Binomial Queues

Andrew W. Appel, 2016

Several pictures & captions from

Robert Sedgewick, *Algorithms 3rd Edition*

These slides are best viewed in your PDF viewer in whole-page (page-at-a-time) mode, not scrolling mode.

Priority Queues

Operations: (there is a total order on the element type)

Create an empty queue

Insert an element into the queue

Delete-max: delete and return the largest element

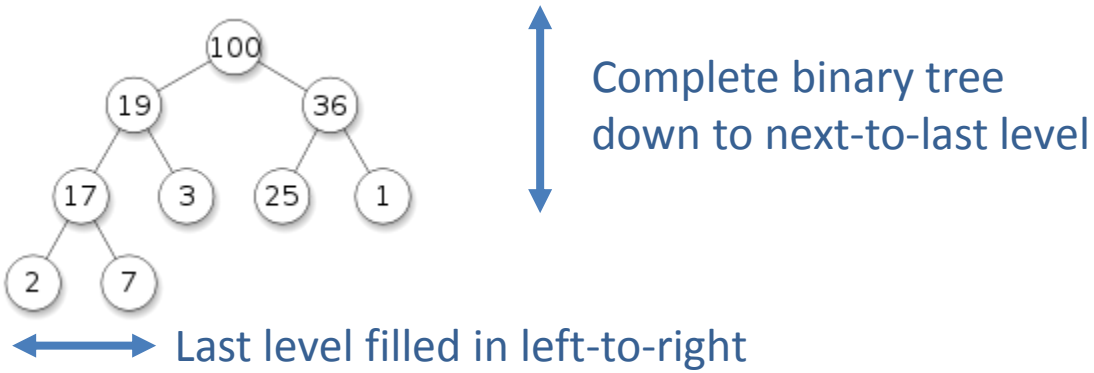
Join: merge two queues

Persistence: save the state of a queue, or restore to saved state

Binary Heap

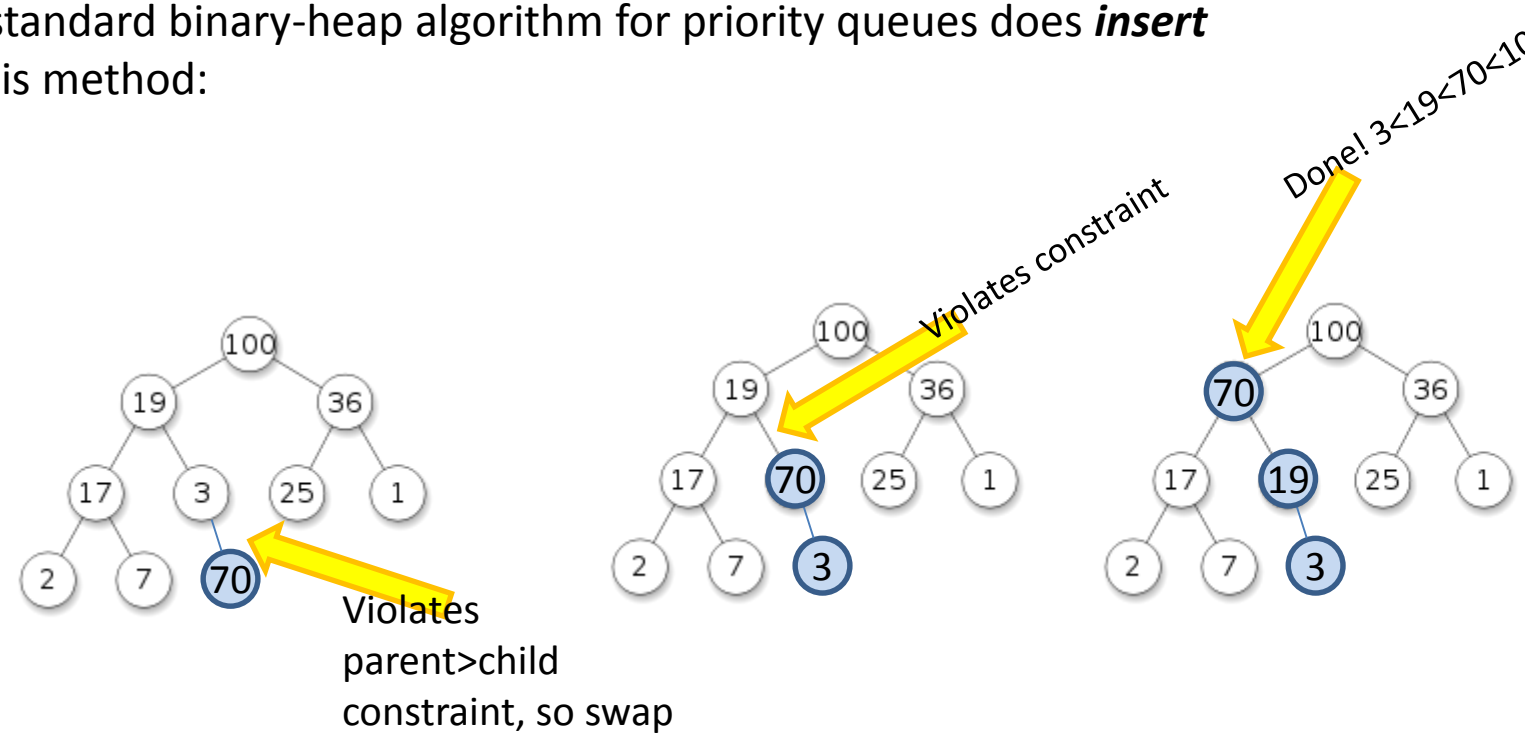
A *binary heap* is a tree with these properties:

