Rank-Pairing Heaps

Bernard Haeupler Siddhartha Sen Robert E. Tarjan

Presentation by Alexander Pokluda

Cheriton School of Computer Science, University of Waterloo, Canada

SIAM JOURNAL ON COMPUTING Vol. 40, No. 6 (2011), pp. 1463-1485

Outline

What is a Rank-Pairing Heap?

Main Concept

The rank-pairing heap combines the asymptotic efficiency of Fibonacci heaps with the simplicity of pairing heaps. The heap operation *delete-min* takes $O(\log n)$ amortized time while all other heap operations take $O(1)$ amortized time, matching the theoretical lower bounds.

Definition of a Heap

A *heap* is a data structure consisting of a set of items, each with a distinct real-valued key that supports the following operations:

make-heap return a new, empty heap

insert(x, H) insert item x, with predefined key, into heap H

find-min(H) return the item in heap H with minimum key

delete-min(H) if H is not empty, delete from H the item of minimum key

meld(H_1, H_2) return a heap containing all the items in disjoint heaps H_1 and H_2, destroying H_1 and H_2

decrease-key(x, Δ, H) decrease the key of item x in heap H by the amount $\Delta > 0$

Time Bounds

Lower Bounds

Since n numbers can be sorted by doing n insertions into an initially empty heap followed by n *delete-min* operations, the classical $\Omega(n \log n)$ lower bound on the number of binary comparisons needed for sorting implies that either insertion or minimum deletion must take $\Omega(\log n)$ amortized time.

A lot of work has been done to develop data structures in which minimum deletion takes $O(\log n)$ amortized time and each of the other heap operations take $O(1)$ amortized time.

Fibonacci Heaps and Pairing Heaps

Fibonacci Heap

- Invented to support *decrease-key* in $O(1)$ time
- Supports *delete-min* in $O(\log n)$ time and other operations in $O(1)$ amortized time

Pairing Heap

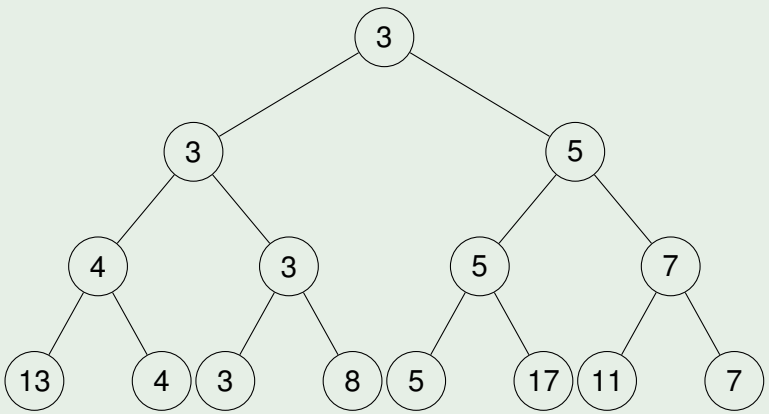
- Self-adjusting implementation
- *decrease-key* requires $\Omega(\log \log n)$ amortized time if other operations are allowed only $O(\log n)$ amortized time
- Best upper bound known for *delete-min* is $O(2^{2\sqrt{\lg \lg n}})$

Fibonacci heaps do not perform well in practice but pairing heaps do. *Rank-pairing heaps* combine the asymptotic efficiency of Fibonacci heaps with the simplicity of pairing heaps.

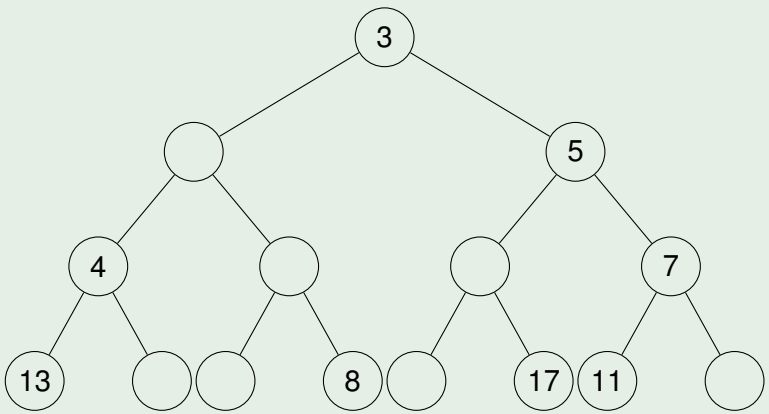
Tournaments

The basis of many heap data structures is the tournament.

