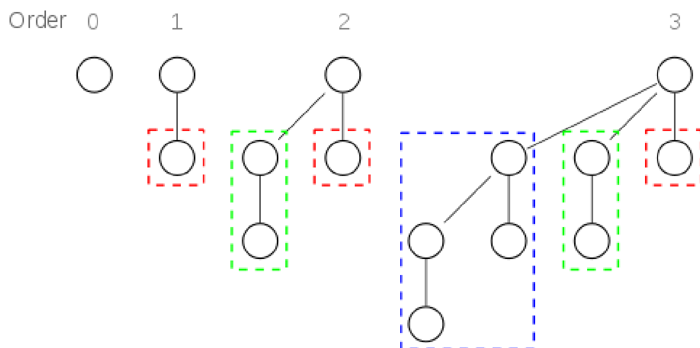
Binomial heap

Contributors to Wikimedia projects ⋮ 10-13 minutes ⋮ 6/27/2003
DOI: 10.1145/359460.359478, Show Details

In computer science, a **binomial heap** is a data structure that acts as a priority queue but also allows pairs of heaps to be merged. It is important as an implementation of the mergeable heap abstract data type (also called meldable heap), which is a priority queue supporting merge operation. It is implemented as a heap similar to a binary heap but using a special tree structure that is different from the complete binary trees used by binary heaps.[1] Binomial heaps were invented in 1978 by Jean Vuillemin.[1][2]

Binomial heap[edit]

A binomial heap is implemented as a set of **binomial** trees (compare with a binary heap, which has a shape of a single binary tree), which are defined recursively as follows:[1]



Binomial trees of order 0 to 3: Each tree has a root node with subtrees of all lower ordered binomial trees, which have been highlighted. For example, the order 3 binomial tree is connected to an order 2, 1, and 0 (highlighted as blue, green and red respectively) binomial tree.

A binomial tree of order k has 2^k nodes, and height k. The name comes from the shape: a binomial tree of order k has $\binom{k}{d}$ nodes at depth d, a binomial coefficient. Because of its structure, a binomial tree of order k can be constructed from two trees of order $k-1$ by attaching one of them as the leftmost child of the root of the other tree. This feature is central to the *merge* operation of a binomial heap, which is its major advantage over other conventional heaps.[1][3]

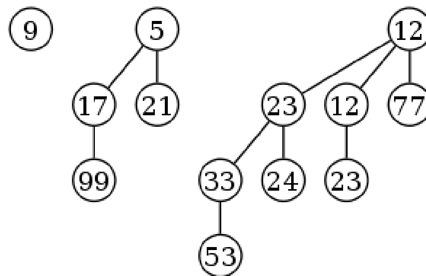
Structure of a binomial heap[edit]

A binomial heap is implemented as a set of binomial trees that satisfy the *binomial heap properties*:[1]

- Each binomial tree in a heap obeys the *minimum-heap property*: the key of a node is greater than or equal to the key of its parent.
- There can be at most one binomial tree for each order, including zero order.

The first property ensures that the root of each binomial tree contains the smallest key in the tree. It follows that the smallest key in the entire heap is one of the roots.[1]

The second property implies that a binomial heap with n nodes consists of at most $1 + \log_2 n$ binomial trees, where $\log_2$ is the binary logarithm. The number and orders of these trees are uniquely determined by the number of nodes n: there is one binomial tree for each nonzero bit in the binary representation of the number n. For example, the decimal number 13 is 1101 in binary, $2^3 + 2^2 + 2^0$, and thus a binomial heap with 13 nodes will consist of three binomial trees of orders 3, 2, and 0 (see figure below).[1][3]



Example of a binomial heap containing 13 nodes with distinct keys.
The heap consists of three binomial trees with orders 0, 2, and 3.

