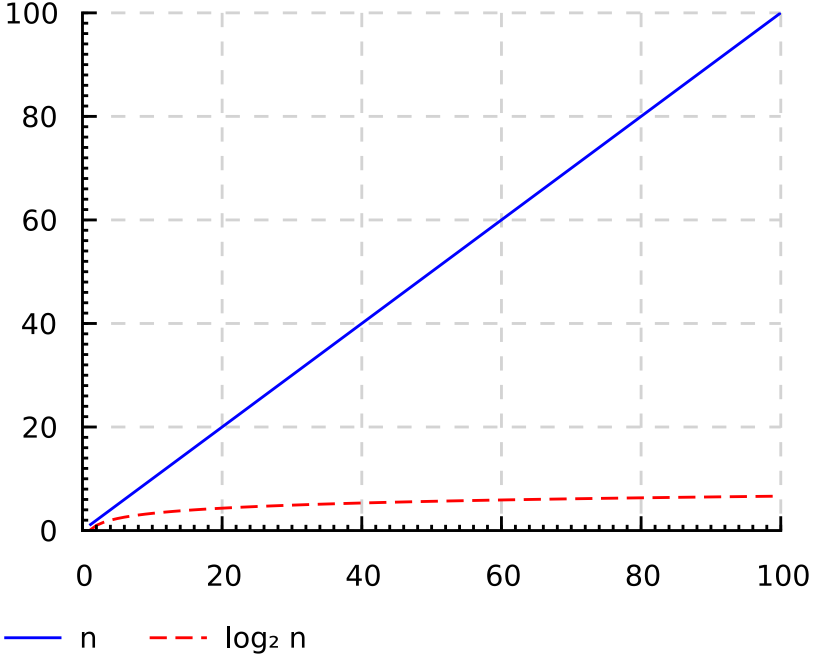
Some applications:

- Scheduling of processes.
- Sorting.
- Dijkstra's algorithm (3rd assignment).

2nd assignment: Implement priority queue.

If you implement the priority queue ADT with lists, what is the worst case time complexity of `insert` and `delete-min`?

- `insert`: $\Theta(1)$, `delete-min`: $\Theta(1)$.
- `insert`: $\Theta(1)$, `delete-min`: $\Theta(n)$.
- `insert`: $\Theta(n)$, `delete-min`: $\Theta(1)$.
- `insert`: $\Theta(n)$, `delete-min`: $\Theta(n)$.



100
80
60
40
20
0
0
20
40
60
80
100
n
log₂ n

Binary heaps

Binary heaps

Heap-ordered complete binary trees.

Heap-ordered

Every node is smaller than or equal to its children.

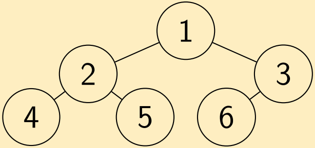
Complete binary tree

As low as possible, every level completely filled except possibly the last one, which is filled from the left.

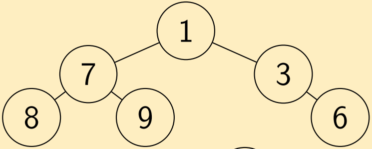
A binary heap of size n has height $\Theta(\log n)$.

Identify the binary heaps.

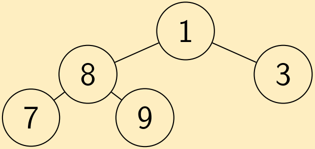
A:



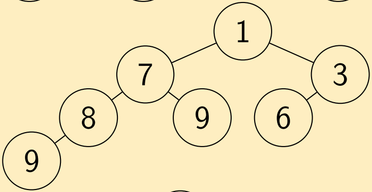
B:



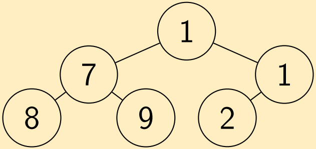
C:



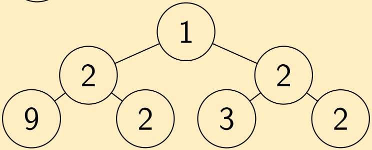
D:



E:



F:



Implementation of binary heaps

- Empty queue: Empty tree.
- `find-min`: Return the root.
- `insert`: Insert at the end. Percolate up.
- `delete-min`: Remove the root.
  Move the final element to the top.
  Percolate down.
- `decrease-key`: Change priority, percolate up (or down for `increase-key`).

Percolate up/down until the tree is heap-ordered.