Example



full representation

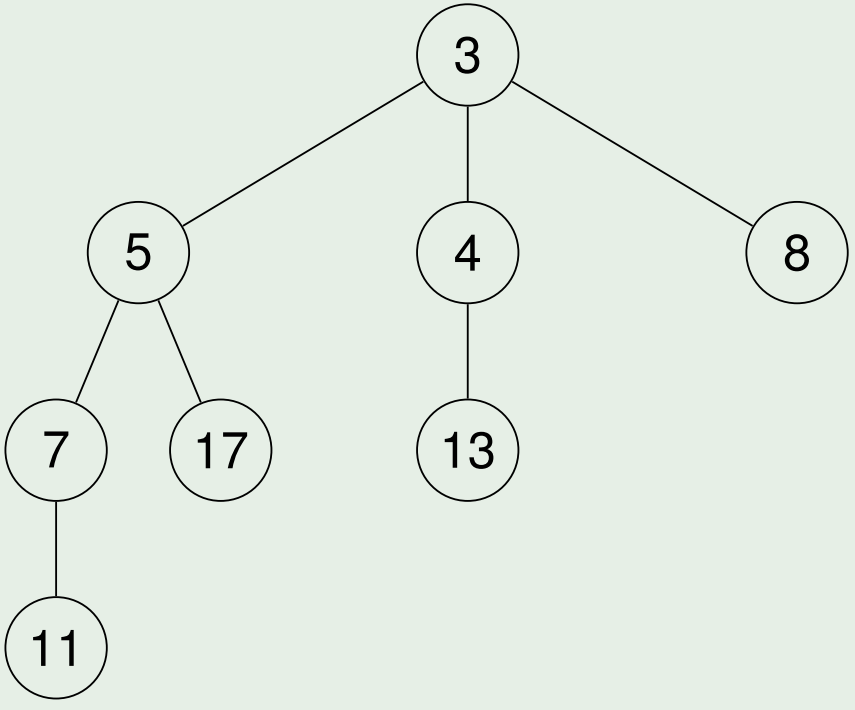


half-full representation

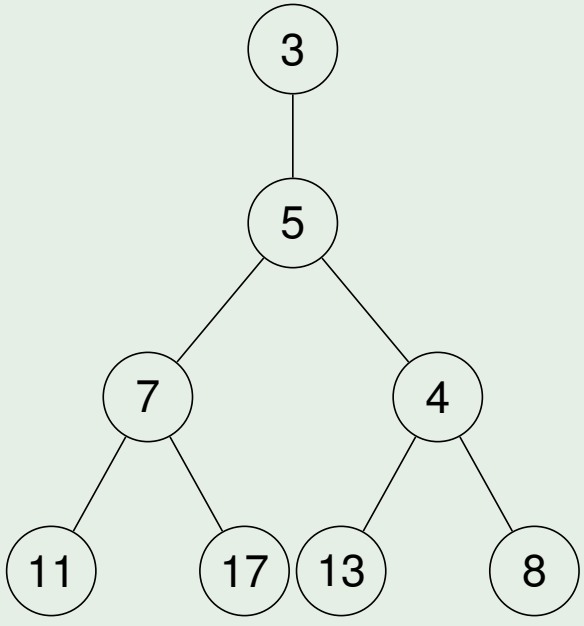
Tournaments

The basis of many heap data structures is the tournament.

Example



heap-ordered representation



half-ordered representation

Half Trees

The half-ordered representation of a tournament is a half tree.

- It is a binary tree whose root has a missing right subtree
- It is half-ordered: the key of any item is less than that of all items in its left subtree

A fundamental operation on half trees is linking:

1. Compare the keys of the roots
2. If x and y are the roots of smaller and larger key, detach the old left subtree of x and make it the right subtree of y
3. Then make the tree rooted at y the new left subtree of x

A link takes $O(1)$ time.