The number of different ways that n items with distinct keys can be arranged into a binomial heap equals the largest odd divisor of $n!$. For $n = 1, 2, 3, \dots$ these numbers are

1, 1, 3, 3, 15, 45, 315, 315, 2835, 14175, ... (sequence A049606 in the OEIS)

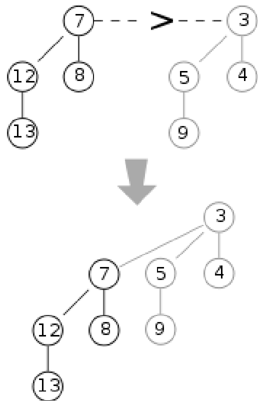
If the n items are inserted into a binomial heap in a uniformly random order, each of these arrangements is equally likely.[3]

Implementation[edit]

Because no operation requires random access to the root nodes of the binomial trees, the roots of the binomial trees can be stored in a linked list, ordered by increasing order of the tree. Because the number of children for each node is variable, it does not work well for each node to have separate links to each of its children, as would be common in a binary tree; instead, it is possible to implement this tree using links from each node to its highest-order child in the tree, and to its sibling of the next smaller order than it. These sibling pointers can be interpreted as the next pointers in a linked list of

the children of each node, but with the opposite order from the linked list of roots: from largest to smallest order, rather than vice versa. This representation allows two trees of the same order to be linked together, making a tree of the next larger order, in constant time.[1][3]

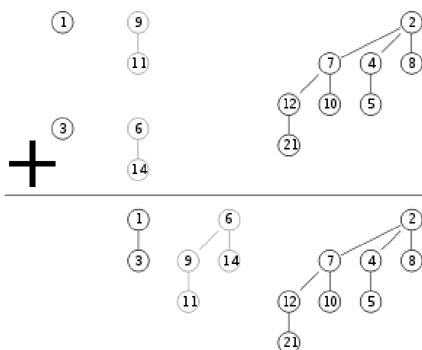
Merge[edit]



To merge two binomial trees of the same order, first compare the root key. Since 7>3, the black tree on the left (with root node 7) is attached to the grey tree on the right (with root node 3) as a subtree. The result is a tree of order 3.

The operation of **merging** two heaps is used as a subroutine in most other operations. A basic subroutine within this procedure merges pairs of binomial trees of the same order. This may be done by comparing the keys at the roots of the two trees (the smallest keys in both trees). The root node with the larger key is made into a child of the root node with the smaller key, increasing its order by one:[1][3]

```
function mergeTree(p, q)
    if p.root.key <= q.root.key
        return p.addSubTree(q)
    else
        return q.addSubTree(p)
```



This shows the merger of two binomial heaps. This is accomplished by merging two binomial trees of the same order one by one. If the resulting merged tree has the same order as one binomial tree in one of the two heaps, then those two are merged again.

To merge two heaps more generally, the lists of roots of both heaps are traversed simultaneously in a manner similar to that of the merge algorithm, in a sequence from smaller orders of trees to larger orders. When only one of the two heaps being merged contains a tree of order j, this tree is moved to the output heap. When both of the two heaps contain a tree of order j, the two trees are merged to one tree of order $j+1$ so that the minimum-heap property is satisfied. It may later become necessary to merge this tree with some other tree of order $j+1$ in one of the two input heaps. In the course of the algorithm, it will examine at most three trees of any order, two from the two heaps we merge and one composed of two smaller trees.[1][3]

```
function merge(p, q)
    while not (p.end() and q.end())
        tree = mergeTree(p.currentTree(), q.currentTree())

        if not heap.currentTree().empty()
            tree = mergeTree(tree, heap.currentTree())

        heap.addTree(tree)
        heap.next(); p.next(); q.next()
```

Because each binomial tree in a binomial heap corresponds to a bit in the binary representation of its size, there is an analogy between the merging of two heaps and the binary addition of the *sizes* of the two heaps, from right-to-left. Whenever a carry occurs during addition, this corresponds to a merging of two binomial trees during the merge.[1][3]

Each tree has order at most $\log_2 n$ and therefore the running time is $O(\log n)$.[1][3]

Insert[edit]