- The *key* at each node is greater than all the keys in the subtrees of that node
- Each node has at most 2 children
- The tree is *near complete*:



Binary Heap

The standard binary-heap algorithm for priority queues does ***insert*** by this method:



This takes worst-case log(N) time, since the height of a nearly full binary tree with N nodes is log(N). Any educated computer scientist should know how this works.

We will not be using “standard” binary heaps, but binomial heaps; let’s file this away and move on.

Implementation choices

The standard implementation of a priority queue is a *binary heap*, implemented in an array. https://en.wikipedia.org/wiki/Binary_heap
It has efficient **insert** and **delete-max** but other operations are linear time.

	Heap in array			
Create	O(N)			
Insert	O(log N)			
Delete-max	O(log N)			
Join	O(N)			
Persistence	O(N)			

Implementation choices

One can also use a *balanced binary search tree*, such as a *red-black tree.*
https://en.wikipedia.org/wiki/Red-black_tree
A conventional imperative implementation, where insert and delete-max modify the tree destructively, requires tree-copying for persistence.
To join two trees, insert each node of the smaller tree into the bigger one; still O(N) if the trees are about equal in size.

	Heap in array	**Red-black tree (Imp)**		
Create	O(N)	O(1)		
Insert	O(log N)	O(log N)		
Delete-max	O(log N)	O(log N)		
Join	O(N)	O(N)		
Persistence	O(N)	O(N)		

Implementation choices

A pure-functional implementation of a balanced search tree,
with nondestructive update, gets persistence "for free,"
since the **insert** or **delete-max** does not destroy the old tree value.

	Heap in array	Red-black tree (Imp)	Red-black tree (Fun)	
Create	O(N)	O(1)	O(1)	
Insert	O(log N)	O(log N)	O(log N)	
Delete-max	O(log N)	O(log N)	O(log N)	
Join	O(N)	O(N)	O(N)	
Persistence	O(N)	O(N)	O(1)	

Implementation choices

Binomial queues, invented by Jean Vuillemin in 1978, allow all of these operations (including **join**) in O(log N) time. The *imperative* implementation has cost O(N) for persistence, versus O(1) for the *functional* implementation. The explanation you'll see in the next few pages can apply to either implementation.

	Heap in array	Red-black tree (Imp)	Red-black tree (Fun)	Binomial Queue
Create	O(N)	O(1)	O(1)	O(1)
Insert	O(log N)	O(log N)	O(log N)	O(log N)
Delete-max	O(log N)	O(log N)	O(log N)	O(log N)
Join	O(N)	O(N)	O(N)	O(log N)
Persistence	O(N)	O(N)	O(1)	O(1)

The textbook

A good explanation of the Binomial Queue data structure is in Robert Sedgewick's book, *Algorithms Third Edition,* published by Addison-Wesley.

