Further Data Structures

- The story so far
  - Saw some fundamental operations as well as advanced operations on arrays, stacks, and queues
  - Saw a dynamic data structure, the linked list, and its applications.
  - Saw the hash table so that insert/delete/find can be supported efficiently.
  - Saw how hierarchical data can be stored in a tree along with applications.
- This week we will
  - Introduce a new operation : delete`Min` and its applications.
  - Propose data structures for an efficient `deleteMin`.

Motivation

- Consider a job scheduling environment
  - such as a printer facility
  - several jobs accepted
  - printed in a certain order. typically, first in first out.
- Another example could be an office
  - several files to be cleared.
  - Which ones are taken up first?
  - The order is not entirely FIFO.

Motivation

- So, need to store the jobs in a data structure so that
  - the next job according to priority can be located easily.
  - a new job can be added
- Let us consider existing data structures and their suitability.

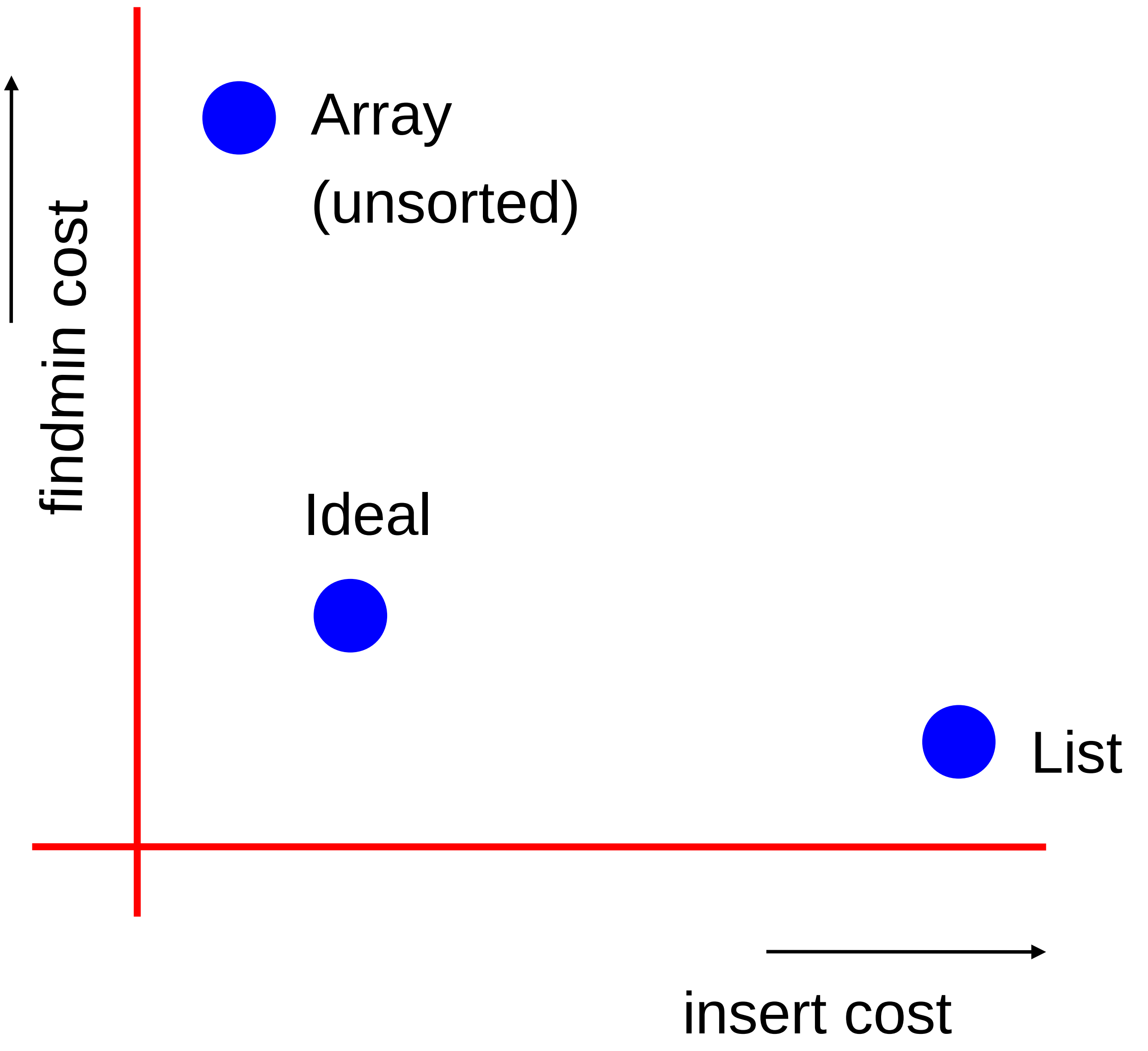
Use an Array/Queue

- Can store jobs in consecutive indices.
- Adding a new job is easy.
- But, deleting the job with the highest priority requires scanning the entire array/queue.
- Keeping the array sorted also does not help.
  - May use binary search to locate the required job.
  - But, deleting an element at some index requires moving all other elements.
- At any rate, cost of insert and delete the job with highest priority takes O(n) time for n jobs.

Use a Linked List

- Can store jobs in sorted order of priority.
- The job with the highest priority can be at the beginning of the list.
- Can be removed also easily.
- But, insert is costly. Need to scan the entire list to place the new job.
- Cost of insert = O(n), cost of finding the job with highest priority = O(1)

Comparing Solutions



Other Solutions?

- Can use a binary search tree?
- Use height balanced so that each operation is O(log n) only.
- However, may be a too elaborate solution when we require only a few operations.
- Our solution will be based on trees but with some simplification.

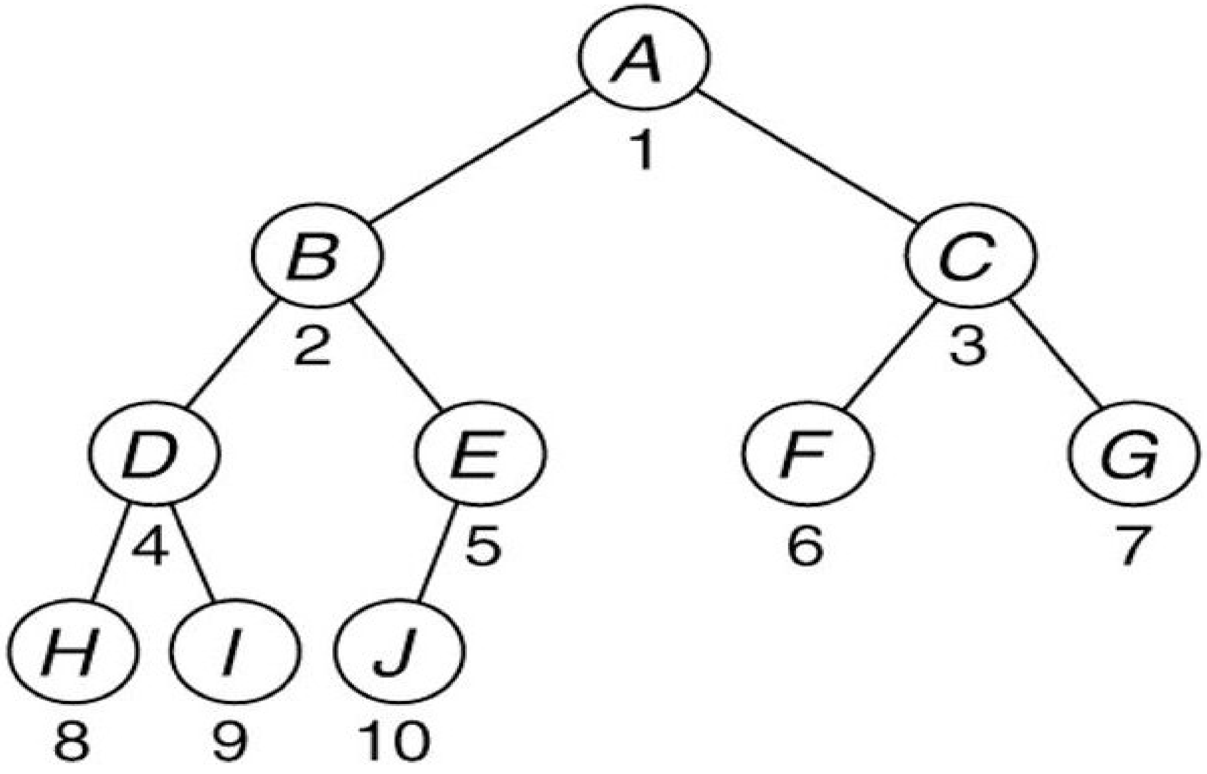
A New Solution

- Let us assume that a smaller value indicates a larger priority.
  - like nice value in Unix.
- First, we will say that our operations are:
  - insert(x) : insert a new item with a given priority
  - deleteMin() : delete the item with the highest priority.
- We will use a binary tree with the following property.

Our Solution

- Consider a binary tree of n nodes where the leaf nodes are spread across at most two levels.
- This also suggests that the depth of the tree is at most ceil(log n).
- Another way to say the above : the tree is completely occupied by nodes with the exception of the last level where nodes are filled from left to right.