Time complexity

If the nodes can be found quickly:

- Empty queue: $\Theta(1)$.
- `find-min`: $\Theta(1)$.
- `insert`: $O(\log n)$ (maybe amortised).
- `delete-min`: $O(\log n)$ (maybe amortised).
- `decrease-key`: $O(\log n)$.

(Assuming that comparisons take constant time.)

Most nodes are located “close” to the leaves.
Average time complexity of `insertion`: $O(1)$.

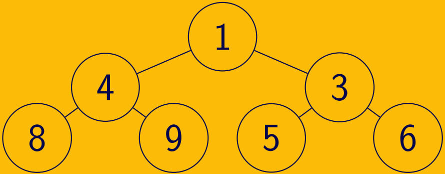
Implementation of binary heaps

One can represent the tree using an array (2nd assignment).

- The root at position 1.
- The last element at position n.
- The first empty cell at position $n+1$.
- Node i's left child: $2i$.
- Node i's right child: $2i+1$.
- Node i's parent ($i > 1$): $\lfloor i/2 \rfloor$.

What is the result of applying `delete-min`

to



?

A:

3	4	5	6	8	9

B:

3	5	6	4	8	9

C:

3	4	8	9	5	6

D:

3	4	5	8	9	6

`decrease-key`

`decrease-key`:
How can the node be located quickly?

Can use extra data structure.

Example: Hash table.

Leftist heaps

`merge`

- Merging two binary heaps, implemented using arrays, seems to be inefficient.
- With *leftist heaps*: $O(\log n)$ (assuming $O(1)$ comparisons).

Leftist heaps

- Heap-ordered (pointer-based) binary trees, with extra invariant (later).
- Basic operation: `merge`.
- Easy to implement `insert`, `delete-min` in terms of `merge`.

Leftist heaps, first attempt

```
-- Invariant for Node x l r:
-- * x is smaller than or equal to
--   all elements in l and r.
data PriorityQueue a
  = Empty
  | Node a (PriorityQueue a) (PriorityQueue a)

empty :: PriorityQueue a
empty = Empty

isEmpty :: PriorityQueue a -> Bool
isEmpty Empty        = True
isEmpty (Node _ _ _) = False
```

Leftist heaps, first attempt

```
insert :: Ord a =>
          a -> PriorityQueue a -> PriorityQueue a
insert x t = merge (Node x Empty Empty) t

-- Precondition: The queue must not be empty.
findMin :: PriorityQueue a -> a
findMin Empty        = error "findMin: Empty queue."
findMin (Node x _ _) = x

-- Precondition: The queue must not be empty.
deleteMin :: Ord a =>
             PriorityQueue a -> PriorityQueue a
deleteMin Empty        = error "deleteMin: Empty."
deleteMin (Node _ l r) = merge l r
```

Leftist heaps, first attempt

`merge` implemented by going down right spines:

```
merge :: Ord a =>
         PriorityQueue a -> PriorityQueue a ->
         PriorityQueue a
merge Empty              r                   = r
merge l                  Empty               = l
merge l@(Node xl ll rl) r@(Node xr lr rr) =
  if xl <= xr then
    Node xl ll (merge rl r)
  else
    Node xr lr (merge l rr)
```

What is the worst case time complexity of `merge`? Assume that both queues have n elements, and that comparisons take constant time.

- $\Theta(1)$.
- $\Theta(\log n)$.
- $\Theta(n)$.
- $\Theta(n \log n)$.
- $\Theta(n^2)$.
- $\Theta(n^2 \log n)$.

Leftist heaps

- Trees may be very unbalanced:
  no left children, only right children.
- This makes `merge` linear (in the worst case).
- Solution: Ensure right spine is short.

Leftist heaps

Null path length

- -1 for empty trees.
- Otherwise: Number of steps from root to closest node with at most one child.

```
npl :: PriorityQueue a -> Integer
npl Empty          = -1
npl (Node _ l r) = 1 + min (npl l) (npl r)
```

Leftist heaps

Null path length

- -1 for empty trees.
- Otherwise: Number of steps from root to closest node with at most one child.

```
height :: PriorityQueue a -> Integer
height Empty          = -1
height (Node _ l r) = 1 + max (height l) (height r)
```

Leftist heaps

Leftist

For `Node x l r`, `npl l` $\geq$ `npl r`.

This implies:

- Number of nodes on right spine: $1 +$ `npl t`.
- $1 +$ `npl t` is $O(\log n)$, where n is the size of `t`.

Thus the right spine is short.

