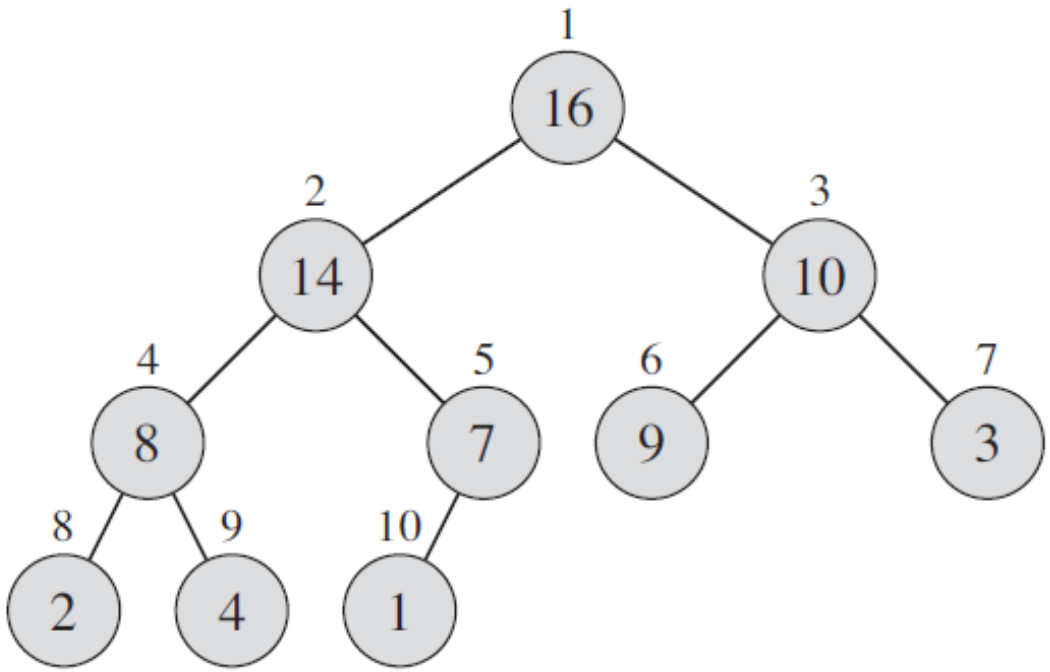
Binary, Binomial Heaps

Applications: Heap Sort, Priority Queue

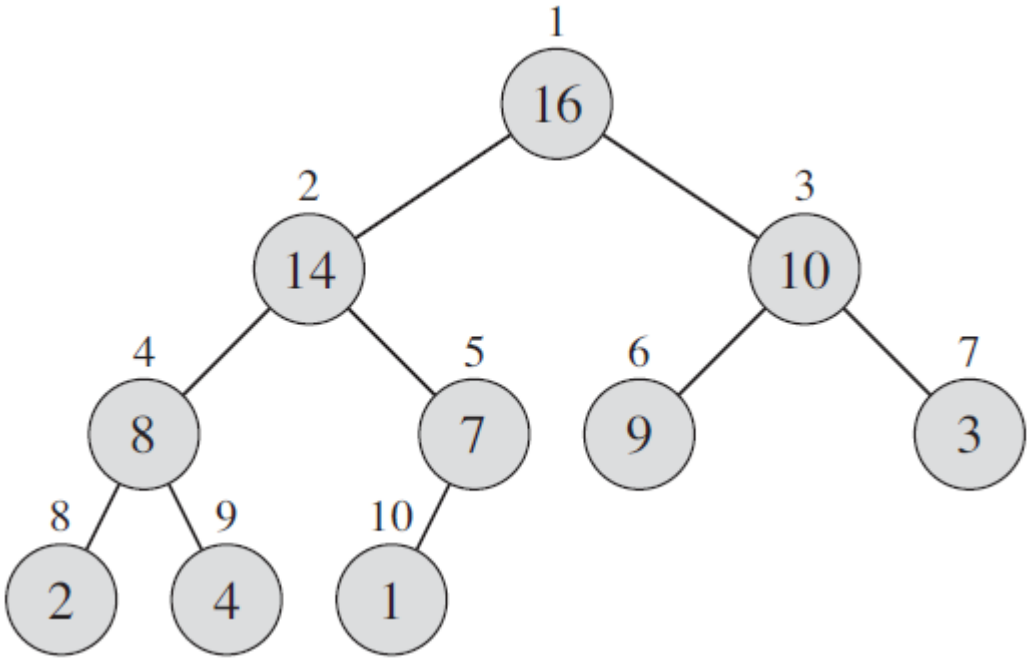
Heap

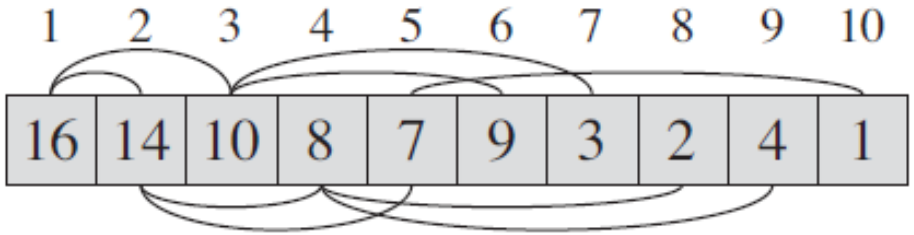
- The (binary) heap data structure is an array object that we can view as a nearly complete binary tree.
- Two key properties:
  - The parent node of any given node has a higher priority (or a higher value) than its children.
  - The tree is complete, meaning all levels of the tree are filled except possibly the last level, which is filled from left to right.



Heaps come in two main varieties: Max heaps and Min heaps.

- In a Max heap, the parent node has a greater value than its children
- In a Min heap, the parent node has a smaller value than its children





In a ***max-heap***, the ***max-heap property*** is that for every node i other than the root,

$$A[\text{PARENT}(i)] \geq A[i]$$

```
PARENT(i)
1  return ⌊i/2⌋
```

```
LEFT(i)
1  return 2i
```

```
RIGHT(i)
1  return 2i + 1
```

Complexity

- Viewing a heap as a tree, the ***height*** of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf
- the height of the heap to be the height of its root.

- Since a heap of n elements is based on a complete binary tree, its height is O(logn)

- the basic operations on heaps run in time at most proportional to the height of the tree and thus take O(logn) time.

Heap: Insertion

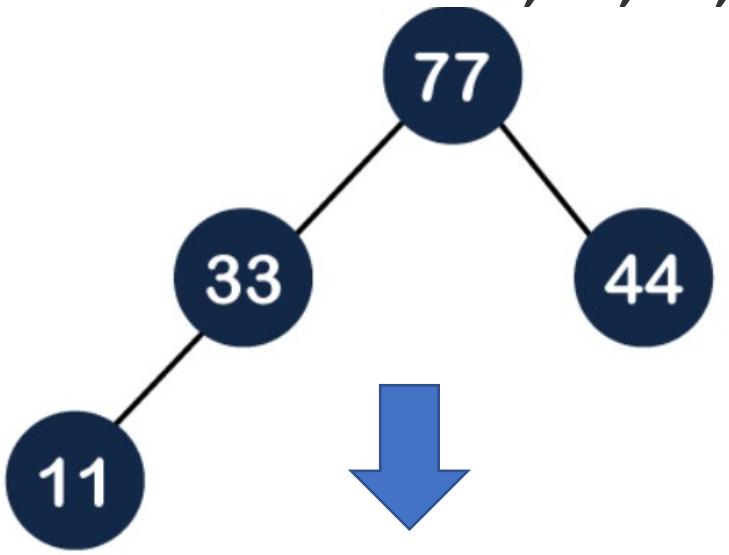
1. Add the new element to the next available spot in the **array (!)** that represents the **binary heap**. This is typically the first empty spot at the end of the array.
2. Compare the new element with its parent. If the new element is greater (or smaller) than its parent and violates the heap property, swap the new element with its parent.

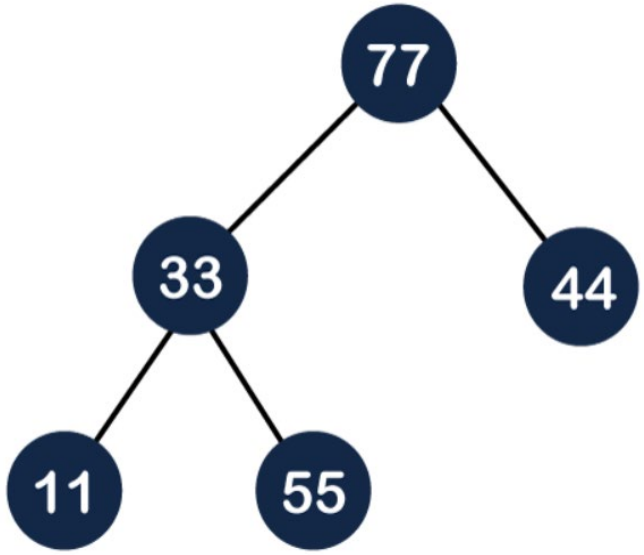
3. Repeat step 2 until the heap property is restored.

