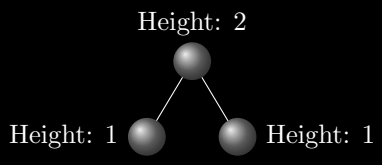
In-place CartesianTreeSort variant

Francisco Alejandro Arganis Ramírez

NOTE. The following conventions are used:

- Array indices start at 1
  The valid range of an index i for an array with size n is $1 \leq i \leq n$.

- Node heights start at 1
  The height of a node is the number of nodes in the longest downward path to a leaf from that node.



  The height of a tree its the height of its root.

`CartesianTreeSort` is an adaptive sorting algorithm with $O(n \log n)$ time and $O(n)$ space worst case complexities. The standard algorithm uses a min cartesian tree, which is an implicit binary tree placed in an array with the two additional properties

1. The tree is also a min heap.
2. An in-order traversal of the nodes yields the values in the same order in which they appear in the array.

`CartesianTreeSort` works by first building the min cartesian tree and then successively finding the next smallest value. This idea can be adapted to be done in-place, but the improvement in memory usage results in an non-adaptive algorithm.

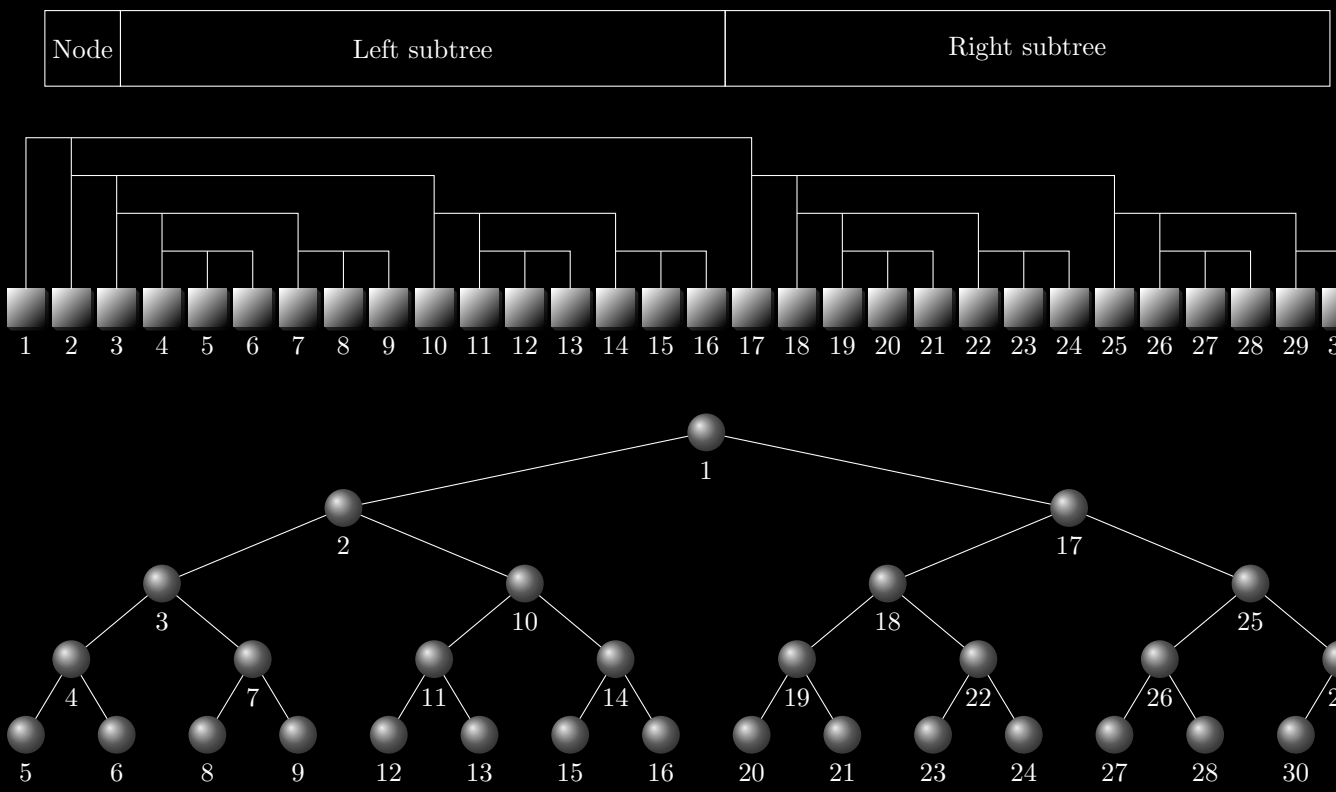
The variant, refered to as `InplaceCartesianTreeSort`, uses a modified min cartesian tree, defined as follows: A min left oriented cartesian tree (loct) is an implicit binary tree placed in an array with the two additional properties

