Binary heap

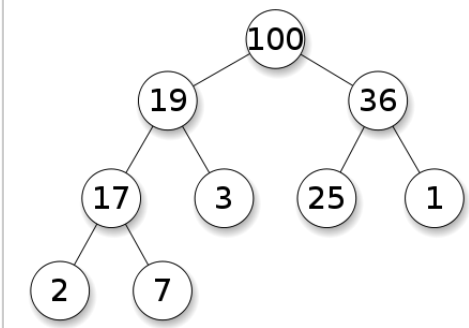
From Wikipedia, the free encyclopedia

A **binary heap** is a heap data structure created using a binary tree. It can be seen as a binary tree with two additional constraints:[2]

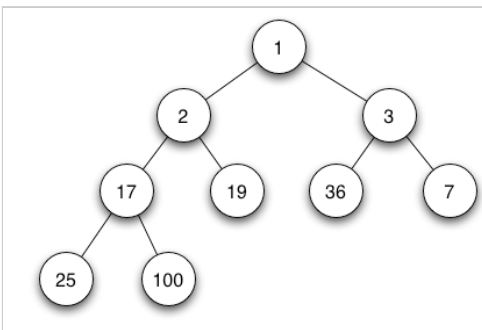
Shape property
: A binary heap is a *complete binary tree*; that is, all levels of the tree, except possibly the last one (deepest) are fully filled, and, if the last level of the tree is not complete, the nodes of that level are filled from left to right.

Heap property
: All nodes are *either* ***greater than or equal to*** *or* ***less than or equal to*** *each of its children, according to a comparison predicate defined for the heap*.

Binary Heap		
Type	Tree	
Time complexity in big O notation		
	Average	Worst case
Space	O(n)	O(n)
Search	O(n)	O(n)
Insert	O(1) [1]	O(log n)
Delete	O(log n)	O(log n)



Example of a complete binary max heap



Example of a complete binary min heap

Heaps with a mathematical "greater than or equal to" (≥) comparison predicate are called *max-heaps*; those with a mathematical "less than or equal to" (≤) comparison predicate are called *min-heaps*. Min-heaps are often used to implement priority queues.[3][4]

Since the ordering of siblings in a heap is not specified by the heap property, a single node's two children can be freely interchanged unless doing so violates the shape property (compare with treap). Note, however, that in the common array-based heap, simply swapping the children might also necessitate moving the children's sub-tree nodes to retain the heap property.

The binary heap is a special case of the d-ary heap in which d = 2.

Contents

Heap operations

Both the insert and remove operations modify the heap to conform to the shape property first, by adding or removing from the end of the heap. Then the heap property is restored by traversing up or down the heap. Both operations take O(log *n*) time.

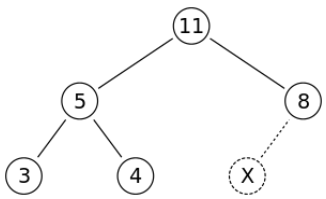
Insert

To add an element to a heap we must perform an *up-heap* operation (also known as *bubble-up*, *percolate-up*, *sift-up*, *trickle-up*, *heapify-up*, or *cascade-up*), by following this algorithm:

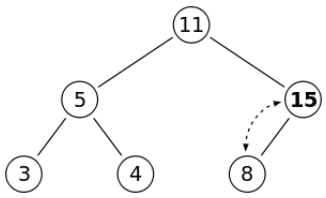
1. Add the element to the bottom level of the heap.
2. Compare the added element with its parent; if they are in the correct order, stop.
3. If not, swap the element with its parent and return to the previous step.

The number of operations required is dependent on the number of levels the new element must rise to satisfy the heap property, thus the insertion operation has a time complexity of O(log *n*). However, in 1974, Thomas Porter and Istvan Simon proved that the function for the average number of levels an inserted node moves up is upper bounded by the constant 1.6067.[1] The average number of operations required for an insertion into a binary heap is 2.6067 since one additional comparison is made that does not result in the inserted node moving up a level. Thus, on average, binary heap insertion has a constant, O(1), time complexity. Intuitively, this makes sense since approximately 50% of the elements are leaves and approximately 75% of the elements are in the bottom two levels, it is likely that the new element to be inserted will only move a few levels upwards to maintain the heap.