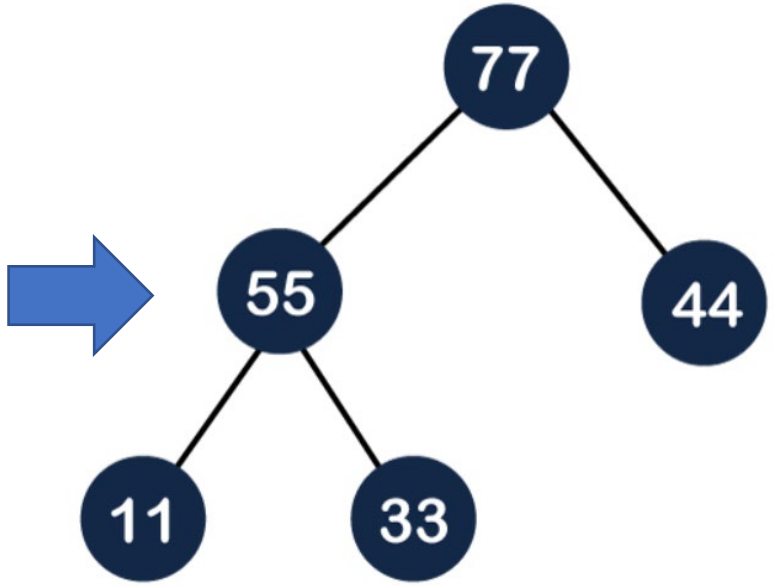
Overall Complexity of insert operation is O(log n), n is the total no of nodes.

Heap: Insertion

44, 33, 77, 11, 55, 88, 66








```
function insertMaxHeap(heap, value):
    heap.append(value)  # add new element to end of array
    index = len(heap) - 1
    parent = index // 2
    while index > 1 and heap[index] > heap[parent]:
        # swap new element with its parent if it violates heap property
        heap[index], heap[parent] = heap[parent], heap[index]
        index = parent
        parent = index // 2
```

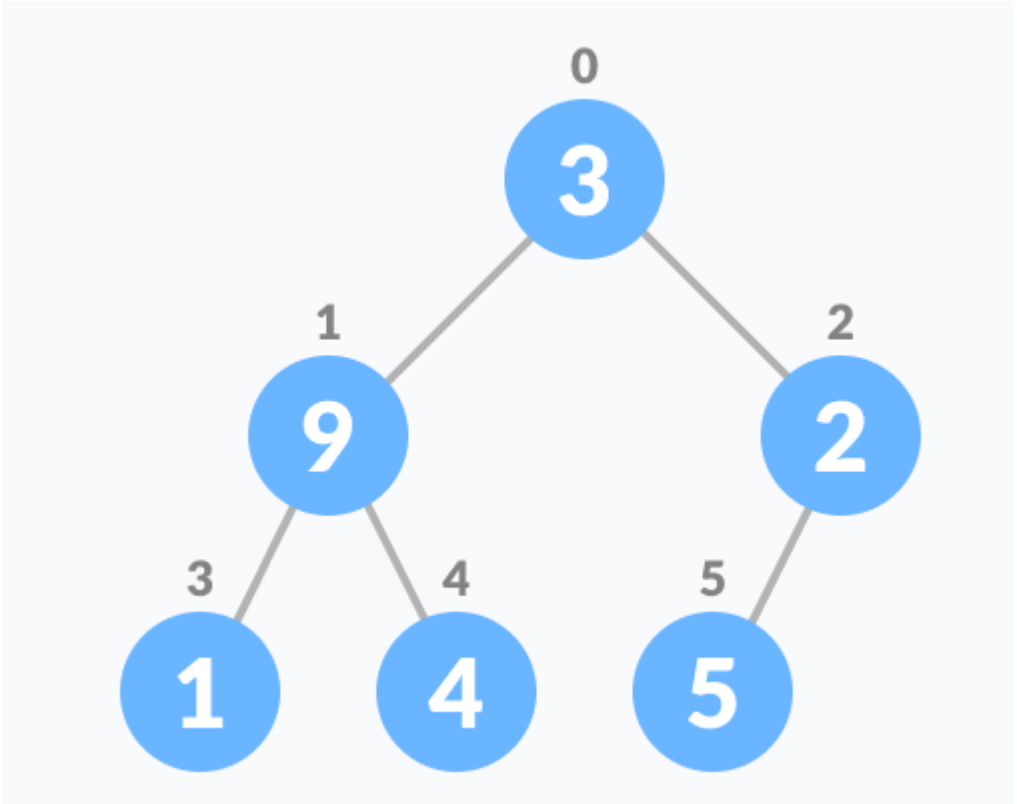
Heapify

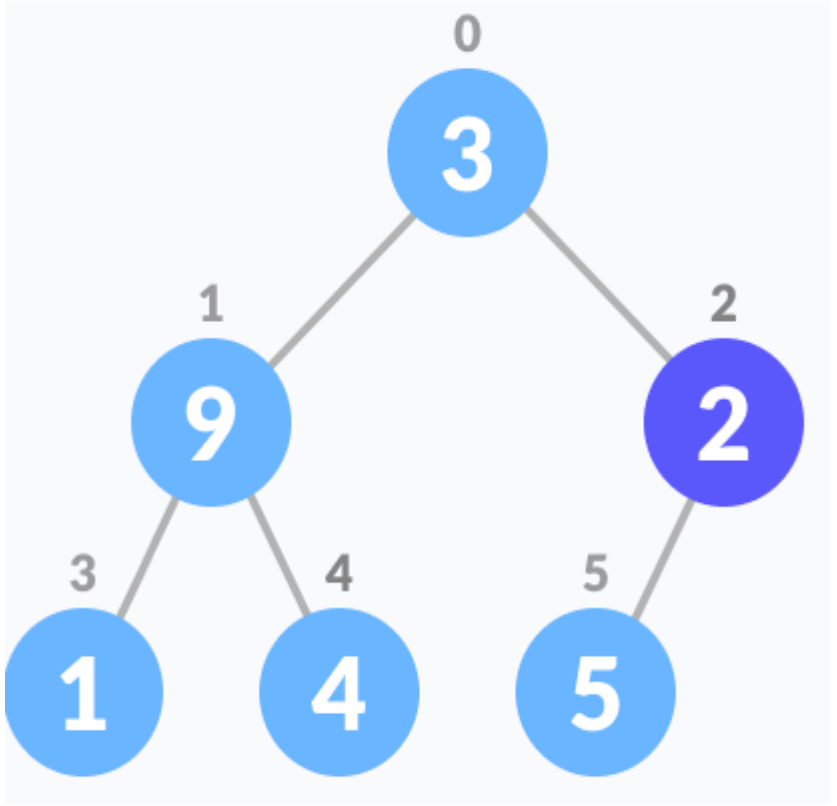
- A process of converting an array or a list of elements into a heap data structure, such as a binary heap, without changing the relative order of the elements.

- A time complexity of O(n), where n is the number of elements in the array.

```python
def heapify_max(arr):
    # start from the last parent node and work backwards
    for i in range(len(arr) // 2, 0, -1):
        _heapify_max(arr, i)

def _heapify_max(arr, i):
    # get indices of left and right children
    left = 2 * i
    right = 2 * i + 1

    # assume parent is the maximum value
    largest = i

    # check if left child is larger than parent
    if left <= len(arr) - 1 and arr[left] > arr[largest]:
        largest = left

    # check if right child is larger than parent
    if right <= len(arr) - 1 and arr[right] > arr[largest]:
        largest = right

    # if either child is larger, swap with parent and recursively heapify
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        _heapify_max(arr, largest)
```

