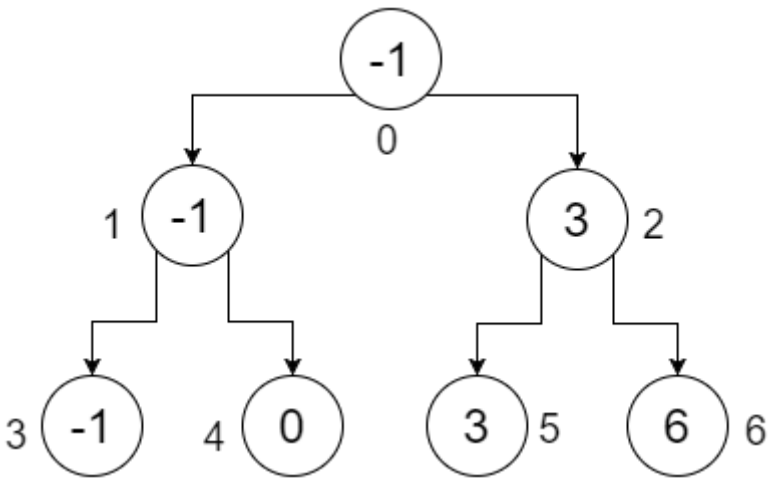
Implementation of Segment Tree Using Array

Example

Let's say, we have an array: `Item = {-1, 0, 3, 6}`. We want to construct **SegmentTree** array to find out the minimum value in a given range. Our segment tree will look like:



The numbers below the nodes show the indices of each values that we'll store in our **SegmentTree** array. We can see that, to store 4 elements, we needed an array of size 7. This value is determined by:

```
Procedure DetermineArraySize(Item):
multiplier := 1
n := Item.size
while multiplier < n
    multiplier := multiplier * 2
end while
size := (2 * multiplier) - 1
Return size
```

So if we had an array of length 5, the size of our **SegmentTree** array would be: (**8** * **2**) - **1** = **15**. Now, to determine the position of left and right child of a node, if a node is in index **i**, then the position of its:

Left Child is denoted by: **(2** * **i)** + **1**.
Right Child is denoted by: **(2** * **i)** + **2**.

And the index of the **parent** of any **node** in index **i** can be determined by: **(i** - **1)**/**2**.

So the **SegmentTree** array representing our example would look like:

```
   0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+
|  -1 |  -1 |  3  |  -1 |  0  |  3  |  6  |
+-----+-----+-----+-----+-----+-----+-----+
```

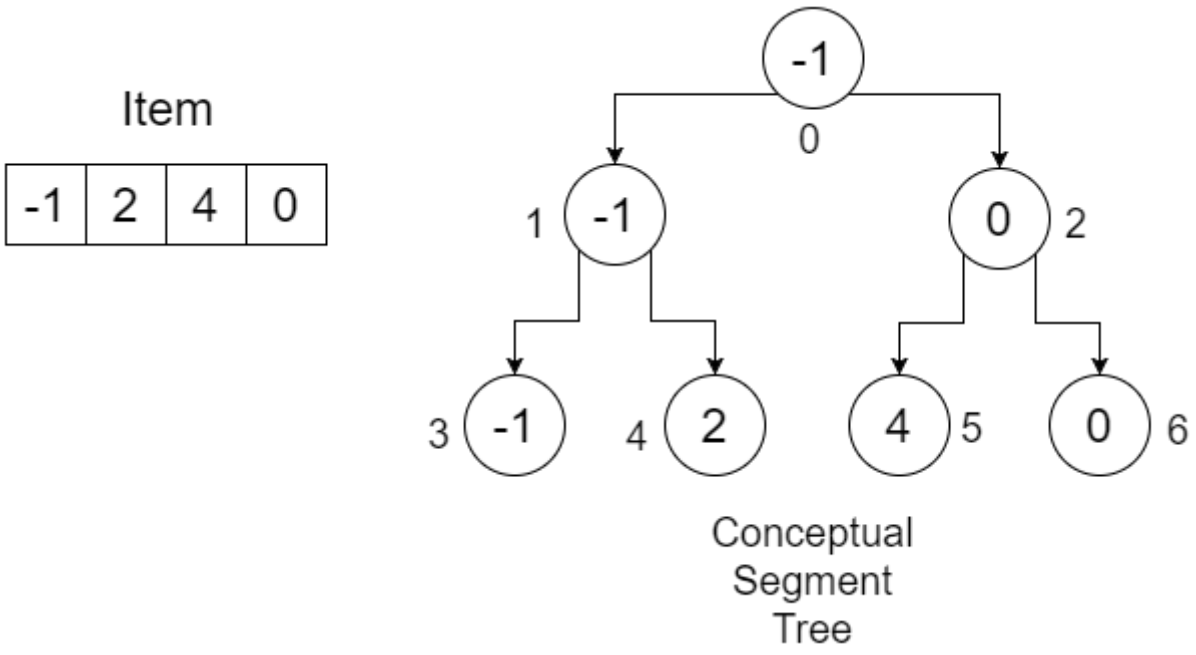
Let's look at the pseudo-code to construct this array:

```
Procedure ConstructTree(Item, SegmentTree, low, high, position):
if low is equal to high
    SegmentTree[position] := Item[low]
else
    mid := (low+high)/2
    constructTree(Item, SegmentTree, low, mid, 2*position+1)
    constructTree(Item, SegmentTree, mid+1, high, 2*position+2)
    SegmentTree[position] := min(SegmentTree[2*position+1], SegmentTree[2*position+2])
end if
```

