

Heap

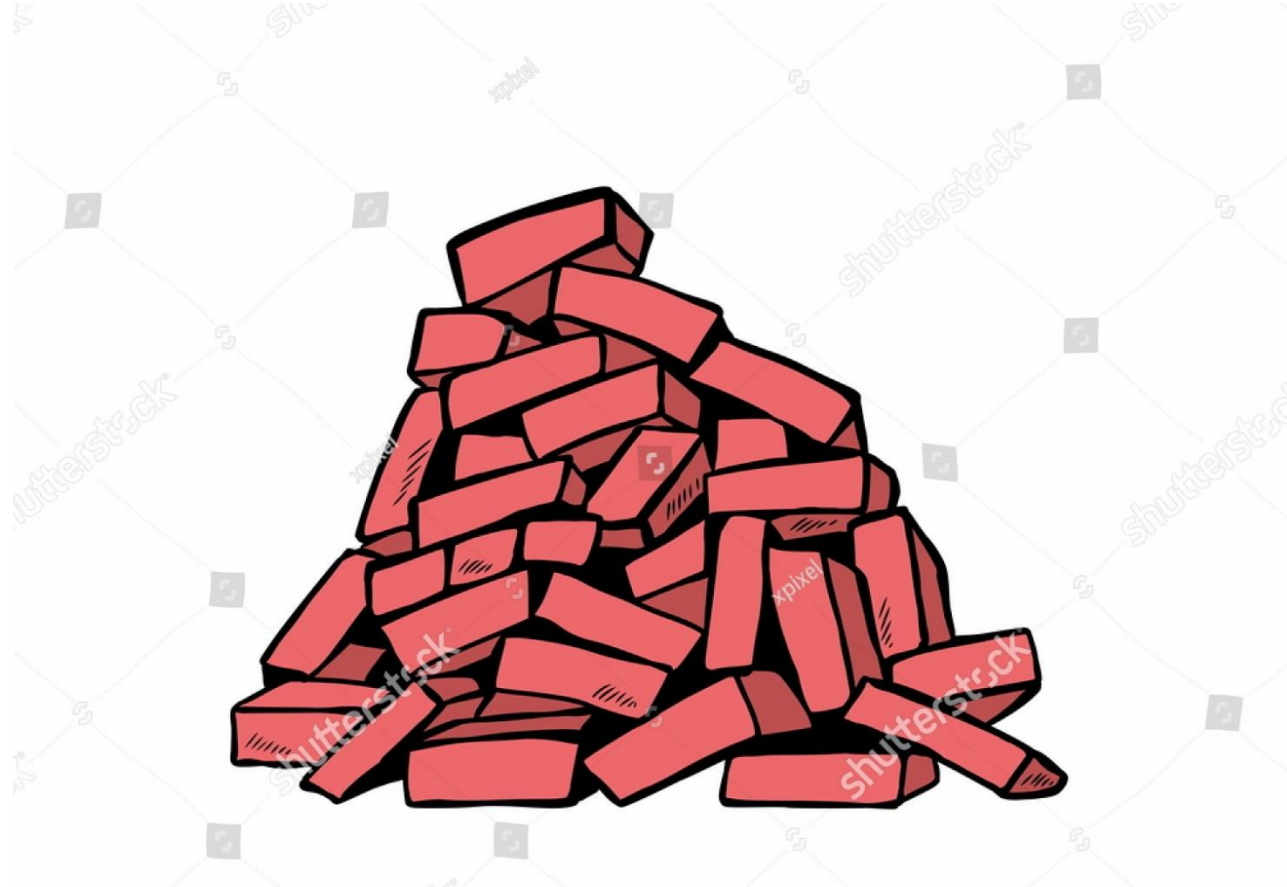
“Tree in specific order”

Prerequisite: Tree

Reference:
CLRS 3rd Edition, Chapter 6

Md. Saidul Hoque Anik
onix.hoque.mist@gmail.com

Heap

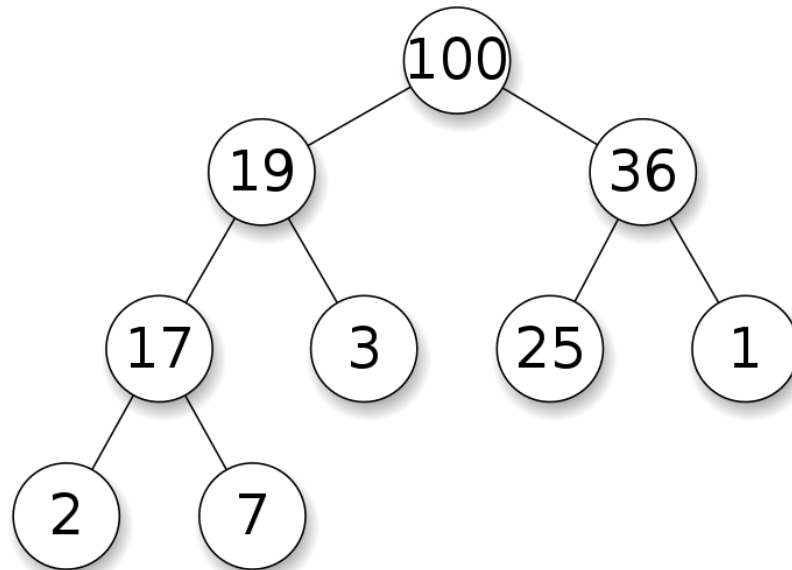


Heap

General Definition

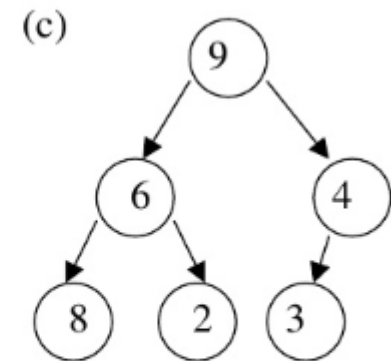
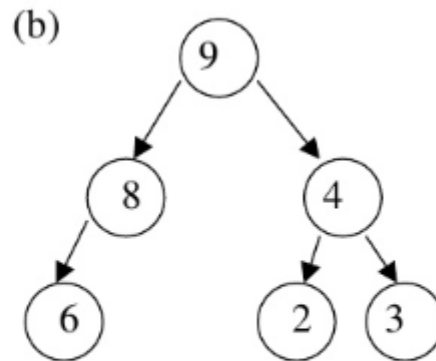
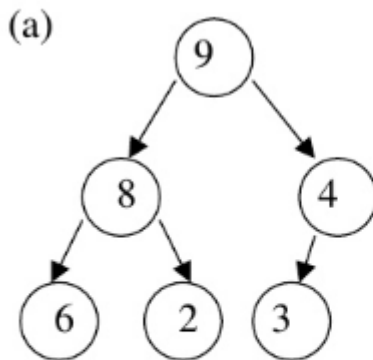
A tree based data structure in which all the nodes of tree are in a specific order. (Known as Heap-order Property)

Let's say if X is a parent node of Y, then the value of X follows some specific order/rule with respect to value of Y and the same order/rule will be followed across the tree.



Binary Heap

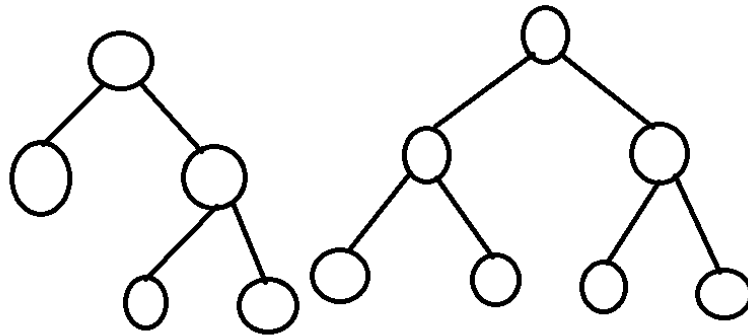
- A Complete Binary Tree (Each level is completely filled except the bottom level. Bottom level is filled from left to right)
- Satisfies Heap-order Property
 - If it a max-heap, then data in parent is greater or equal to data in its children
 - If it a min-heap, then data in parent is lesser or equal to data in its children



- In the above examples **only (a) is a heap**. (b) is not a heap as it is not complete and (c) is complete but does not satisfy the second property defined for heaps.

