# Heaps

1

2

3

4

5

6

8

9

7

A heap, or a maxheap, is a binary tree with two additional properties: 1) each node is larger than either of its children and 2) it is complete.

(A minheap is a heap in which each node is smaller than its children.)

Strangely enough, we do not code heaps using TreeNode. Rather, we represent the heap as an array! We do not use the zero cell. This way, any arbitrary node k has children in locations 2k and 2k+1, and a parent in k/2. Make sure these formulas work.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 99 | 80 | 85 | 17 | 30 | 84 | 2 | 16 | 1 |  |  |  |
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] | [11] | [12] |

Why must the heap be a *complete* binary tree? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_So there are no gaps in the array

# Exercises

1. Which of the following trees is a heap? C and E

b.

f.

e.

d.

c.

a.

2. Which of the following arrays have the heap property?

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 99 | 88 | 66 | 44 | 33 | 55 | 77 | 33 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

a.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 99 | 88 | 77 | 66 | 55 | 44 | 33 | 22 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

b.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 99 | 88 | 44 | 77 | 22 | 33 | 55 | 66 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

c.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 99 | 88 | 66 | 77 | 22 | 33 | 44 | 55 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

d.

# heapDown

1

2

3

4

5

6

8

9

7

10

1

2

3

4

5

6

8

9

7

10

# heapUp and heapDown are the two basic heap operations. Whenever you add to or remove from a heap, the new item has to rise or fall to its proper place so that the heap is kept in heap order. Each method works in O(log n).

# Let's do heapDown. For variety, we will use char data.

# Let’s just put in a new value at the root, for example, put in 'b'. heapDown swaps, if necessary, that value with its largest child, and keeps swapping, until the value is in place, i.e, the heap order has been restored. Trace the movement if you put a new 'b' at the root:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [0] | [1] | [2] | [3] | [4] | [5] | [6] | [7] | [8] | [9] | [10] |  |  |  |  |
| null | 'b' | e | d | c | d | a | c | a | b | b | insert ‘b’ | | | |
|  |  |  |  |  |  |  |  |  |  |  | swap with largest child | | | |
|  |  |  |  |  |  |  |  |  |  |  | swap with largest child | | | |

# Go back to the original heap. Insert an 'f' at the root. What happens? There is no swapping, because 'f' is larger than its children; it has found its proper place, which is one base case. Going off the end of the array is another base case.

# The heapDown algorithm is easy to say, but several things are going on: swap the current value with its largest child, then recur. Don’t go off the end of array. If it helps, draw a flowchart. Then write pseudocode.

# If the children are null, less than the value or equal to the value, Stop parsing

# If the left child is larger than the right, swap values and recur for the new array

# If the right child is larger than the right swap values and recur for the new array

# 

# The header below is for the recursive version. (heapDown can also be done iteratively.) Assume that you have a working swap method. lastIndex is what it says. We are passing it here because later, in heapSort, lastIndex will be reduced by 1 during each recursive call.

# private static void heapDown(double[] array, int k, int lastIndex)