1. The tree is also a min heap.
2. An pre-order traversal of the nodes yields the values in the same order in which they appear in the array.

Alternatively, a right oriented cartesian tree (roct) that satisfies the post-order property could be defined instead. In this case a max roct is required to sort the array.

`InplaceCartesianTreeSort` replicates the behavior of `CartesianTreeSort`, by first building a min loct and then successively finding the next smallest value in total $O(n \log n)$ time and $O(1)$ space worst case. However, for almost-sorted arrays `CartesianTreeSort` will run in $O(n)$ time while `InplaceCartesianTreeSort` will still run in $O(n \log n)$.

The advantage of using pre-order over in-order is that node indices can always be directly calculated through a simple formula. The tree structure scheme for a loct and the corresponding index distribution on the implicit tree are shown below.



With this scheme a node at index i and height h has their left child at index $i + 1$ and right child at index $i + 2^{h-1}$.

`leftChild`. Receives an index i and returns the index of its left child.

```
leftChild (i)
begin
        return i + 1
end
```

`rightChild`. Receives an index i with height h and returns the index of its right child.

```
rightChild (i, h)
begin
        return i + (1 << (h - 1))
end
```

The min loct building requieres only $O(n)$ time, just like building a regular min cartesian tree, and it works in the same way as the regular `heapify` algorithm from `HeapSort`. First, the height H of the implicit tree is calculated.

`getMaxHeight`. Receives the size n of an array and returns the height of the implicit tree, that is, returns an integer H such that $2^{H-1} - 1 < n \leq 2^H - 1$.

```
getMaxHeight (n)
begin
        height <- floor (log2 (n))
        if (1 << height) = n + 1 do
                return height
        else do
                return height + 1
        end if
end
```

All nodes with height 1 are ignored as they are a valid loct by themselves. Then, for each height from 2 to H, the values at nodes with that height are sifted down.

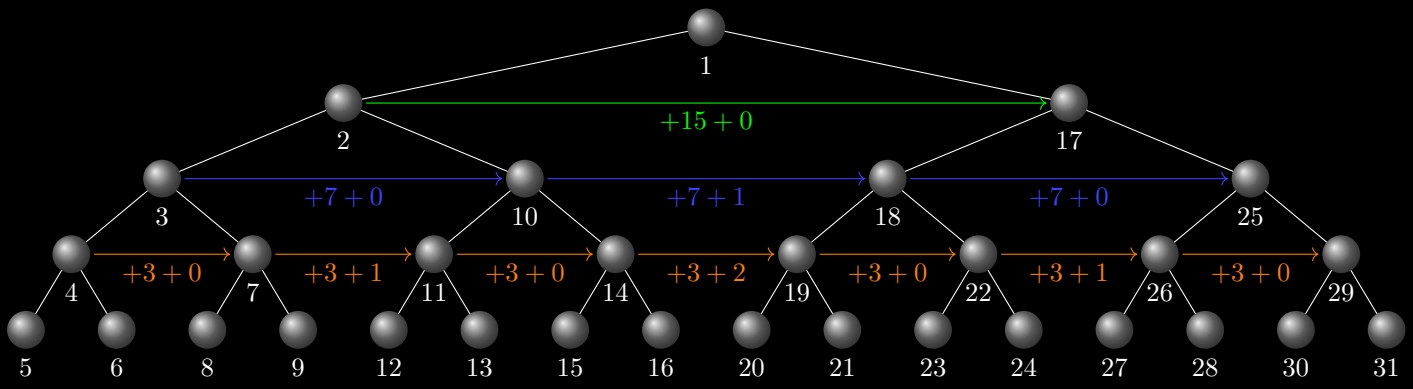
`tripleMin`. Receives an array a and three indices i, j, k and returns an index m such that $a[m] = \min\{a[i], a[j], a[k]\}$.

```
tripleMin (a, i, j, k)
begin
        min <- i
        if j <= a.size && a [j] < a [min] do
                min <- j
        end if
        if k <= a.size && a [k] < a [min] do
                min <- k
        end if
        return min
end
```