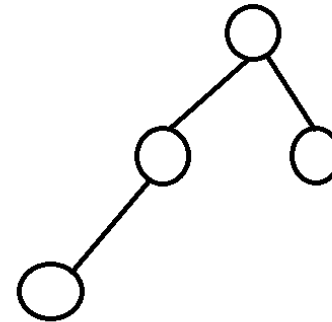
Note

Difference between Complete and Full Binary Tree

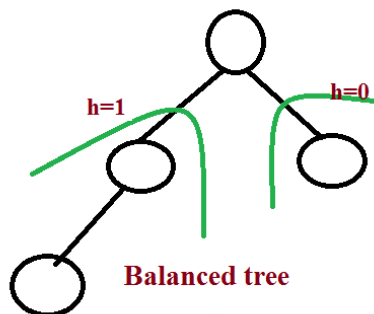


Binary Tree

Fully (perfect) Binary Tree

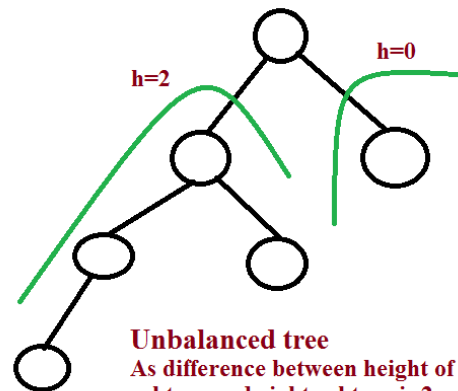


Complete Binary Tree



Balanced tree

As difference between height of left subtree and right subtree is 1

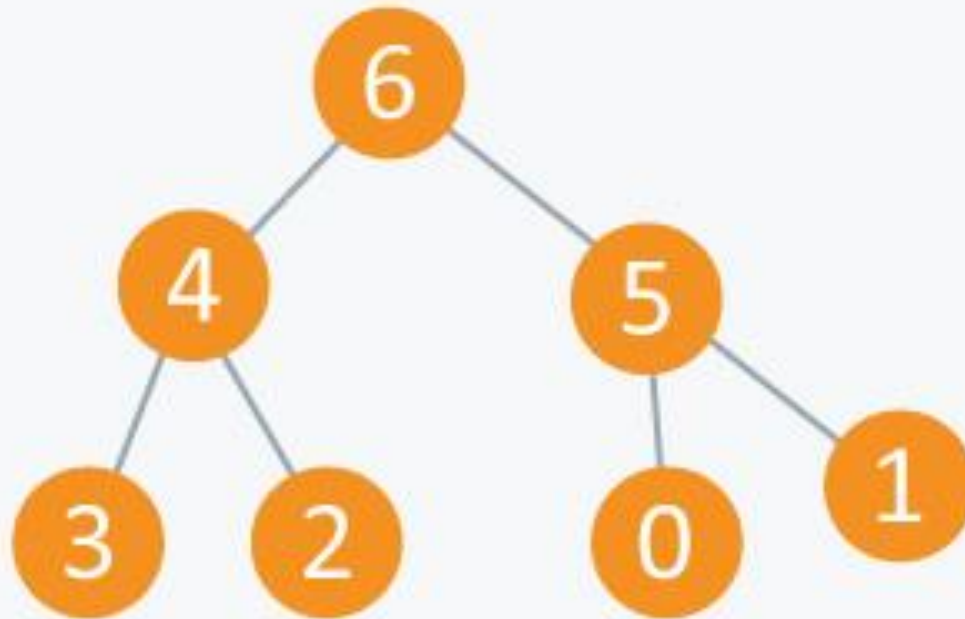


Unbalanced tree

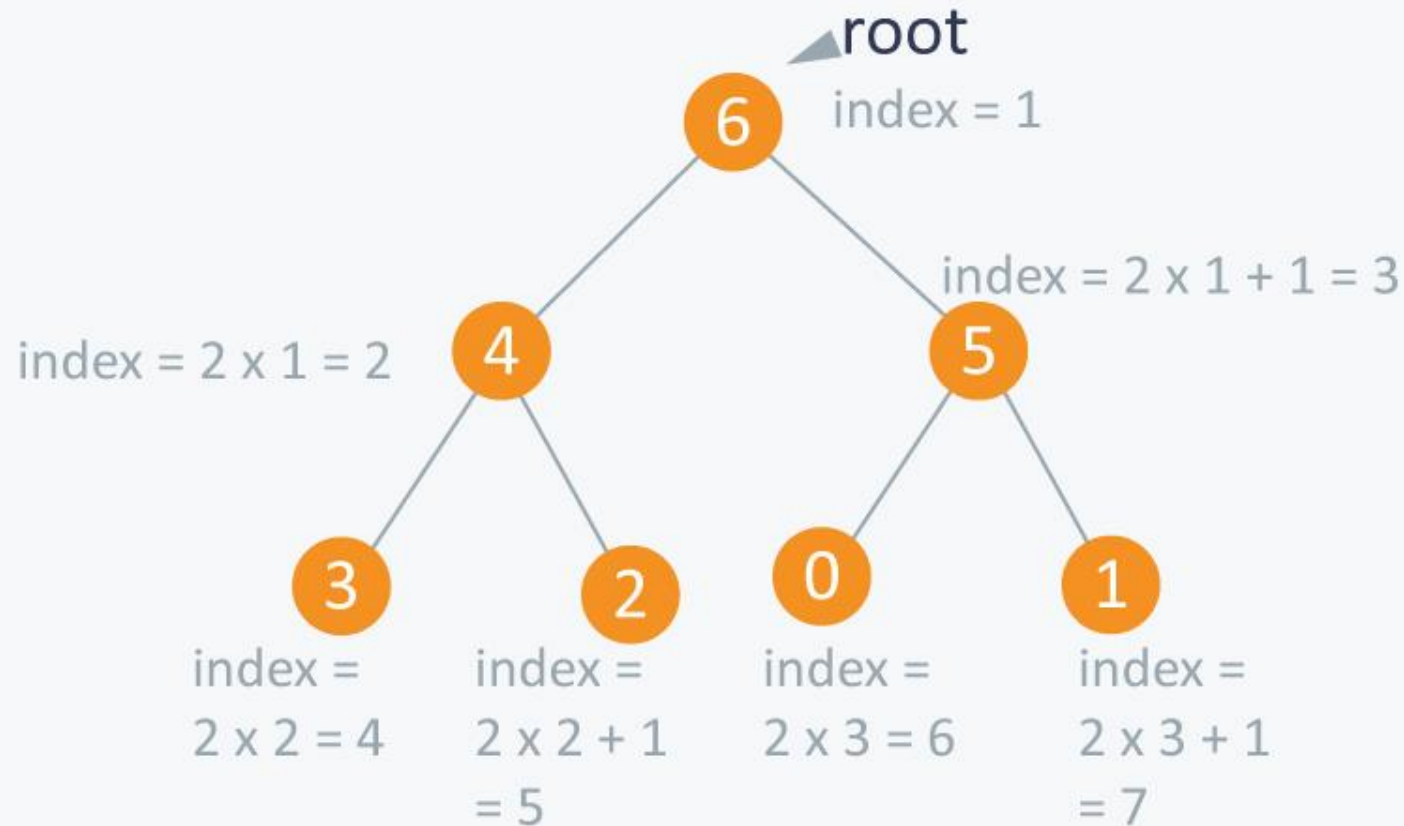
As difference between height of left subtree and right subtree is 2

Example of Heap

Job Scheduling



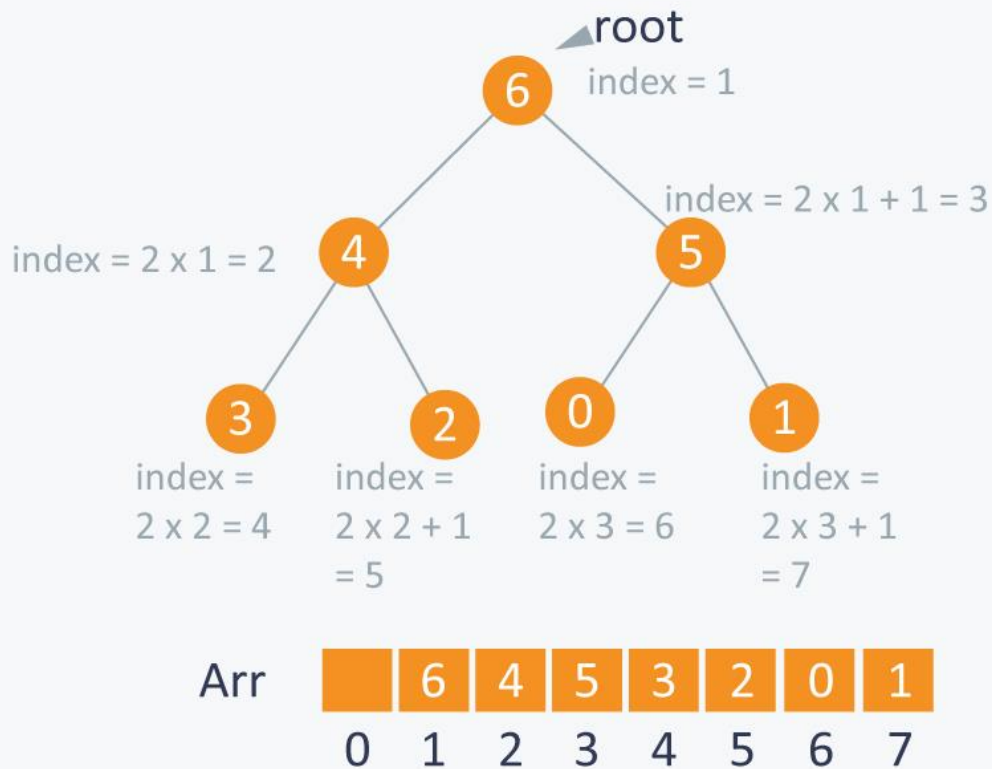
Heap Implementation



Arr

	6	4	5	3	2	0	1
0	1	2	3	4	5	6	7

Heap Implementation



```
left_child(i)
{ return 2 * i; }
```

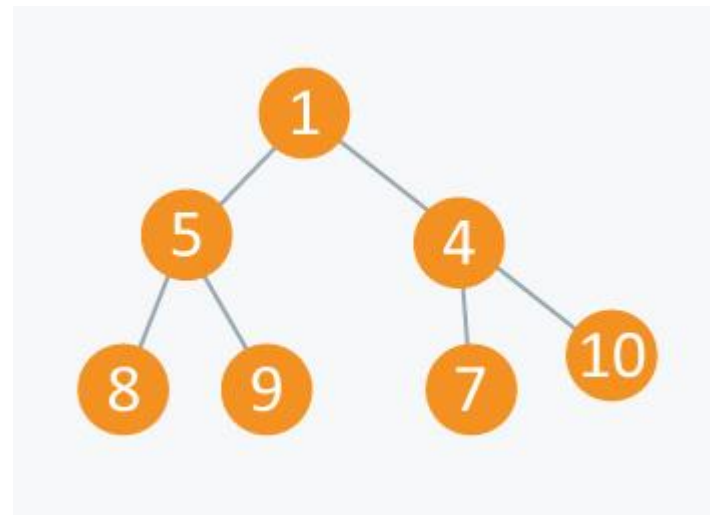
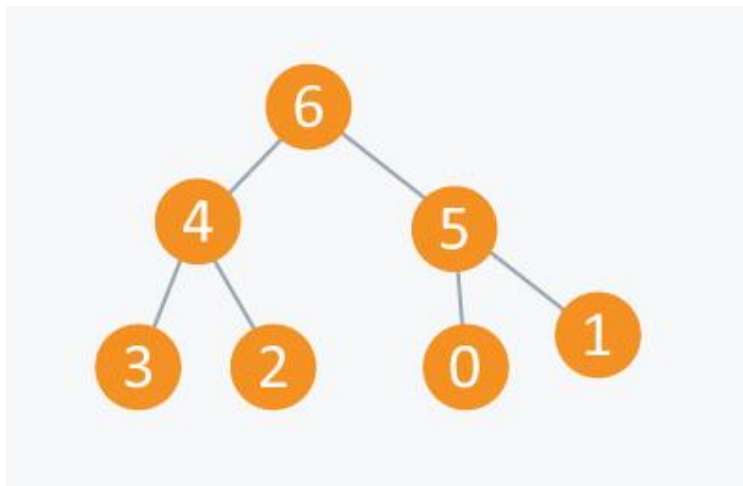
```
right_child(i)
{ return 2 * i + 1; }
```

```
parent(i)
{ return i / 2; }
```


Types of Heap

Max Heap: In this type of heap, the value of parent node will always be greater than or equal to the value of child node across the tree and the node with highest value will be the root node of the tree.

Min Heap: In this type of heap, the value of parent node will always be less than or equal to the value of child node across the tree and the node with lowest value will be the root node of tree.




In a Max Heap, both children of a node are two max heaps.
In a Min Heap, both children of a node are two min heaps.

Task

Write the following array into the heap tree.