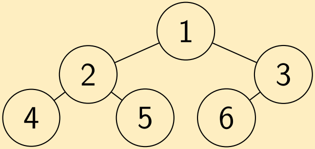
Leftist heaps

Leftist heap invariants

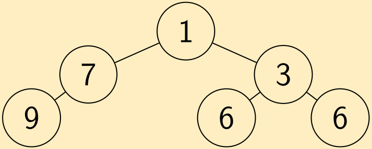
1. Heap-ordered.
2. Leftist.

Identify the leftist heaps.

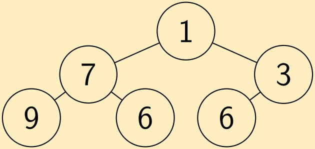
A:



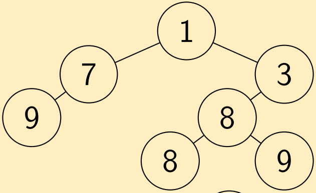
B:



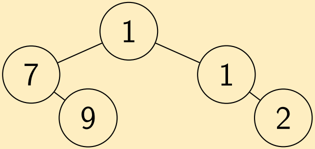
C:



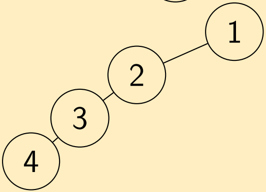
D:



E:



F:



Leftist heaps

- The previous implementation of `merge` sometimes breaks the leftist invariant.
- Simple fix: When necessary, swap the left and right subtrees.

Leftist heaps

Old code:

```
merge :: Ord a =>
         PriorityQueue a -> PriorityQueue a ->
         PriorityQueue a
merge Empty                r                 = r
merge l                    Empty             = l
merge l@(Node xl ll rl) r@(Node xr lr rr) =
  if xl <= xr then
    Node xl ll (merge rl r)
  else
    Node xr lr (merge l rr)
```

Leftist heaps

New code:

```
merge :: Ord a =>
         PriorityQueue a -> PriorityQueue a ->
         PriorityQueue a
merge Empty              r                  = r
merge l                  Empty              = l
merge l@(Node xl ll rl) r@(Node xr lr rr) =
  if xl <= xr then
    node xl ll (merge rl r)
  else
    node xr lr (merge l rr)
```

Leftist heaps

Smart constructor used to enforce leftist invariant:

```
-- Precondition for node x l r:
-- * x is smaller than or equal to
--   all elements in l and r.
node :: a -> PriorityQueue a -> PriorityQueue a ->
        PriorityQueue a
node x l r =
  if npl l >= npl r then
    Node x l r
  else
    Node x r l
```

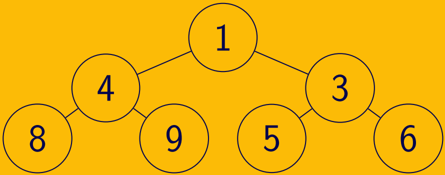
Leftist heaps

One final tweak:

- The recursive calculation of `npl` is unnecessary.
- Fix: Store `npl` values in nodes.

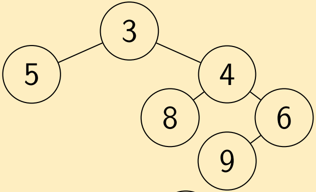
```
-- Invariants: ...
data PriorityQueue a
  = Empty
  | Node Integer a (PriorityQueue a)
                   (PriorityQueue a)

npl :: PriorityQueue a -> Integer
npl Empty          = -1
npl (Node n _ _ _) = n
```

What is the result of applying `deleteMin` to

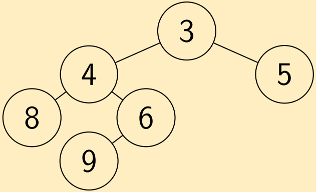


?

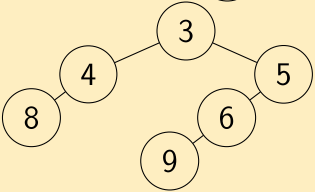
A:



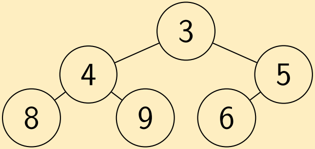
B:



C:



D:



Time complexities

	Binary heap	Leftist heap (immutable)
`find-min`	$O(1)$	$O(1)$
`delete-min`	$O(\log n)$	$O(\log n)$
`insert`	$O(1)$ (average)	$O(\log n)$
`decrease-key`	$O(\log n)$	$O(n)$
`merge`	$O(n)$	$O(\log n)$

(Assuming that comparisons take constant time.)

Other priority queue data structures

Comparison on Wikipedia.

Summary

- Binary trees.
- Priority queues.
  - Binary heaps.
  - Leftist heaps.

Next time:

- Hash tables.