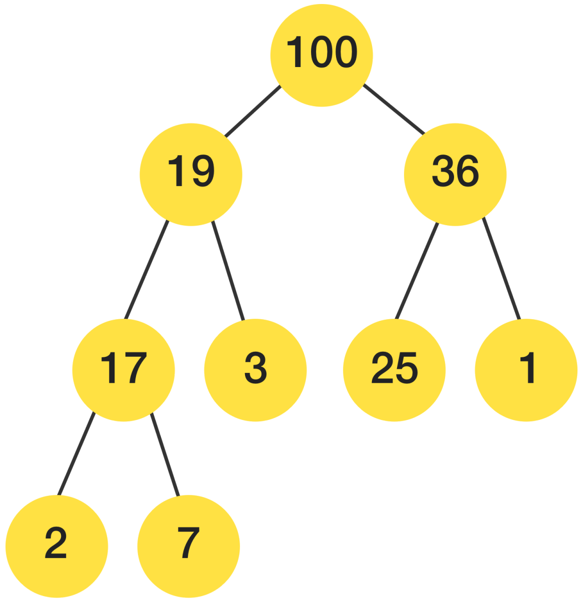
Heaps

Heaps are tree-based data structures constrained by a **heap property**. Heaps are used in many famous algorithms such as Dijkstra's algorithm for finding the shortest path, the heap sort sorting algorithm, implementing priority queues, and more. Essentially, heaps are the data structure you want to use when you want to be able to access the maximum or minimum element very quickly.

There are many variations of heaps, each offering advantages and tradeoffs.



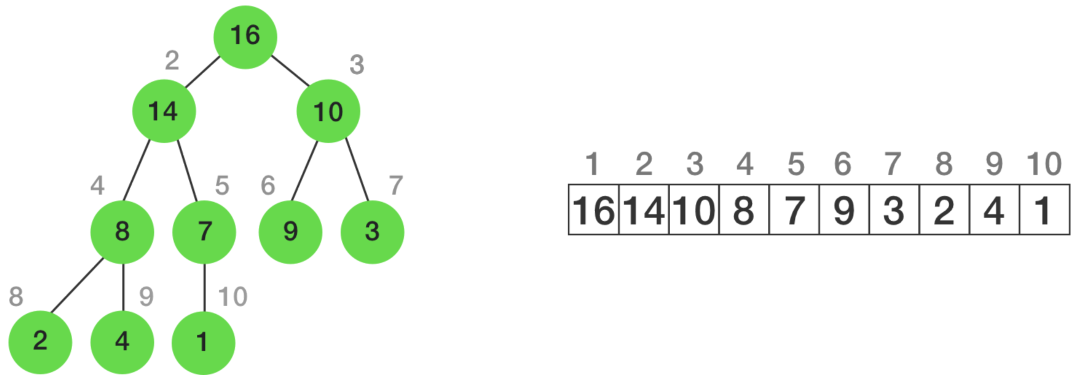
Example of a complete binary max-heap with node keys being integers from 1 to 100[1]

Contents

General Structure

A heap is a data structure that is usually implemented with an array that can be thought of as a tree (they can be implemented using pointers, however). For example, in the image below, notice how the elements in the array map to the tree to create a max-heap (a heap where the parent node has a larger value than its children).

In terms of the tree, the **root** of the heap is the top most element. In the image below, the root is 16. The **height** of a given node in the tree is defined by the longest path from it to a **leaf**, where a leaf is a node at the bottom of the tree.



Example of a Max Heap (note: 1 indexing)[2]

There are two general versions of the heap property: a **min-heap** property and a **max-heap** property.

Max-heap property

If A is an array representation of a heap, then in Max-heap:

$$A[\text{parent}(i)] \geq A[i],$$

which means that a node can't have a greater value than its parent. In a max-heap, the largest element is stored at the root, and the minimum elements are in the leaves.

Min-heap property

Similarly, if A is an array representation of a heap then, in Min-heap:

$$A[\text{parent}(i)] \leq A[i],$$

which means that a parent node can't have a greater value than its children. Thus, the minimum element is located at the root, and the maximum elements are located in the leaves.