0	1	2	3	4	5	6	7
7	4	7	8	9	2	6	5

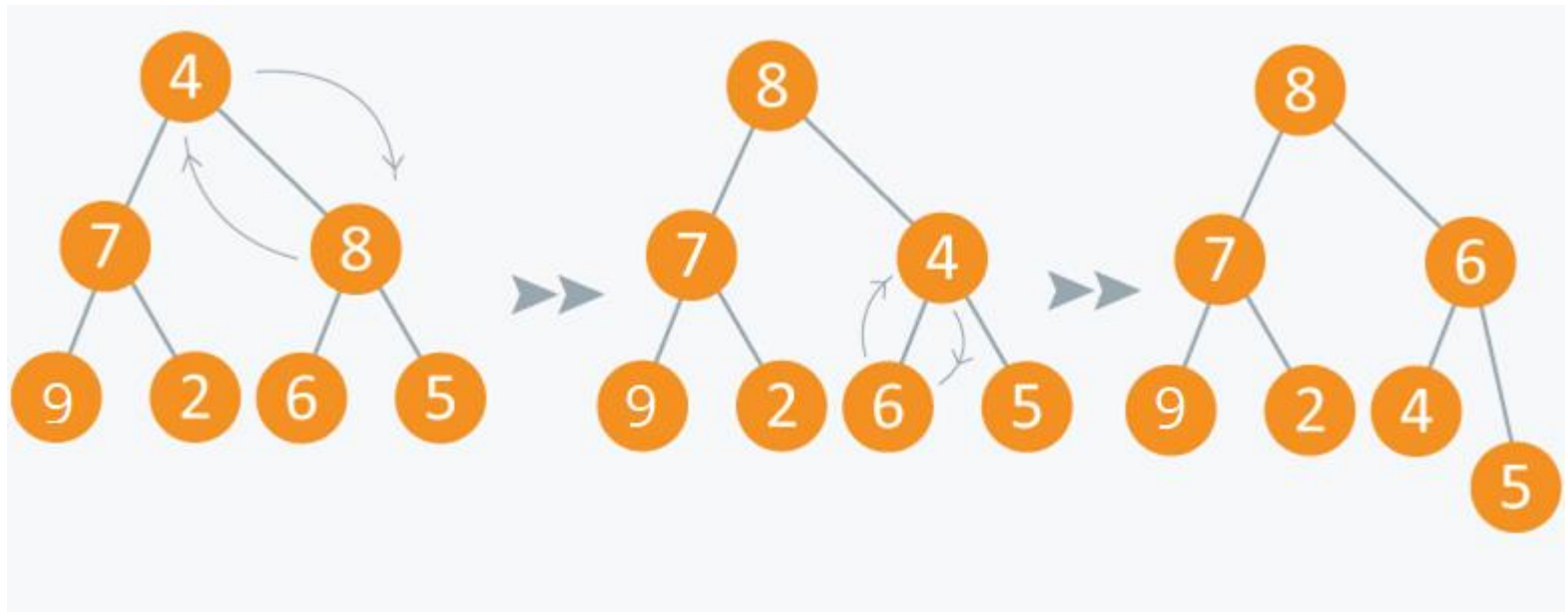


Index 0 holds the number
of nodes in the heap

Implementation of Max Heap

Two functions are needed to convert an array to a Max Heap

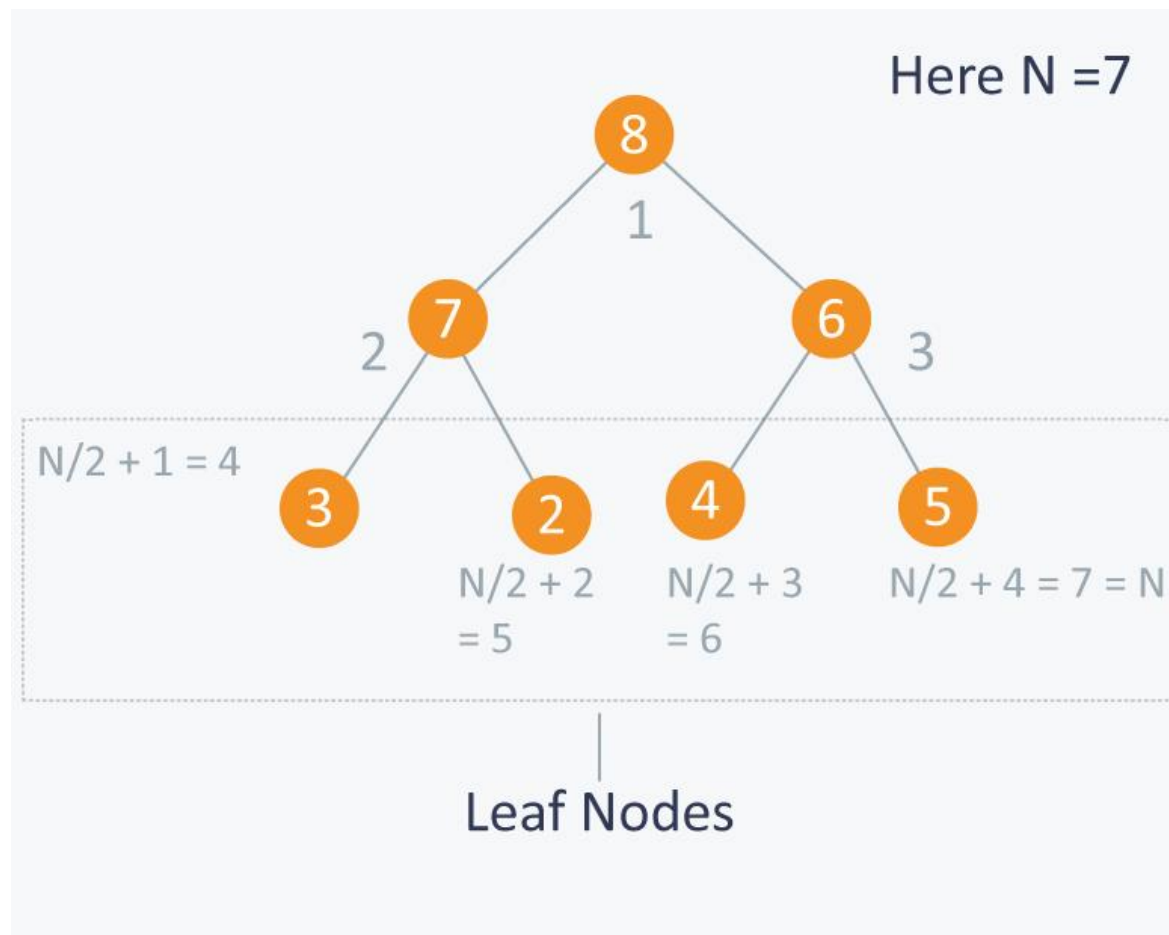
1. **Max Heapify** : Takes a node i , and goes upto leaf to make sure that the trees in that path is in max-heap-order



Implementation of Max Heap

Two functions are needed to convert an array to a Max Heap

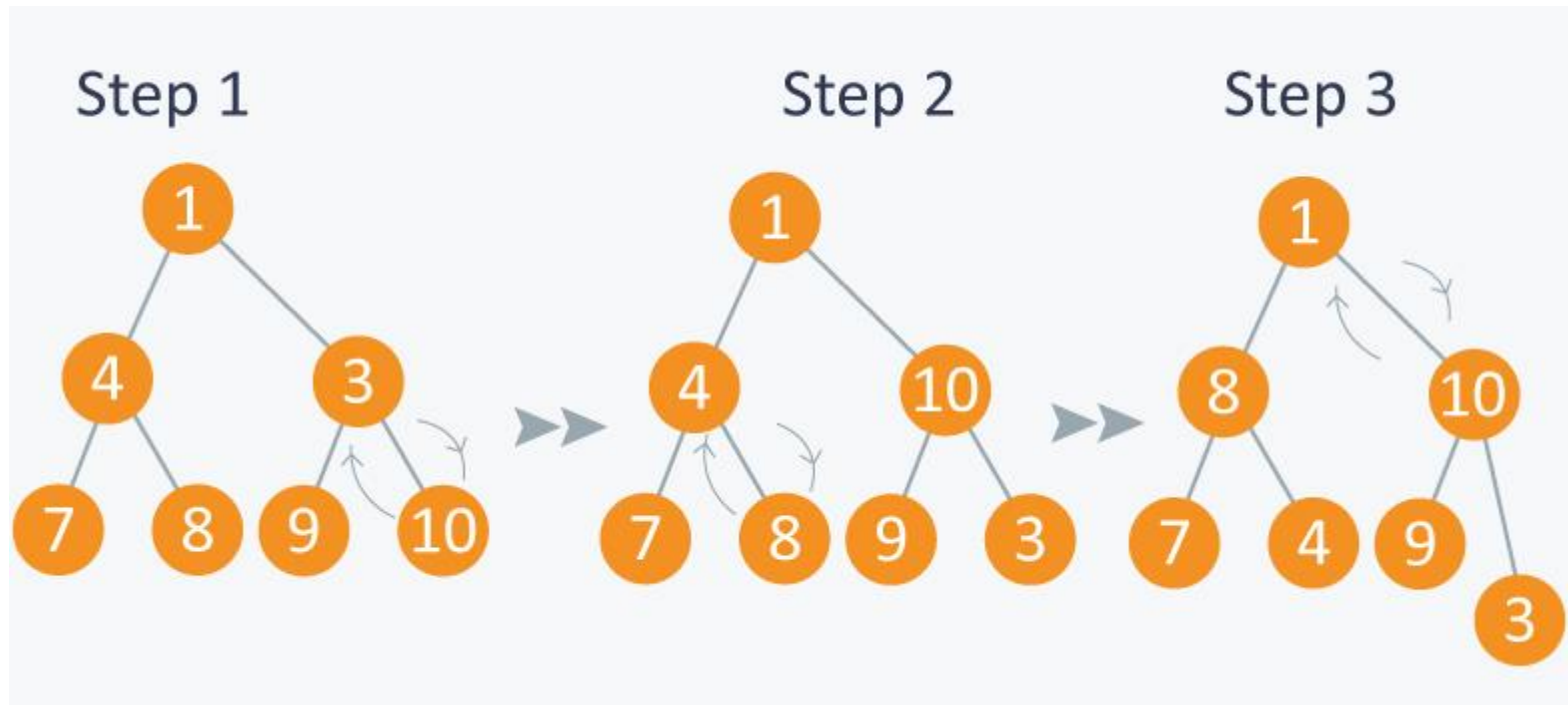
2. **Build Heap** : Take nodes from $n/2$ to 1, call max_heapify for each node



Implementation of Max Heap

Two functions are needed to convert an array to a Max Heap

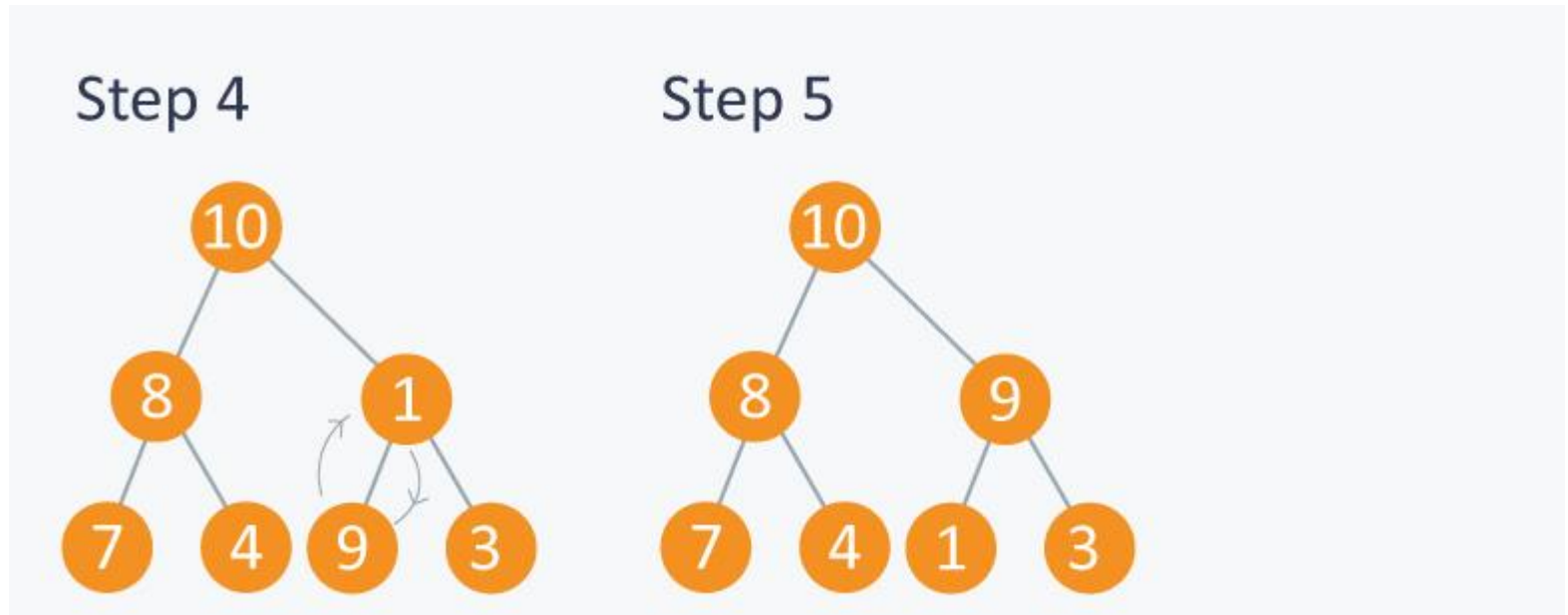
2. **Build Heap** : Take nodes from $n/2$ to 1, call `max_heapify()` for each node.



