Heapsort

All the three algorithms we have used so far didn't require any specific data structure (https://www.codesdope.com/blog/tag/data-strucutre/?tag=data-strucutre) but heapsort does. It is implemented using heaps (https://www.codesdope.com/blog/article/heap-binary-heap/). As stated earlier in this course, some of the algorithms we will study in this course will be implemented using some data structures. So, it is recommended to read articles on data structure (https://www.codesdope.com/blog/tag/data-strucutre/?tag=data-strucutre) first. However, if you are familiar with data structures, then you are good to go.

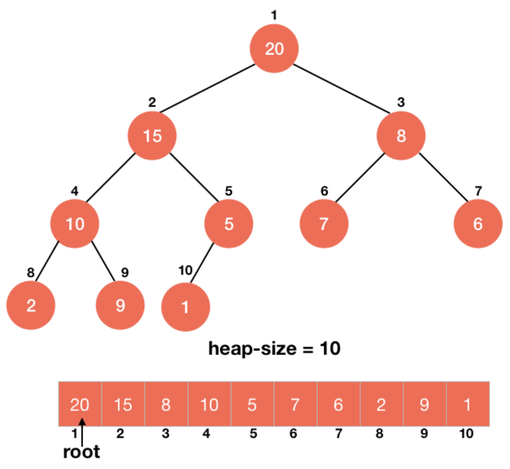
Heaps Recap

- Heaps are complete binary trees implemented using an array
- Parent of a node i is $\lfloor \frac{i}{2} \rfloor$
  Left child of the node i is $2*i$
  Right child of the node i is $(2*i)+1$
- Max-heap property → The value of a node is greater than or equal to the value of its children i.e., $A[Parent[i]] \geq A[i]$ for all nodes i>1.
  Min-heap property → The value of a node is either smaller or equal to the value of its children i.e., $A[Parent[i]] \leq A[i]$ for all nodes i>1.
- HEAPIFY is a function used to maintain the heap property of any heap. It is applied to a node when the children of the node are following the property of a heap but the node itself may be violating it. It runs in $O(\lg n)$ time.
- BUILD-HEAP is a function used to make a heap from an array. It runs in $O(n)$ time.

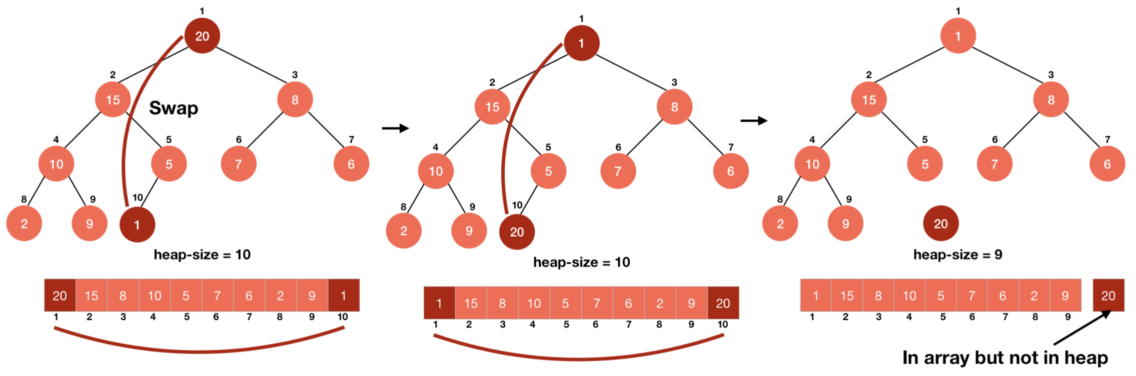
Let's start with heapsort

Working of Heapsort

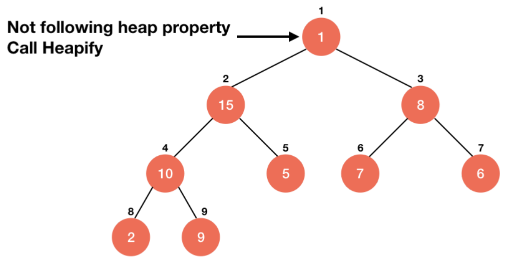
Heapsort is implemented using a max-heap. In a max-heap, the maximum element is at the root of the tree and is also the first element of the array i.e., A[1].

