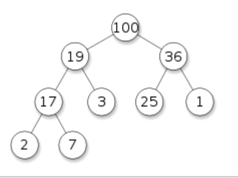
Heap (data structure)

From Wikipedia, the free encyclopedia

This article is about the programming data structure. For the dynamic memory area, see Dynamic memory allocation.

This article includes a list of references, but **its sources remain unclear** because it has **insufficient inline citations**. Please help to improve this article by introducing more precise citations. *(November 2013)*



Example of a complete binary max-heap with node keys being integers from 1 to 100

In computer science, a **heap** is a specialized tree-based abstract data type that satisfies the *heap property:* If A is a parent node of B then the key of node A is ordered with respect to the key of node B with the same ordering applying across the heap. Heaps can be classified further as either a "**max heap**" or a "**min heap**". In a max heap, the keys of parent nodes are always greater than or equal to those of the children and the highest key is in the root node. In a min heap, the keys of parent nodes are less than or equal to those of the children and the lowest key is in the root node. Heaps are crucial in several efficient graph algorithms such as Dijkstra's algorithm, and in the sorting algorithm heapsort. A common implementation of a heap is the binary heap, in which the tree is a complete binary tree (see figure).

In a heap, the highest (or lowest) priority element is always stored at the root, hence the name **heap**. A heap is not a sorted structure and can be regarded as partially ordered. As visible from the Heap-diagram, there is no particular relationship among nodes on any given level, even among the siblings. When a heap is a complete binary tree, it has a smallest possible height—a heap with N nodes always has log N height. A heap is a useful data structure when you need to remove the object with the highest (or lowest) priority.

Note that, as shown in the graphic, there is no implied ordering between siblings or cousins and no implied sequence for an in-order traversal (as there would be in, e.g., a binary search tree). The heap relation mentioned above applies only between nodes and their parents, grandparents, etc. The maximum number of children each node can have depends on the type of heap, but in many types it is at most two, which is known as a binary heap.

The heap is one maximally efficient implementation of an abstract data type called a priority queue, and in fact priority queues are often referred to as "heaps", regardless of how they may be implemented. Note that despite the similarity of the name "heap" to "stack" and "queue", the latter two are abstract data types, while a heap is a specific data structure, and "priority queue" is the proper term for the abstract data type.[citation needed]

A *heap* data structure should not be confused with *the heap* which is a common name for the pool of memory from which dynamically allocated memory is allocated. The term was originally used only for the data structure.

Contents

Operations [edit]

The common operations involving heaps are:

Basic

- *find-max* or *find-min*: find the maximum item of a max-heap or a minimum item of a min-heap (a.k.a. *peek*)
- *insert*: adding a new key to the heap (a.k.a., *push*[1])
- *extract-min* [or *extract-max*]: returns the node of minimum value from a min heap [or maximum value from a max heap] after removing it from the heap (a.k.a., *pop*[2])
- *delete-max* or *delete-min*: removing the root node of a max- or min-heap, respectively
- *replace*: pop root and push a new key. More efficient than pop followed by push, since only need to balance once, not twice, and appropriate for fixed-size heaps.[3]

Creation

- *create-heap*: create an empty heap
- *heapify*: create a heap out of given array of elements
- *merge* (*union*): joining two heaps to form a valid new heap containing all the elements of both, preserving the original heaps.
- *meld*: joining two heaps to form a valid new heap containing all the elements of both, destroying the original heaps.

Inspection

- *size*: return the number of items in the heap.
- *is-empty*: return true if the heap is empty, false otherwise – an optimized form of size when total size is not needed.

Internal

- *increase-key* or *decrease-key*: updating a key within a max- or min-heap, respectively
- *delete*: delete an arbitrary node (followed by moving last node and sifting to maintain heap)
- *sift-up*: move a node up in the tree, as long as needed; used to restore heap condition after insertion. Called "sift" because node moves up the tree until it reaches the correct level, as in a sieve.
- *sift-down*: move a node down in the tree, similar to sift-up; used to restore heap condition after deletion or replacement.