Implementation of Max Heap

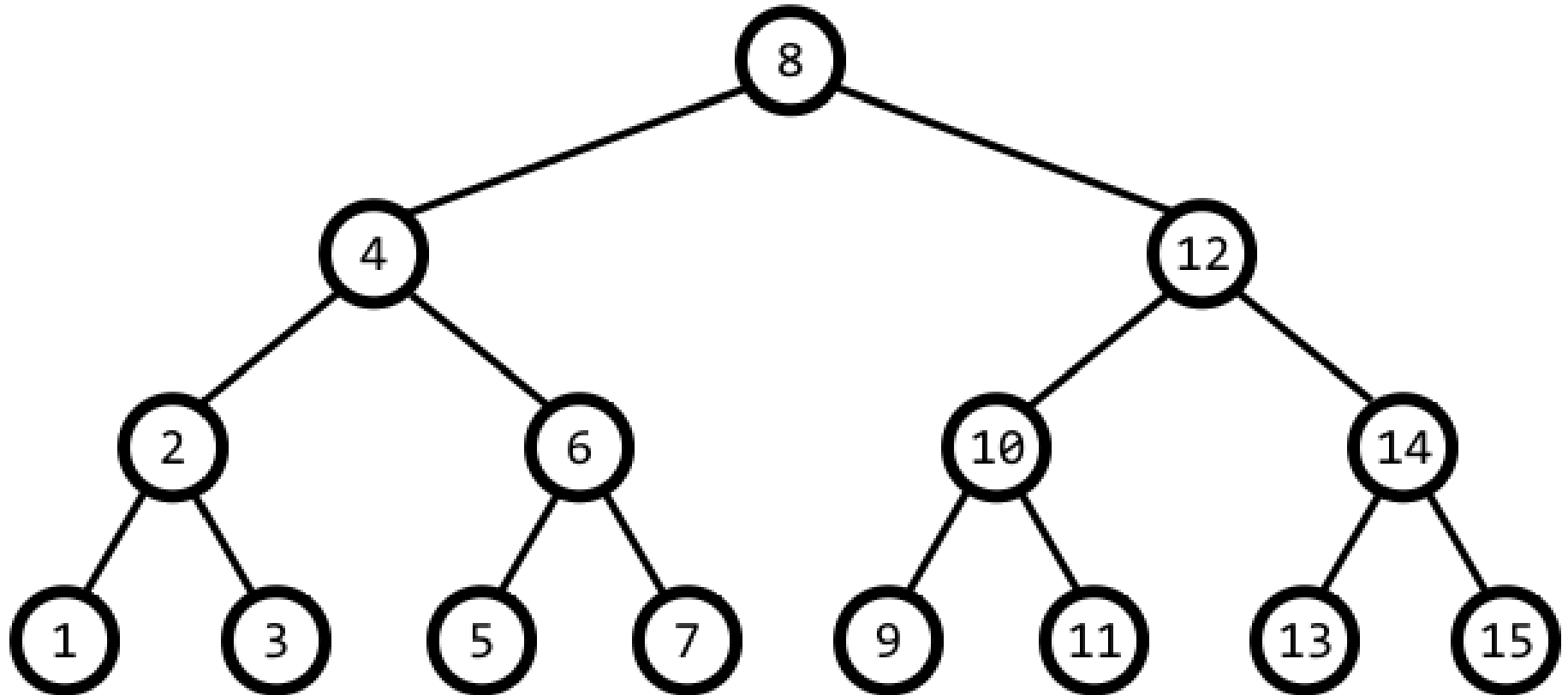
Two functions are needed to convert an array to a Max Heap

- 2. Build Heap** : Take nodes from $n/2$ to 1, call `max_heapify` for each node



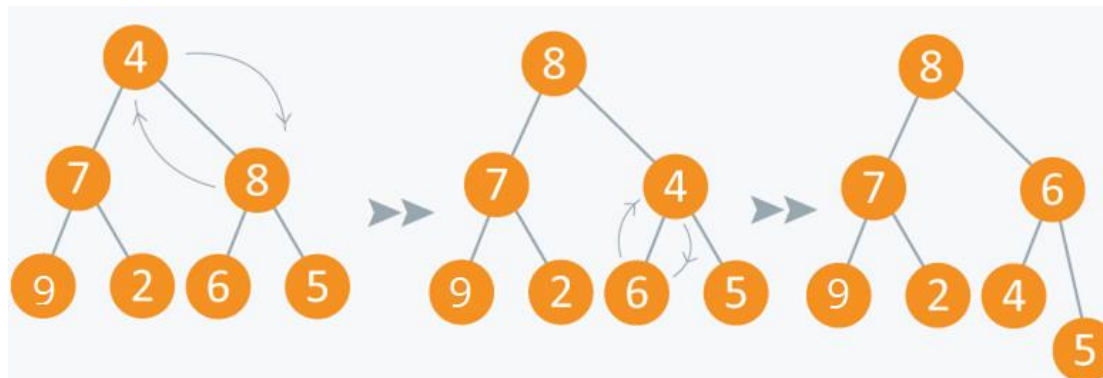
Task

For the following tree, draw each step of `build_heap` until the tree becomes a Max-Heap



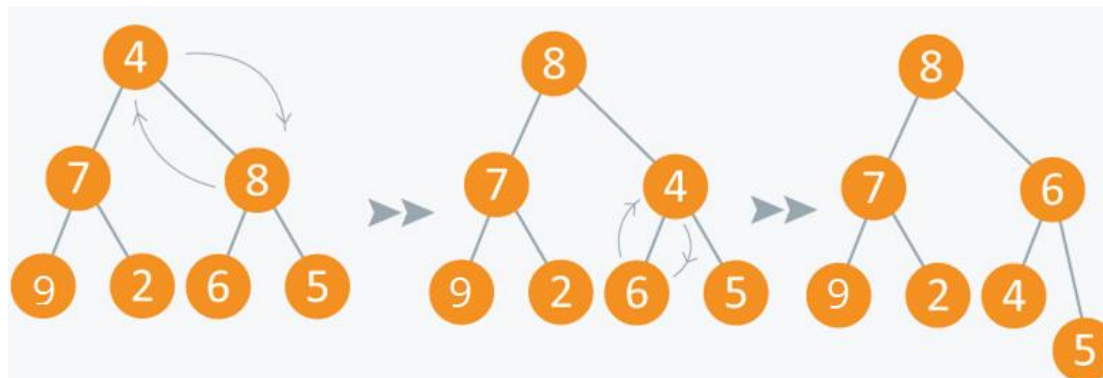
Pseudocode

```
void max_heapify (int Arr[ ], int i, int N)
{
    int left = 2*i                //left child
    int right = 2*i +1            //right child
```



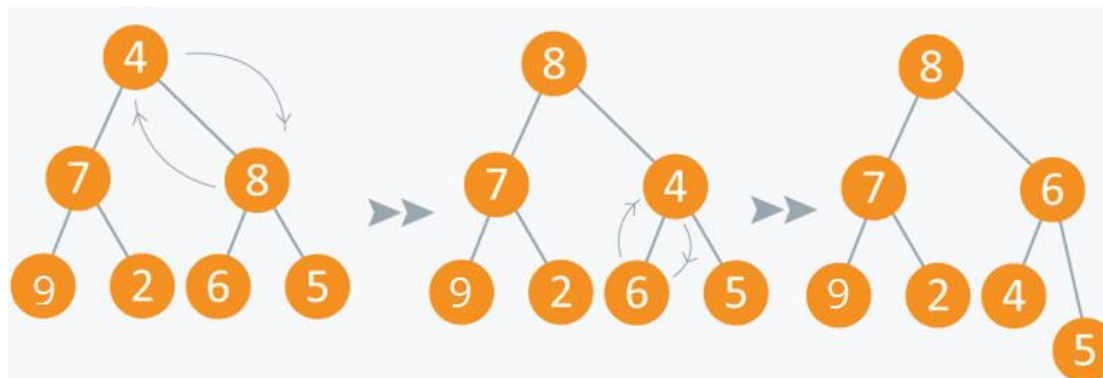
Pseudocode

```
void max_heapify (int Arr[ ], int i, int N)
{
    int left = 2*i                //left child
    int right = 2*i +1            //right child
    if(left<= N and Arr[left] > Arr[i] )
        largest = left;
    else
        largest = i;
}
```



Pseudocode

```
void max_heapify (int Arr[ ], int i, int N)
{
    int left = 2*i           //left child
    int right = 2*i +1       //right child
    if(left<= N and Arr[left] > Arr[i] )
        largest = left;
    else
        largest = i;
    if(right <= N and Arr[right] > Arr[largest] )
        largest = right;
```



Pseudocode

```
void max_heapify (int Arr[ ], int i, int N)
{
    int left = 2*i                //left child
    int right = 2*i +1            //right child
    if(left<= N and Arr[left] > Arr[i] )
        largest = left;
    else
        largest = i;
    if(right <= N and Arr[right] > Arr[largest] )
        largest = right;
    if(largest != i )
    {
        swap (Arr[i] , Arr[largest]);
        max_heapify (Arr, largest,N);
    }
}
```

Pseudocode

```
void build_maxheap (int Arr[ ])
{
    // Pseudocode for building a max heap
    // 1. Find the root of the tree (index 1)
    // 2. Compare the root with its children
    // 3. If the root is smaller than its children, swap it with the larger child
    // 4. Repeat steps 2 and 3 for the new root until it is larger than its children
    // 5. Repeat steps 1-4 for all non-leaf nodes (nodes from index n/2 to 1)
}
// End of build_maxheap function
```

Pseudocode

```
void build_maxheap (int Arr[ ])
{
    for(int i = N/2 ; i >= 1 ; i-- )
    {
        max_heapify (Arr, i) ;
    }
}
```

Task

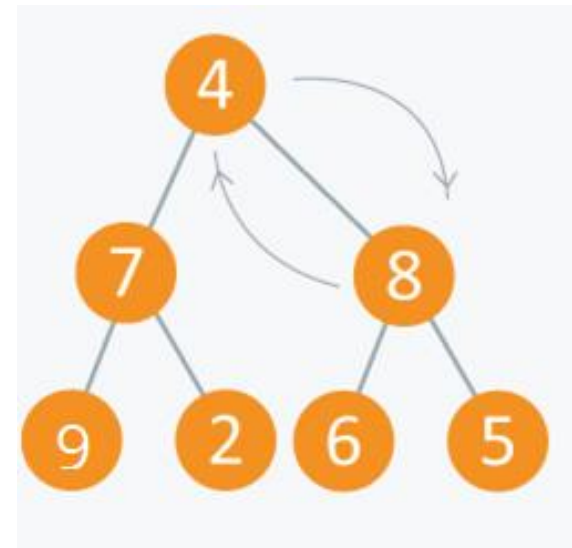
Write the pseudocode for

- `min_heapify()`
- `build_minheap()`

Complexity Analysis

max_heapify

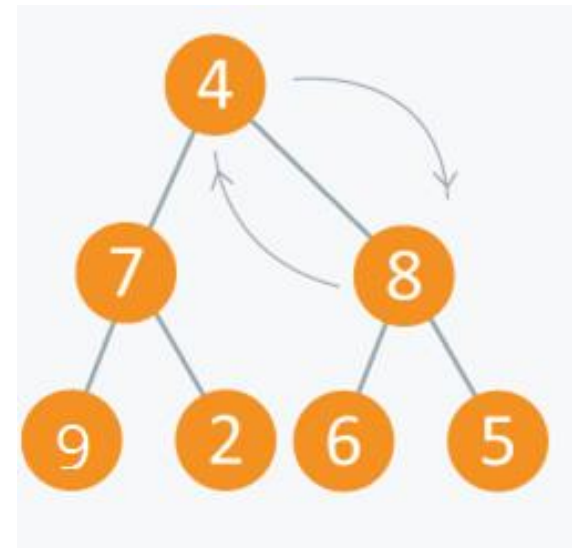
```
void max_heapify (int Arr[ ], int i, int N)
{
    int left = 2*i           //left child
    int right = 2*i +1       //right child
    if(left<= N and Arr[left] > Arr[i] )
        largest = left;
    else
        largest = i;
    if(right <= N and Arr[right] > Arr[largest] )
        largest = right;
    if(largest != i )
    {
        swap (Ar[i] , Arr[largest]);
        max_heapify (Arr, largest,N);
    }
}
```



Complexity Analysis

build_maxheap

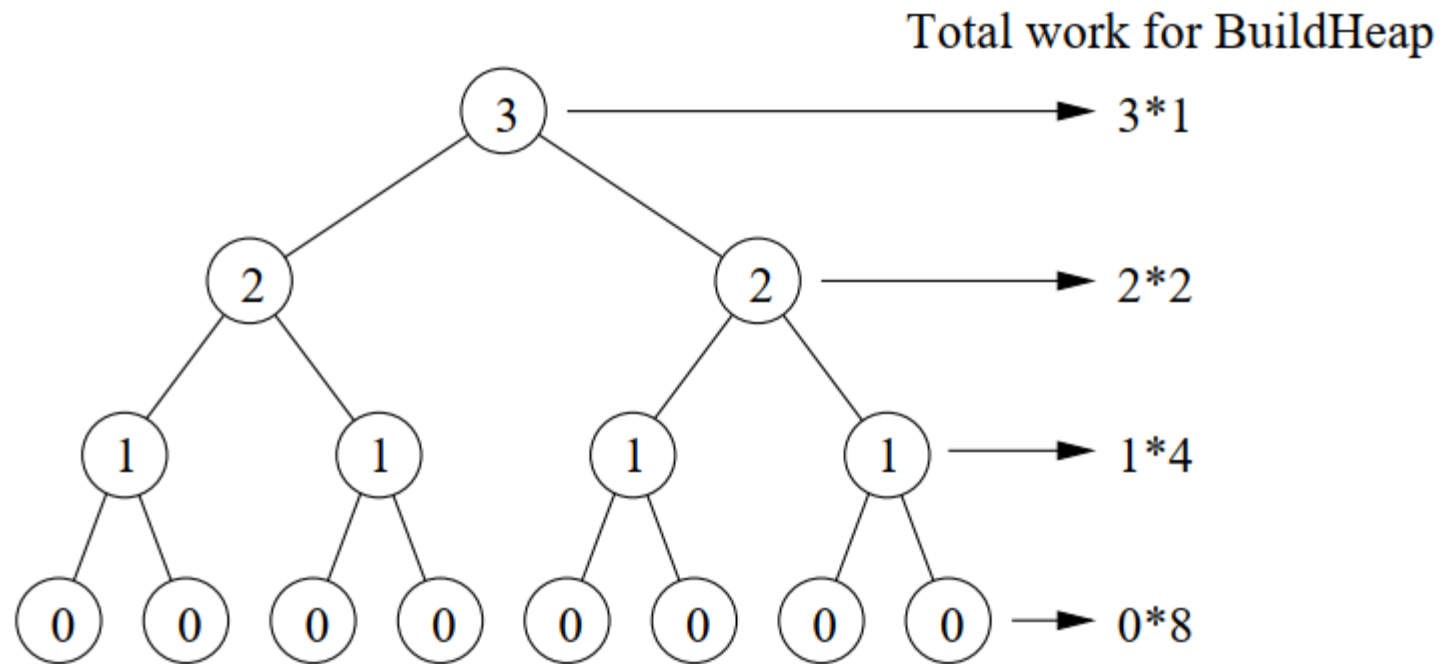
```
void build_maxheap (int Arr[ ])  
{  
    for(int i = N/2 ; i >= 1 ; i-- )  
    {  
        max_heapify (Arr, i) ;  
    }  
}
```



Complexity: ?