`minSiftDown`. Receives an array a and an index i with height h and sifts down $a[i]$ to its appropriate position in the min loct.

```
minSiftDown (a, i, h)
begin
        while h > 1 do
                left <- leftChild (i)
                right <- rightChild (i, h)
                min <- tripleMin (a, i, left, right)
                if min = i do
                        break
                end if
                a.swap (i, min)
                i <- min
                h <- h - 1
        end while
end
```

The difference of two consecutive nodes with the same height h is $r + 2^h$ where r is the number of roots between them with greater height. This can be calculated as $2^h - 1 + g$ where g is a gap given by the binary carry sequence (OEIS A007814). The first few values of g are $0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 5, \ldots$ and have a closed-form expression.



`binaryCarrySequence`. Receives an integer c and returns the exponent of highest power of 2 dividing c.

```
binaryCarrySequence (c)
begin
        return log2 (c - (c & (c - 1)))
end
```

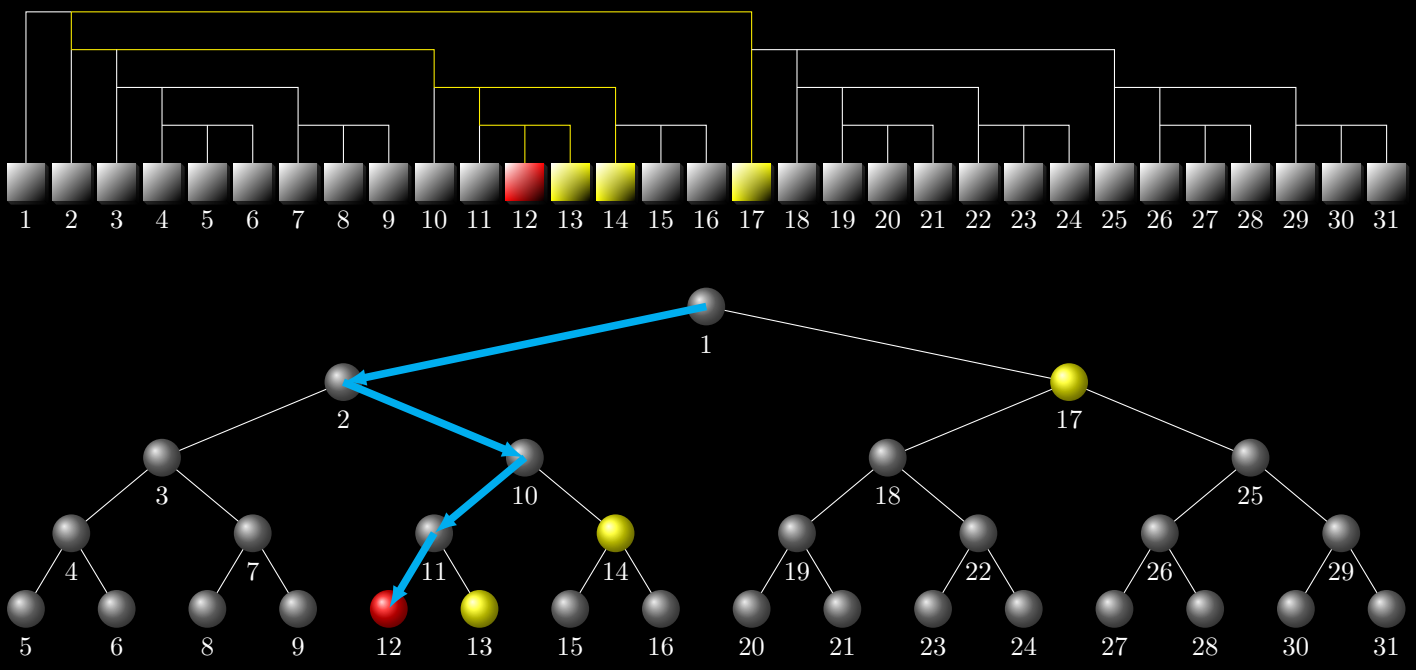
`buildMinLoct`. Receives an array a and the height of the implicit tree H and builds a min loct.

```
buildMinLoct (a, H)
begin
        height <- 2
        while height <= H do
                counter <- 1
                increment <- (1 << height) - 1
                node <- 1 + H - height
                while node < a.size do
                        minSiftDown (a, node, height)
                        node <- node + increment + binaryCarrySequence (counter)
                        counter <- counter + 1
                end while
        height <- height + 1
        end while
end
```