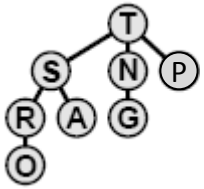
Professor Sedgewick and his publisher have given permission for free access to this section of the book, at

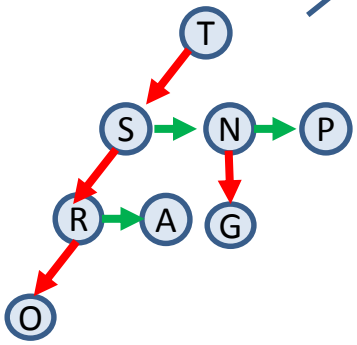
http://www.cs.princeton.edu/~appel/BQ.pdf

In the rest of these slides, the figures and captions labeled "Fig. 9.xx" are from Sedgewick's book.

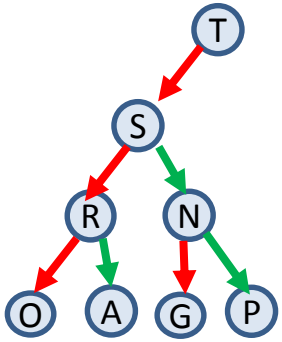
Representing N-ary trees as 2-ary trees

Here's a heap.
Notice that each node is greater
(in alphabetical order)
than its children.
But it's not a binary heap,
i.e. some nodes have >2 children, and it's
not "complete" at the next-to-last level. That's OK.

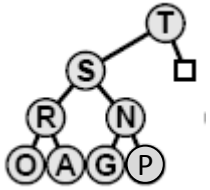




You can always represent an N-ary tree as a binary tree, using the "first-child", ↙ "next-child" → representation.

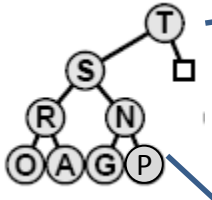


Same tree again, just tilted



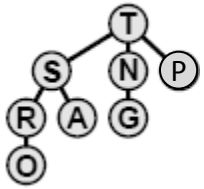
Left-heap-order invariant

In this 2-ary-tree representation of an N-ary heap, the invariant is, "the key at each node is greater than all the keys in its left subtree."



Notice that N⊀P, but that's OK, N<P not required because P is the right child of N, not the left child.

In this N-ary heap, the heap invariant is, "the key at each node is greater than the key at each of the node's children"

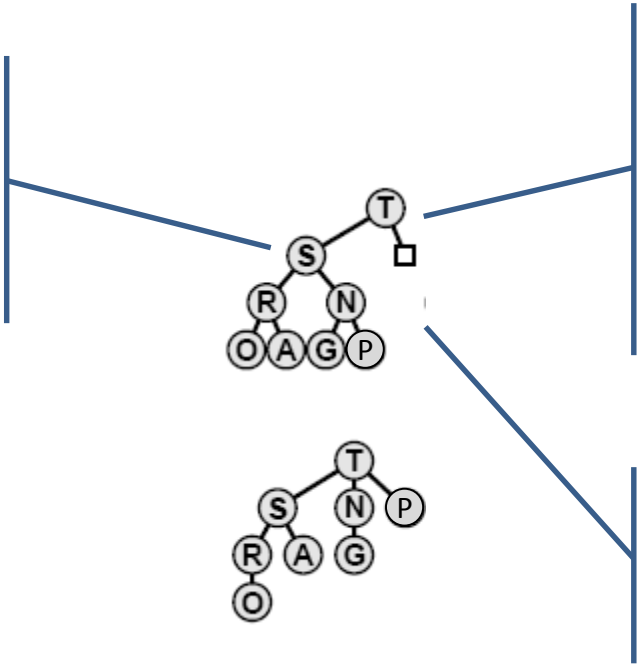


?

Convince yourself that the N-ary-heap invariant *does correspond* to the left-heap-order invariant on the 2-ary-tree!