Complexity Analysis

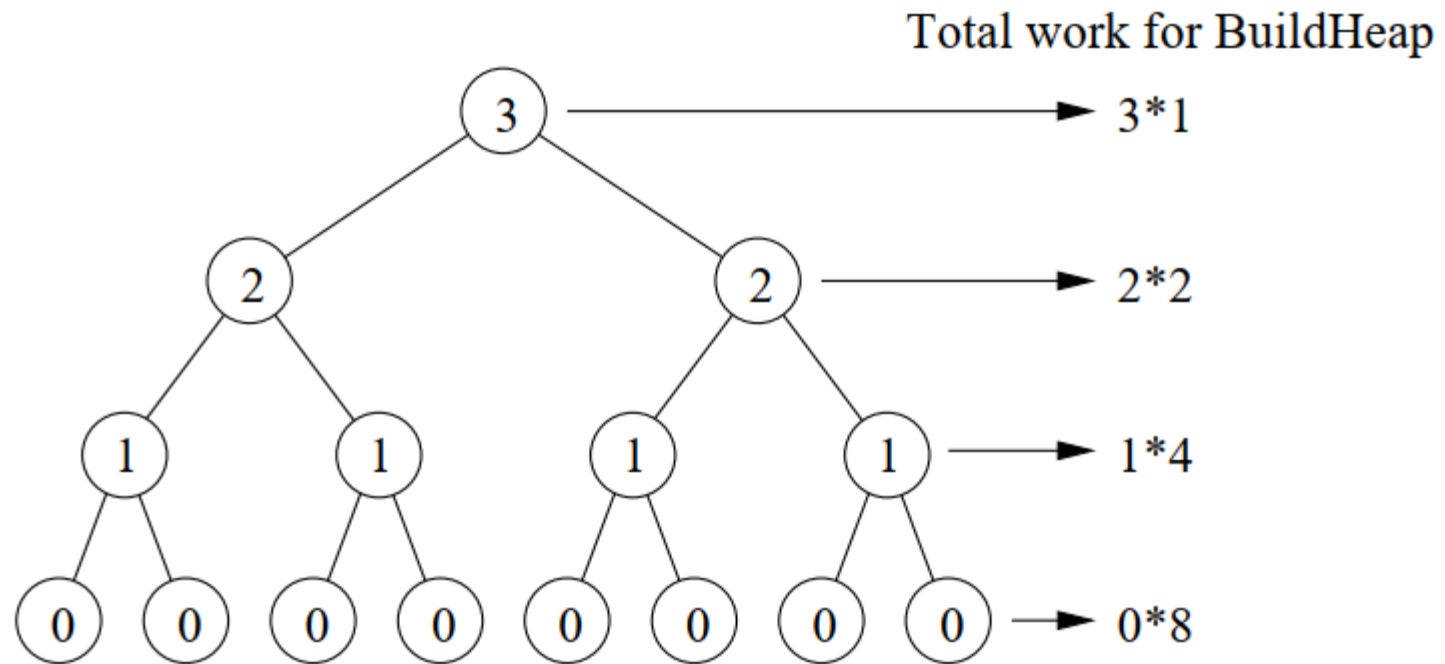
build_maxheap



$$n = 2^{h+1} - 1, \text{ where } h \text{ is the height of the tree}$$

Complexity Analysis

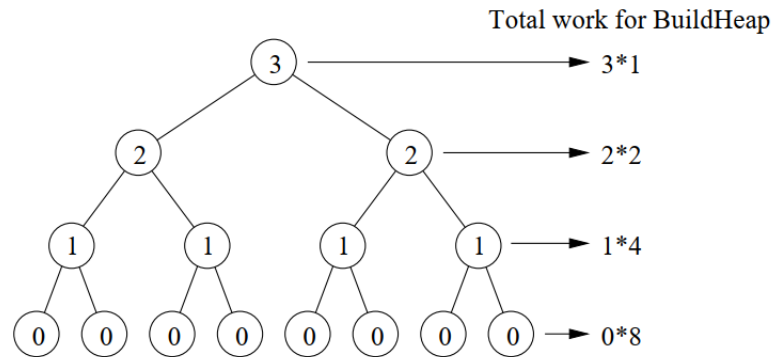
build_maxheap



$$n = 2^{h+1} - 1, \text{ where } h \text{ is the height of the tree}$$

Complexity Analysis

build_maxheap



At the bottommost level there are 2^h nodes, but we do not call Heapify on any of these so the work is 0. At the next to bottommost level there are 2^{h-1} nodes, and each might sift down 1 level. At the 3rd level from the bottom there are 2^{h-2} nodes, and each might sift down 2 levels. In general, at level j from the bottom there are 2^{h-j} nodes, and each might sift down j levels. So, if we count from bottom to top, level-by-level, we see that the total time is proportional to

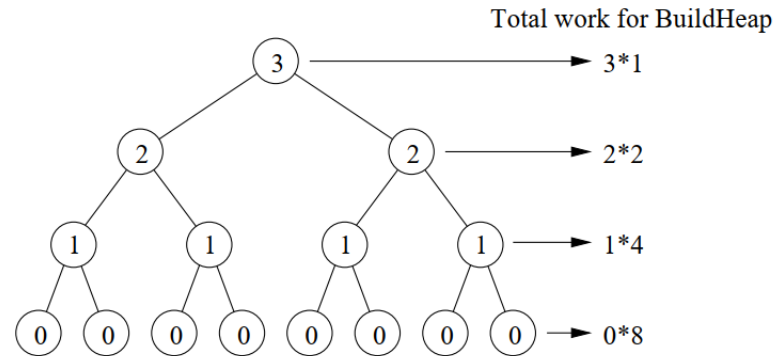
$$T(n) = \sum_{j=0}^h j 2^{h-j} = \sum_{j=0}^h j \frac{2^h}{2^j}.$$

If we factor out the 2^h term, we have

$$T(n) = 2^h \sum_{j=0}^h \frac{j}{2^j}.$$

Complexity Analysis

build_maxheap



First, write down the infinite general geometric series,
for any constant $x < 1$.

$$\sum_{j=0}^{\infty} x^j = \frac{1}{1-x}.$$

Then take the derivative of both sides with respect to x , and multiply by x giving:

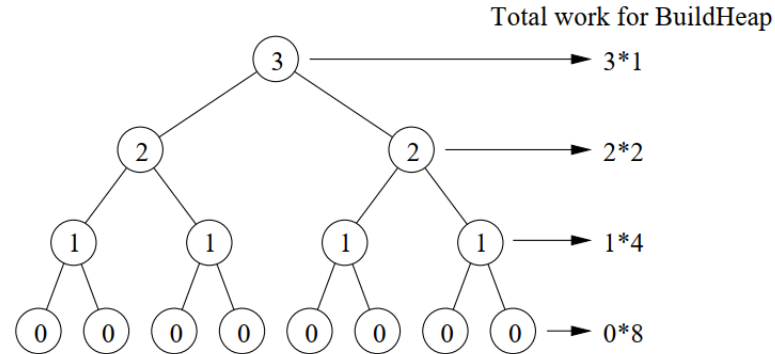
$$\sum_{j=0}^{\infty} jx^{j-1} = \frac{1}{(1-x)^2} \qquad \sum_{j=0}^{\infty} jx^j = \frac{x}{(1-x)^2},$$

and if we plug $x = 1/2$, then voila! we have the desired formula:

$$\sum_{j=0}^{\infty} \frac{j}{2^j} = \frac{1/2}{(1 - (1/2))^2} = \frac{1/2}{1/4} = 2.$$

Complexity Analysis

build_maxheap



$$\sum_{j=0}^{\infty} \frac{j}{2^j} = \frac{1/2}{(1 - (1/2))^2} = \frac{1/2}{1/4} = 2.$$

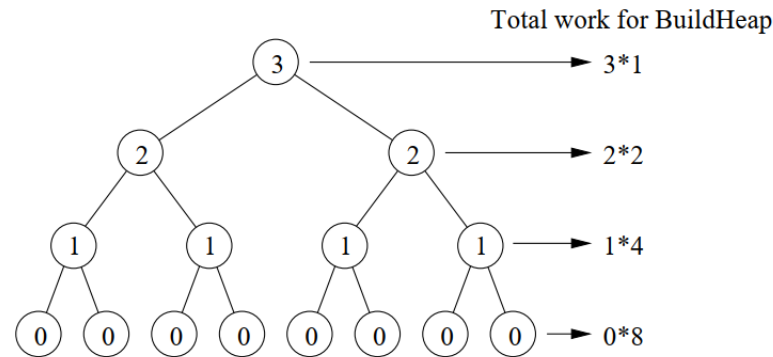
Using this we have

$$T(n) = 2^h \sum_{j=0}^h \frac{j}{2^j} \leq 2^h \sum_{j=0}^{\infty} \frac{j}{2^j} \leq 2^h \cdot 2 = 2^{h+1}.$$

Now recall that $n = 2^{h+1} - 1$, so we have $T(n) \leq n + 1 \in O(n)$. Clearly the algorithm takes at least $\Omega(n)$ time (since it must access every element of the array at least once) so the total running time for BuildHeap is $\Theta(n)$.

Complexity Analysis

build_maxheap



Perhaps a more intuitive way to describe what is happening here is to observe an important fact about binary trees. This is that the vast majority of nodes are at the lowest level of the tree. For example, in a complete binary tree of height h there is a total of $n \approx 2^{h+1}$ nodes in total, and the number of nodes in the bottom 3 levels alone is

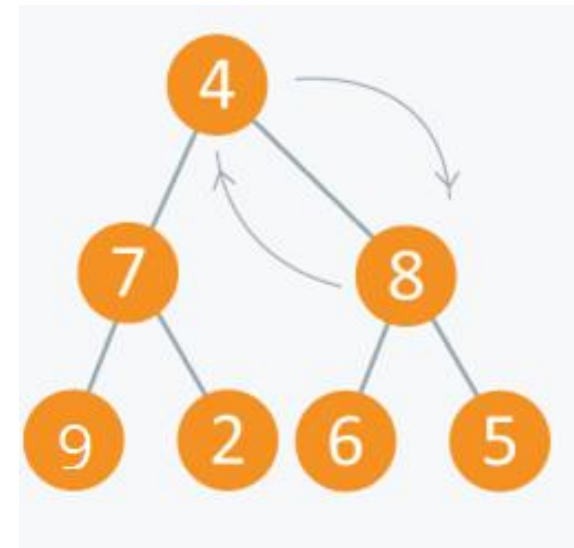
$$2^h + 2^{h-1} + 2^{h-2} = \frac{n}{2} + \frac{n}{4} + \frac{n}{8} = \frac{7n}{8} = 0.875n.$$

That is, almost 90% of the nodes of a complete binary tree reside in the 3 lowest levels. Thus the lesson to be learned is that when designing algorithms that operate on trees, it is important to be most efficient on the bottommost levels of the tree (as BuildHeap is) since that is where most of the weight of the tree resides.