Our Solution

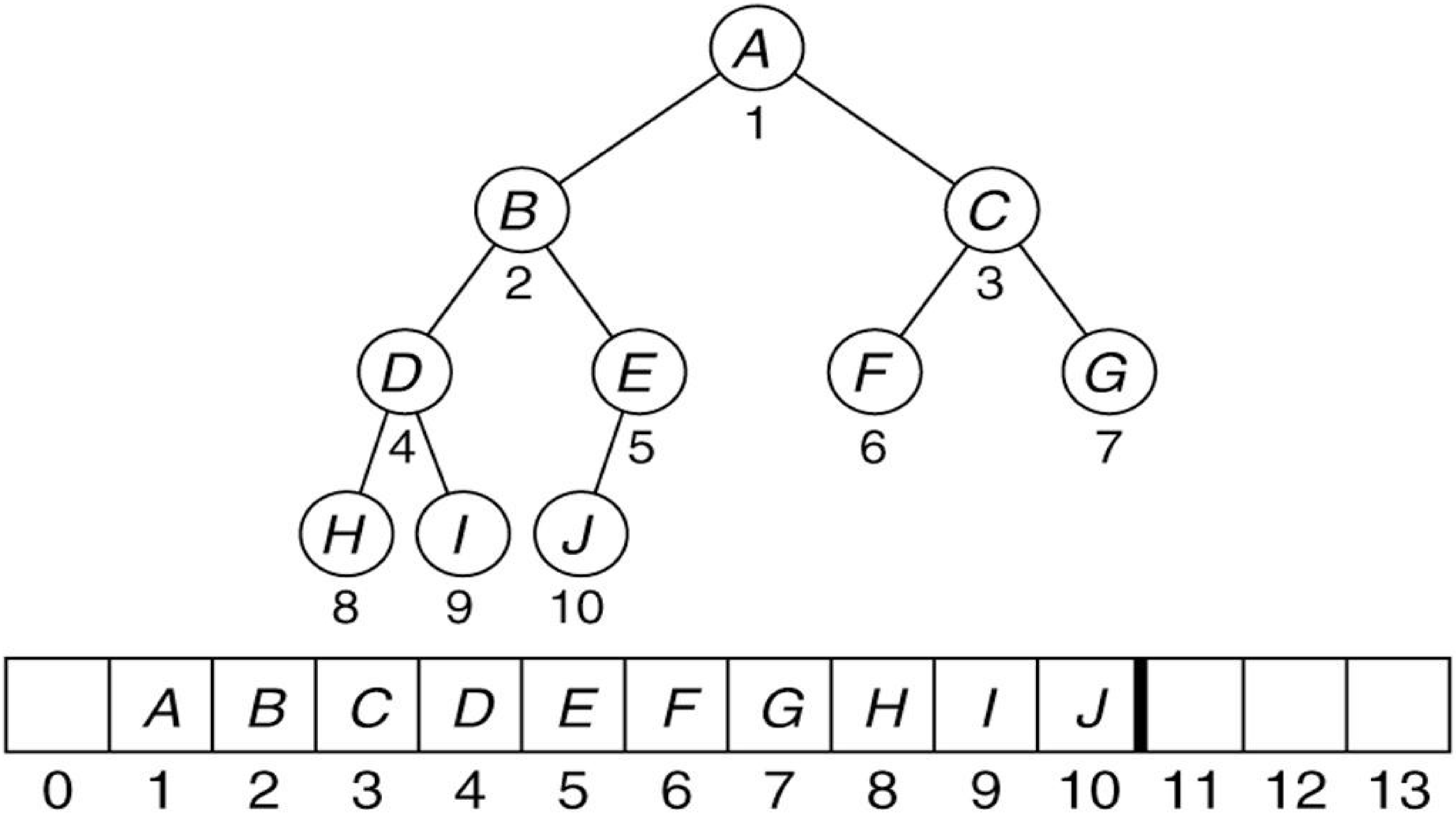


- Such a tree is also called as a complete binary tree.
- Thus, a complete binary tree of height h has between 2^h and $2^{h+1} – 1$ nodes.

Our Solution

- Instead of using a tree, we can use an array to represent such a complete binary tree.
  - Alike level order traversal..
- The root is stored at index 1.
- Its children are stored at indices 2 and 3.
- In general, the children of node at index k are stored at indices 2k and 2k+1.

An Example



Our Solution

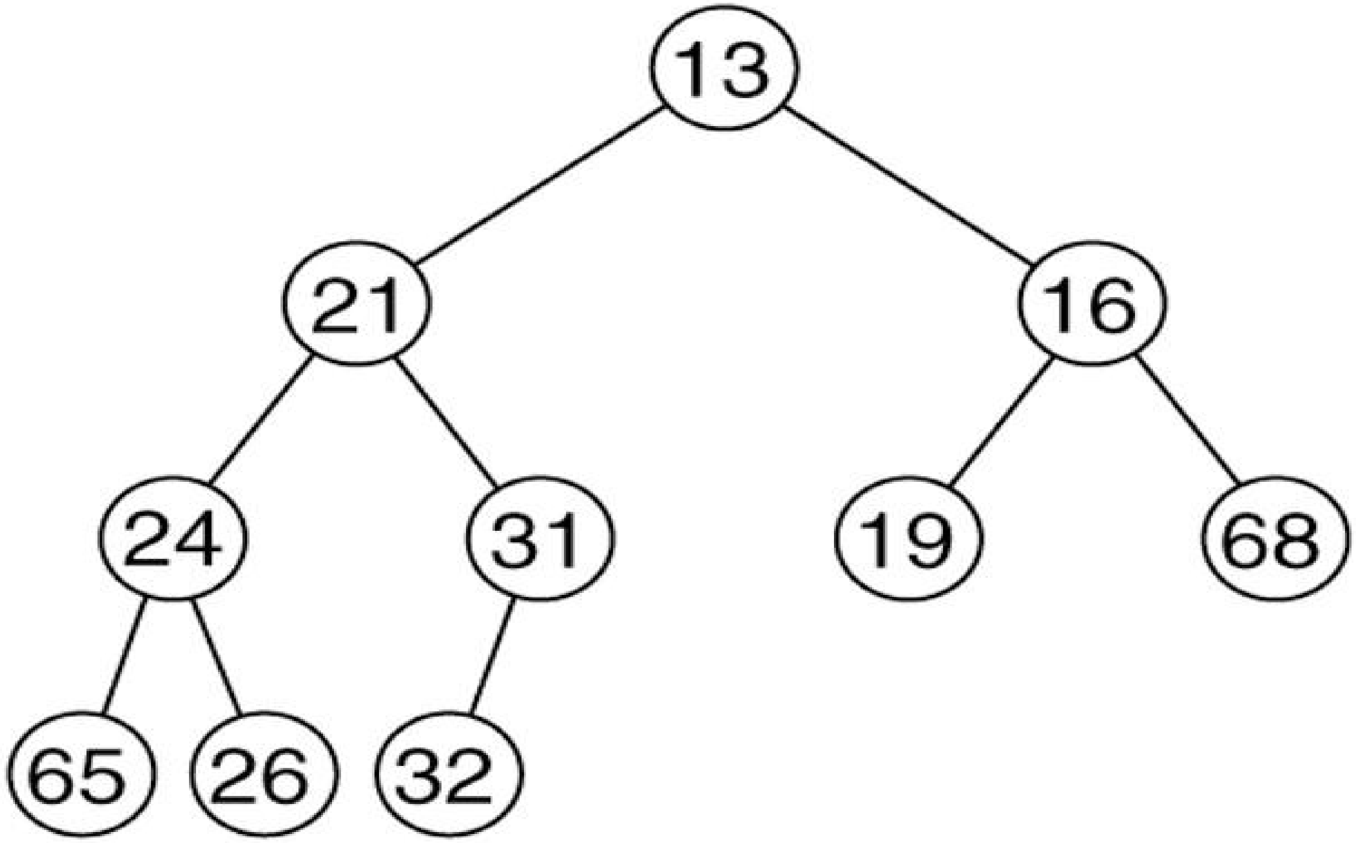
- Now, a tree alone is not enough.
- Need to have some order in the items.
- We propose the following property.

```
The value at any node will be smaller
  than the value of its children.
```