Left-leaning heap

We call this a
“left-heap-ordered”
tree, with its invariant,
“the key at each node >
all the keys in its left subtree”



The left child of the root is a
complete binary tree;
the right child is empty.
Therefore the tree has 2^k nodes,
where k is the height of the
left subtree.

Therefore we call this a
“left-heap-ordered
power-of-2 heap”

Binomial Queues

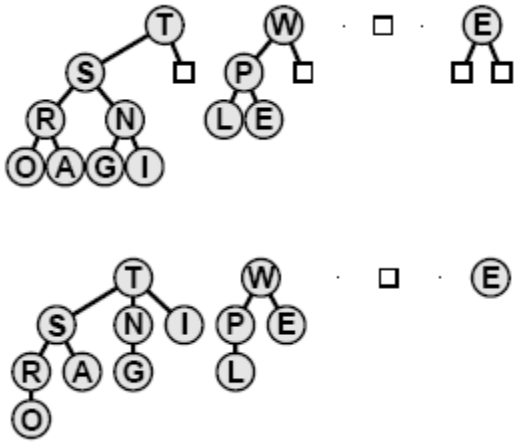


Figure 9.15
A binomial queue of size 13

A binomial queue of size N is a list of left-heap-ordered power-of-2 heaps, one for each bit in the binary representation of N. Thus, a binomial queue of size 13 = 1101_2 consists of an 8-heap, a 4-heap, and a 1-heap. Shown here are the left-heap-ordered power-of-2 heap representation (top) and the heap-ordered binomial-tree representation (bottom) of the same binomial queue.

Well, actually, not a list of heaps; a list of (option(heap)). The heap may be present or absent at each position.

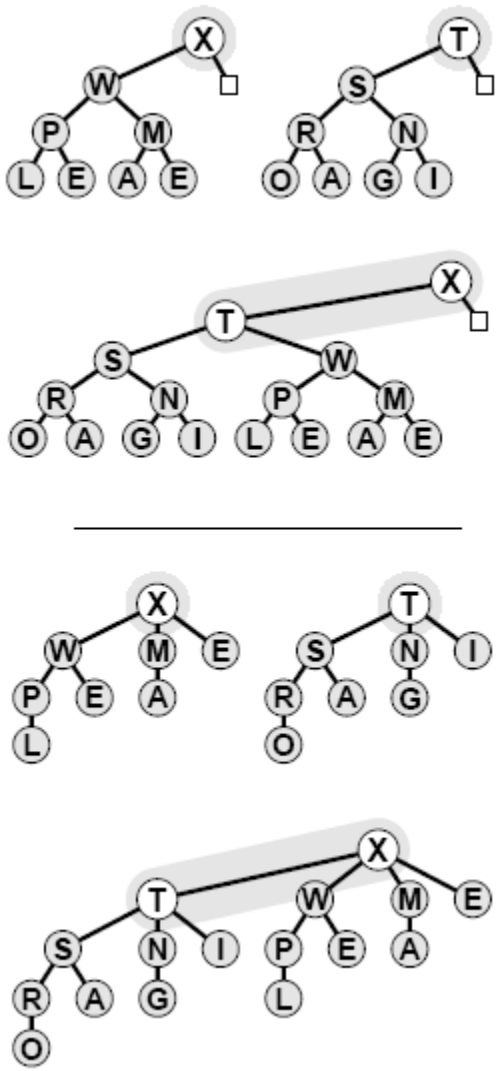


Figure 9.16
Joining of two equal-sized power-of-2 heaps.

We join two power-of-two heaps (top) *by putting the larger of the roots at the root, with that root's (left) subtree as the right subtree of the other original root. If the operands have 2^n nodes, the result has 2^{n+1} nodes. If the operands are left-heap ordered, then so is the result, with the largest key at the root. The heap-ordered binomial-tree representation of the same operation is shown below the line.*

Time: O(1), constant time (only the white-colored nodes Ⓧ Ⓣ are examined or touched).