Complexity Analysis

build_maxheap

```
void build_maxheap (int Arr[ ])  
{  
    for(int i = N/2 ; i >= 1 ; i-- )  
    {  
        max_heapify (Arr, i) ;  
    }  
}
```



Amortized Complexity: $O(N)$

Amortized complexity

Amortized complexity is the total expense per operation, evaluated over a sequence of operations.

The idea is to guarantee the total expense of the entire sequence, while permitting individual operations to be much more expensive than the amortized cost.

Example: The behavior of C++ `std::vector<>`. When `push_back()` increases the vector size above its pre-allocated value, it doubles the allocated length.

So a single `push_back()` may take $O(N)$ time to execute (as the contents of the array are copied to the new memory allocation).

However, because the size of the allocation was doubled, the next $N-1$ calls to `push_back()` will each take $O(1)$ time to execute. So, the total of N operations will still take $O(N)$ time; thereby giving `push_back()` an amortized cost of $O(1)$ per operation.

Application of Heap

1. Priority Queue
2. Heapsort

Priority Queue

Priority Queue is similar to queue where we insert an element from the back and remove an element from front, but with a one difference that the logical order of elements in the priority queue depends on the priority of the elements.

Naive Approach:

Suppose we have N elements and we have to insert these elements in the priority queue. We can use list and can insert elements in $O(N)$ time and can sort them to maintain a priority queue in $O(N \log N)$ time.

Efficient Approach:

We can use heaps to implement the priority queue. It will take $O(\log N)$ time to insert and delete each element in the priority queue.

Priority Queue

Supported Operations for max priority queue:

1. **maximum**(Arr) : It returns maximum element from the Arr.
2. **extract_maximum** (Arr) - It removes and return the maximum element from the Arr.
3. **increase_val** (Arr, i , val) - It increases the key of element stored at index i in Arr to new value val.
4. **insert_val** (Arr, val) - It inserts the element with value val in Arr.

Complexity

HEAP-MAXIMUM(A)

1 **return** $A[1]$

Complexity

HEAP-EXTRACT-MAX(A)

```
1  if  $A.heap\text{-}size < 1$   
2      error “heap underflow”  
3   $max = A[1]$   
4   $A[1] = A[A.heap\text{-}size]$   
5   $A.heap\text{-}size = A.heap\text{-}size - 1$   
6  MAX-HEAPIFY( $A, 1$ )  
7  return  $max$ 
```

Complexity

HEAP-INCREASE-KEY(A, i, key)

```
1  if  $key < A[i]$ 
2      error “new key is smaller than current key”
3   $A[i] = key$ 
4  while  $i > 1$  and  $A[\text{PARENT}(i)] < A[i]$ 
5      exchange  $A[i]$  with  $A[\text{PARENT}(i)]$ 
6       $i = \text{PARENT}(i)$ 
```

Complexity

MAX-HEAP-INSERT(A, key)

1 $A.heap-size = A.heap-size + 1$

2 $A[A.heap-size] = -\infty$

3 HEAP-INCREASE-KEY($A, A.heap-size, key$)

Priority Queue in STL

Supported functions:

- empty()
- size()
- top()
- push()
- pop()

```
#include <stdio.h>
#include <queue>
#include <vector>
#include <functional>
using namespace std;
```

```
int main()
{
    ///priority_queue<int> q;
    priority_queue<int, vector<int>, greater<int> > q;
}
```

^ Min-heap Priority Queue

Heapsort

HEAPSORT(A)

```
1  BUILD-MAX-HEAP( $A$ )
2  for  $i = A.length$  downto 2
3      exchange  $A[1]$  with  $A[i]$ 
4       $A.heap-size = A.heap-size - 1$ 
5      MAX-HEAPIFY( $A, 1$ )
```