Depending on the type of heap used, the heap property may have additional requirements.

In order to maintain the max-heap property (or min-heap property), heapsort uses a procedure called `max_heapify(A,i)`. It takes an array A and an index in the array i as input. This can easily be adapted to a `min-heapify` function.

Here is a Python implementation of `max_heapify`:

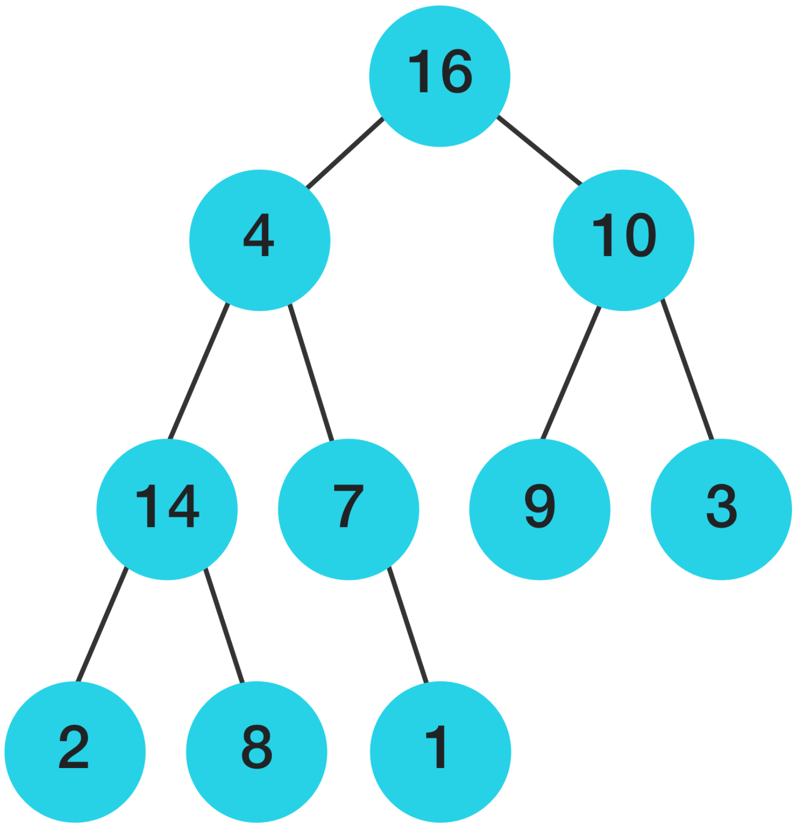
Python

```
def max_heapify(A, heap_size, i):
    left = 2 * i + 1
    right = 2 * i + 2
    largest = i
    if left < heap_size and A[left] > A[largest]:
        largest = left
    else:
        largest = i
    if right < heap_size and A[right] > A[largest]:
        largest = right
    if largest != i:
        A[i], A[largest] = A[largest], A[i]
        max_heapify(A, heap_size, largest)
```

Essentially, if an element $A[i]$ violates the max-heap property, `max_heapify` will correct it by trickling the element down the tree, until the subtree rooted at index i is a max heap (and therefore the violation is corrected).

TRY IT YOURSELF

Which parent node violates the max-heap property?



- ○ 1
- ○ 4
- ○ 8
- ○ 9
- ○ 16

Minimum Functionalities of Heaps

There are several operations that all heaps should implement. Depending on the specific type of heap used, the way that these operations are implemented may vary. No matter what operation is done in any given heap implementation, the heap properties associated with the implementation must be satisfied when the operation is complete. For example, if performing an operation on a min-heap implemented using a binary heap violates the min-heap property, the violation must be fixed before the operation is complete. Here are a few basic heap operations.