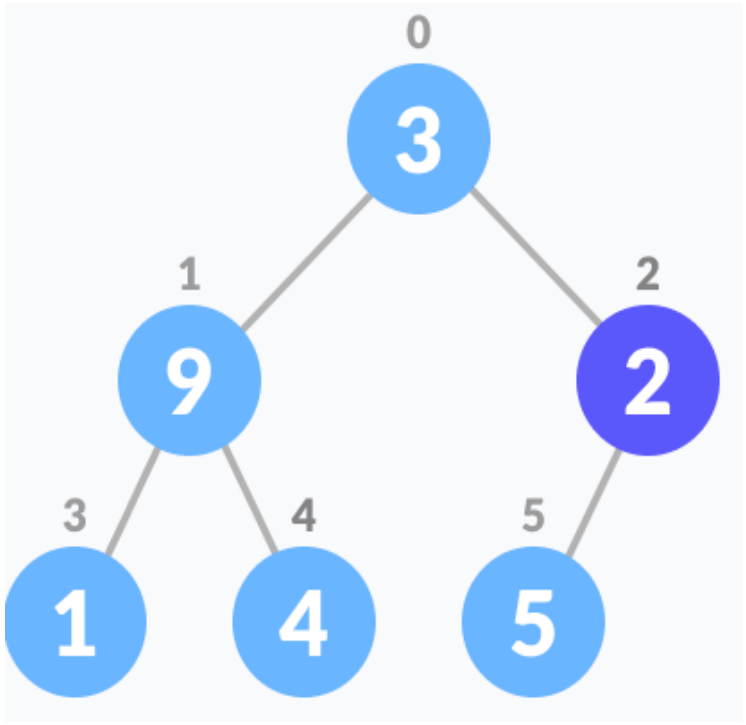
Heapify

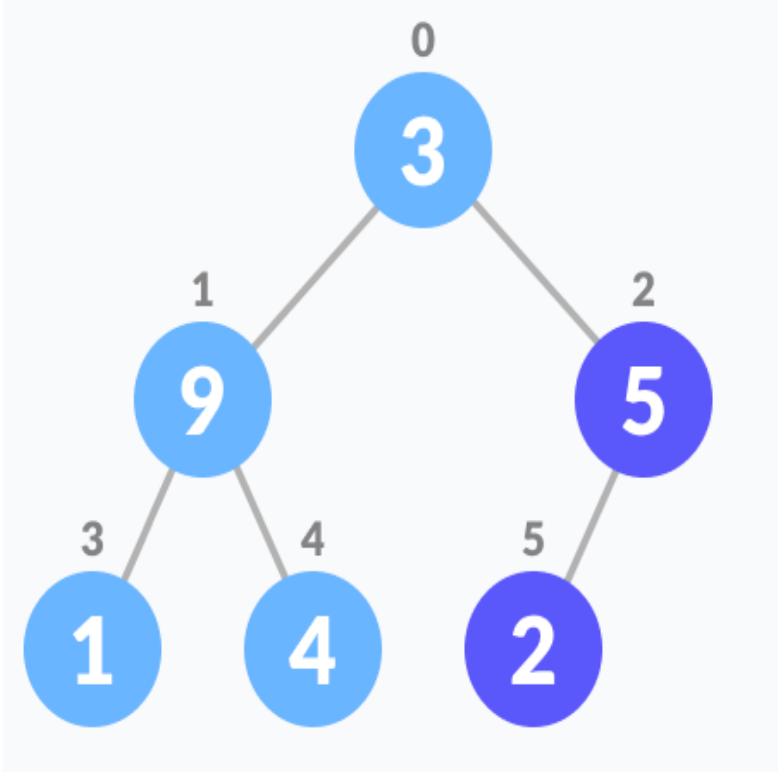


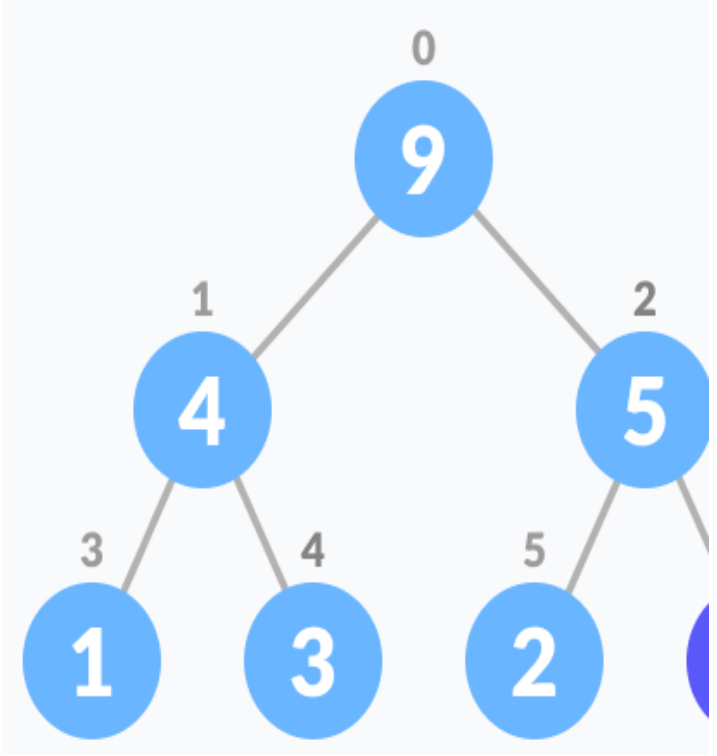


Heapify

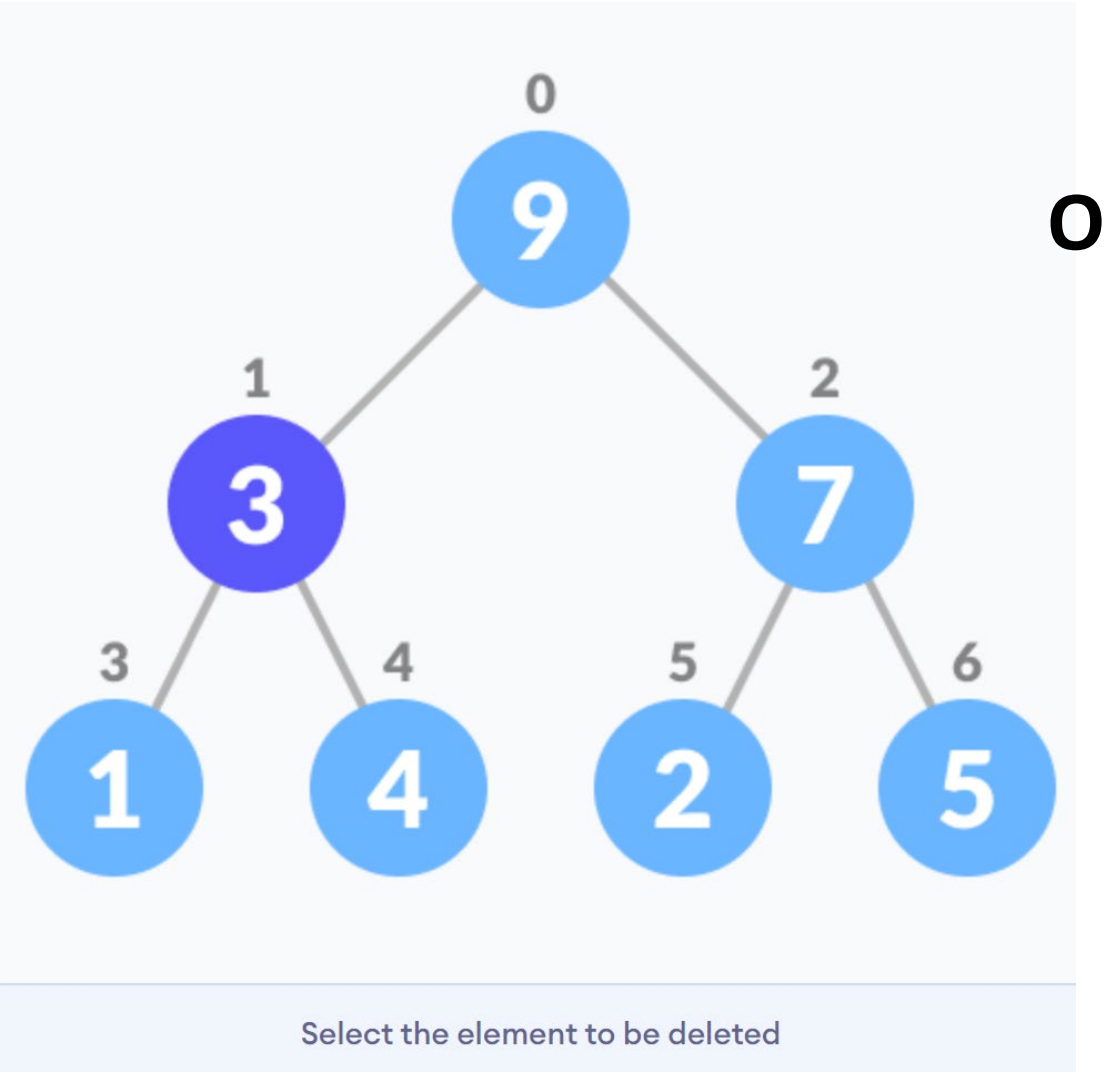
Start from the first index of non-leaf node whose index is given by n/2 - 1