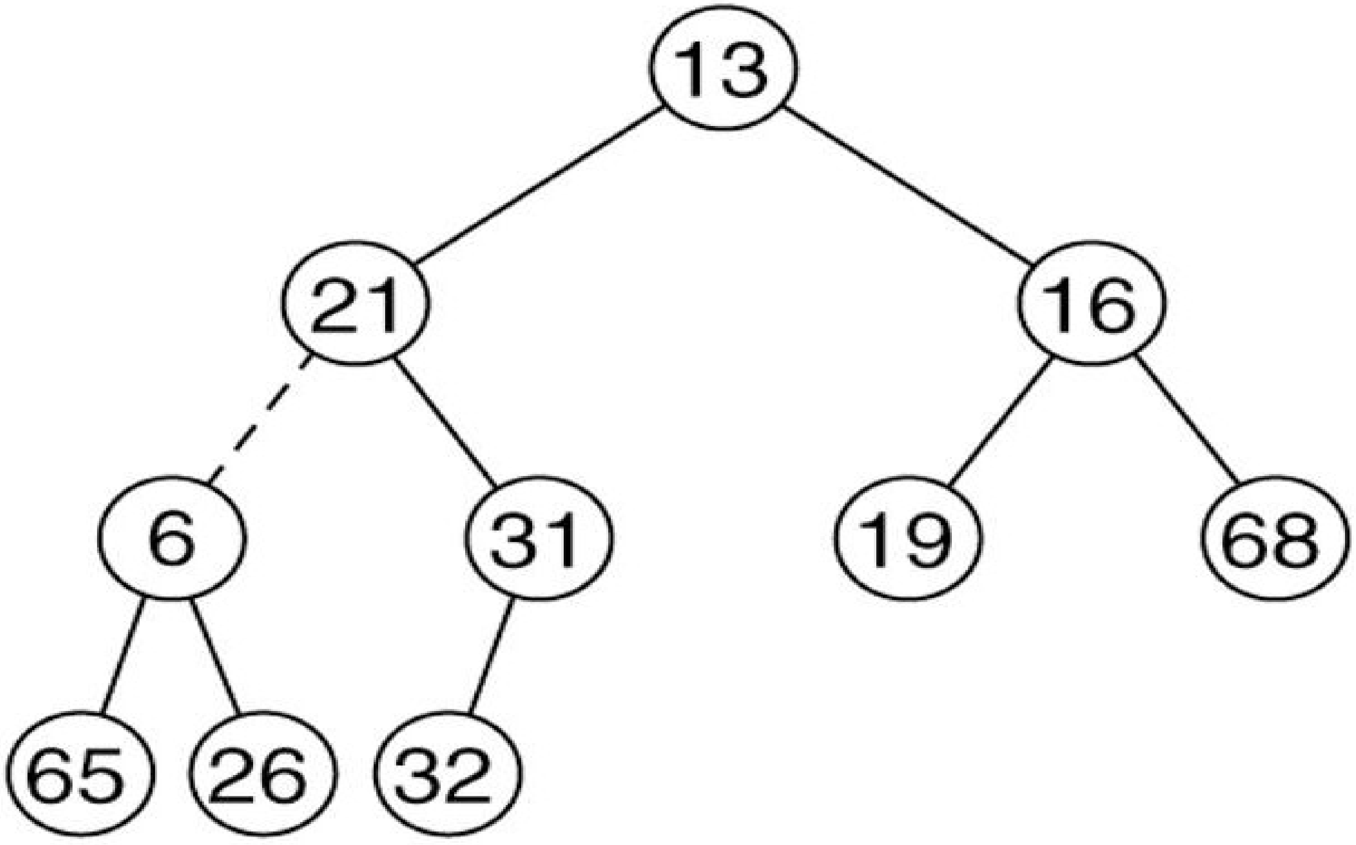
Our Solution

- We will call our data structure as a heap.
- The property has the following intuition.
  - It is reasonable to expect that the root contains the minimum element.
  - Each subtree of the root could also be viewed as our data structure, we can also expect that the root of the left subtree contain the minimum element among the elements in the left subtree, and
  - The root of the right subtree contain the minimum element among the elements of the right subtree.

Example



Example not a Heap



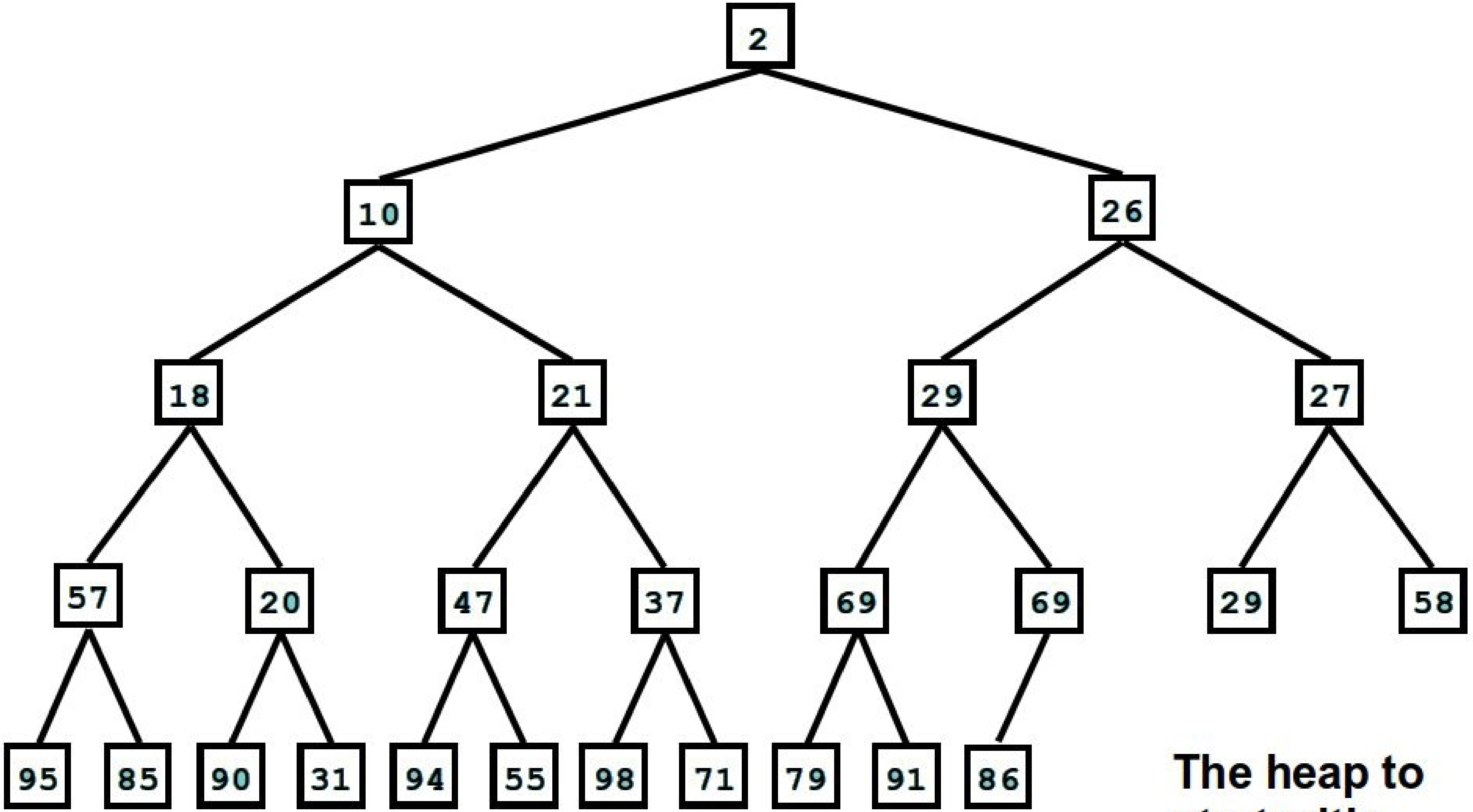
Our Solution

- while implementing our operations, we have to
  - keep the structure as a complete binary tree and
  - maintain the heap property
- Let us see how to implement the operations.

Operation `Insert(x)`

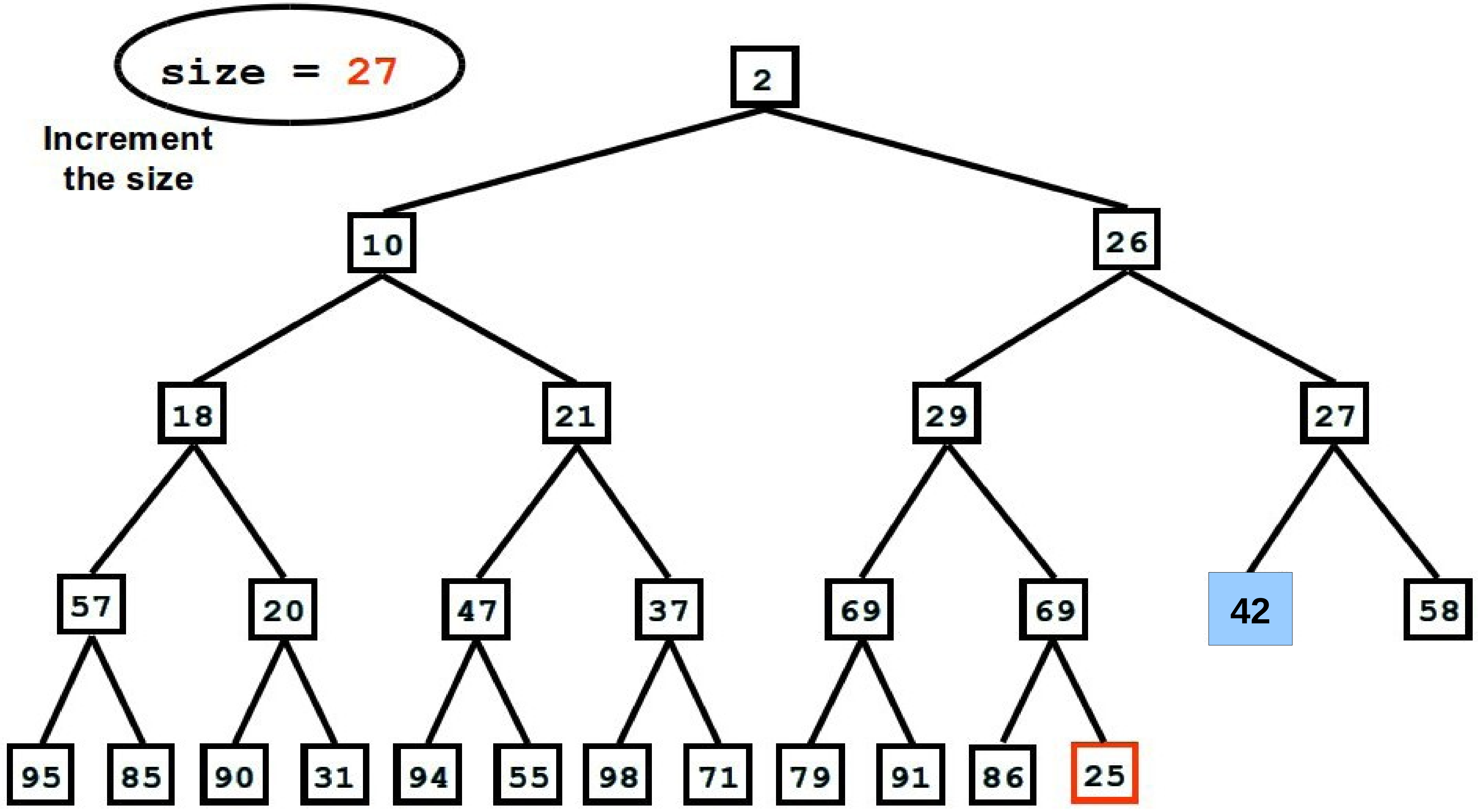
- We have to satisfy the complete binary tree.
- So, add the new item as the rightmost leaf.