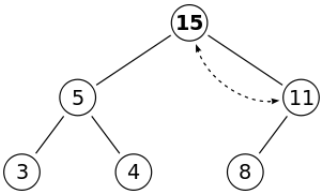
As an example of binary heap insertion, say we have a max-heap



and we want to add the number 15 to the heap. We first place the 15 in the position marked by the X. However, the heap property is violated since 15 > 8, so we need to swap the 15 and the 8. So, we have the heap looking as follows after the first swap:



However the heap property is still violated since 15 > 11, so we need to swap again:



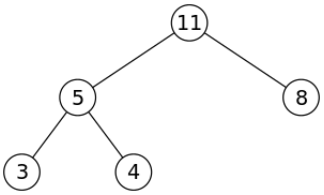
which is a valid max-heap. There is no need to check the left child after this final step: at the start, the max-heap was valid, meaning 11 > 5; if 15 > 11, and 11 > 5, then 15 > 5, because of the transitive relation.

Extract

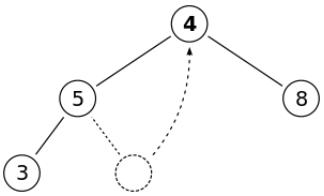
The procedure for deleting the root from the heap (effectively extracting the maximum element in a max-heap or the minimum element in a min-heap) and restoring the properties is called *down-heap* (also known as *bubble-down*, *percolate-down*, *sift-down*, *trickle down*, *heapify-down*, *cascade-down*, and *extract-min/max*).

1. Replace the root of the heap with the last element on the last level.
2. Compare the new root with its children; if they are in the correct order, stop.
3. If not, swap the element with one of its children and return to the previous step. (Swap with its smaller child in a min-heap and its larger child in a max-heap.)

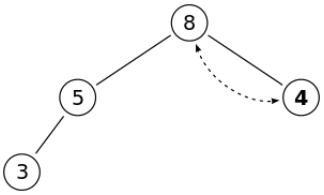
So, if we have the same max-heap as before



We remove the 11 and replace it with the 4.



Now the heap property is violated since 8 is greater than 4. In this case, swapping the two elements, 4 and 8, is enough to restore the heap property and we need not swap elements further:



The downward-moving node is swapped with the *larger* of its children in a max-heap (in a min-heap it would be swapped with its smaller child), until it satisfies the heap property in its new position. This functionality is achieved by the **Max-Heapify** function as defined below in pseudocode for an array-backed heap *A* of length *heap_length*[*A*]. Note that "A" is indexed starting at 1, not 0 as is common in many real programming languages.

Max-Heapify (*A*, *i*):
    left ← 2*i*
    right ← 2*i* + 1
    largest ← *i*
    if *left* ≤ *heap_length*[*A*] **and** *A*[*left*] > A[*largest*] **then**:
        largest ← *left*
    if *right* ≤ *heap_length*[*A*] **and** *A*[*right*] > *A*[*largest*] **then**:
        largest ← *right*
    if *largest* ≠ *i* **then**:
        swap *A[*i*]* ↔ A*[*largest*]*
        Max-Heapify(*A*, *largest*)

For the above algorithm to correctly re-heapify the array, the node at index *i* and its two direct children must violate the heap property. If they do not, the algorithm will fall through with no change to the array. The down-heap operation (without the preceding swap) can also be used to modify the value of the root, even when an element is not being deleted.

In the worst case, the new root has to be swapped with its child on each level until it reaches the bottom level of the heap, meaning that the delete operation has a time complexity relative to the height of the tree, or O(log *n*).

Building a heap

A heap could be built by successive insertions. This approach requires $O(n \log n)$ time because each insertion takes $O(\log n)$ time and there are n elements. However this is not the optimal method. The optimal method starts by arbitrarily putting the elements on a binary tree, respecting the shape property (the tree could be represented by an array, see below). Then starting from the lowest level and moving upwards, shift the root of each subtree downward as in the deletion algorithm until the heap property is restored. More specifically if all the subtrees starting at some height h (measured from the bottom) have already been "heapified", the trees at height $h+1$ can be heapified by sending their root down along the path of maximum valued children when building a max-heap, or minimum valued children when building a min-heap. This process takes $O(h)$ operations (swaps) per node. In this method most of the heapification takes place in the lower levels. Since the height of the heap is $\lfloor \log(n) \rfloor$, the number of nodes at height h is $\leq \left\lceil 2^{(\log n - h) - 1} \right\rceil = \left\lceil \frac{2^{\log n}}{2^{h+1}} \right\rceil = \left\lceil \frac{n}{2^{h+1}} \right\rceil$. Therefore, the cost of heapifying all subtrees is:

$$\begin{aligned}\sum_{h=0}^{\lceil \log n \rceil} \frac{n}{2^{h+1}} O(h) &= O\left(n \sum_{h=0}^{\lceil \log n \rceil} \frac{h}{2^{h+1}}\right) \\ &\leq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) \\ &= O(n)\end{aligned}$$