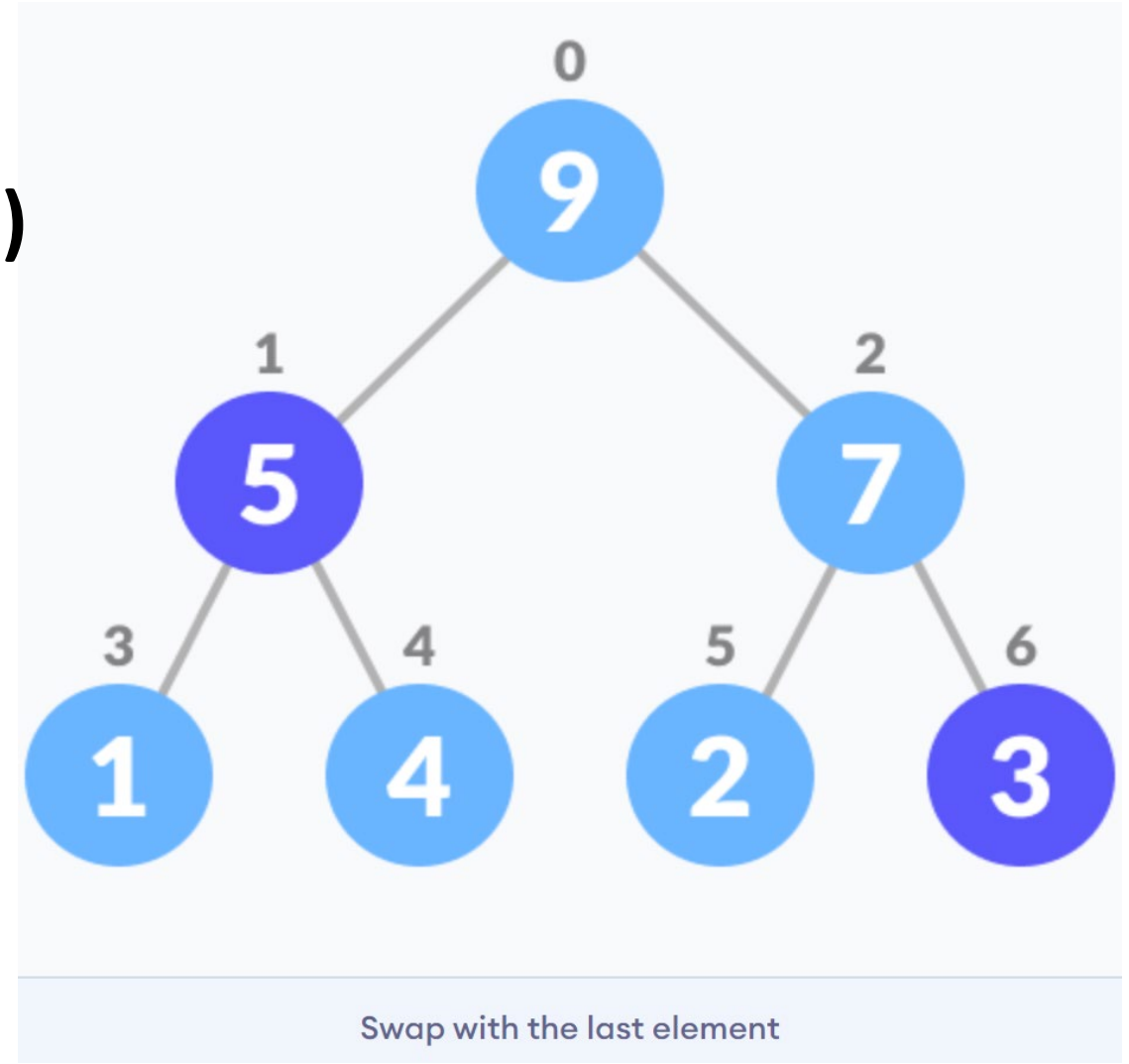


Delete



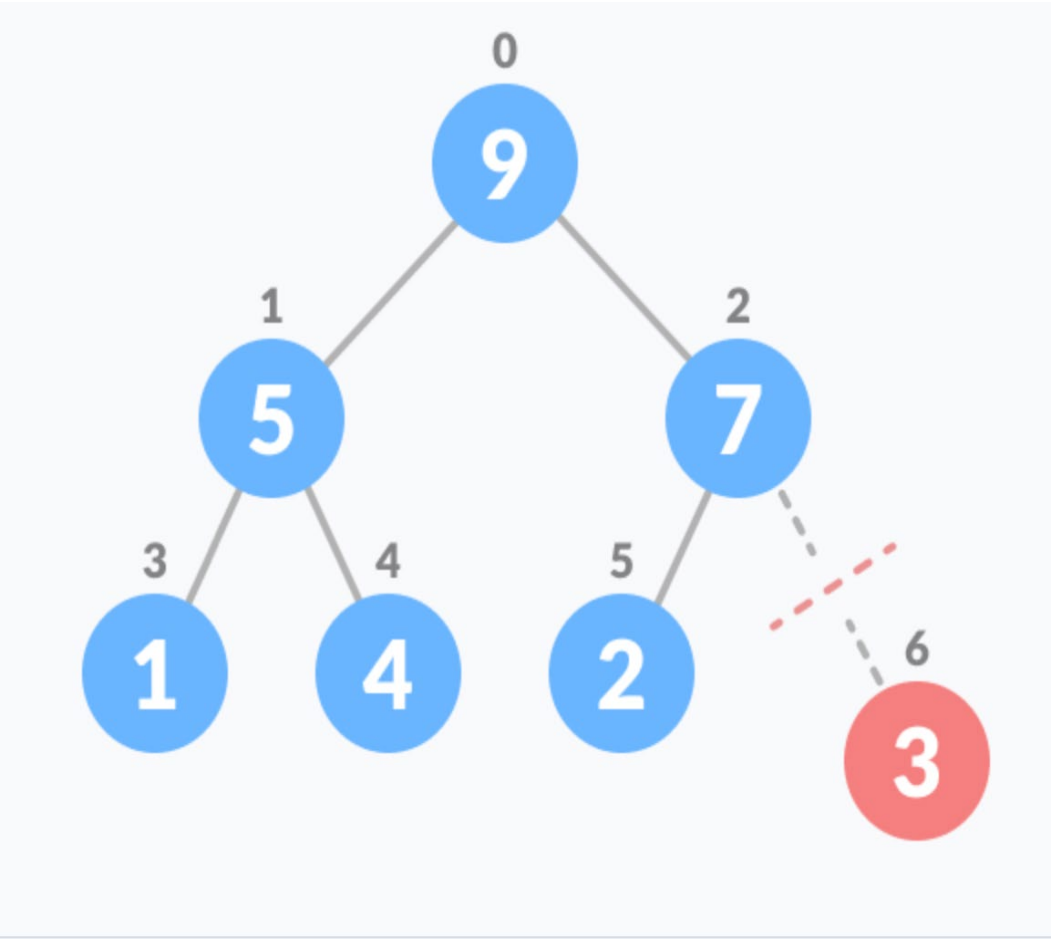
Select the element to be deleted

O(log n)