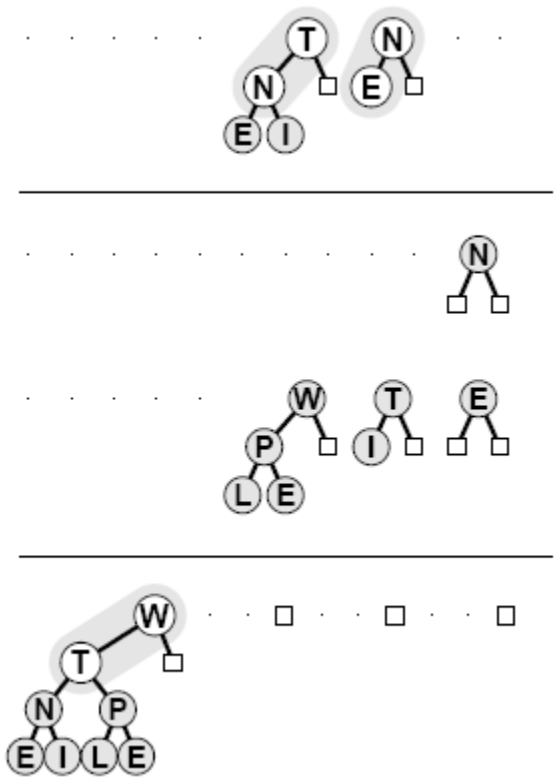


Figure 9.17
Insertion of a new element into a binomial queue

Adding an element to a binomial queue of seven nodes is analogous to performing the binary addition $111_2 + 1 = 1000_2$, with carries at each bit. The result is the binomial queue at the bottom, with an 8-heap and null 4-, 2-, and 1-heaps.

```
      1  1  0      <-- carry bits
         1         operand x
      1  1  1      operand y
   ---------
   1  0  0  0      <-- x+y
```

Time: log N, because that is (about) the length of the list of option(heap) if there are N nodes (total) in all the heaps in the list.

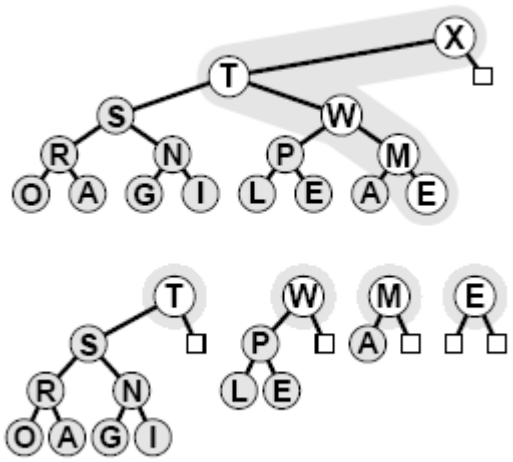


Figure 9.18
Removal of the maximum in a power-of-2 heap

Taking away the root gives a forest of power-of-2 heaps, all left-heap ordered, with roots from the right spine of the tree.

Time: log N, which is the length of the "right spine"

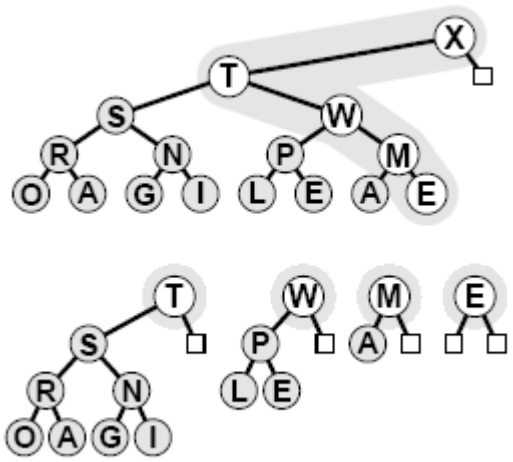


Figure 9.18
Removal of the maximum in a power-of-2 heap

Taking away the root gives a forest of power-of-2 heaps, all left-heap ordered, with roots from the right spine of the tree. This operation leads to a way to remove the maximum element from a binomial queue: Take away the root of the power-of-2 heap that contains the largest element, then use the join operation to merge the resulting binomial queue with remaining power-of-2 heaps in the original binomial queue.

Time: log N, which is the length of the "right spine;" assuming that "join" can also be done in log N . . .