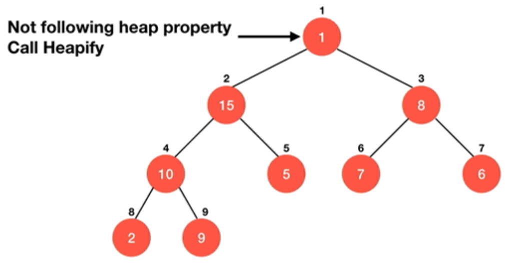


Now, we want this maximum element to be at the last element of our sorted array. So, we swap it with the last element and then discard this element from the heap by reducing the size of the heap.

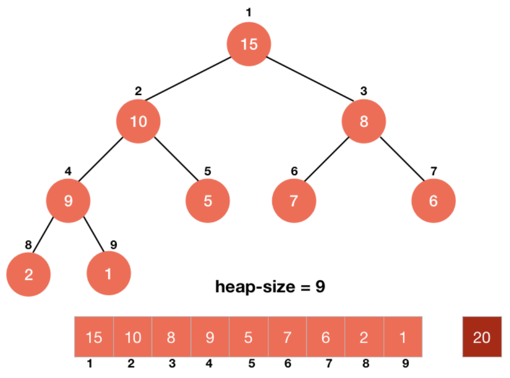


So, we have correctly put the largest element at the correct position, but in doing so, we have disturbed our heap or more precisely, the root of the heap because we haven't touched any child of the root yet. Thus they are still following the max-heap property.