Heap at Present



The heap to start with...

Inserting 25 – Step 1



Insert 25

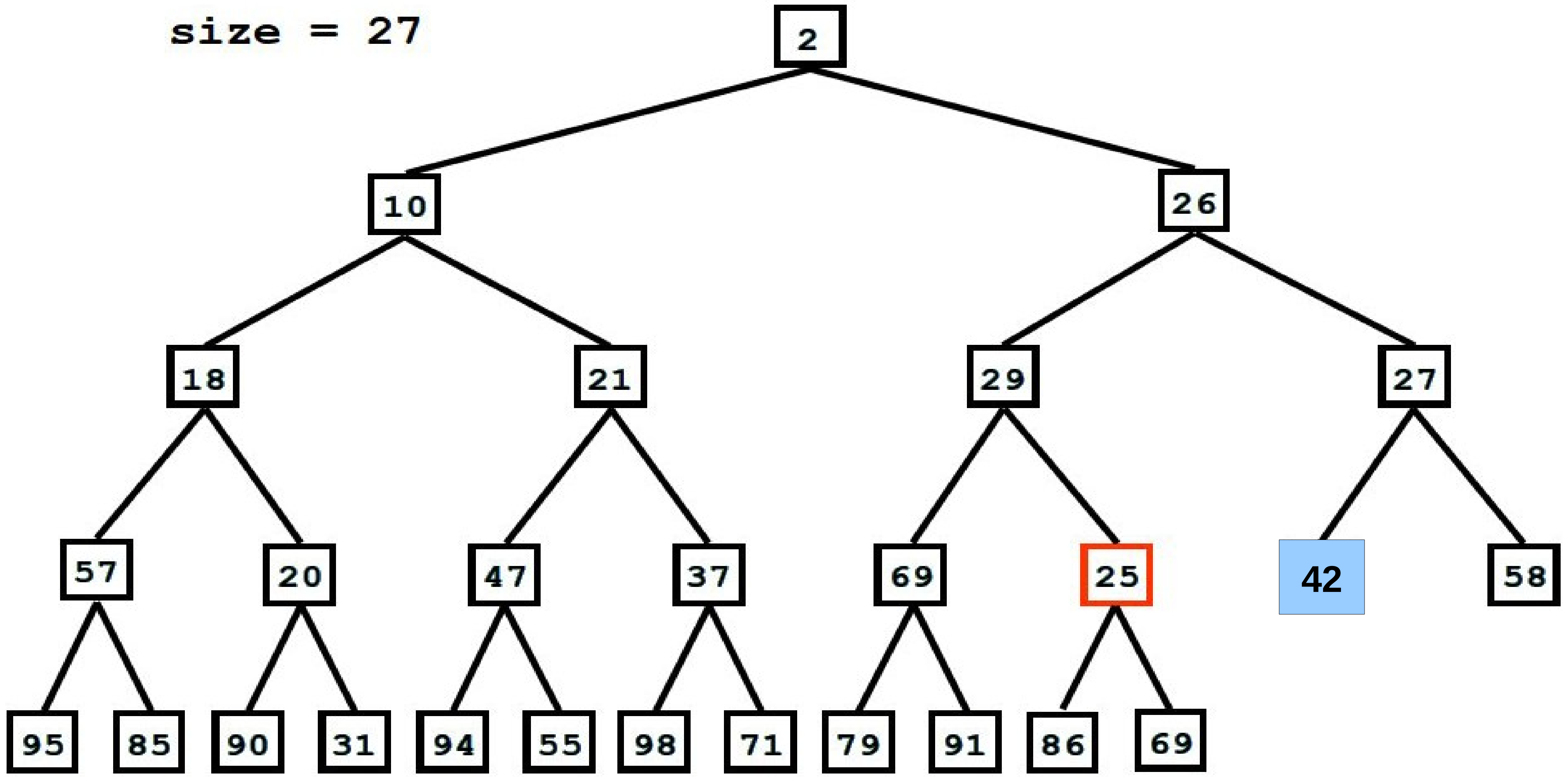
Insert(x) – Step 2

- Step 1 may not satisfy heap property.
- To restore the heap property, we have to push the inserted element towards the root.
- This process is called as ShuffleUp.

Shuffle Up

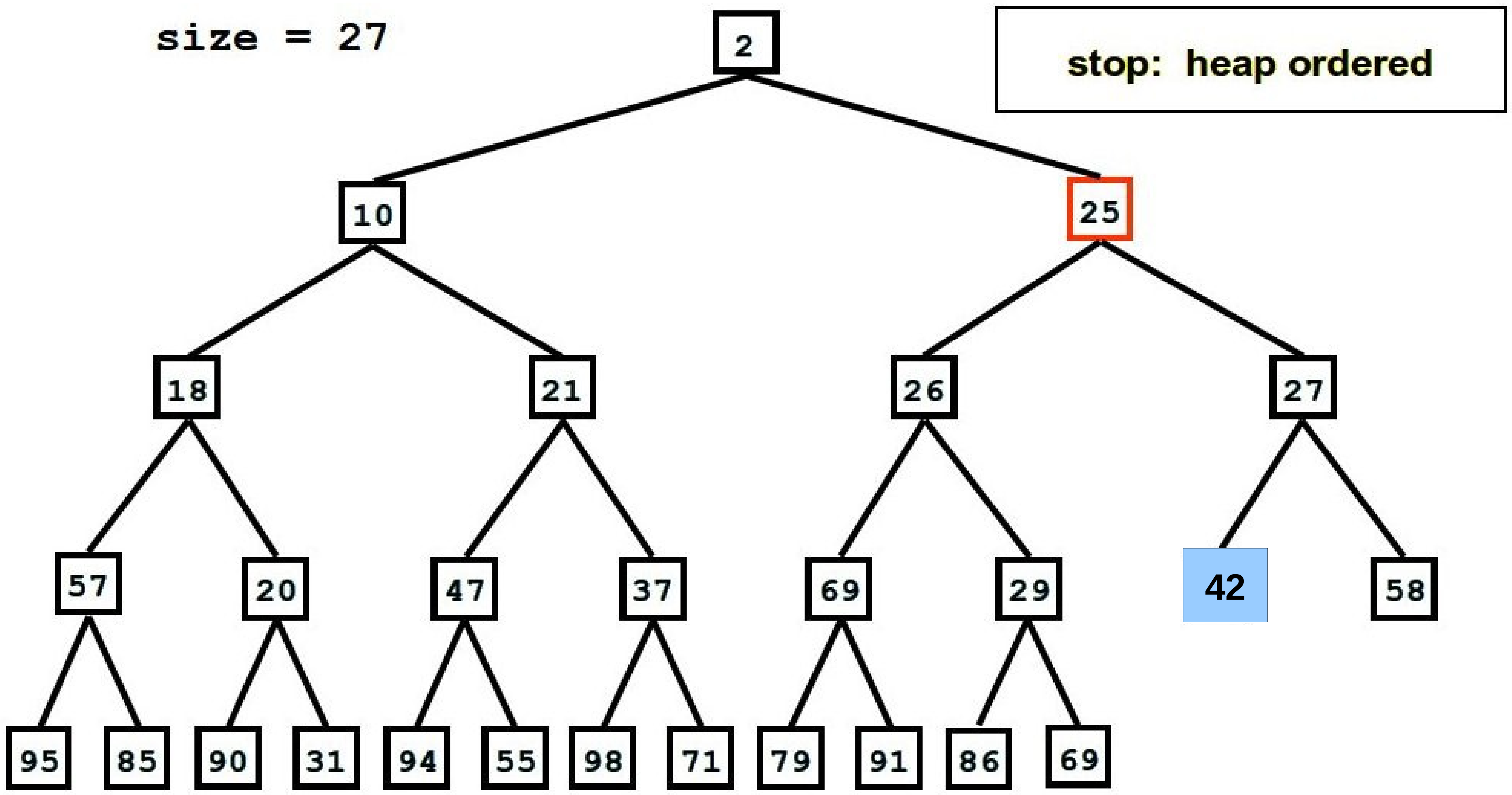
- In the Shuffleup operation, the value of a position is compared with that the value of its parent.
- If the value is smaller than its parent, then they are swapped.
- This is done until either value is bigger than the value at its parent or we have reached the root.

