# Heap vs Binary Search Tree (BST)

Heap just guarantees that elements on higher levels are greater (for max-heap) or smaller (for min-heap) than elements on lower levels, whereas BST guarantees order (from "left" to "right"). If you want sorted elements, go with BST

Heap is better at findMin/findMax (O(1)), while BST is good at all finds (O(logN)). Insert is O(logN) for both structures. If you only care about findMin/findMax (e.g. priority-related), go with heap. If you want everything sorted, go with BST.

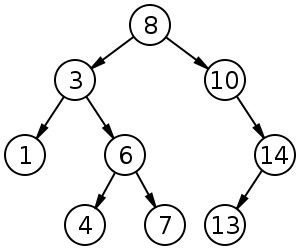
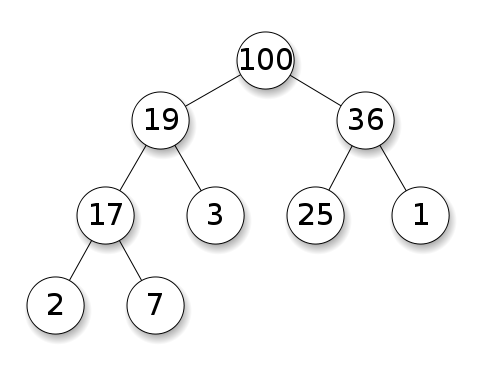
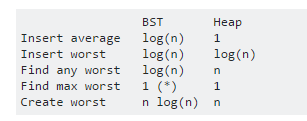


Figure : Heap

BST have average of O(logn) for insertion, deletion, and search.  
Binary Heaps have average O(1)O(1) for findMin/findMax and O(logn) for insertion and deletion



**Advantages of binary heap over a balanced BST**

average time insertion into a binary heap is O(1), for BST is O(log(n)). **This** is the killer feature of heaps.

There are also other heaps which reach O(1) amortized (stronger) like the [Fibonacci Heap](https://en.wikipedia.org/wiki/Fibonacci_heap), and even worst case, like the [Brodal queue](https://en.wikipedia.org/wiki/Brodal_queue), although they may not be practical because of non-asymptotic performance: <http://stackoverflow.com/questions/30782636/are-fibonacci-heaps-or-brodal-queues-used-in-practice-anywhere>

binary heaps can be efficiently implemented on top of arrays, BST cannot.

So we don't have to store 3 pointers per node (left, right, parent) plus balancing data (e.g. RB-ness), saving memory by a constant factor.

binary heap creation [is O(n) worst case](https://en.wikipedia.org/wiki/Binary_heap#Building_a_heap), O(n log(n)) for BST.

**Advantage of BST over binary heap**

* search for arbitrary elements is O(log(n)). **This** is the killer feature of BSTs.

For heap, it is O(n) in general, except for the largest element which is O(1).

**"False" advantage of heap over BST**

* heap is O(1) to find max, BST O(log(n)).

This is a common misconception, because it is trivial to modify a balanced BST to keep track of the largest element, and update it whenever that element could be changed: on insertion of a larger one swap, on removal find the second largest. <http://stackoverflow.com/questions/7878622/can-we-use-binary-search-tree-to-simulate-heap-operation> (mentioned [by Yeo](http://stackoverflow.com/a/27074221/895245)).

Actually, this is a limitation of heaps compared to BSTs: the only efficient search is that for the largest element.

**Average binary heap insert is O(1)**

Sources:

* Paper: <http://i.stanford.edu/pub/cstr/reports/cs/tr/74/460/CS-TR-74-460.pdf>
* WSU slides: <http://www.eecs.wsu.edu/~holder/courses/CptS223/spr09/slides/heaps.pdf>

Intuitive argument:

* bottom tree levels have exponentially more elements than top levels, so new elements are almost certain to go at the bottom
* heap insertion [starts from the bottom](https://en.wikipedia.org/wiki/Binary_heap#Insert), BST must start from the top

In a binary heap, increasing the value at a given index is also O(1) for the same reason. But if you want to do that, it is likely that you will want to keep an extra index up-to-date on heap operations <http://stackoverflow.com/questions/17009056/how-to-implement-ologn-decrease-key-operation-for-min-heap-based-priority-queu> e.g. for Dijikstra. Possible at no extra time cost.

**BST cannot be efficiently implemented on an array**

Heap operations only need to bubble up or down a single tree branch, so O(log(n)) worst case swaps, O(1) average.

Keeping a BST balanced requires tree rotations, which can change the top element for another one, and would require moving the entire array around (O(n)).

**Philosophy**

* BSTs maintain a global property between a parent and all descendants (left smaller, right bigger).

The top node of a balanced BST is the middle element, which requires global knowledge to maintain (knowing how many smaller and larger elements are there).

This global property is more expensive to maintain (log n insert), but gives more powerful searches (log n search).

* Heaps maintain a local property between parent and direct children (parent > children).

The top note of a heap is the big element, which only requires local knowledge to maintain (knowing your parent).

**Doubly-linked list**

A doubly linked list can be seen as subset of the heap where first item has greatest priority, so let's compare them here as well:

* insertion:
  + position:
    - doubly linked list: the inserted item must be either the first or last, as we only have pointers to those elements.
    - binary heap: the inserted item can end up in any position. Less restrictive than heap.
  + time:
    - doubly linked list: O(1) worst case we have pointers to the items, and the update is really simple
    - binary heap: O(1) average, thus worse. Tradeoff for having more general insertion position.
* search: O(n) for both

An use case for this is when the key of the heap is the current timestamp: in that case, new entries will always go to the beginning of the list. So we can even forget the exact timestamp altogether, and just keep the position in the list as the priority.

This can be used to implement an [LRU cache](http://stackoverflow.com/a/34206517/895245). Just like [for heap applications like Dijkstra](http://stackoverflow.com/questions/14252582/how-can-i-use-binary-heap-in-the-dijkstra-algorithm), you will want to keep an additional hashmap from the key to the corresponding node of the list, to find which node to update quickly.