Implementation [edit]

Heaps are usually implemented in an array (fixed size or dynamic array), and do not require pointers between elements. After an element is inserted into or deleted from a heap, the heap property may be violated and the heap must be balanced by internal operations.

Full and almost full binary heaps may be represented in a very space-efficient way (as an implicit data structure) using an array alone. The first (or last) element will contain the root. The next two elements of the array contain its children. The next four contain the four children of the two child nodes, etc. Thus the children of the node at position *n* would be at positions **2n** and **2n + 1** in a one-based array, or **2n + 1** and **2n + 2** in a zero-based array. This allows moving up or down the tree by doing simple index computations. Balancing a heap is done by shift-up or shift-down operations (swapping elements which are out of order). As we can build a heap from an array without requiring extra memory (for the nodes, for example), heapsort can be used to sort an array in-place.

Different types of heaps implement the operations in different ways, but notably, insertion is often done by adding the new element at the end of the heap in the first available free space. This will generally violate the heap property, and so the elements are then sifted up until the heap property has been reestablished. Similarly, deleting the root is done by removing the root and then putting the last element in the root and sifting down to rebalance. Thus replacing is done by deleting the root and putting the *new* element in the root and sifting down, avoiding a sifting up step compared to pop (sift down of last element) followed by push (sift up of new element).

Construction of a binary (or *d*-ary) heap out of a given array of elements may be performed in linear time using the classic Floyd algorithm, with the worst-case number of comparisons equal to $2N - 2s_2(N) - e_2(N)$ (for a binary heap), where $s_2(N)$ is the sum of all digits of the binary representation of N and $e_2(N)$ is the exponent of 2 in the prime factorization of N.[4] This is faster than a sequence of consecutive insertions into an originally empty heap, which is log-linear (or linearithmic).[a]

Variants [edit]

- 2–3 heap
- B-heap
- Beap
- Binary heap
- Binomial heap
- Brodal queue
- *d*-ary heap
- Fibonacci heap
- Leftist heap

- Leftist heap
- Pairing heap
- Skew heap
- Soft heap
- Weak heap
- Leaf heap
- Radix heap
- Randomized meldable heap
- Ternary heap

Comparison of theoretic bounds for variants [edit]

In the following time complexities[5] *O*(*f*) is an asymptotic upper bound and *Θ*(*f*) is an asymptotically tight bound (see Big O notation). Function names assume a min-heap.

Operation	Binary[5]	Binomial[5]	Fibonacci[5]	Pairing[6]	Brodal[7][b]	Rank-pairing[9]	Strict Fibonacci[10]
find-min	*Θ*(1)	*Θ*(1)	*Θ*(1)	*Θ*(1)	*Θ*(1)	*Θ*(1)	*Θ*(1)
delete-min	*Θ*(log *n*)	*Θ*(log *n*)	*O*(log *n*)[c]	*O*(log *n*)[c]	*O*(log *n*)	*O*(log *n*)[c]	*O*(log *n*)
insert	*Θ*(log *n*)	*Θ*(1)[c]	*Θ*(1)	*Θ*(1)	*Θ*(1)	*Θ*(1)	*Θ*(1)
decrease-key	*Θ*(log *n*)	*Θ*(log *n*)	*Θ*(1)[c]	*o*(log *n*)[c][d]	*Θ*(1)	*Θ*(1)[c]	*Θ*(1)
merge	*Θ*(*m* log *n*)[e]	*O*(log *n*)[f]	*Θ*(1)	*Θ*(1)	*Θ*(1)	*Θ*(1)	*Θ*(1)