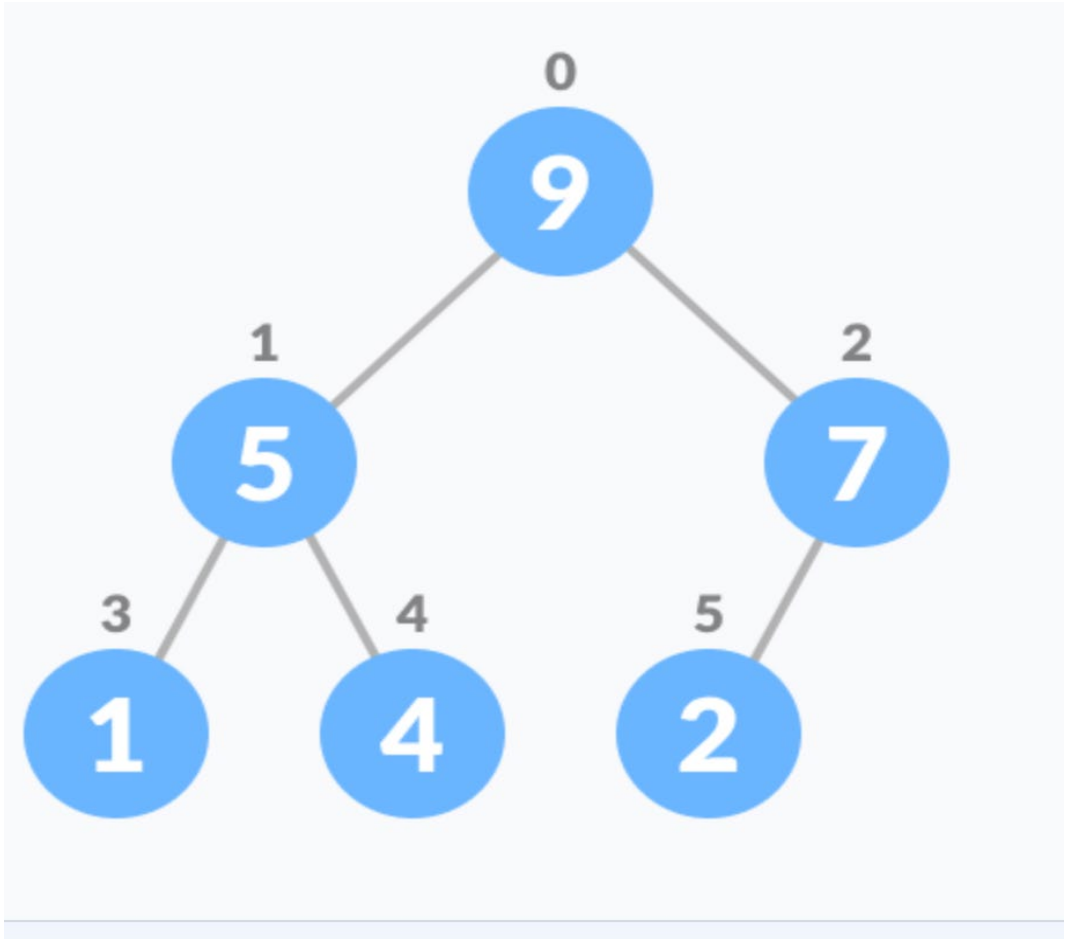


Swap with the last element

Delete



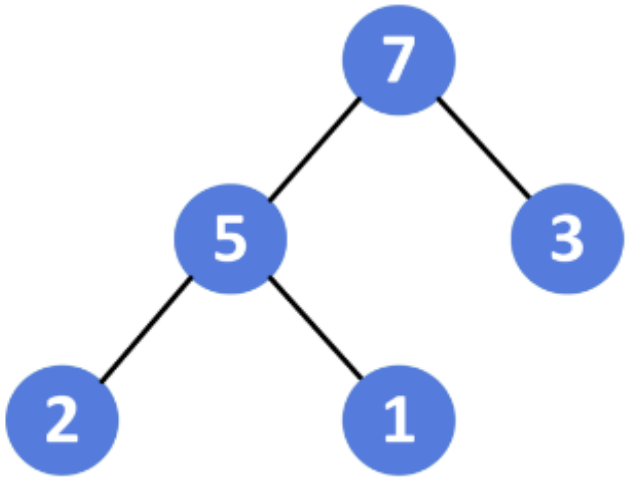
Remove the last element

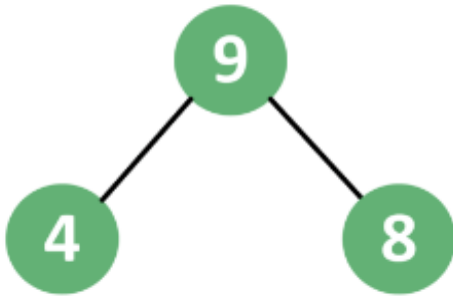


Heapify the array

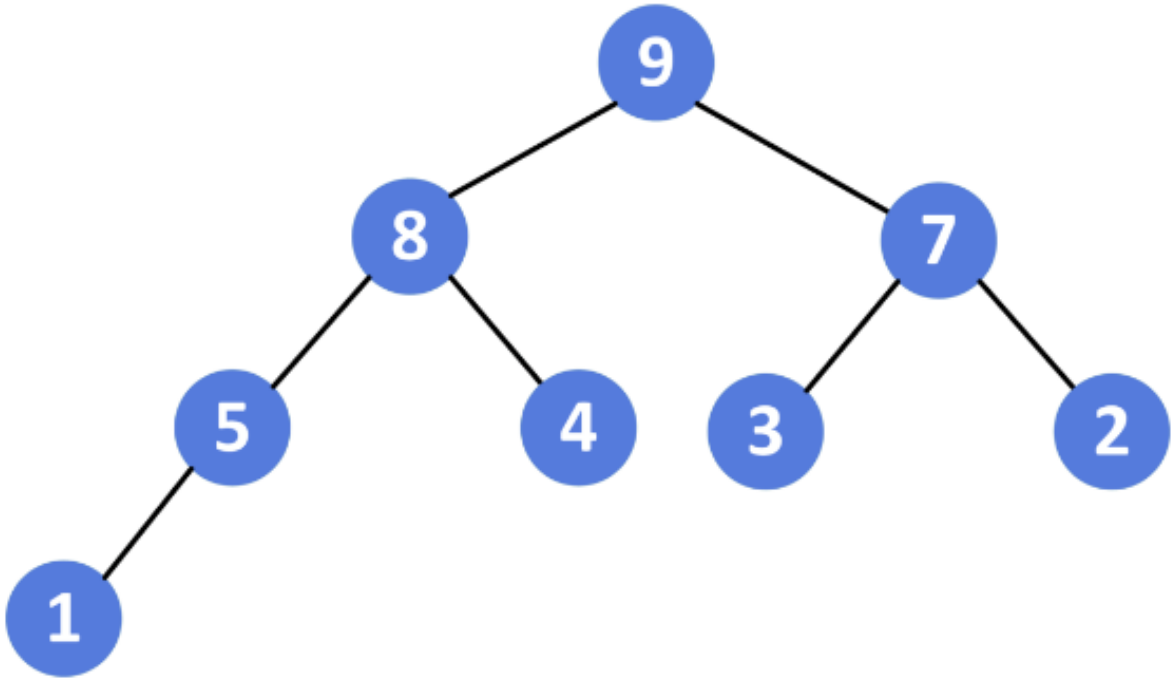
Merge

- H_1: $[7, 5, 3, 2, 1]$
- H_2: $[9, 4, 8]$





- H : $[9, 8, 7, 5, 4, 3, 2, 1]$



The time complexity of this approach is $\mathbf{O(N + M)}$, where N is the number of vertices of the first heap H_1, and M is the number of vertices of the second heap H_2. The reason behind this complexity is the same as the complexity of building a max heap.

The space complexity here is $\mathbf{O(N + M)}$. since we're storing the vertices of the new heap.

Merge

If you add the elements one by one using the heap insert operation, the cost of adding n elements to the heap and heapifying it would be O(n log(n)). This is because each insertion operation takes O(log(n)) time, and you perform n such operations.

However, if you have all n elements in an unsorted array, you can build a heap from the array using the bottom-up heap construction algorithm in O(n) time.

The time complexity of this approach is $\mathbf{O(N+M)}$, where N is the number of vertices of the first heap H_1, and M is the number of vertices of the second heap H_2. The reason behind this complexity is the same as the complexity of building a max heap.

The space complexity here is $\mathbf{O(N+M)}$. since we're storing the vertices of the new heap.

Heap Operation

❑Peek (Find max/min)

Peek operation returns the maximum element from Max Heap or minimum element from Min Heap without deleting the node.

For both Max heap and Min Heap

❑Extract-Max/Min

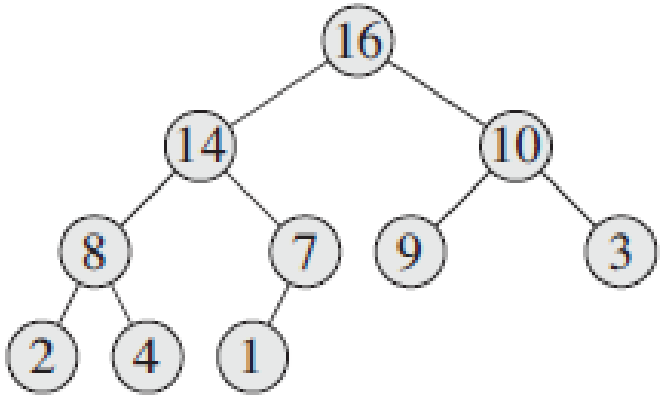
- Extract-Max returns the node with maximum value after removing it from a Max Heap
- Extract-Min returns the node with the minimum value after removing it from Min Heap.