At first, we take input of the values and initialize the **SegmentTree** array with `infinity` using the length of the **Item** array as its size. We call the the procedure using:

low = Starting index of **Item** array.
high = Finishing index of **Item** array.
position = 0, indicates the **root** of our Segment Tree.

Now, let's try to understand the procedure using an example:



The size of our **Item** array is **4**. We create an array of length **(4 * 2) - 1 = 7** and initialize them with `infinity`. You can use a very large value for it. Our array would look like:

```
   0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+
| inf | inf | inf | inf | inf | inf | inf |
+-----+-----+-----+-----+-----+-----+-----+
```

Since this is a recursive procedure, we'll see the operation of the `ConstructTree` using a recursion table that keeps track of `low`, `high`, `position`, `mid` and `calling line` at each call. At first, we call **ConstructTree(Item, SegmentTree, 0, 3, 0)**. Here, `low` is not same as `high`, we'll get a `mid`. The `calling line` indicates which `ConstructTree` is called after this statement. We denote the `ConstructTree` calls inside the procedure as **1** and **2** respectively. Our table will look like:

```
+-----+------+----------+-----+--------------+
| low | high | position | mid | calling line |
+-----+------+----------+-----+--------------+
|  0  |   3  |     0    |  1  |       1      |
+-----+------+----------+-----+--------------+
```

So when we call `ConstructTree-1`, we pass: `low = 0`, `high = mid = 1`, `position = 2*position+1 = 2*0+1 = 1`. One thing you can notice, that is `2*position+1` is the left child of **root**, which is **1**. Since `low` is not equal to `high`, we get a `mid`. Our table will look like:

```
+-----+------+----------+-----+--------------+
| low | high | position | mid | calling line |
+-----+------+----------+-----+--------------+
|  0  |   3  |     0    |  1  |       1      |
+-----+------+----------+-----+--------------+
|  0  |   1  |     1    |  0  |       1      |
+-----+------+----------+-----+--------------+
```

In the next recursive call, we pass `low = 0`, `high = mid = 0`, `position = 2*position+1 = 2*1+1=3`. Again the left child of index **1** is **3**. Here, `low` is e`high`, so we set `SegmentTree[position] = SegmentTree[3] = Item[low] = Item[0] = -1`. Our **SegmentTree** array will look like:

```
   0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+
| inf | inf | inf |  -1 | inf | inf | inf |
+-----+-----+-----+-----+-----+-----+-----+
```