a. ^ Each insertion takes O(log(*k*)) in the existing size of the heap, thus $\sum_{k=1}^{n} O(\log k)$. Since $\log n/2 = (\log n) - 1$, a constant factor (half) of these insertions are within a constant factor of the maximum, so asymptotically we can assume $k = n$; formally the time is $nO(\log n) - O(n) = O(n \log n)$. This can also be readily seen from Stirling's approximation.
b. ^ Brodal and Okasaki later describe a persistent variant with the same bounds except for decrease-key, which is not supported. Heaps with *n* elements can be constructed bottom-up in *O*(*n*).[8]
c. ^ *a b c d e f g* Amortized time.
d. ^ Bounded by $\Omega(\log\log n), O(2^{2\sqrt{\log\log n}})$[11][12]
e. ^ *n* is the size of the larger heap and *m* is the size of the smaller heap.
f. ^ *n* is the size of the larger heap.

Applications [edit]

The heap data structure has many applications.

- Heapsort: One of the best sorting methods being in-place and with no quadratic worst-case scenarios.
- Selection algorithms: A heap allows access to the min or max element in constant time, and other selections (such as median or kth-element) can be done in sub-linear time on data that is in a heap.[13]
- Graph algorithms: By using heaps as internal traversal data structures, run time will be reduced by polynomial order. Examples of such problems are Prim's minimal-spanning-tree algorithm and Dijkstra's shortest-path algorithm.
- Priority Queue: A priority queue is an abstract concept like "a list" or "a map"; just as a list can be implemented with a linked list or an array, a priority queue can be implemented with a heap or a variety of other methods.
- Order statistics: The Heap data structure can be used to efficiently find the kth smallest (or largest) element in an array.

Implementations [edit]

- The C++ Standard Library provides the `make_heap`, `push_heap` and `pop_heap` algorithms for heaps (usually implemented as binary heaps), which operate on arbitrary random access iterators. It treats the iterators as a reference to an array, and uses the array-to-heap conversion. It also provides the container adaptor `priority_queue`, which wraps these facilities in a container-like class. However, there is no standard support for the decrease/increase-key operation.
- The Boost C++ libraries include a heaps library. Unlike the STL it supports decrease and increase operations, and supports additional types of heap: specifically, it supports *d*-ary, binomial, Fibonacci, pairing and skew heaps.
- The Java 2 platform (since version 1.5) provides the binary heap implementation with class `java.util.PriorityQueue<E>` in Java Collections Framework. However, there is no support for the decrease/increase-key operation.
- Python has a `heapq` module that implements a priority queue using a binary heap.
- PHP has both max-heap (`SplMaxHeap`) and min-heap (`SplMinHeap`) as of version 5.3 in the Standard PHP Library.
- Perl has implementations of binary, binomial, and Fibonacci heaps in the `Heap` distribution available on CPAN.
- The Go language contains a `heap` package with heap algorithms that operate on an arbitrary type that satisfy a given interface.
- Apple's Core Foundation library contains a `CFBinaryHeap` structure.
- Pharo has an implementation in the Collections-Sequenceable package along with a set of test cases. A heap is used in the implementation of the timer event loop.

See also [edit]

- Sorting algorithm
- Stack (abstract data type)
- Queue (abstract data type)
- Tree (data structure)
- Treap, a form of binary search tree based on heap-ordered trees

References [edit]