Shuffle Up in Example



Insert 25

Shuffle Up Example



Procedure Insert(x)

```
Procedure Insert(x)
begin
   H(k + 1) = x; i = k + 1;
   done = false;
   while i != 1 and not done
   begin
      parent = i/2;
      if H(parent) > H(i) then
         swap(H(parent);H(i));
         i = parent;
      else done = true;
   end-while
end
```

Procedure DeleteMin()

- Recall that in a heap, the minimum element is always available at the root.
- The difficult part is to physically delete the element.
  - Lazy deletion is not an option here.

Operation DeleteMin()

- Since the number of elements after deletion has to go down by 1, the element placed at index n presently has to be relocated eleswhere.

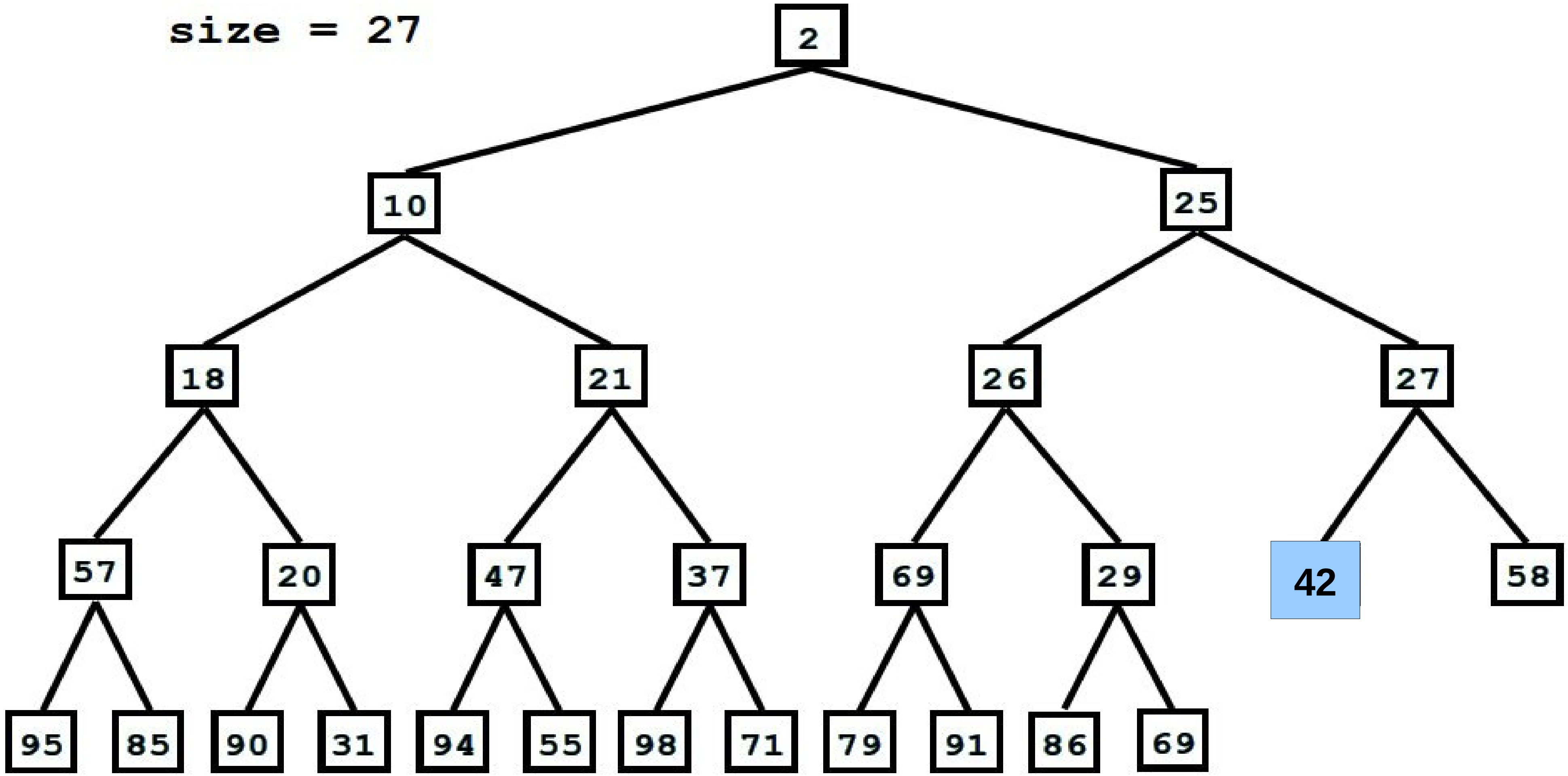
Operation DeleteMin()

- This is done by first relocating it to the root, satisfying the complete binary tree property.
- This relocation may however violate the heap property.
- To restore the heap property, we perform what is known as Shuffle-Down.

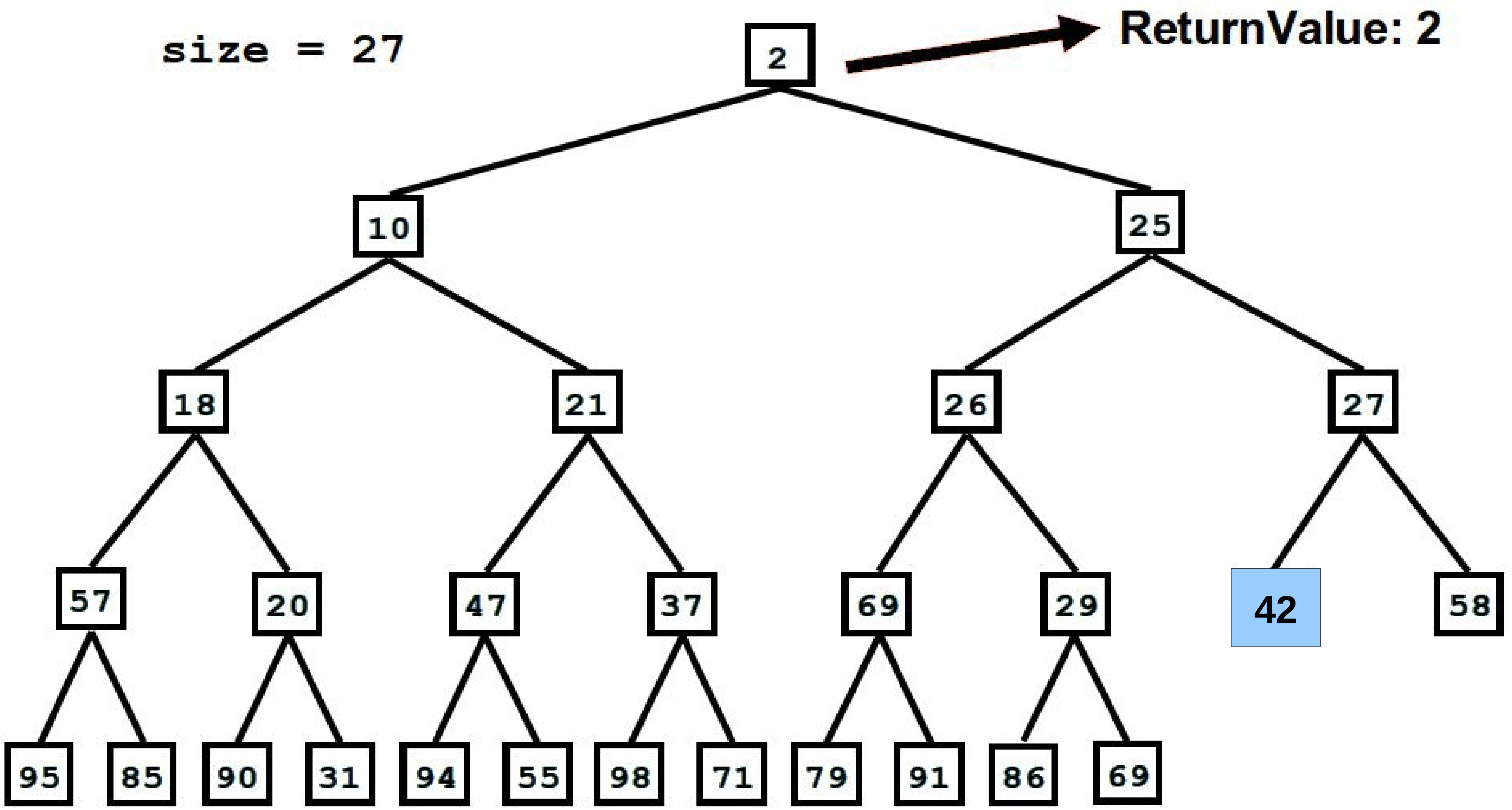
Shuffle Down

- In Shuffle-Down, analogous to Shuffle-Up, we move the element down the tree
- until its value is at most the value of any of its children. Every time the value at a node X is bigger than the
- value of either of its children, the smallest child and X are swapped.