Example

Linking two trees

Generic Implementation

Generic Binomial Queue Implementation

A heap consists of a set of half trees whose nodes are the items in the heap, represented by a singly-linked circular list of the tree roots. Access to the root list is by a pointer to the root of minimum key, which we call the *min-root*.

Do the various heap operations, excluding key decrease and arbitrary deletion as follows:

make-heap create an empty list of roots

insert(x, H**)** make x a one-node half tree, insert it into the list of roots after the min-root, make it the new min-root if necessary

find-min(H**)** return the min-root

Generic Implementation

meld(H_1, H_2) concatenate the two root lists, and make the old min-root of smaller key the min-root of the new list

delete-min(H) disassemble the half tree rooted at the min-root x as follows:

1. Let $y = \text{left}(x)$
2. Delete x and cut each edge on the right spine of y
3. Add the new half trees to the remaining half trees
4. Find the root of minimum key and make it the new min-root

Example

Delete-min on heap H

Generic Implementation

meld(H_1, H_2) concatenate the two root lists, and make the old min-root of smaller key the min-root of the new list

delete-min(H) disassemble the half tree rooted at the min-root x as follows:

1. Let $y = \text{left}(x)$
2. Delete x and cut each edge on the right spine of y
3. Add the new half trees to the remaining half trees
4. Find the root of minimum key and make it the new min-root

Example

Delete-min on heap H

Additionally, after each heap operation, do zero or more links of half trees to reduce their number.

Linking Half Trees

This data structure is efficient only if the the links are done carefully.

Pairing heaps link half trees whose roots are adjacent in the root list

All other implementations use *node ranks* as as proxy for size and link two half trees only if they are of equal rank; after the link, increase the rank of the winning root by one

If all links are done this way, all half trees are *perfect* and the result is a *binomial queue*

Linking Half Trees

- In the original version of binomial queues, links are done eagerly to maintain the invariant that the heap contains at most one root per rank
  - This gives an $O(\log n)$ worst-case time bound for insert, meld and delete-min
- Doing links lazily, specifically during minimum deletion, gives better amortized efficiency
- Fibonacci heaps and all similar structures do as many links as possible after a minimum deletion, leaving at most one root per rank
  - This method, called *multipass linking*, is used for rp-heaps as well
- A simpler, lazier version called *one-pass linking* also works

One-pass Linking

To implement one-pass linking:

1. Maintain a set of buckets, one per rank
2. During minimum deletion, process each half tree by adding it to the bucket for its rank
3. If the bucket is not empty, link the tree with the half tree in the bucket, leaving the bucket empty
4. Add the new half trees to the list of trees in the new heap
5. Once all half trees have been processed, add any half trees still in a bucket to the list of trees in the new heap

Example

One-pass linking during minimum deletion

Theorem

The amortized time for an operation on a one-pass or multipass binomial queue is $O(1)$ *for a make-heap, find-min, insert or meld, and* $O(\log n)$ *for a delete-min.*

Proof.

A make-heap, find-min, insert or meld takes $O(1)$ actual time. We can prove minimum deletion takes $O(\log n)$ amortized time by defining a potential function of a heap to be twice the number of half trees. □

Rank-Pairing Heaps

The main goal is to implement key decrease so that it takes $O(1)$ amortized time. We extend one-pass and multipass binomial queues to support key decrease. We call the resulting data structure the rank-pairing heap.

We can implement decrease-key(x, Δ, H) as follows:

1. Reduce the key of x
2. To restore half order, create a new half tree rooted at x by detaching the subtrees rooted at x and $y = \text{right}(x)$, reattaching the subtree rooted at y in place of x, and adding x to the list of roots

We call this a cut at x.

Rank-Pairing Heaps

The main goal is to implement key decrease so that it takes $O(1)$ amortized time. We extend one-pass and multipass binomial queues to support key decrease. We call the resulting data structure the rank-pairing heap.