Heap Sort

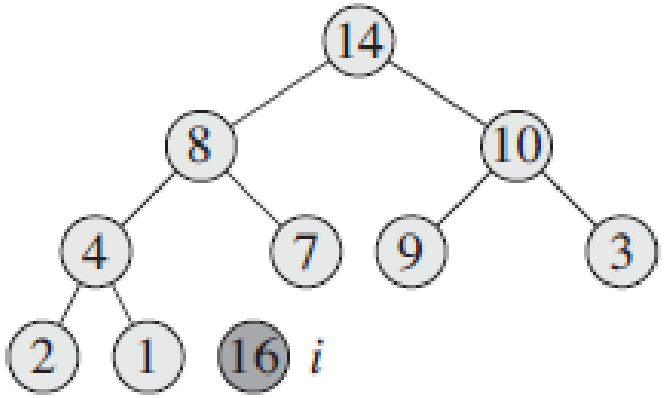
```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

The HEAPSORT procedure takes time $O(n \lg n)$, since the call to BUILD-MAX-HEAP takes time $O(n)$ and each of the $n - 1$ calls to MAX-HEAPIFY takes time $O(\lg n)$.

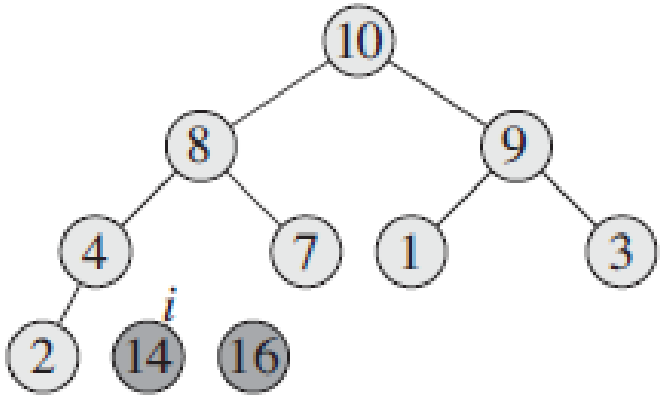
Heap Sort



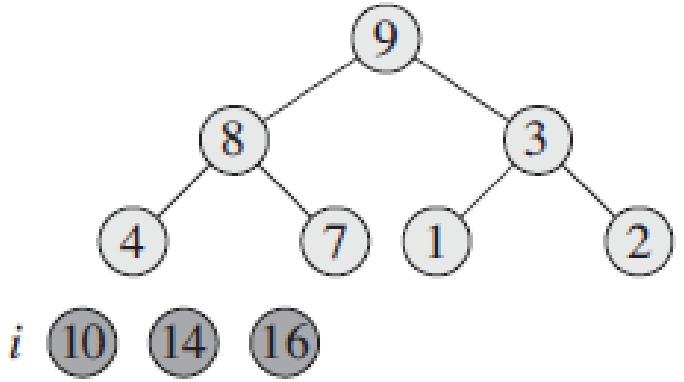
(a)



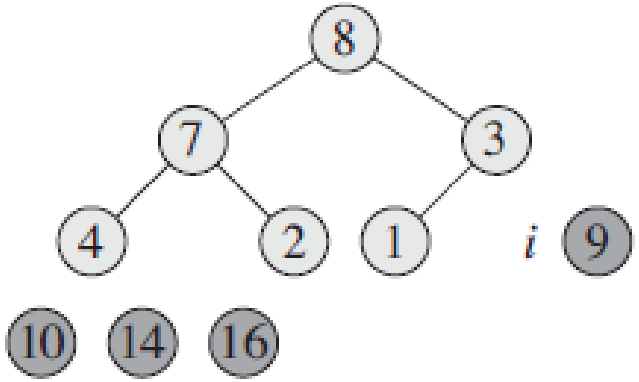
(b)



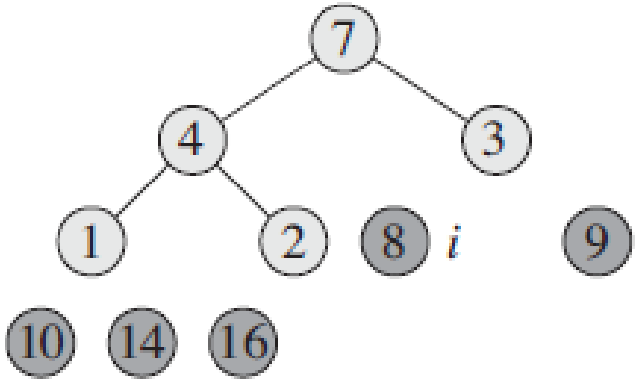
(c)



(d)

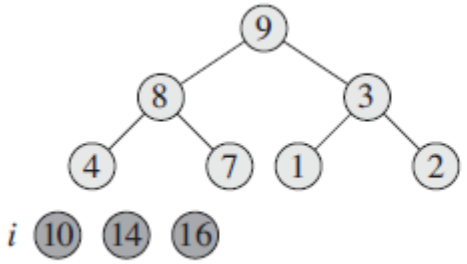


(e)

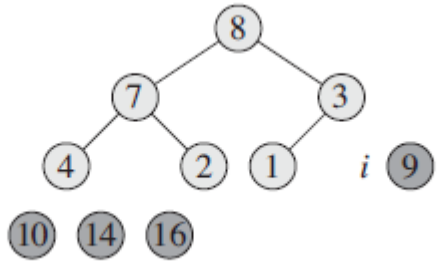


(f)

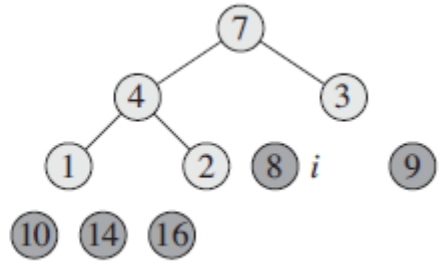
Heap Sort



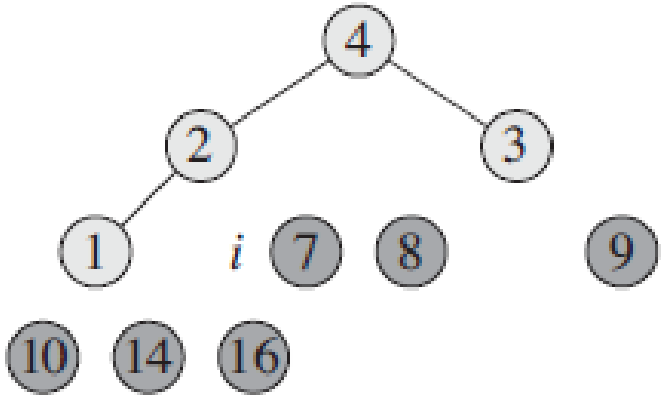
(d)



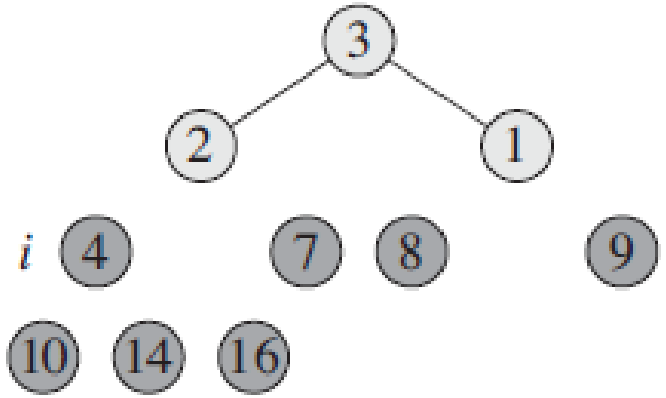
(e)



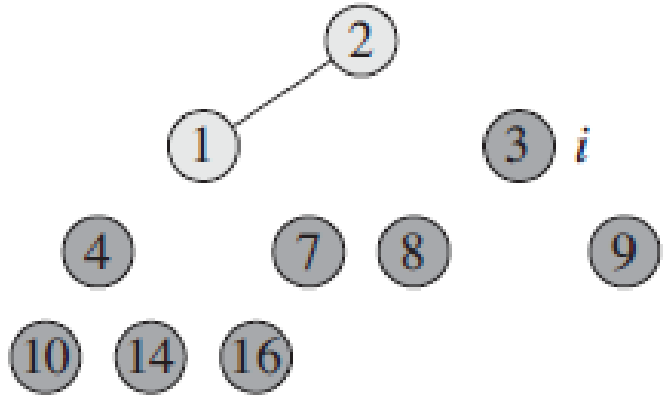
(f)



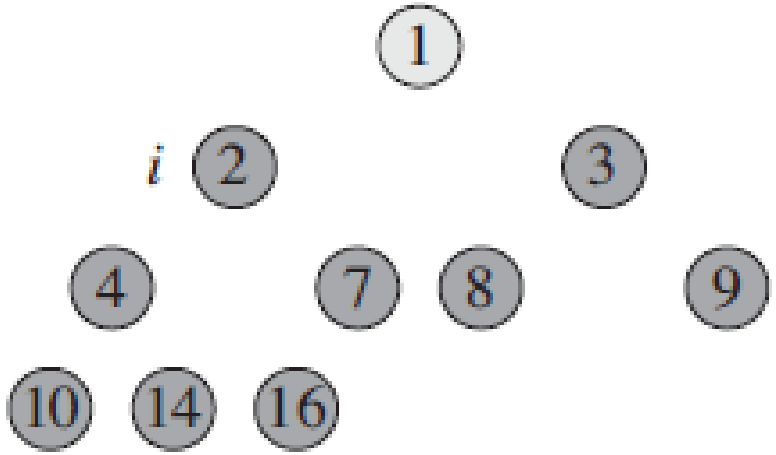
(g)



(h)



(i)



(j)

A	1	2	3	4	7	8	9	10	14	16

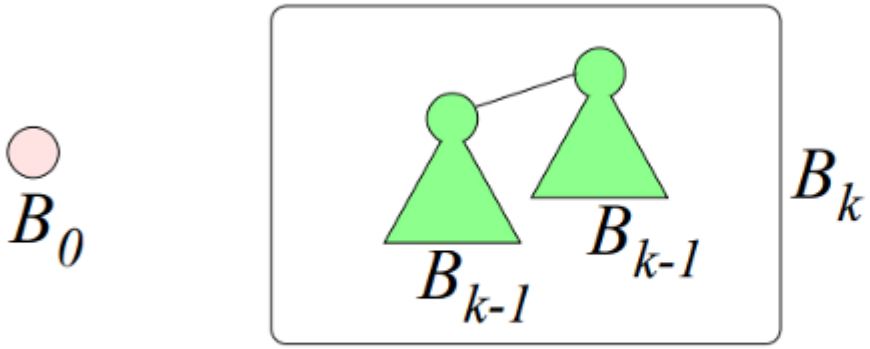
(k)

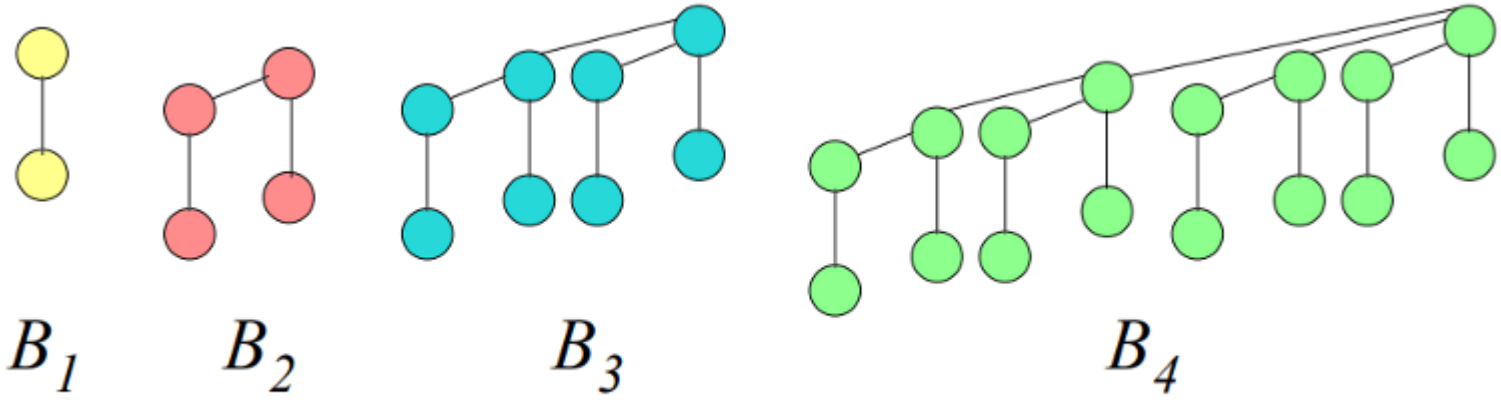
Binomial Heap

Binomial Tree

A binomial tree B_k is an ordered tree defined recursively.

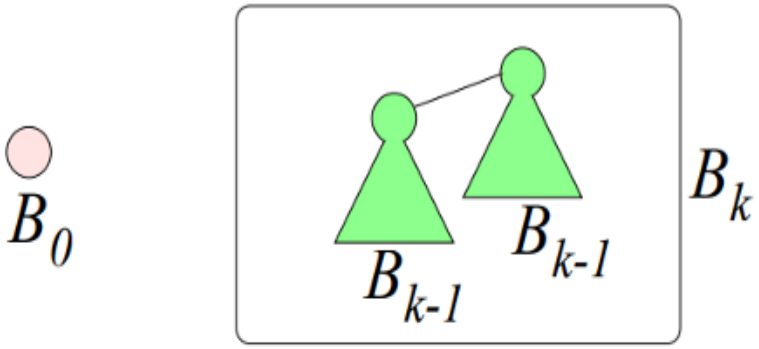
- B_0 consists of a single node.
- For $k \geq 1$, B_k is a pair of B_{k-1} trees, where the root of one B_{k-1} becomes the leftmost child of the other.

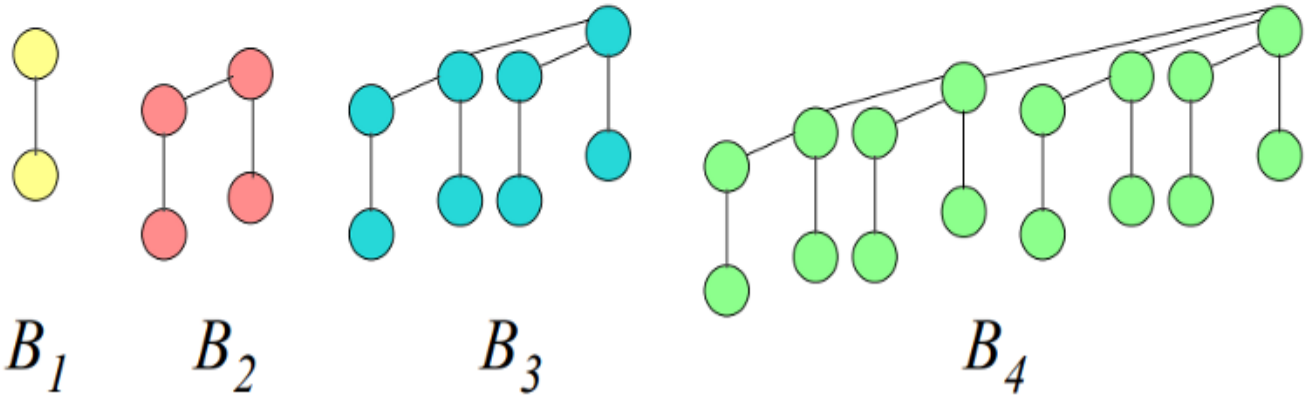




Binomial Heap

Binomial Tree





how many different ways you can choose *i* items from k items set without repetition and without order.