Heapsort

Arr

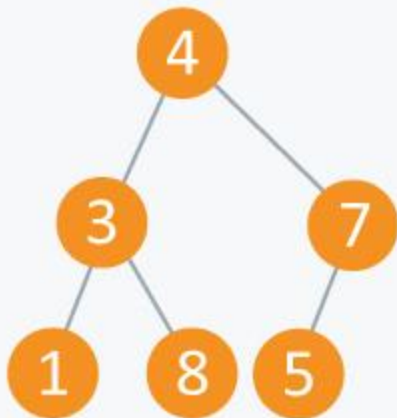
		4	3	7	1	8	5
	0	1	2	3	4	5	6

Heapsort

Arr

	4	3	7	1	8	5
0	1	2	3	4	5	6

Initial Elements

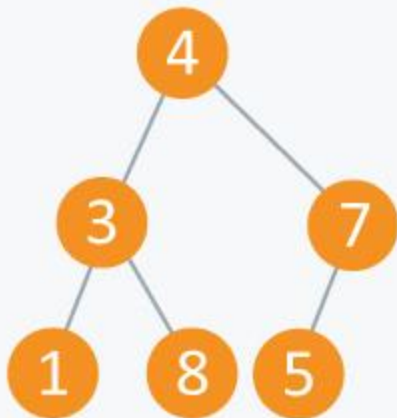


Heapsort

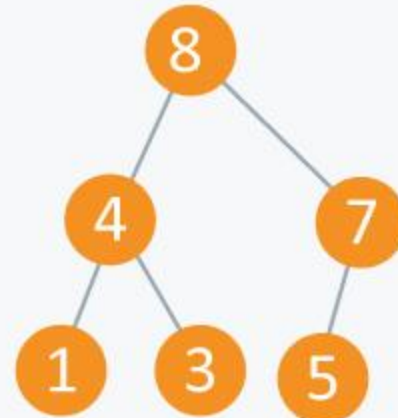
Arr

	4	3	7	1	8	5
0	1	2	3	4	5	6

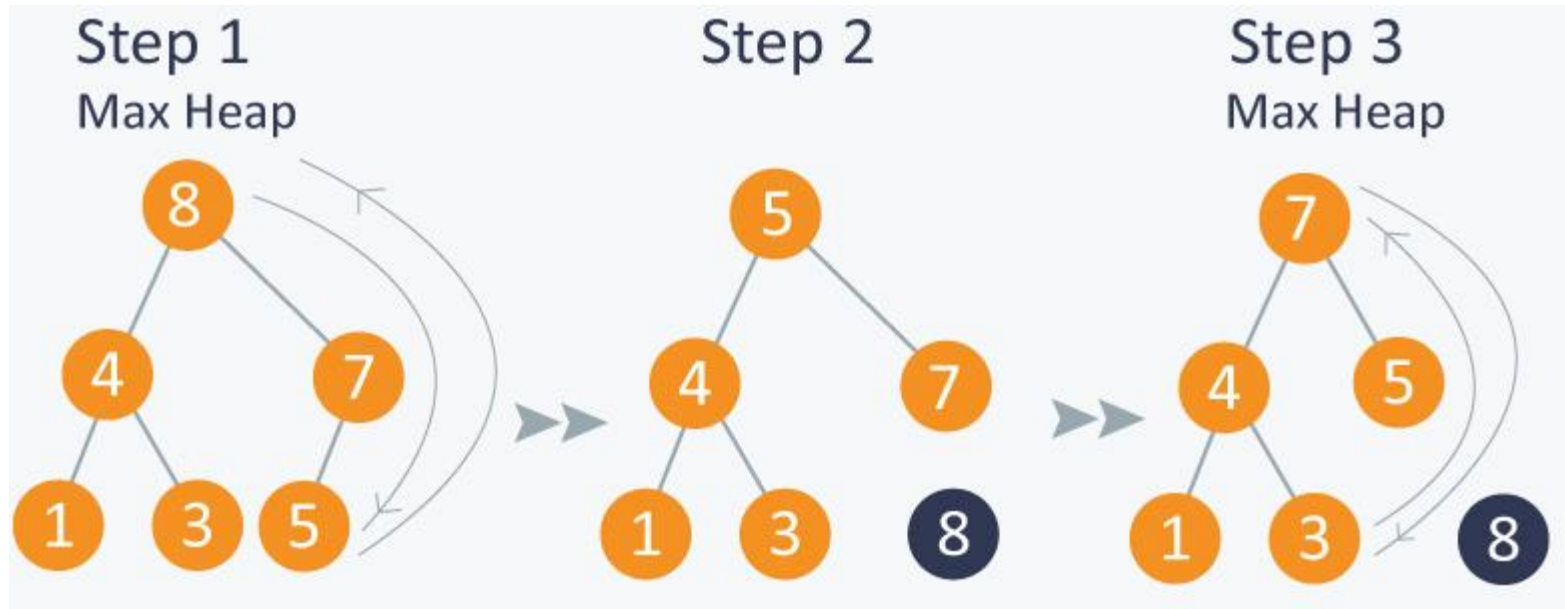
Initial Elements



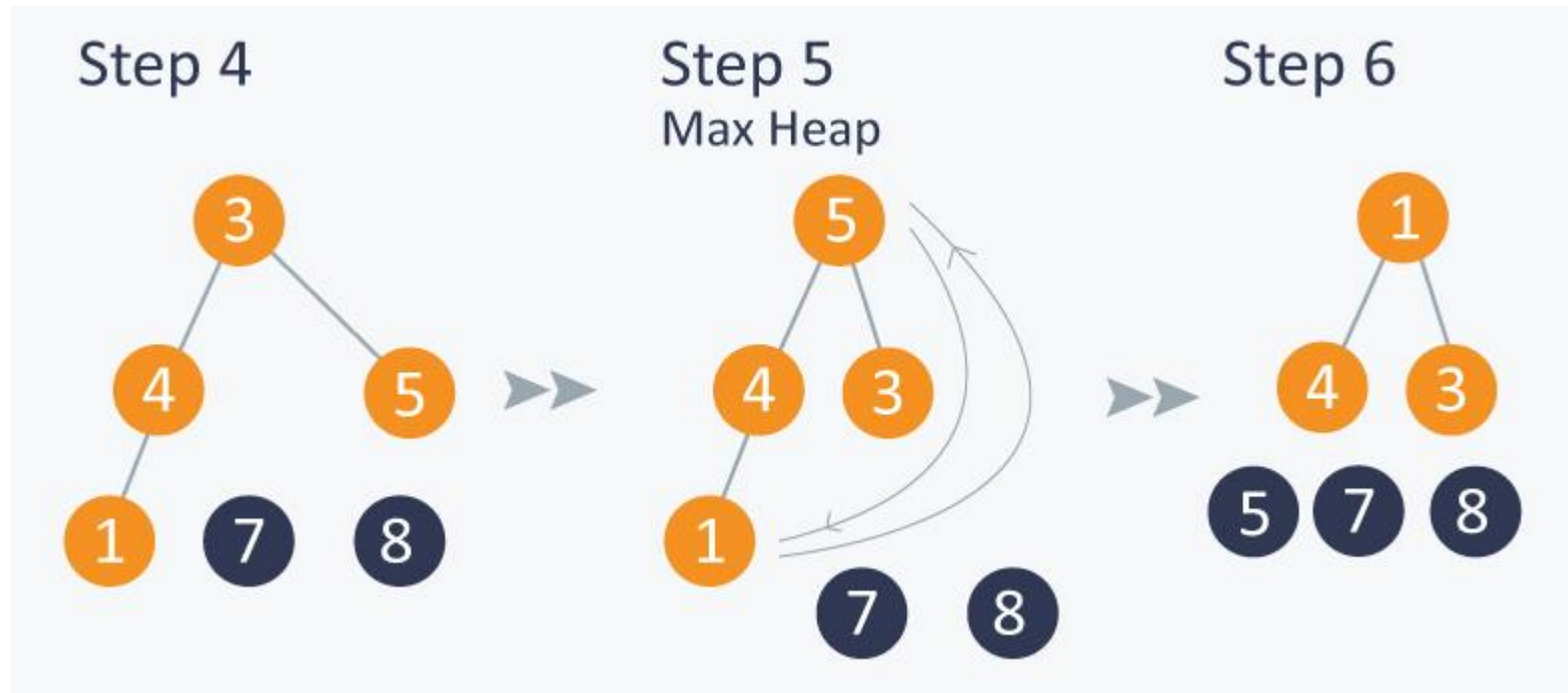
Max Heap



Heapsort

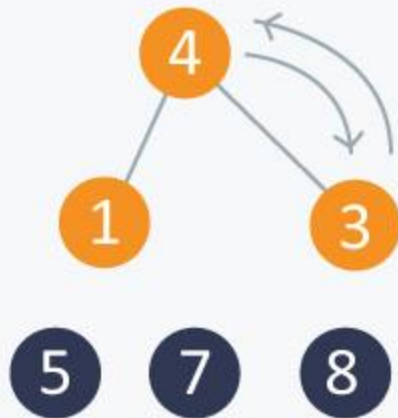


Heapsort

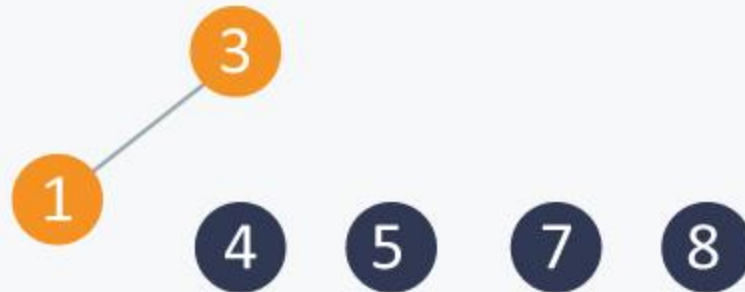


Heapsort

Step 7
Max Heap



Step 8



Heapsort

Step 9
Max Heap

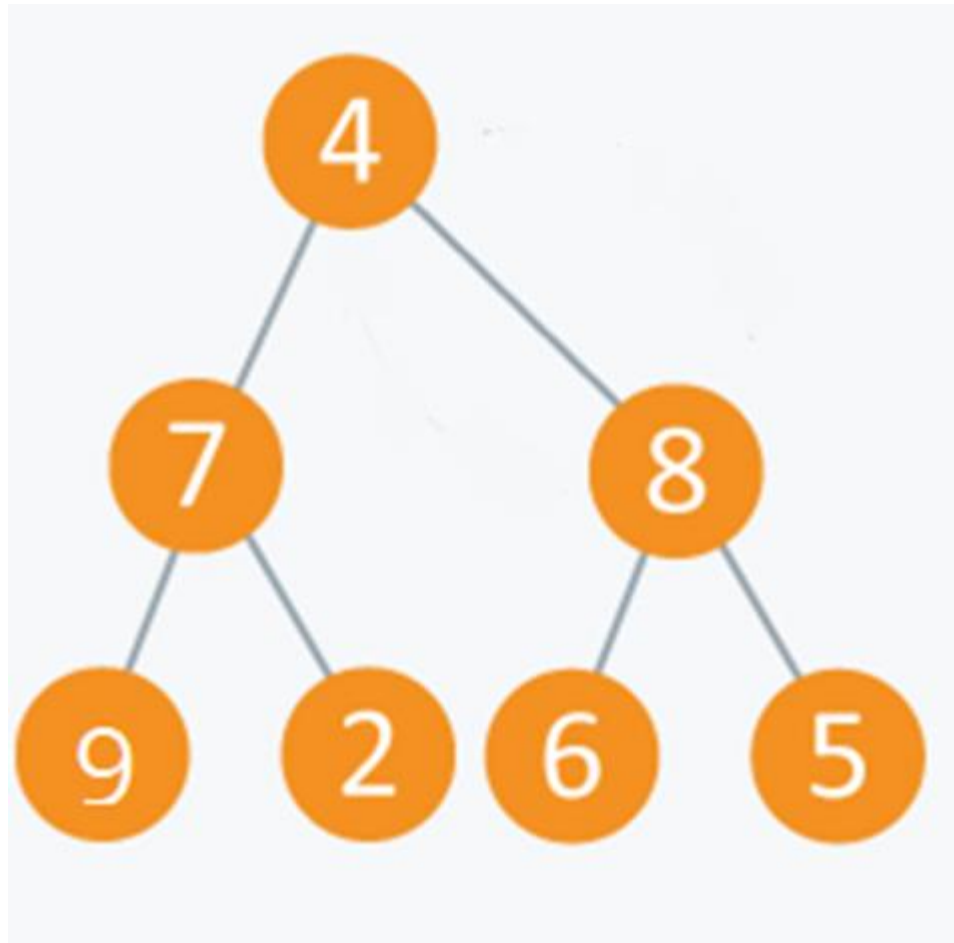


Step 10



Task

Apply Build-heap and show the steps of heapsort for the following graph



Heapsort Complexity

HEAPSORT(*A*)

```
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size - 1
5      MAX-HEAPIFY(A, 1)
```

Complexity:

max_heapify has complexity $O(\log N)$,

build_maxheap has complexity $O(N)$

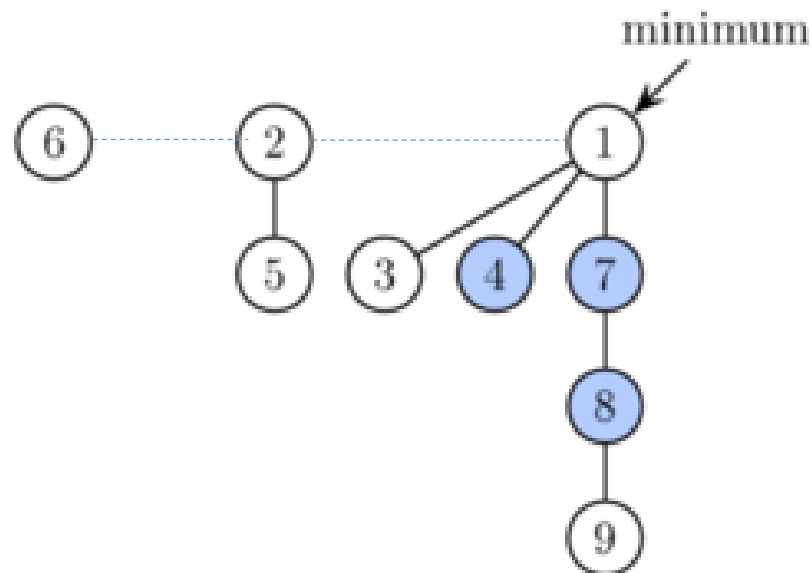
we run max_heapify $N-1$ times in heap_sort function,

therefore complexity of heap_sort function is $O(N \log N)$.

Fibonacci Heap

A Fibonacci heap is a collection of trees satisfying the minimum-heap property.

- Can be divided into sub-heaps
- All roots are connected
- A pointer to min node is present
- All nodes are connected via doubly linked list