- **Build Heap**: It is important to be able to construct a heap. `Build-Heap` is usually implemented using the `Insert` and `Heapify` function repeatedly. So starting from an empty heap, nodes are added with `Insert` and then `Heapify` is called to make sure the heap maintains the heap properties at each step.
- **Heapify**: Used to maintain the heap properties (described in above sections).
- **Insert**: It is important to be able to add elements to the heap.
- **Remove**: It is important to be able to delete elements from the heap.
- **Find Minimum/Maximum** and **Extract Minimum/Maximum**: Depending on the purpose of the heap, the largest or smallest elements are often of interest so these operations are useful to have.
- **Decrease/Increase Key**: This is used to change the key of a particular node. The key determines the place in the heap where the node will be. So adjusting the key allows the algorithm to rearrange parts of the heap.
- **Merge**: Sometimes called **meld**, the merge function is a useful operation to have to combine heaps.

Types of Heaps

There are several different types of heaps, each with a different implementation and various advantages and disadvantages. However, each heap type satisfies the heap property and can be used for the same types of tasks.

Type of Heap	Insert	Delete	Decrease Key	Merge	Extract Min

Binary Heap	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$\Theta(\log n)$
Fibonacci Heap*	$O(1)$	$O(\log n)$	$O(1)$	$O(1)$	$O(\log n)$
Binomial Heap	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Pairing Heap*	$O(1)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

*Amortized running times

Python Implementation of a Heap

This is partial Python code from Python's documentation site describing the internal configuration of heaps in the Python language. This is just a sketch of how heap operations work. There are parts of the code that are missing here, for the sake of brevity, and this code is meant as a learning tool to get an idea of how Python implements a few basic heap operations. [3]

Python

```python
def heappush(heap, item):
    """Push item onto heap, maintaining the heap invariant."""
    heap.append(item)
    _siftdown(heap, 0, len(heap)-1)

def heappop(heap):
    """Pop the smallest item off the heap, maintaining the heap invariant."""
    lastelt = heap.pop()    # raises appropriate IndexError if heap is empty
    if heap:
        returnitem = heap[0]
        heap[0] = lastelt
        _siftup(heap, 0)
    else:
        returnitem = lastelt
    return returnitem

def heapify(x):
    """Transform list into a heap, in-place, in O(len(x)) time."""
    n = len(x)
    # Transform bottom-up.  The largest index there's any point to looking at
    # is the largest with a child index in-range, so must have 2*i + 1 < n,
    # or i < (n-1)/2.  If n is even = 2*j, this is (2*j-1)/2 = j-1/2 so
    # j-1 is the largest, which is n//2 - 1.  If n is odd = 2*j+1, this is
    # (2*j+1-1)/2 = j so j-1 is the largest, and that's again n//2-1.
    for i in reversed(xrange(n//2)):
        _siftup(x, i)

def _heappushpop_max(heap, item):
    """Maxheap version of a heappush followed by a heappop."""
    if heap and cmp_lt(item, heap[0]):
        item, heap[0] = heap[0], item
        _siftup_max(heap, 0)
    return item

def _heapify_max(x):
    """Transform list into a maxheap, in-place, in O(len(x)) time."""
    n = len(x)
    for i in reversed(range(n//2)):
        _siftup_max(x, i)

def nlargest(n, iterable):
    """Find the n largest elements in a dataset.

    Equivalent to:  sorted(iterable, reverse=True)[:n]
    """
    if n < 0:
        return []
    it = iter(iterable)
    result = list(islice(it, n))
    if not result:
        return result
    heapify(result)
    _heappushpop = heappushpop
    for elem in it:
        _heappushpop(result, elem)
    result.sort(reverse=True)
    return result

def nsmallest(n, iterable):
    """Find the n smallest elements in a dataset.

    Equivalent to:  sorted(iterable)[:n]
    """
    if n < 0:
        return []
    it = iter(iterable)
    result = list(islice(it, n))
    if not result:
        return result
    _heapify_max(result)
    _heappushpop = _heappushpop_max
    for elem in it:
        _heappushpop(result, elem)
    result.sort()
    return result

# 'heap' is a heap at all indices >= startpos, except possibly for pos.  pos
# is the index of a leaf with a possibly out-of-order value.  Restore the
# heap invariant.
def _siftdown(heap, startpos, pos):
    newitem = heap[pos]
    # Follow the path to the root, moving parents down until finding a place
    # newitem fits.
    while pos > startpos:
        parentpos = (pos - 1) >> 1
        parent = heap[parentpos]
        if cmp_lt(newitem, parent):
            heap[pos] = parent
            pos = parentpos
            continue
        break
    heap[pos] = newitem

def _siftup(heap, pos):
    endpos = len(heap)
```

```python
    startpos = pos
    newitem = heap[pos]
    # Bubble up the smaller child until hitting a leaf.
    childpos = 2*pos + 1    # leftmost child position
    while childpos < endpos:
        # Set childpos to index of smaller child.
        rightpos = childpos + 1
        if rightpos < endpos and not cmp_lt(heap[childpos], heap[rightpos]):
            childpos = rightpos
        # Move the smaller child up.
        heap[pos] = heap[childpos]
        pos = childpos
        childpos = 2*pos + 1
    # The leaf at pos is empty now.  Put newitem there, and bubble it up
    # to its final resting place (by sifting its parents down).
    heap[pos] = newitem
    _siftdown(heap, startpos, pos)

def _siftdown_max(heap, startpos, pos):
    'Maxheap variant of _siftdown'
    newitem = heap[pos]
    # Follow the path to the root, moving parents down until finding a place
    # newitem fits.
    while pos > startpos:
        parentpos = (pos - 1) >> 1
        parent = heap[parentpos]
        if cmp_lt(parent, newitem):
            heap[pos] = parent
            pos = parentpos
            continue
        break
    heap[pos] = newitem

def _siftup_max(heap, pos):
    'Maxheap variant of _siftup'
    endpos = len(heap)
    startpos = pos
    newitem = heap[pos]
    # Bubble up the larger child until hitting a leaf.
    childpos = 2*pos + 1    # leftmost child position
    while childpos < endpos:
        # Set childpos to index of larger child.
        rightpos = childpos + 1
        if rightpos < endpos and not cmp_lt(heap[rightpos], heap[childpos]):
            childpos = rightpos
        # Move the larger child up.
        heap[pos] = heap[childpos]
        pos = childpos
        childpos = 2*pos + 1
    # The leaf at pos is empty now.  Put newitem there, and bubble it up
    # to its final resting place (by sifting its parents down).
    heap[pos] = newitem
    _siftdown_max(heap, startpos, pos)
```