Properties of Binomial Trees

Lemma A For all integers $k \geq 0$, the following properties hold:

1. B_k has 2^k nodes.
2. B_k has height k.
3. For $i = 0, \ldots, k$, B_k has exactly $\binom{k}{i}$ nodes at depth i.
4. The root of B_k has degree k and all other nodes in B_k have degree smaller than k.
5. If $k \geq 1$, then the children of the root of B_k are $B_{k-1}, B_{k-2}, \cdots, B_0$ from left to right.

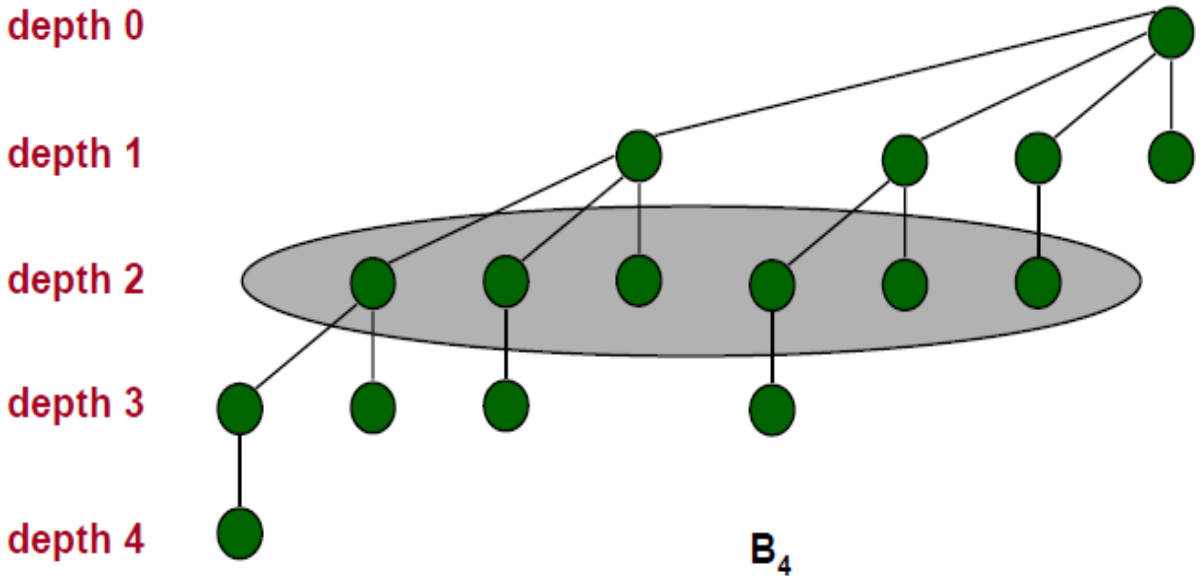
n choose k or binomial coefficient or simply combinations

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Binomial Heap

Binomial Tree

$\binom{4}{2} = 6$ $\binom{4}{2} = \frac{4!}{2!(4-2)!} = \frac{4 \cdot 3}{2 \cdot 1} = 6$



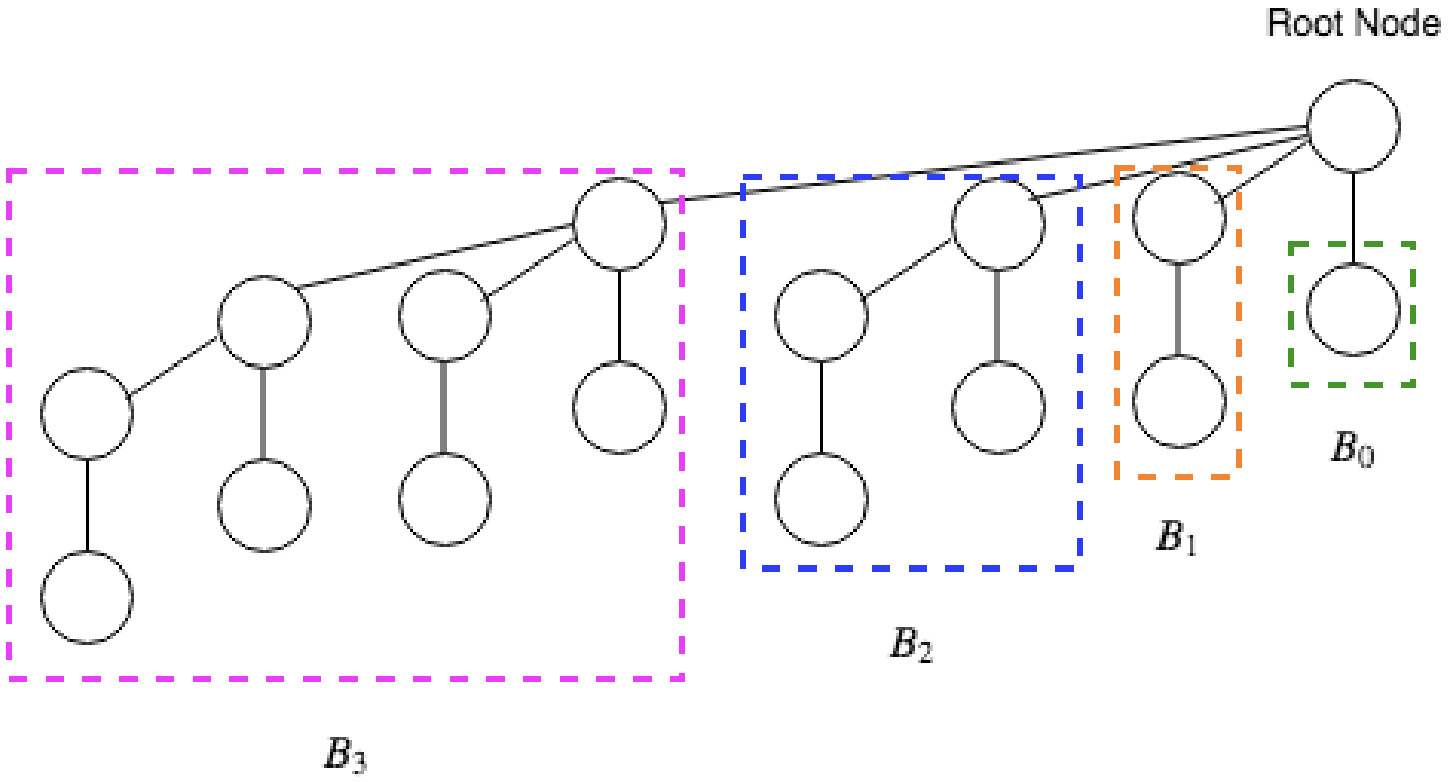
Properties of Binomial Trees

Lemma A For all integers $k \geq 0$, the following properties hold:

1. B_k has 2^k nodes.
2. B_k has height k.
3. For $i = 0, \ldots, k$, B_k has exactly $\binom{k}{i}$ nodes at depth i.
4. The root of B_k has degree k and all other nodes in B_k have degree smaller than k.
5. If $k \geq 1$, then the children of the root of B_k are $B_{k-1}, B_{k-2}, \cdots, B_0$ from left to right.

Binomial Heap

Binomial Tree



A binomial tree of **order 4** whose children are the binomial trees of **order 3, 2, 1, and 0**.

Properties of Binomial Trees

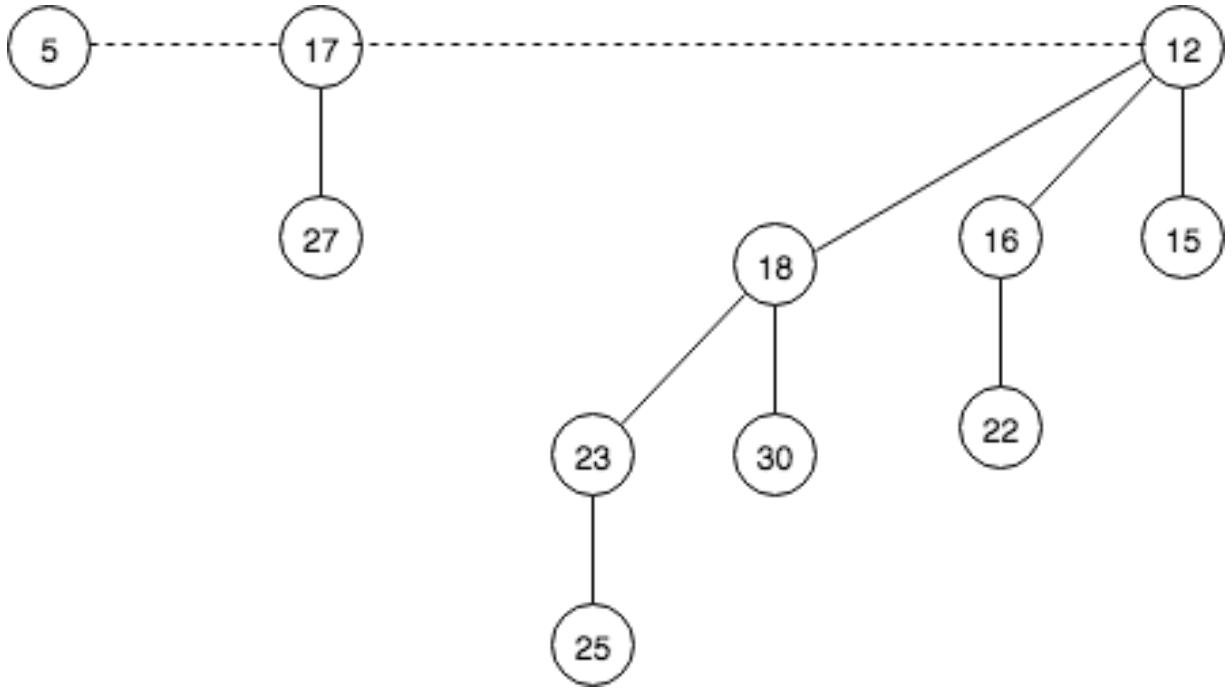