This uses the fact that the given infinite series $h / 2^h$ converges to 2.

The exact value of the above (the worst-case number of comparisons during the heap construction) is known to be equal to:

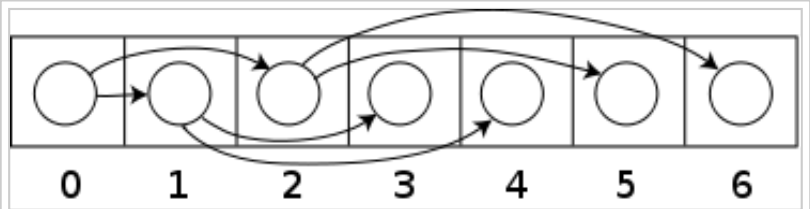
$$2n - 2s_2(n) - e_2(n),$$[5]

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.

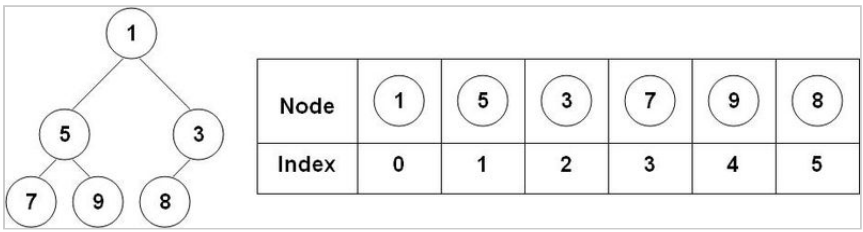
The **Build-Max-Heap** function that follows, converts an array *A* which stores a complete binary tree with n nodes to a max-heap by repeatedly using **Max-Heapify** in a bottom up manner. It is based on the observation that the array elements indexed by *floor*(n/2) + 1, *floor*(n/2) + 2, ..., n are all leaves for the tree, thus each is a one-element heap. **Build-Max-Heap** runs **Max-Heapify** on each of the remaining tree nodes.

Build-Max-Heap (*A*):
    heap_length[*A*] ← *length*[*A*]
    for *i* ← *floor*(*length*[*A*]/2) **downto** 1 **do**
        Max-Heapify(*A*, *i*)

Heap implementation



A small complete binary tree stored in an array



Comparison between a binary heap and an array implementation.

Heaps are commonly implemented with an array. Any binary tree can be stored in an array, but because a binary heap is always a complete binary tree, it can be stored compactly. No space is required for pointers; instead, the parent and children of each node can be found by arithmetic on array indices. These properties make this heap implementation a simple example of an implicit data structure or Ahnentafel list. Details depend on the root position, which in turn may depend on constraints of a programming language used for implementation, or programmer preference. Specifically, sometimes the root is placed at index 1, sacrificing space in order to simplify arithmetic. The *peek* operation (*find-min* or *find-max*) simply returns the value of the root, and is thus O(1).

Let *n* be the number of elements in the heap and *i* be an arbitrary valid index of the array storing the heap. If the tree root is at index 0, with valid indices 0 through $n - 1$, then each element *a* at index *i* has

- children at indices $2i + 1$ and $2i + 2$
- its parent *floor*($(i - 1) / 2$).

Alternatively, if the tree root is at index 1, with valid indices 1 through *n*, then each element *a* at index *i* has

- children at indices $2i$ and $2i + 1$

- its parent at index *floor*(i / 2).

This implementation is used in the heapsort algorithm, where it allows the space in the input array to be reused to store the heap (i.e. the algorithm is done in-place). The implementation is also useful for use as a Priority queue where use of a dynamic array allows insertion of an unbounded number of items.