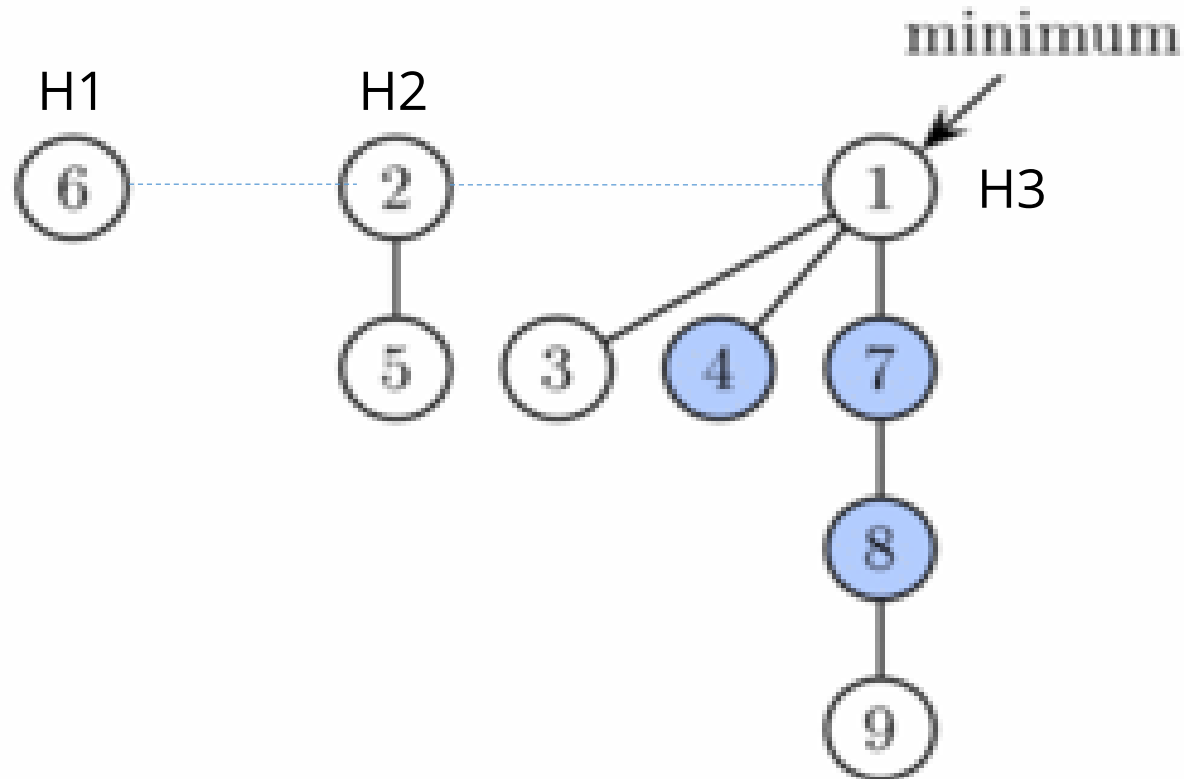
Rank of a sub-heap

Max number of children of a sub-heap

Rank of H1 = 0

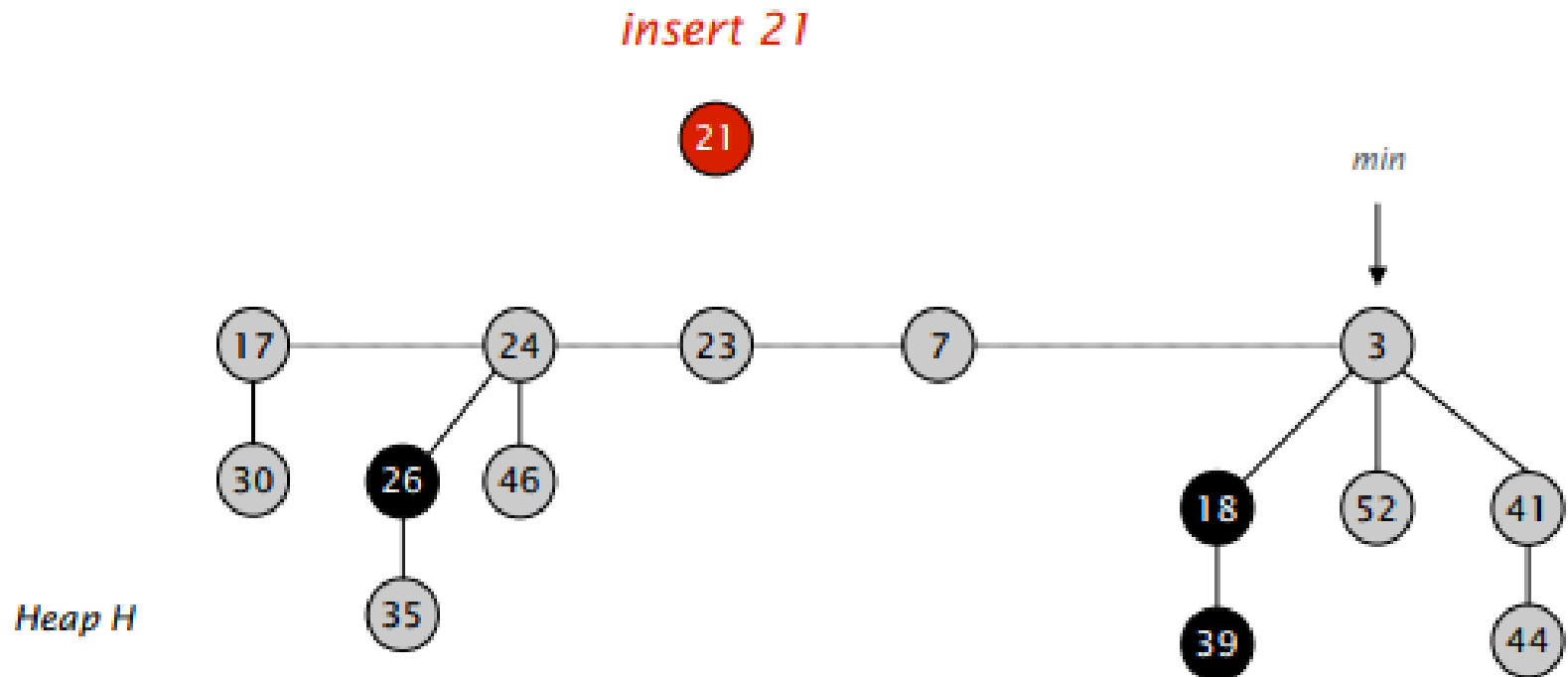
Rank of H2 = 1

Rank of H3 = 3



Push Operation

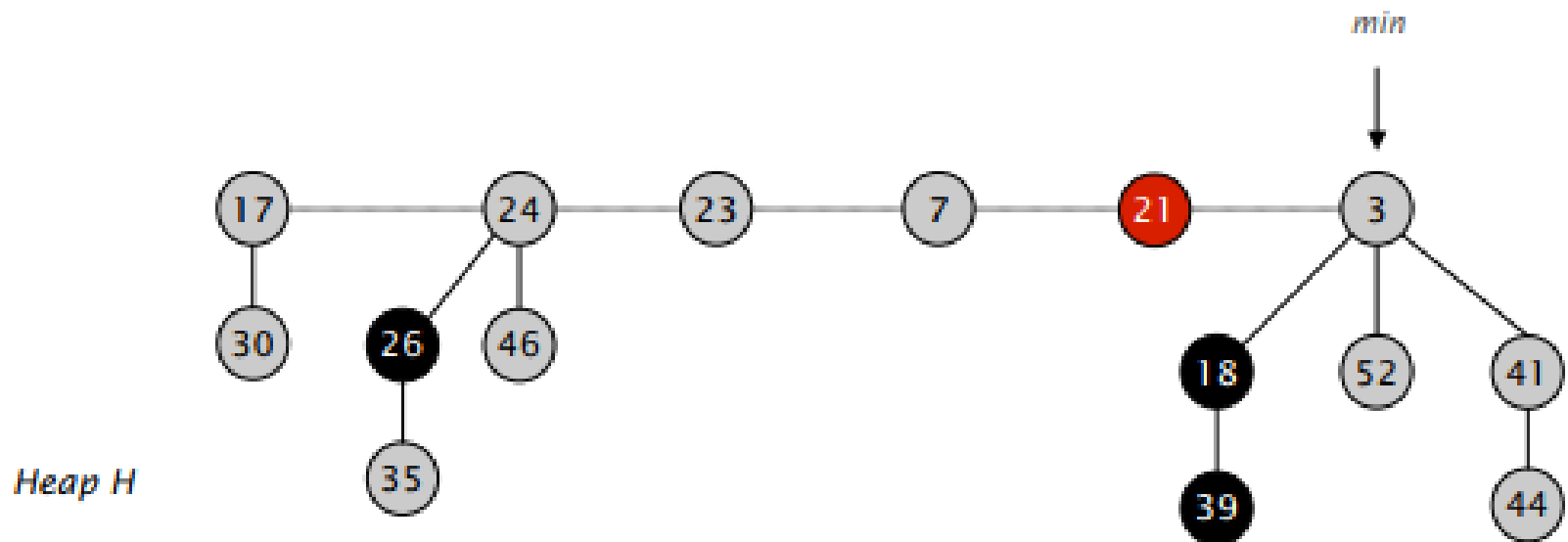
A singleton Tree is created, update min pointer if necessary



Push Operation

A singleton Tree is created, update min pointer if necessary

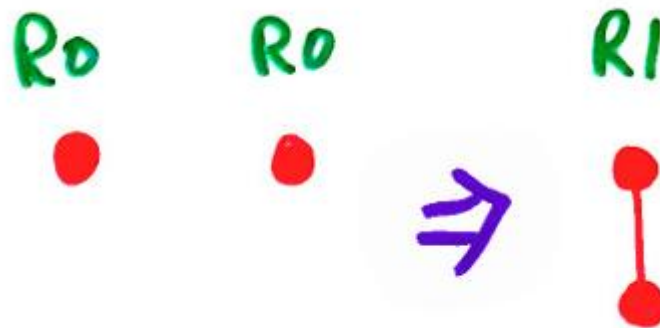
insert 21



Pop (Extract Min)

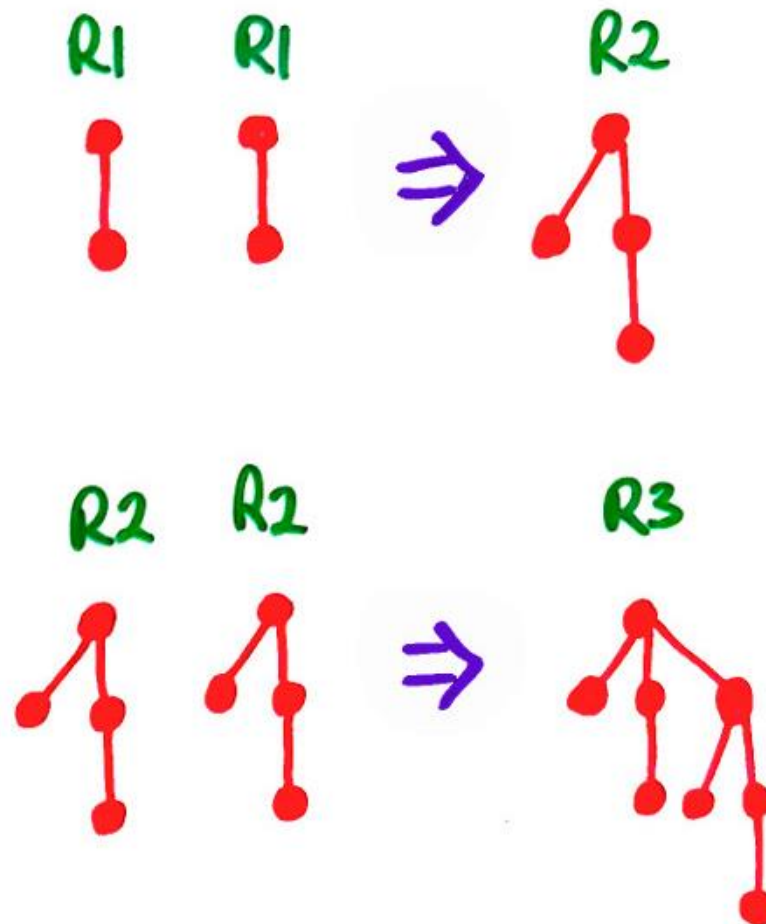
Cleanup (Merging) is done after popping a node.

As part of the extract-min operation in a fibonacci heap, the heap “cleans up” itself by merging sub-heap trees with the same number of immediate children, called its rank, starting at the smallest rank first. If two rank 0 (no children) sub-heap trees are found in the heap, they are merged into a single rank 1 (1 child) sub-heap.



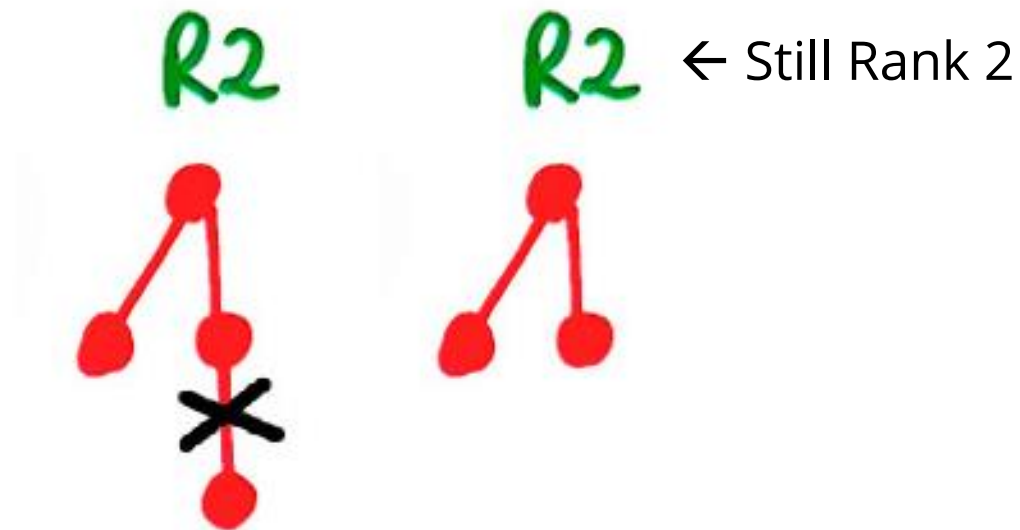
Pop (Extract Min)

This process is repeated. If two rank 1 sub-heaps now exist, they are merged into a single rank 2 sub-heap, etc.



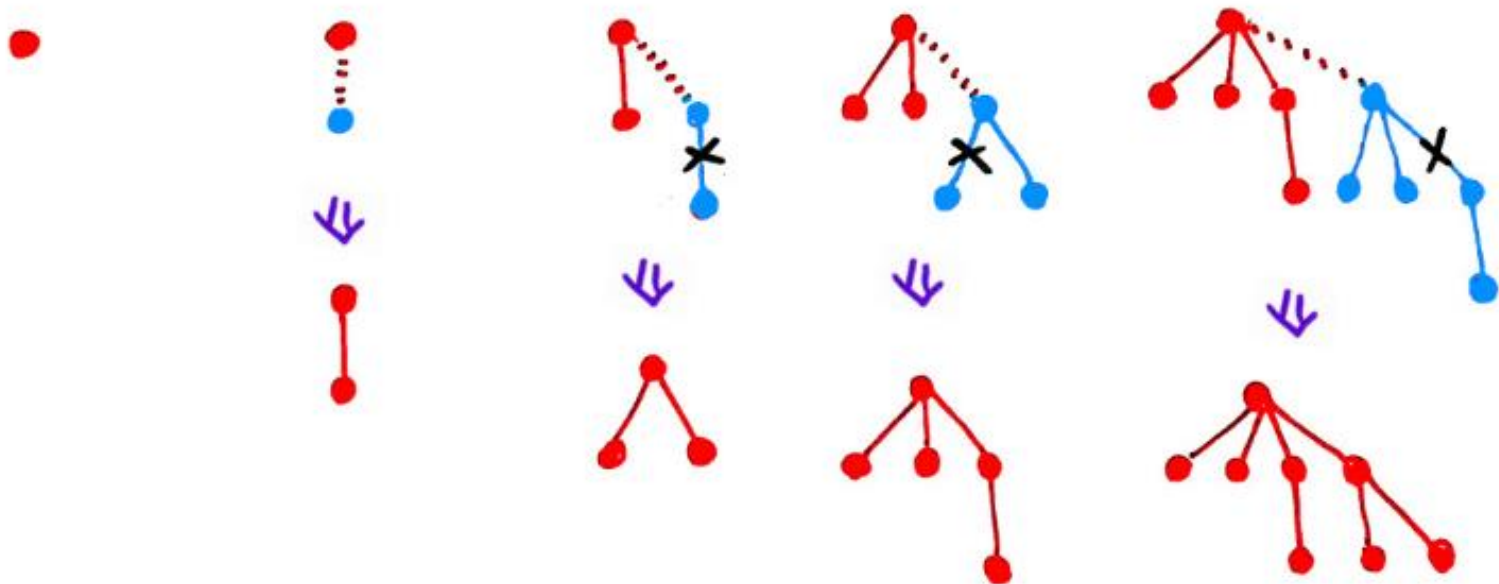
Pop (Extract Min)

Another important point to note is that when two trees are merged, the resulting tree can have more than the minimum required number of nodes for its rank.



Pop (Extract Min)

Notice anything about the final node counts? 1, 2, 3, 5, 8... which is the Fibonacci Sequence



Reference

- <https://www.hackerearth.com/practice/notes/heaps-and-priority-queues/>
- <https://amoffat.github.io/blog/fibonacci-heap>
- <https://www.cs.princeton.edu/~wayne/teaching/fibonacci-heap.pdf>