Our recursion table will look like:

```
+-----+------+----------+-----+--------------+
| low | high | position | mid | calling line |
+-----+------+----------+-----+--------------+
|  0  |   3  |     0    |  1  |       1      |
+-----+------+----------+-----+--------------+
|  0  |   1  |     1    |  0  |       1      |
+-----+------+----------+-----+--------------+
|  0  |   0  |     3    |     |              |
+-----+------+----------+-----+--------------+
```

So you can see, **-1** has got its correct position. Since this recursive call is complete, we go back to the previous row of our recursion table. The table:

```
+-----+------+----------+-----+--------------+
| low | high | position | mid | calling line |
+-----+------+----------+-----+--------------+
|  0  |   3  |     0    |  1  |       1      |
+-----+------+----------+-----+--------------+
|  0  |   1  |     1    |  0  |       1      |
+-----+------+----------+-----+--------------+
```

In our procedure, we execute the `ConstructTree-2` call. This time, we pass `low = mid+1 = 1`, `high = 1`, `position = 2*position+2 = 2*1+2 = 4`. Our `calling line` changes to **2**. We get:

```
+-----+------+----------+-----+--------------+
| low | high | position | mid | calling line |
+-----+------+----------+-----+--------------+
|  0  |   3  |     0    |  1  |       1      |
+-----+------+----------+-----+--------------+
|  0  |   1  |     1    |  0  |       2      |
+-----+------+----------+-----+--------------+
```

Since, `low` is equal to `high`, we set: `SegmentTree[position] = SegmentTree[4] = Item[low] = Item[1] = 2`. Our **SegmentTree** array:

```
   0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+
| inf | inf | inf |  -1 |  2  | inf | inf |
+-----+-----+-----+-----+-----+-----+-----+
```

Our recursion table:

```
+-----+------+----------+-----+--------------+
| low | high | position | mid | calling line |
+-----+------+----------+-----+--------------+
|  0  |   3  |     0    |  1  |       1      |
+-----+------+----------+-----+--------------+
|  0  |   1  |     1    |  0  |       2      |
+-----+------+----------+-----+--------------+
|  1  |   1  |     4    |     |              |
+-----+------+----------+-----+--------------+
```

Again you can see, **2** has got its correct position. After this recursive call, we go back to the previous row of our recursion table. We get:

```
+-----+------+----------+-----+--------------+
| low | high | position | mid | calling line |
+-----+------+----------+-----+--------------+
|  0  |   3  |     0    |  1  |       1      |
+-----+------+----------+-----+--------------+
|  0  |   1  |     1    |  0  |       2      |
+-----+------+----------+-----+--------------+
```

We execute the last line of our procedure, `SegmentTree[position] = SegmentTree[1] = min(SegmentTree[2*position+1], SegmentTree[2*position+2]) = min(SegmentTree[3], SegmentTree[4]) = min(-1,2) = -1`. Our **SegmentTree** array:

```
   0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+
| inf |  -1 | inf |  -1 |  2  | inf | inf |
+-----+-----+-----+-----+-----+-----+-----+
```

Since this recursion call is complete, we go back to the previous row of our recursion table and call `ConstructTree-2`:

```
+-----+------+----------+-----+--------------+
| low | high | position | mid | calling line |
+-----+------+----------+-----+--------------+
|  0  |   3  |     0    |  1  |       2      |
+-----+------+----------+-----+--------------+
```

We can see that the left portion of our segment tree is complete. If we continue in this manner, after completing the whole procedure we'll finally get a completed **SegmentTree** array, that'll look like:

```
   0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+
|  -1 |  -1 |  0  |  -1 |  2  |  4  |  0  |
+-----+-----+-----+-----+-----+-----+-----+
```

The time and space complexity of constructing this **SegmentTree** array is: `O(n)`, where **n** denotes the number of elements in **Item** array. Our constructed **SegmentTree** array can be used to perform *Range Minimum Query(RMQ)*. To construct an array to perform *Range Maximum Query*, we need to replace the line:

```
SegmentTree[position] := min(SegmentTree[2*position+1], SegmentTree[2*position+2])
```

with:

```
SegmentTree[position] := max(SegmentTree[2*position+1], SegmentTree[2*position+2])
```

‹ Previous Next ›