The upheap/downheap operations can then be stated in terms of an array as follows: suppose that the heap property holds for the indices $b, b+1, ..., e$. The sift-down function extends the heap property to $b-1, b, b+1, ..., e$. Only index $i = b-1$ can violate the heap property. Let j be the index of the largest child of $a[i]$ (for a max-heap, or the smallest child for a min-heap) within the range $b, ..., e$. (If no such index exists because $2i > e$ then the heap property holds for the newly extended range and nothing needs to be done.) By swapping the values $a[i]$ and $a[j]$ the heap property for position i is established. At this point, the only problem is that the heap property might not hold for index j. The sift-down function is applied tail-recursively to index j until the heap property is established for all elements.

The sift-down function is fast. In each step it only needs two comparisons and one swap. The index value where it is working doubles in each iteration, so that at most $\log_2 e$ steps are required.

For big heaps and using virtual memory, storing elements in an array according to the above scheme is inefficient: (almost) every level is in a different page. B-heaps are binary heaps that keep subtrees in a single page, reducing the number of pages accessed by up to a factor of ten.[6]

The operation of merging two binary heaps takes $\Theta(n)$ for equal-sized heaps. The best you can do is (in case of array implementation) simply concatenating the two heap arrays and build a heap of the result.[7] A heap on n elements can be merged with a heap on k elements using $O(\log n \log k)$ key comparisons, or, in case of a pointer-based implementation, in $O(\log n \log k)$ time.[8] An algorithm for splitting a heap on n elements into two heaps on k and $n-k$ elements, respectively, based on a new view of heaps as an ordered collections of subheaps was presented in.[9] The algorithm requires $O(\log n * \log n)$ comparisons. The view also presents a new and conceptually simple algorithm for merging heaps. When merging is a common task, a different heap implementation is recommended, such as binomial heaps, which can be merged in $O(\log n)$.

Additionally, a binary heap can be implemented with a traditional binary tree data structure, but there is an issue with finding the adjacent element on the last level on the binary heap when adding an element. This element can be determined algorithmically or by adding extra data to the nodes, called "threading" the tree—instead of merely storing references to the children, we store the inorder successor of the node as well.

It is possible to modify the heap structure to allow extraction of both the smallest and largest element in $O(\log n)$ time.[10] To do this, the rows alternate between min heap and max heap. The algorithms are roughly the same, but, in each step, one must consider the alternating rows with alternating comparisons. The performance is roughly the same as a normal single direction heap. This idea can be generalised to a min-max-median heap.

Derivation of index equations

In an array-based heap, the children and parent of a node can be located via simple arithmetic on the node's index. This section derives the relevant equations for heaps with their root at index 0, with additional notes on heaps with their root at index 1.

To avoid confusion, we'll define the **level** of a node as its distance from the root, such that the root itself occupies level 0.

Child nodes

For a general node located at index i (beginning from 0), we will first derive the index of its right child, $\text{right} = 2i + 2$.

Let node i be located in level L, and note that any level l contains exactly 2^l nodes. Furthermore, there are exactly $2^{l+1} - 1$ nodes contained in the layers up to and including layer l (think of binary arithmetic; 0111...111 = 1000...000 - 1). Because the root is stored at 0, the kth node will be stored at index $(k - 1)$. Putting these observations together yields the following expression for the **index of the last node in layer l**.

$$\text{last}(l) = (2^{l+1} - 1) - 1 = 2^{l+1} - 2$$

Let there be j nodes after node i in layer L, such that

$$\begin{aligned} i = &\quad \text{last}(L) - j \\ = &\quad (2^{L+1} - 2) - j \end{aligned}$$

Each of these j nodes must have exactly 2 children, so there must be $2j$ nodes separating i's right child from the end of its layer ($L + 1$).

$$\begin{aligned} \text{right} = &\quad \text{last}(\text{L} + 1) - 2j \\ = &\quad (2^{L+2} - 2) - 2j \\ = &\quad 2(2^{L+1} - 2 - j) + 2 \\ = &\quad 2i + 2 \end{aligned}$$

As required.

Noting that the left child of any node is always 1 place before its right child, we get $\text{left} = 2i + 1$.

If the root is located at index 1 instead of 0, the last node in each level is instead at index $2^{l+1} - 1$. Using this throughout yields $\text{left} = 2i$ and $\text{right} = 2i + 1$ for heaps with their root at 1.