We can implement decrease-key(x, Δ, H) as follows:

1. Reduce the key of x
2. To restore half order, create a new half tree rooted at x by detaching the subtrees rooted at x and $y = \text{right}(x)$, reattaching the subtree rooted at y in place of x, and adding x to the list of roots

We call this a cut at x.

This implementation is correct but not efficient.

Implementing Key Decrease Efficiency

A number of implementations try to implement decrease-key efficiently by maintaining some balance condition.

Key Concept

The main insight is that no such balance condition is necessary; it suffices just to update ranks, in particular to decrease the ranks of certain ancestors. The only restructuring is the cut at x.

Denote $p(x)$ and $r(x)$ the parent and rank of a node x. Define rank difference $\Delta r(x) = r(p(x)) - r(x)$.

- A root whose left child has rank difference i is an i-node
- A non-root whose left child and right child have rank difference i and j is an i, j-node

In a binomial queue, every root is a 1-node and every non-root is a 1,1-node. We shall relax the second half of this invariant.

Type 1 Rank-Pairing Heaps

Key Observation

Each node has a number of descendants at least exponential in its rank even if we allow 0, i-nodes in addition to 1,1-nodes

Definition

type-1 rank rule: every root is a 1-node and every child is a 1,1-node or a 0, i-node for some $i > 0$

A type-1 rp-heap is a set of heap-ordered half trees whose nodes have ranks that obey the type-1 rank rule

Lemma

In a type-1 rp-heap, every node of rank k has at least 2^k descendants including itself, at least $2^{k+1} - 1$ if it is a child. Hence $k < \lg n$.

Operations on Rank-Pairing Heaps

The operations make-heap, find-min, insert, meld are exactly the same as for binomial queues. delete-min is the same except for one change: during half-tree disassembly, give each new root a rank that is one greater than its left child.
decrease-key is implemented as follows:

1. Reduce the key of x
2. If x is a root, make it the min-root if its key is now minimum
3. If x is not a root, detach the subtrees rooted at x and $y = \text{right}(x)$, reattach the subtree rooted at y in place of x, add x to the list of roots
4. Restore the rank rule: Make the rank of x one greater than its left child; and, starting at the new parent of y walk up through ancestors, reducing ranks to obey the rank rule

Example

Decrease-key in a type-1 rp-heap

Amortized Efficiency of Rank-Pairing Heaps

The efficiency of type-1 and type-2 rank pairing heaps can be analyzed using a potential function. For type-2 heaps:

- Let the potential of a root with an i-child be i
- Let the potential of a 1,1-node be 0
- Let the potential of other nodes be $i + j - 1$

Theorem

The amortized time for make-heap, find-min, insert, meld or decrease-key on a type-1 or type-2 rp-heap is $O(1)$ *and* $O(\log n)$ *for delete-min.*

Proof.

make-heap, find-min, and meld operations take $O(1)$ actual time; insert takes $O(1)$ time and increases the potential by 1.

Amortized Efficiency of Rank-Pairing Heaps

Proof. (continued)

For delete-min in type-2 heaps, the disassembly increases the potential by at most $\log_\phi n$. The entire minimum deletion takes at most $h + O(1)$ time. At most $\log_\phi n + 1$ half trees remain after all the links, so there are at least $h - \log_\phi n - 1$ links. The amortized time of minimum deletion is thus at most $2\log_\phi n + O(1)$.

For decrease-key, we must consider the cascading changes in node ranks. If x is a root, decrease-key takes $O(1)$ actual time and does not change the potential. The amortized time for all rank reductions is at most four (three units of added potential plus one unit of time for last rank reduction). Thus, decrease-key takes $O(1)$ amortized time.

The analysis for type-1 rp-heaps is similar.

Conclusion

Rank-pairing heaps combine the performance guarantees of Fibonacci heaps with the simplicity of pairing heaps.

- Ranks are used to guarantee efficiency
- Rank-pairing heaps build on previous work
- Simpler methods of doing key decreases do not have the desired efficiency

Open problems:

- Is there a simpler analysis of type-1 rp-heaps?
- Can one obtain an $O(1)$ amortized time bound for insert, meld, and decrease-key and $O(\log m)$ for delete min if only $O(1)$ rank changes are made after each key decrease?