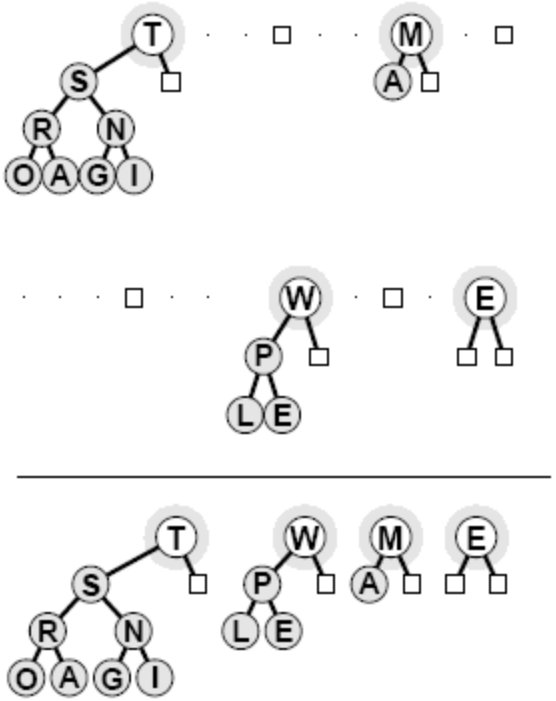


Figure 9.19
Joining of two binomial queues (no carry)

When two binomial queues to be joined do not have any power-of-2 heaps of the same size, the join operation is a simple merge. Doing this operation is analogous to adding two binary numbers without ever encountering $1 + 1$ *(no carry). Here, a binomial queue of 10 nodes is merged with one of 5 nodes to make one of 15 nodes, corresponding to* $1010_2 + 0101_2 = 1111_2$.

Time: log N, because that is (about) the length of the list of option(heap) if there are N nodes (total) in all the heaps in the list.

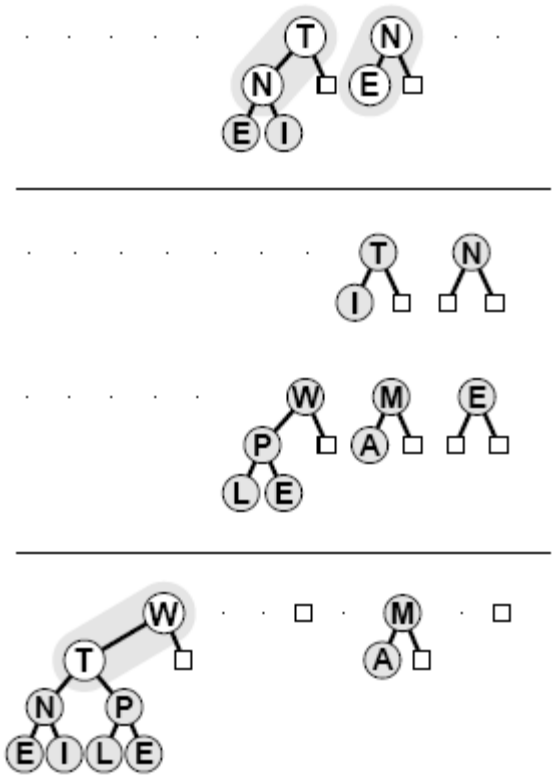


Figure 9.20
Joining of two binomial queues

Adding a binomial queue of 3 nodes to one of 7 nodes gives one of 10 nodes through a process that mimics the binary addition $011_2 + 111_2 = 1010_2$. *Adding* N *to* E *gives an empty 1-heap in the result with a carry 2-heap containing* N and E. *Then adding the three 2-heaps leaves a 2-heap in the result with a carry 4-heap containing* T N E I. *This 4-heap is added to the other 4-heap, producing the binomial queue at the bottom. Few nodes are touched in the process.*

Time: log N, because that is (about) the length of the list of option(heap) if there are N nodes (total) in all the heaps in the list.

Conclusion

Binomial queues are simple and easy to implement in a functional programming language such as ML or Gallina.

Operations (insert, delete-max, join) are all quite efficient: log(N) time. A pure-functional implementation is naturally persistent, without any extra programming effort.

Proofs of correctness are pretty straightforward too (though not exactly *trivial*).