Inserting a new element to a heap can be done by simply creating a new heap containing only this element and then merging it with the original heap. Because of the merge, a single insertion takes time $O(\log n)$. However, this can be sped up using a merge procedure that shortcuts the merge after it reaches a point where only one of the merged heaps has trees of larger order. With this speedup, across a series of k consecutive insertions, the total time for the insertions is $O(k + \log n)$. Another way of stating this is that (after logarithmic overhead for the first insertion in a sequence) each successive **insert** has an *amortized* time of $O(1)$ (i.e. constant) per insertion.[1][3]

A variant of the binomial heap, the skew binomial heap, achieves constant worst case insertion time by using forests whose tree sizes are based on the skew binary number system rather than on the binary

number system.[4]

Find minimum[edit]

To find the **minimum** element of the heap, find the minimum among the roots of the binomial trees. This can be done in $O(\log n)$ time, as there are just $O(\log n)$ tree roots to examine.[1]

By using a pointer to the binomial tree that contains the minimum element, the time for this operation can be reduced to $O(1)$. The pointer must be updated when performing any operation other than finding the minimum. This can be done in $O(\log n)$ time per update, without raising the overall asymptotic running time of any operation.[*citation needed*]

Delete minimum[edit]

To **delete the minimum element** from the heap, first find this element, remove it from the root of its binomial tree, and obtain a list of its child subtrees (which are each themselves binomial trees, of distinct orders). Transform this list of subtrees into a separate binomial heap by reordering them from smallest to largest order. Then merge this heap with the original heap. Since each root has at most $\log_2 n$ children, creating this new heap takes time $O(\log n)$. Merging heaps takes time $O(\log n)$, so the entire delete minimum operation takes time $O(\log n)$.[1]

```
function deleteMin(heap)
    min = heap.trees().first()
    for each current in heap.trees()
        if current.root < min.root then min = current
    for each tree in min.subTrees()
        tmp.addTree(tree)
    heap.removeTree(min)
    merge(heap, tmp)
```

Decrease key[edit]

After **decreasing** the key of an element, it may become smaller than the key of its parent, violating the minimum-heap property. If this is the case, exchange the element with its parent, and possibly also with its grandparent, and so on, until the minimum-heap property is no longer violated. Each binomial tree has height at most $\log_2 n$, so this takes $O(\log n)$ time.[1] However, this operation requires that the representation of the tree include pointers from each node to its parent in the tree, somewhat complicating the implementation of other operations.[3]

Delete[edit]

To **delete** an element from the heap, decrease its key to negative infinity (or equivalently, to some value lower than any element in the heap) and then delete the minimum in the heap.[1]

Applications[edit]

- Discrete event simulation
- Priority queues

See also[edit]

- Weak heap, a combination of the binary heap and binomial heap data structures

References[edit]

1. ^ Jump up to: *a b c d e f g h i j k l m n o p q* *Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001) [1990]. "Chapter 19: Binomial Heaps". Introduction to Algorithms (2nd ed.). MIT Press and McGraw-Hill. pp. 455–475. ISBN 0-262-03293-7.*
2. ^ *Vuillemin, Jean (1 April 1978). "A data structure for manipulating priority queues". Communications of the ACM.* ***21*** *(4): 309–315. doi:10.1145/359460.359478.*
3. ^ Jump up to: *a b c d e f g h i j* *Brown, Mark R. (1978). "Implementation and analysis of binomial queue algorithms". SIAM Journal on Computing.* ***7*** *(3): 298–319. doi:10.1137/0207026. MR 0483830.*
4. ^ *Brodal, Gerth Stølting; Okasaki, Chris (November 1996), "Optimal purely functional priority queues", Journal of Functional Programming,* ***6*** *(6): 839–857, doi:10.1017/s095679680000201x*

External links[edit]

- Two C implementations of binomial heap (a generic one and one optimized for integer keys)
- Haskell implementation of binomial heap
- Common Lisp implementation of binomial heap