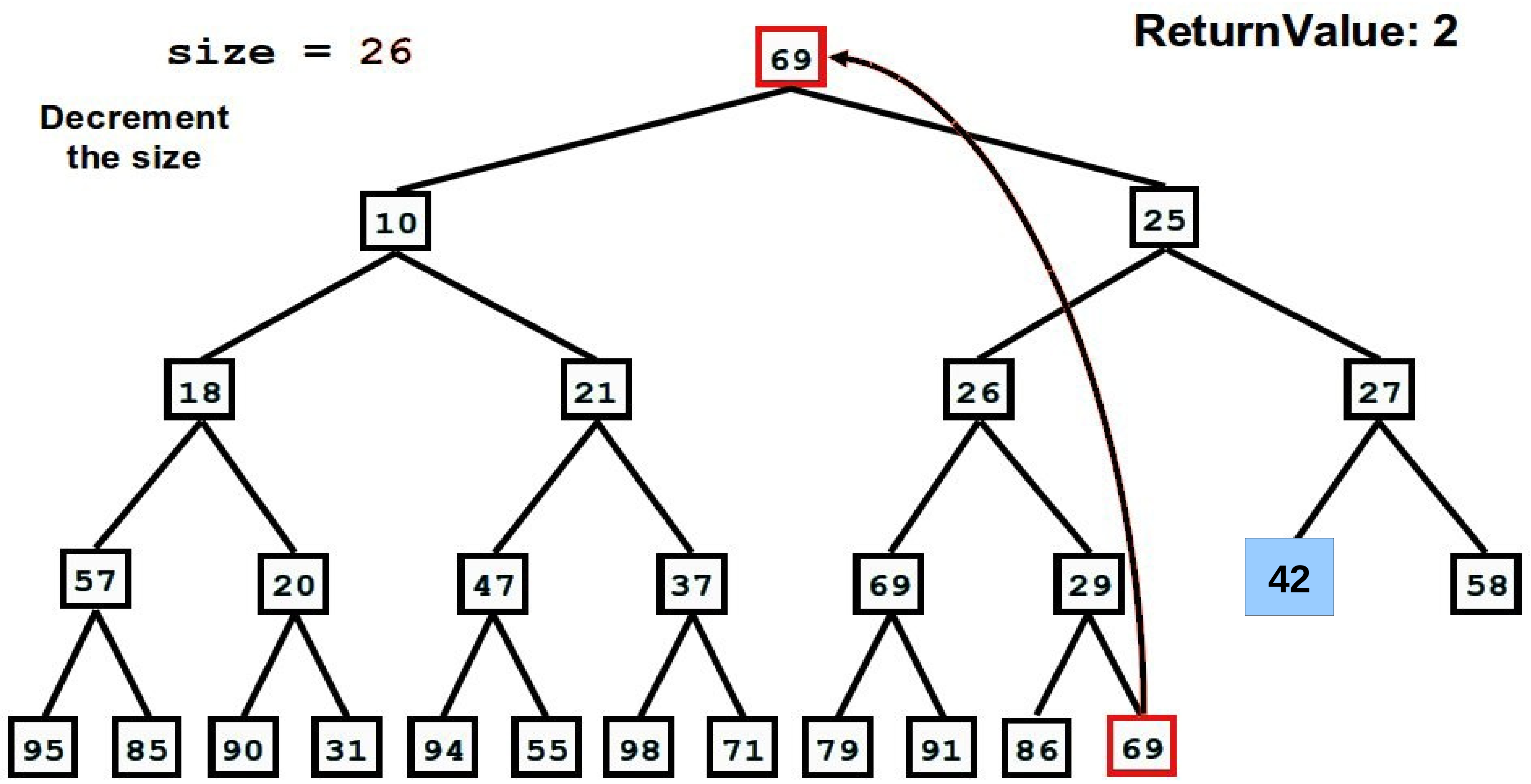
Initial Heap



DeleteMin()

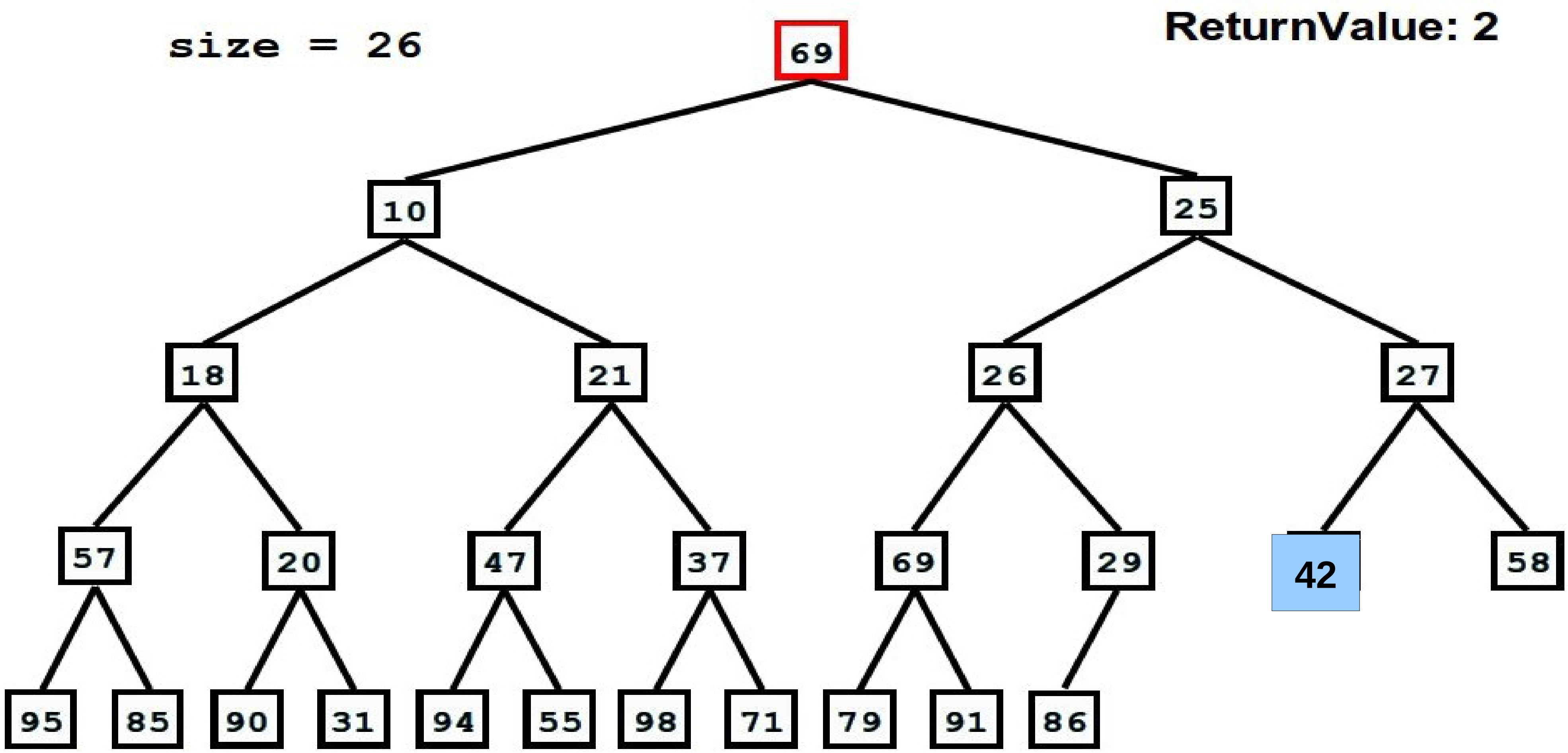

size = 27
ReturnValue: 2
2
10
25
18
21
26
27
57
20
47
37
69
29
42
58
95
85
90
31
94
55
98
71
79
91
86
69