The chosen tree structure scheme converts a linear traversal of the region of any subtree in the array into a pre-order traversal of that subtree. Once the min loct is built, for each node visited during the traversal, the algorithm `findNextMin` compares the value at that index against the values at the roots of the largest possible right subtrees that appear later in the array. As there cannot be more than $\log n$ of these right subtrees, the operation takes $O(\log n)$.

If `findNextMin` finds the next minimum as the root of a right subtree, the minimum is extracted from its respective min loct and the original value is sifted down in $O(\log n)$. So the total time complexity of the traversal step is $O(n \log n)$.

Given an index i, the roots of the largest possible right subtrees after i are found by performing a binary-like search of i. Starting from the root, move to the right child j. If $j > i$ compare $a[i]$ against $a[j]$ and move to the left child. If $j < i$ then move to the right child. Repeat until $j = i$. This search keeps track of the height of the roots so that the sifting can be performed without additional calculations.



`findNextMin`. Receives an array a, an index i and the height of the implicit tree H and returns a pair (m, h) where m is an index of height h such that $a[m] \leq a[k]$ for all $k \geq i$.

```
findNextMin (a, i, H)

begin
        height <- H
        current_node <- 1
        m <- i
        h <- 0
        while current_node != i do
                next_node <- rightChild (current_node, height)
                if next_node > i do
                        if next_node <= a.size && array [next_node] < array [m] do
                                m <- next_node
                                h <- height - 1
                        end if
                        current_node <- leftChild (current_node)
                else do
                        current_node <- next_node
                end if
                height <- height - 1
        end while
        return (m, h)
end
```

`minLoctTraversal`. Receives an array a and the height of the implicit tree H and sorts it if the min loct has been built.

```
minLoctTraversal (a, H)
begin
        index <- 2
        while index < a.size
                (m, h) <- findNextMin (a, index, H)
                if m != index
                        a.swap (index, m)
                        minSiftDown (a, m, h)
                end if
                index <- index + 1
        end while
end
```

The pseudo-code for the complete algorithm is as follows.

`InplaceCartesianTreeSort`. Receives an array a and sorts it.

```
InplaceCartesianTreeSort (a)
begin
        H <- getMaxHeight (a.size)
        buildMinLoct (a, H)
        minLoctTraversal (a, H)
end
```

All the procedures `InplaceCartesianTreeSort` uses are iterative with constant memory requirements. As a result, the total space complexity is $O(1)$ worst case, just like `HeapSort`.