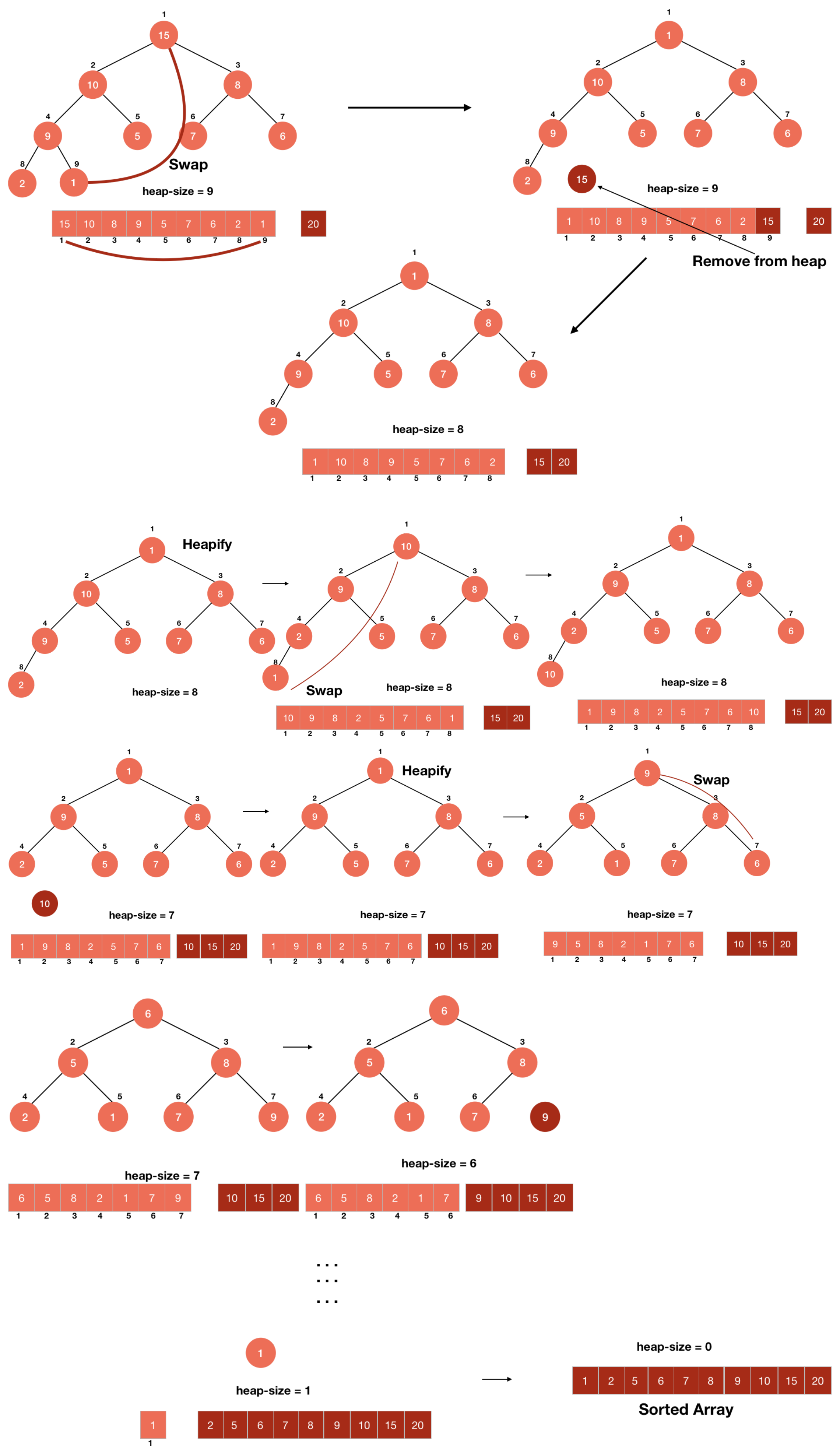


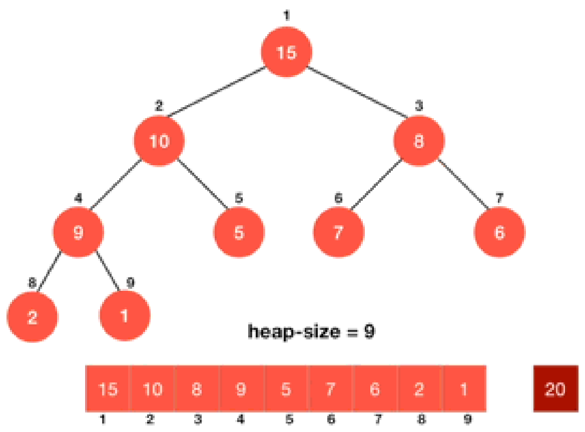


We can easily deal with this problem by calling HEAPIFY on the root of the tree and that will make our tree a max-heap again, but this time the largest element won't be the part of the heap.



Now, we will just repeat this process until every element of the heap is correctly placed in sorted order.


Swap
heap-size = 9
15 10 8 9 5 7 6 2 1
20
heap-size = 9
1 10 8 9 5 7 6 2 15
20
Remove from heap
heap-size = 8
1 10 8 9 5 7 6 2
15 20
Heapify
heap-size = 8
Swap
heap-size = 8
10 9 8 2 5 7 6 1
15 20
heap-size = 8
1 9 8 2 5 7 6 10
15 20
heap-size = 7
1 9 8 2 5 7 6
10 15 20
Heapify
heap-size = 7
1 9 8 2 5 7 6
10 15 20
Swap
heap-size = 7
9 5 8 2 1 7 6
10 15 20
heap-size = 7
6 5 8 2 1 7 9
10 15 20
heap-size = 6
6 5 8 2 1 7
9 10 15 20
. . .
. . .
. . .
heap-size = 1
1
2 5 6 7 8 9 10 15 20
heap-size = 0
1 2 5 6 7 8 9 10 15 20
Sorted Array



Now you have understood the concepts behind the heapsort, so let's write a code for the heapsort.

Coding Heapsort

We can easily implement the above logic in a code. We have to swap the first element of the array with the last element of the heap till the heap exists or the size of the heap is not 0.

```
while A.heap_size > 0
   swap(A[1], A[heap_size])
```

After this, we just have to decrease the size of the heap by 1 to exclude the last element (largest element) form the heap.

```
A.heap_size = A.heap_size-1
```

The last part is to make the array a max-heap again by calling MAX-HEAPIFY function on the root because only the root is disturbed in the entire process.

```
MAX-HEAPIFIY(A, 1)
```

</>
Take note that we are using MAX-HEAPIFY function of a heap which is explained in the data structure course.