DeleteMin

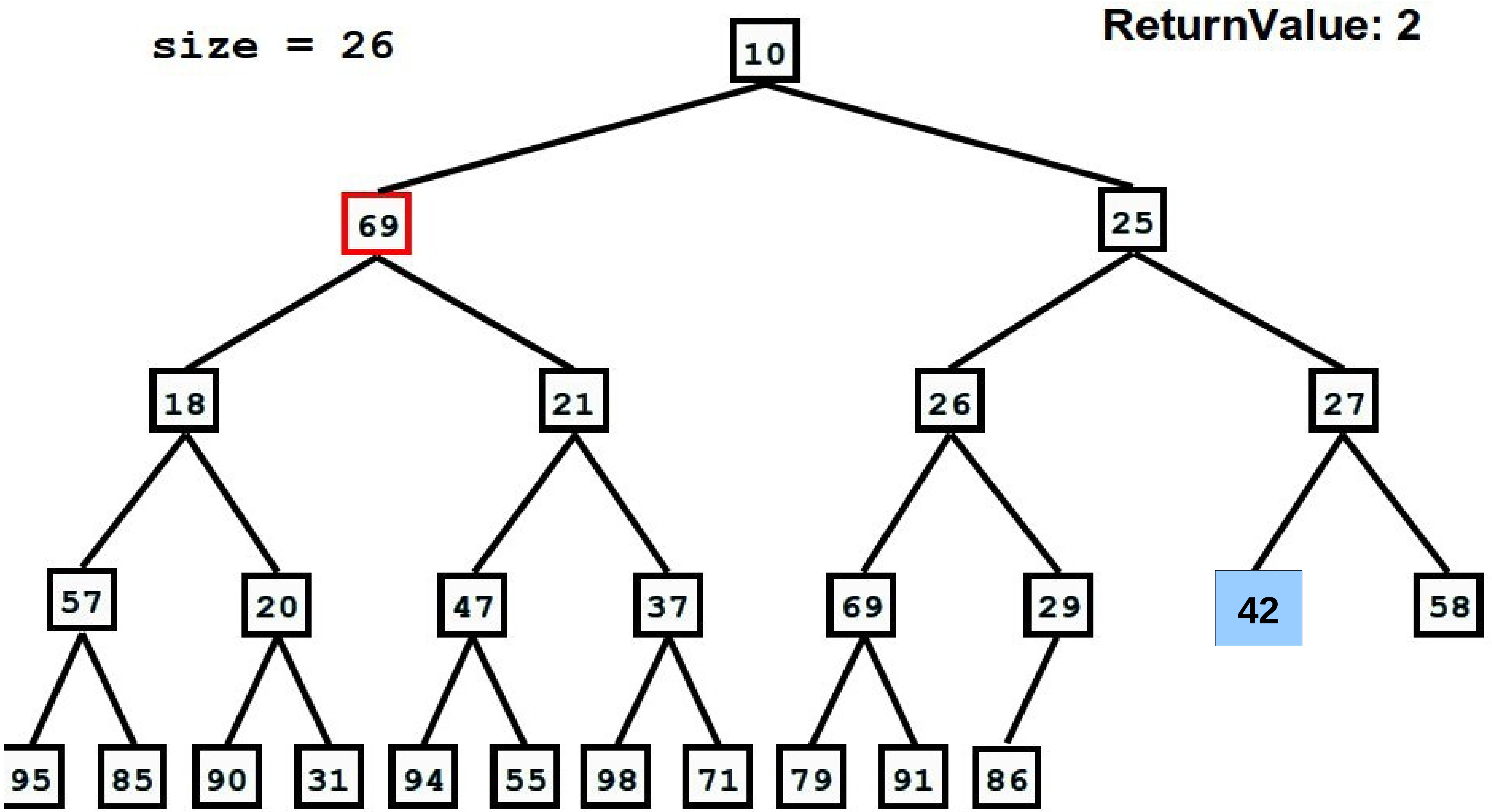


size = 26
Decrement the size
ReturnValue: 2
69
10
25
18
21
26
27
57
20
47
37
69
29
42
58
95
85
90
31
94
55
98
71
79
91
86
69