1. ^ The Python Standard Library, 8.4. heapq — Heap queue algorithm, heapq.heappush
2. ^ The Python Standard Library, 8.4. heapq — Heap queue algorithm, heapq.heappop
3. ^ The Python Standard Library, 8.4. heapq — Heap queue algorithm, heapq.heapreplace
4. ^ Suchenek, Marek A. (2012), "Elementary Yet Precise Worst-Case Analysis of Floyd's Heap-Construction Program", *Fundamenta Informaticae* (IOS Press) **120** (1): 75–92, doi:10.3233/FI-2012-751.
5. ^ *a b c d* Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L. (1990). *Introduction to Algorithms* (1st ed.). MIT Press and McGraw-Hill. ISBN 0-262-03141-8.
6. ^ Iacono, John (2000), "Improved upper bounds for pairing heaps", *Proc. 7th Scandinavian Workshop on Algorithm Theory*, Lecture Notes in Computer Science **1851**, Springer-Verlag, pp. 63–77, doi:10.1007/3-540-44985-X_5
7. ^ Brodal, Gerth S. (1996), "Worst-Case Efficient Priority Queues", *Proc. 7th Annual ACM-SIAM Symposium on Discrete Algorithms* (PDF), pp. 52–58
8. ^ Goodrich, Michael T.; Tamassia, Roberto (2004). "7.3.6. Bottom-Up Heap Construction". *Data Structures and Algorithms in Java* (3rd ed.). pp. 338–341.
9. ^ Haeupler, Bernhard; Sen, Siddhartha; Tarjan, Robert E. (2009). "Rank-pairing heaps" (PDF). *SIAM J. Computing*: 1463–1485.
10. ^ Brodal, G. S. L.; Lagogiannis, G.; Tarjan, R. E. (2012). *Strict Fibonacci heaps* (PDF). Proceedings of the 44th symposium on Theory of Computing - STOC '12. p. 1177. doi:10.1145/2213977.2214082. ISBN 9781450312455. edit
11. ^ Fredman, Michael Lawrence; Tarjan, Robert E. (1987). "Fibonacci heaps and their uses in improved network optimization algorithms" (PDF). *Journal of the Association for Computing Machinery* **34** (3): 596–615. doi:10.1145/28869.28874.
12. ^ Pettie, Seth (2005). "Towards a Final Analysis of Pairing Heaps" (PDF). *Max Planck Institut für Informatik*.
13. ^ Frederickson, Greg N. (1993), "An Optimal Algorithm for Selection in a Min-Heap", *Information and Computation* (PDF) **104** (2), Academic Press, pp. 197–214, doi:10.1006/inco.1993.1030

External links [edit]

- Heap at Wolfram MathWorld

Wikimedia Commons has media related to *Heaps*.

The Wikibook *Data Structures* has a page on the topic of: ***Min and Max Heaps***

v · t · e **Tree data structures**

Search trees 2–3 · 2–3–4 · AA · (a,b) · AVL · B · B+ · B* · B^x · (Optimal) Binary search · Dancing · HTree · Interval · Order statistic · (Left-leaning) Red-black · Scapegoat · Splay · T · Treap · UB ·

(dynamic sets/associative arrays)	Weight-balanced
Heaps	Binary · Binomial · Fibonacci · Leftist · Pairing · Skew · Van Emde Boas
Tries	Hash · Radix · Suffix · Ternary search · X-fast · Y-fast
Spatial data partitioning trees	BK · BSP · Cartesian · Hilbert R · *k*-d (implicit *k*-d) · M · Metric · MVP · Octree · Priority R · Quad · R · R+ · R* · Segment · VP · X
Other trees	Cover · Exponential · Fenwick · Finger · Fusion · Hash calendar · iDistance · K-ary · Left-child right-sibling · Link/cut · Log-structured merge · Merkle · PQ · Range · SPQR · Top

v · t · e

Data structures

Types	Collection · Container
Abstract	Associative array · Double-ended priority queue · Double-ended queue · List · Map · Multimap · Priority queue · Queue · Set (multiset) · Disjoint Sets · Stack
Arrays	Bit array · Circular buffer · Dynamic array · Hash table · Hashed array tree · Sparse array
Linked	Association list · Linked list · Skip list · Unrolled linked list · XOR linked list
Trees	B-tree · Binary search tree (AA · AVL · red-black · self-balancing · splay) · **Heap** (binary · binomial · Fibonacci) · R-tree (R* · R+ · Hilbert) · Trie (Hash tree)
Graphs	Binary decision diagram · Directed acyclic graph · Directed acyclic word graph

List of data structures

Categories: Heaps (data structures)