Parent node

Every node is either the left or right child of its parent, so we know that either of the following is true.

1. $i = 2 \times (\text{parent}) + 1$
2. $i = 2 \times (\text{parent}) + 2$

Hence,

$$\text{parent} = \frac{i - 1}{2} \textbf{or} \frac{i - 2}{2}$$

Now consider the expression $\left\lfloor \frac{i-1}{2} \right\rfloor$.

If node i is a left child, this gives the result immediately, however, it also gives the correct result if node i is a right child. In this case, $(i-2)$ must be even, and hence $(i-1)$ must be odd.

$$\begin{aligned} \left\lfloor \frac{i-1}{2} \right\rfloor &= \left\lfloor \frac{i-2}{2} + \frac{1}{2} \right\rfloor \\ &= \frac{i-2}{2} \\ &= \text{parent} \end{aligned}$$

Therefore, irrespective of whether a node is a left or right child, its parent can be found by the expression:

$$\text{parent} = \left\lfloor \frac{i-1}{2} \right\rfloor$$

See also

- Heap
- Heapsort

Notes

References

1. Thomas Porter; Istvan Simon (1974). "Random Insertion into a Priority Queue Structure" (http://i.stanford.edu/pub/cstr/reports/cs/tr/74/460/CS-TR-74-460.pdf) (PDF). *Stanford University Reports*. Stanford University. p. 13. Retrieved 31 January 2014.
2. eEL,CSA_Dept,IISc,Bangalore, "Binary Heaps" (http://lcm.csa.iisc.ernet.in/dsa/node137.html), *Data Structures and Algorithms* (http://lcm.csa.iisc.ernet.in/dsa/dsa.html)
3. "heapq – Heap queue algorithm" (https://docs.python.org/library/heapq.html). *Python Standard Library*.
4. "Class PriorityQueue" (http://download.oracle.com/javase/6/docs/api/java/util/PriorityQueue.html). *Java™ Platform Standard Ed. 6*.
5. Suchenek, Marek A. (2012), "Elementary Yet Precise Worst-Case Analysis of Floyd's Heap-Construction Program" (http://www.deepdyve.com/lp/ios-press/elementary-yet-precise-worst-case-analysis-of-floyd-s-heap-50NW30HMxU), *Fundamenta Informaticae* (IOS Press) **120** (1): 75–92, doi:10.3233/FI-2012-751 (https://dx.doi.org/10.3233%2FFI-2012-751).
6. Poul-Henning Kamp. "You're Doing It Wrong" (http://queue.acm.org/detail.cfm?id=1814327). ACM Queue. June 11, 2010.
7. Chris L. Kuszmaul. "binary heap" (http://nist.gov/dads/HTML/binaryheap.html). Dictionary of Algorithms and Data Structures, Paul E. Black, ed., U.S. National Institute of Standards and Technology. 16 November 2009.
8. J.-R. Sack and T. Strothotte "An Algorithm for Merging Heaps" (http://www.springerlink.com/content/k24440h5076w013q/), Acta Informatica 22, 171-186 (1985).
9. . J.-R. Sack and T. Strothotte "A characterization of heaps and its applications" (http://www.sciencedirect.com/science/article/pii/089054019090026E) Information and Computation Volume 86, Issue 1, May 1990, Pages 69–86.
10. Atkinson, M.D., J.-R. Sack, N. Santoro, and T. Strothotte (1 October 1986). "Min-max heaps and generalized

priority queues." (http://cg.scs.carleton.ca/~morin/teaching/5408/refs/minmax.pdf) (PDF). Programming techniques and Data structures. Comm. ACM, 29(10): 996–1000.

External links

- Binary Heap Applet (http://people.ksp.sk/~kuko/bak/index.html) by Kubo Kovac
- Using Binary Heaps in A* Pathfinding (http://www.policyalmanac.org/games/binaryHeaps.htm)
- Open Data Structures - Section 10.1 - BinaryHeap: An Implicit Binary Tree (http://opendatastructures.org/versions/edition-0.1e/ods-java/10_1_BinaryHeap_Implicit_Bi.html)
- Implementation of binary max heap in C (http://robin-thomas.github.io/max-heap/) by Robin Thomas
- Implementation of binary min heap in C (http://robin-thomas.github.io/min-heap/) by Robin Thomas