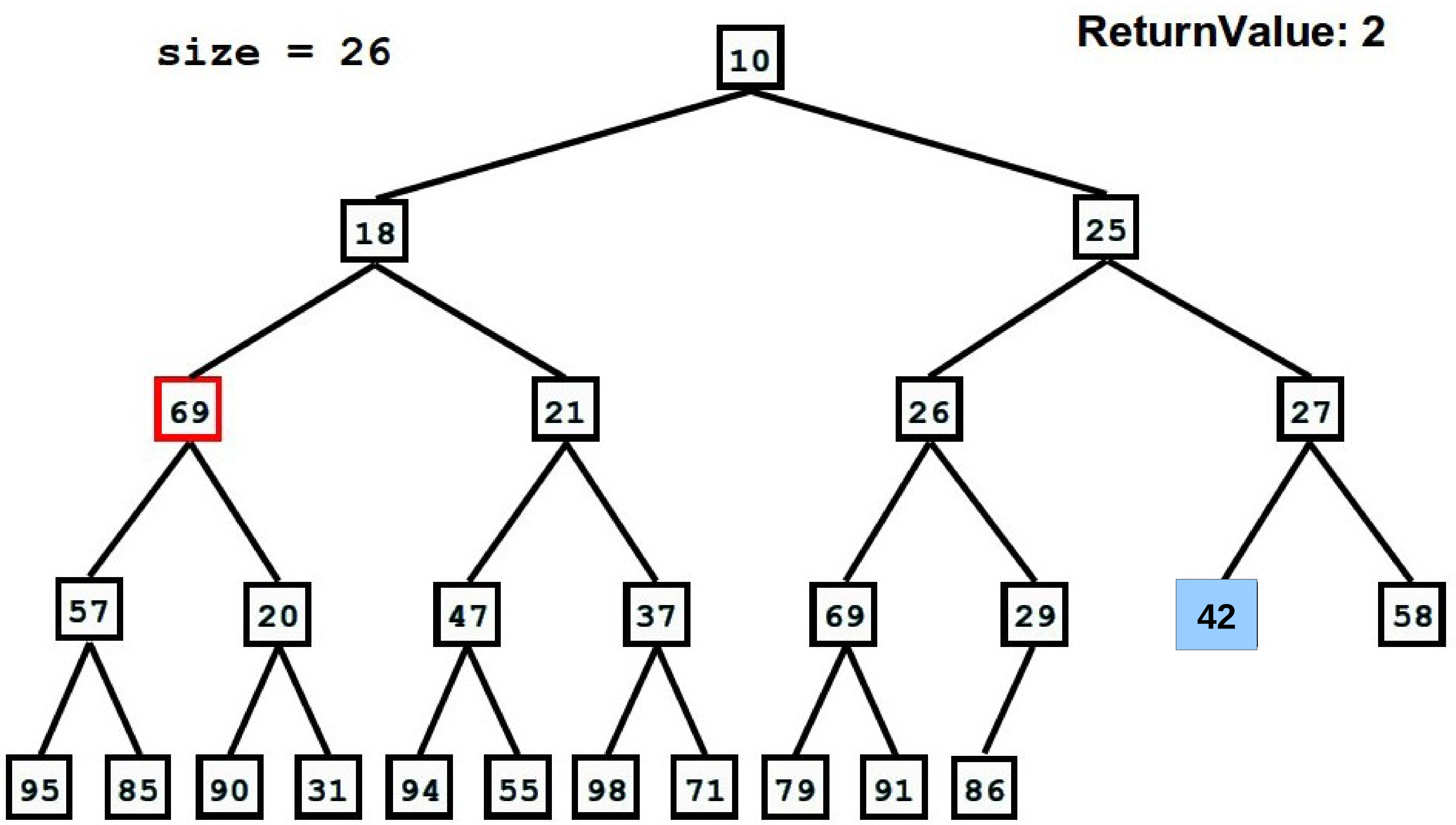
Shuffle Down



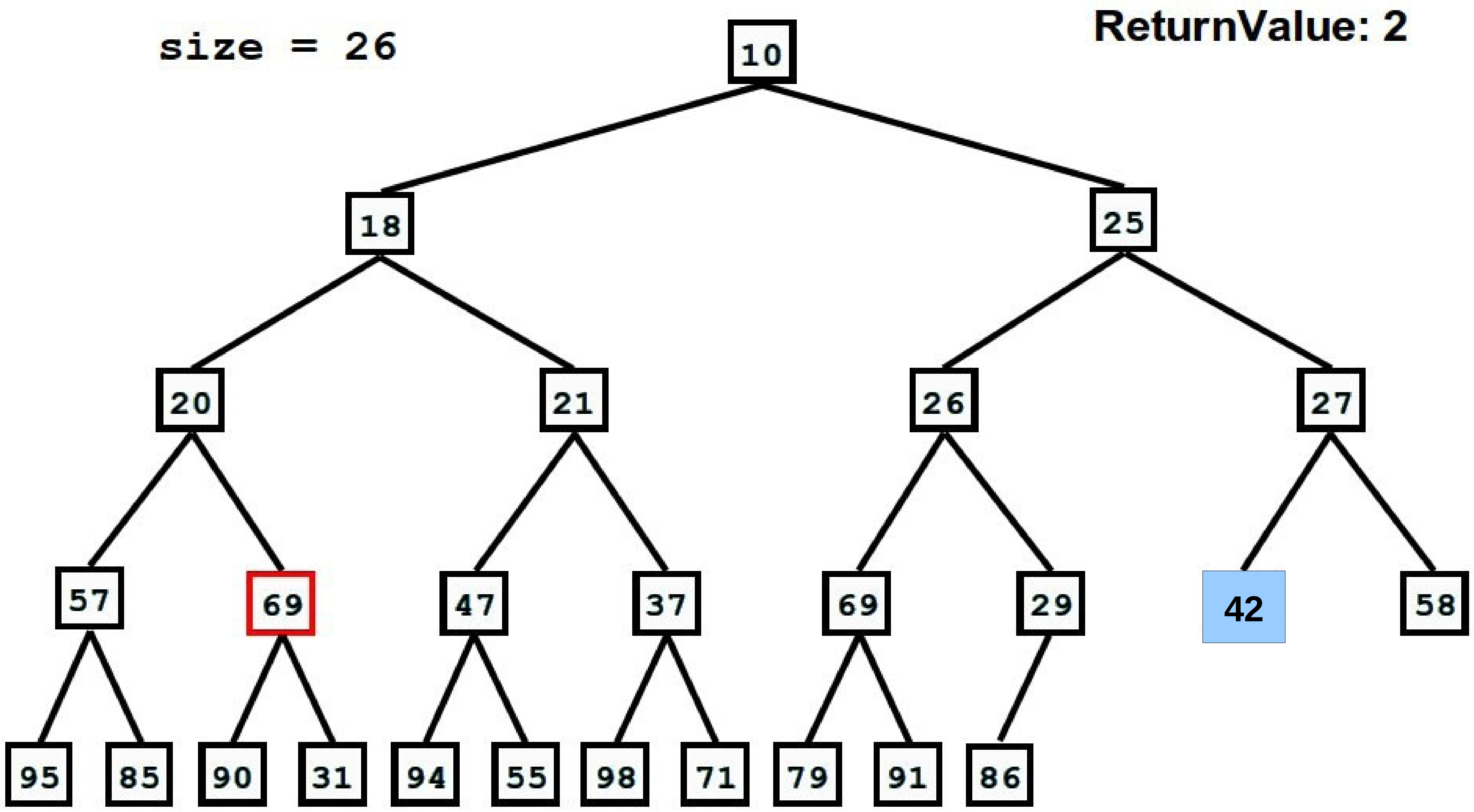
Shuffle Down



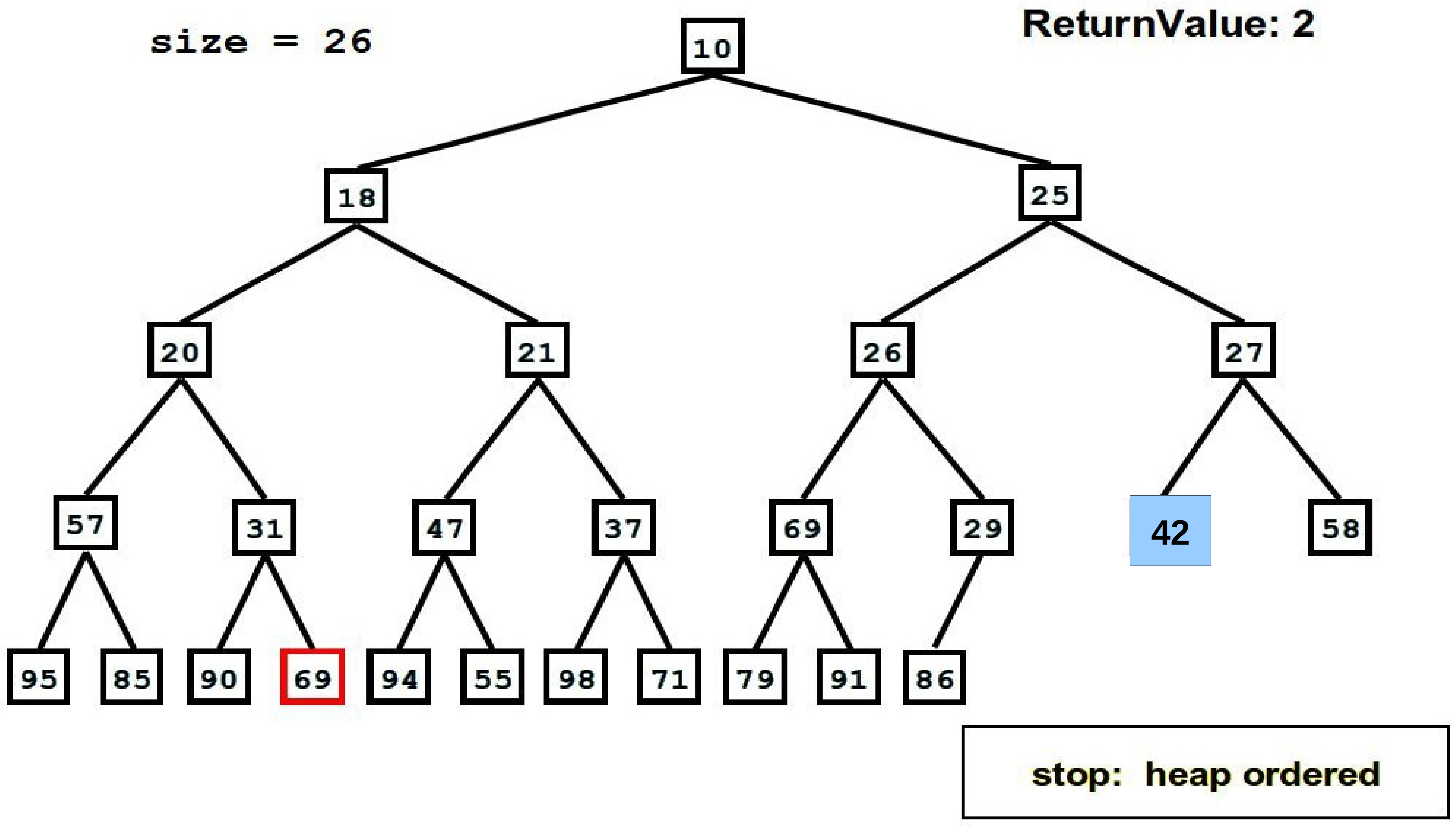
Shuffle Down



Shuffle Down



Shuffle Down



Procedure DeleteMin

```
Procedure DeleteMin(H)
begin
    min = H(1);
    H(1) = H(n);
    n = n-1;
    Shuffle-Down(H(1));
    return min;
end.
```

```
Procedure ShuffleDown(x)
begin
    i =index(x);
    while (i <= n/2) and
    (element > H(2i) or
    element > H(2i + 1)) do
        min = minfH(2i);H(2i+ 1);
        swap(H(i); min);
        i =index(min);
    End-while;
End.
```

Further Applications of the Heap

- The simple data structure has other applications too.
- Other operations such as DecreaseKey()
- We will see an application to sorting.

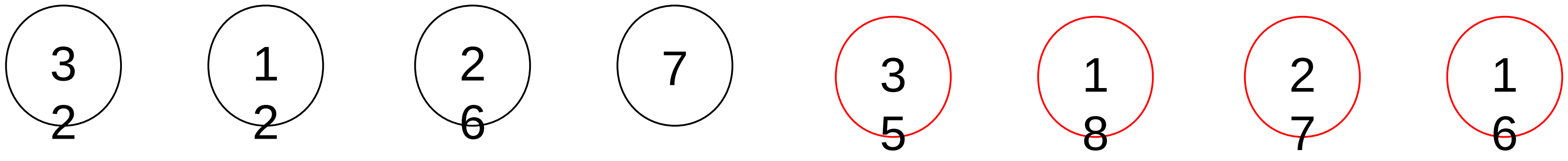
Sorting using a Heap

- Say, given n elements a1, a2, ..., an.
- need to arrange them in increasing order.
- We can first insert each of them starting from an empty heap.
- Now, call deleteMin till the heap is empty.
- We get sorted order of elements.
- Indeed, this procedure is called as heap sort.
- The runtime of this sorting approach is O(nlog n)
  - Call n insert() and n deleteMin() on the heap

Sorting

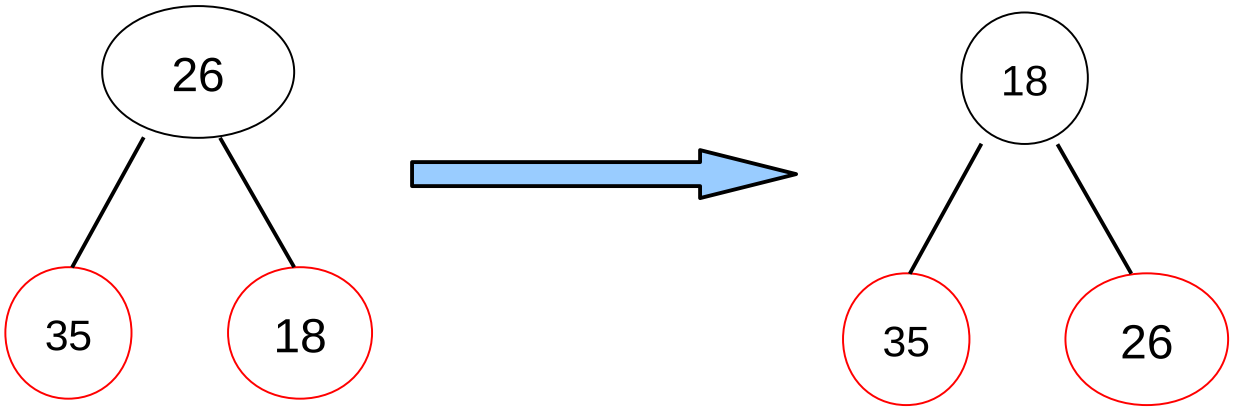
- Requires building a heap of the elements initially.
- Start from an empty heap, insert one element at a time.
- Straight-forward analysis: Each element takes O(log n) time, so total time is O(nlog n)
- But, initial heaps have fewer than n elements.
- The above O(log n) time per insert is an overestimate.

Sorting



- Consider an operation such as BuildHeap that takes n inputs and returns a heap of the n inputs.
- Notice that a one element tree is also a heap.
- So, imagine starting with n/2 one element heaps.

Sorting



- That leaves us with a further n/2 elements.
- Extend each of the one element heaps to a 3-element heap.
  - In a way, two 1-element heaps are joined by a parent.
  - Always maintain heap property

Sorting

- Can do this across all such 1-element heaps.
- Later on, extend these 3-element heaps to 7-element heaps.
  - Combine two 3-element heaps with a new element.

BuildHeap

- The whole procedure now runs in O(n) time for the following reason
- N/2 elements require no time.
- N/4 elements require at most 1 shuffle down.
- N/8 element require at most 2 shuffle down operations.
- Add all these to get O(n) time.