The full implementation can be found on the Python documentation website here.

Applications

Queues

Heaps can be used to implement priority queues where the first object in is the first object to come out of the queue.

EXAMPLE

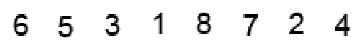
Consider a supermarket line. How can you model a line of customers waiting to pay for their items as a max-heap?

Show Answer

Heap Sort

Heaps are used in the heapsort sorting algorithm. Heapsort is a fast and space efficient sorting algorithm. It works by maintaining heap properties and taking advantage of the ordered nature of min and max heaps.

Here is an animation that shows heapsort. Notice how the heap is built up from the list and how the max-heap property is enforced.



[4]

See Also

- Heap Sort
- Binary Heaps
- Binomial Heaps
- Fibonacci Heap
- Pairing Heaps

References

1. , E. *Max-Heap.svg*. Retrieved June 5, 2016, from https://en.wikipedia.org/wiki/File:Max-Heap.svg
2. Demaine, E., & Devadas, S. *Lecture 4: Heaps and Heap Sort*. Retrieved May 23, 2016, from http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-

2011/lecture-videos/MIT6_006F11_lec04.pdf

3. Panter , M. *heapq.py*. Retrieved June 7, 2016, from https://hg.python.org/cpython/file/2.7/Lib/heapq.py
4. , S. *Heapsort Example*. Retrieved June 7, 2016, from https://en.wikipedia.org/wiki/File:Heapsort-example.gif

Cite as: Heaps. *Brilliant.org*. Retrieved 20:08, May 12, 2020, from https://brilliant.org/wiki/heaps/