Summing up the above code blocks, we can develop the entire code for the heapsort as:

```
HEAPSORT(A)
  while A.heap_size > 0
     swap(A[1], A[A.heap_size])
     A.heap_size = A.heap_size-1
     MAX-HEPAPIFY(A, 1)
```

C **Python** **Java**

```
#include <stdio.h>

int tree_array_size = 11;
int heap_size = 10;

void swap( int *a, int *b ) {
  int t;
  t = *a;
  *a = *b;
  *b = t;
}

//function to get right child of a node of a tree
int get_right_child(int A[], int index) {
  if((((2*index)+1) < tree_array_size) && (index >= 1))
    return (2*index)+1;
  return -1;
}

//function to get left child of a node of a tree
int get_left_child(int A[], int index) {
    if(((2*index) < tree_array_size) && (index >= 1))
        return 2*index;
    return -1;
}

//function to get the parent of a node of a tree
int get_parent(int A[], int index) {
  if ((index > 1) && (index < tree_array_size)) {
    return index/2;
  }
  return -1;
}

void max_heapify(int A[], int index) {
  int left_child_index = get_left_child(A, index);
  int right_child_index = get_right_child(A, index);

  // finding largest among index, left child and right child
  int largest = index;

  if ((left_child_index <= heap_size) && (left_child_index>0)) {
    if (A[left_child_index] > A[largest]) {
      largest = left_child_index;
    }
  }

  if ((right_child_index <= heap_size && (right_child_index>0))) {
    if (A[right_child_index] > A[largest]) {
      largest = right_child_index;
    }
  }

  // largest is not the node, node is not a heap
  if (largest != index) {
    swap(&A[index], &A[largest]);
    max_heapify(A, largest);
  }
}

void build_max_heap(int A[]) {
  int i;
  for(i=heap_size/2; i>=1; i--) {
    max_heapify(A, i);
  }
}

void heapsort(int a[]) {
  while(heap_size > 0) {
    swap(&a[1], &a[heap_size]);
    heap_size = heap_size-1;
    max_heapify(a, 1);
  }
}

int main() {
  //tree is starting from index 1 and not 0
```

?

```
  int A[] = {0, 15, 20, 7, 9, 5, 8, 6, 10, 2, 1};
  build_max_heap(A);
  heapsort(A);

  //printing
  int i;
  for(i=1; i<=10; i++) {
    printf("%d\n",A[i]);
  }
  return 0;
}
```

(/add_ques

Analysis of Heapsort

The analysis of the code is simple. There is a while loop which is running n times and each time it is executing 3 statements. The first two statements (`swap(A[1], A[A.heap_size])` and `A.heap_size = A.heap_size-1`) will take a constant time but the last statement i.e., `MAX-HEPAPIFY(A, 1)` is going to take $O(\lg n)$ time. So, the algorithm is going to take a total of $O(n \lg n)$ time.

</>
One can also argue that the analysis should also include the time of building a heap, assuming that the array we are provided is not a max-heap. Then the heapsort algorithm will first make a call to `BUILD-MAX-HEAP(A)` to make a max-heap and then perform the rest of the tasks. In that case also, the heapsort will take $O(n \lg n)$ time because BUILD-MAX-HEAP will take $O(n)$ time which will be dominated by $O(n \lg n)$.

So, that's it for the sorting algorithms. You can always check the further readings and BlogsDope sorting tag (https://www.codesdope.com/blog/tag/sort/?tag=sort) to read more about sorting. If you don't want to miss any new article from us, consider checking our BlogsDope app (https://play.google.com/store/apps/details?id=com.blogsdope&hl=en_IN).

" Tell me and I forget. Teach me and I remember. Involve me and I learn. "

- Benjamin Franklin

PREV (/course/algorithms-count-and-sort/) (/course/algorithms-divide-and-conquer/) NEXT

# Further Readings

- Trees in Computer Science (/blog/article/trees-in-computer-science/)
- Binary Trees (/blog/article/binary-trees/)
- Binary Trees in C : Array Representation and Traversals (/blog/article/binary-trees-in-c-array-representation-and-travers/)
- Binary Tree in C: Linked Representation & Traversals (/blog/article/binary-tree-in-c-linked-representation-traversals/)
- Binary Tree in Java: Traversals, Finding Height of Node (/blog/article/binary-tree-in-java-traversals-finding-height-of-n/)
- Binary Search Tree (/blog/article/binary-search-tree/)

?