Lemma A For all integers $k \geq 0$, the following properties hold:

1. B_k has 2^k nodes.
2. B_k has height k.
3. For $i = 0, \ldots, k$, B_k has exactly $\binom{k}{i}$ nodes at depth i.
4. The root of B_k has degree k and all other nodes in B_k have degree smaller than k.
5. If $k \geq 1$, then the children of the root of B_k are $B_{k-1}, B_{k-2}, \cdots, B_0$ from left to right.

Binomial Heap

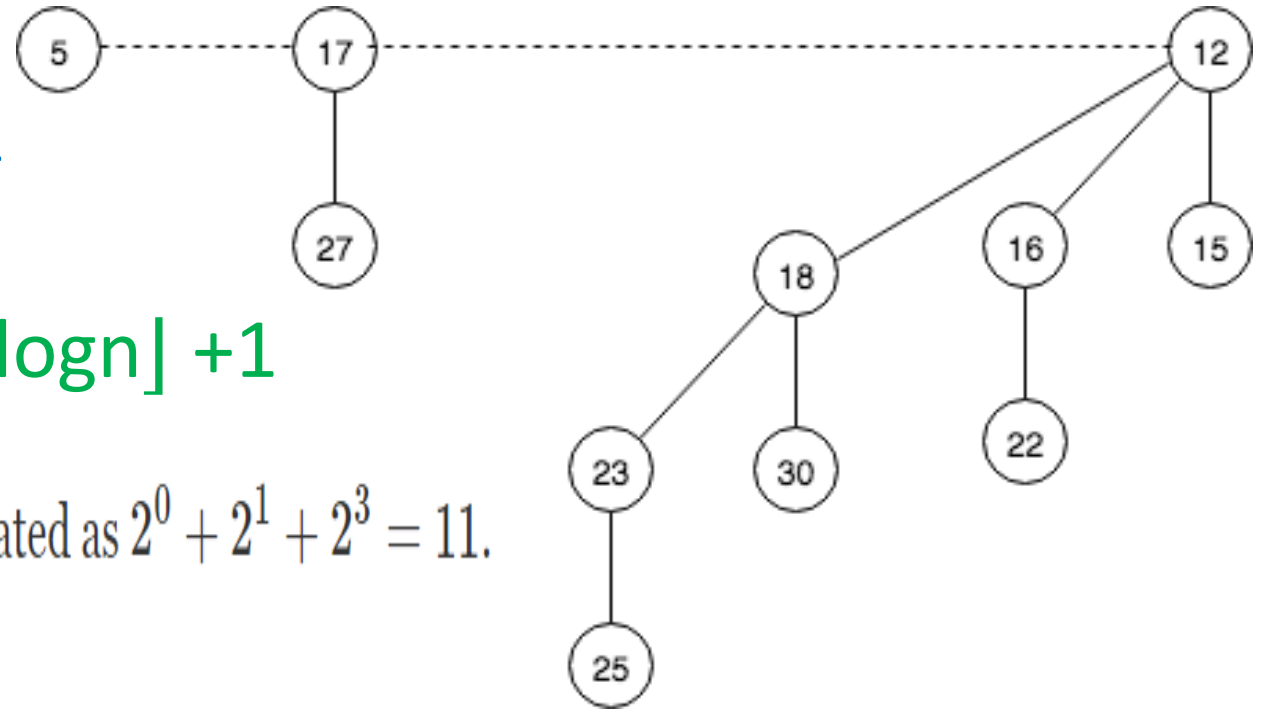
a collection of binomial trees with following binomial heap properties:

- Each binomial tree must be min-heap-ordered (or max-heap-ordered) binomial tree.
  - the root of a min-heap-ordered tree contains the smallest key in the tree
  - if there are m trees, then the smallest key can be found in O(m) time.



Binomial Heap

- For any non-negative integer k, there is at most one binomial tree whose root has degree k.



  - The number of nodes in a binomial tree of degree k is 2^k

an n-node binomial heap consists of at most $\lfloor \log n \rfloor + 1$ binomial trees.

The total number of nodes in the above binomial heap can be calculated as $2^0 + 2^1 + 2^3 = 11$.

binomial trees in a binomial heap of 30 nodes

$$30 = 16 + 8 + 4 + 2 = 2^4 + 2^3 + 2^2 + 2^1$$

the heap contains 4 binomial trees of degree 4, 3, 2, 1.

- if there are m trees, then the smallest key can be found in O(m) time.

Binomial Heap

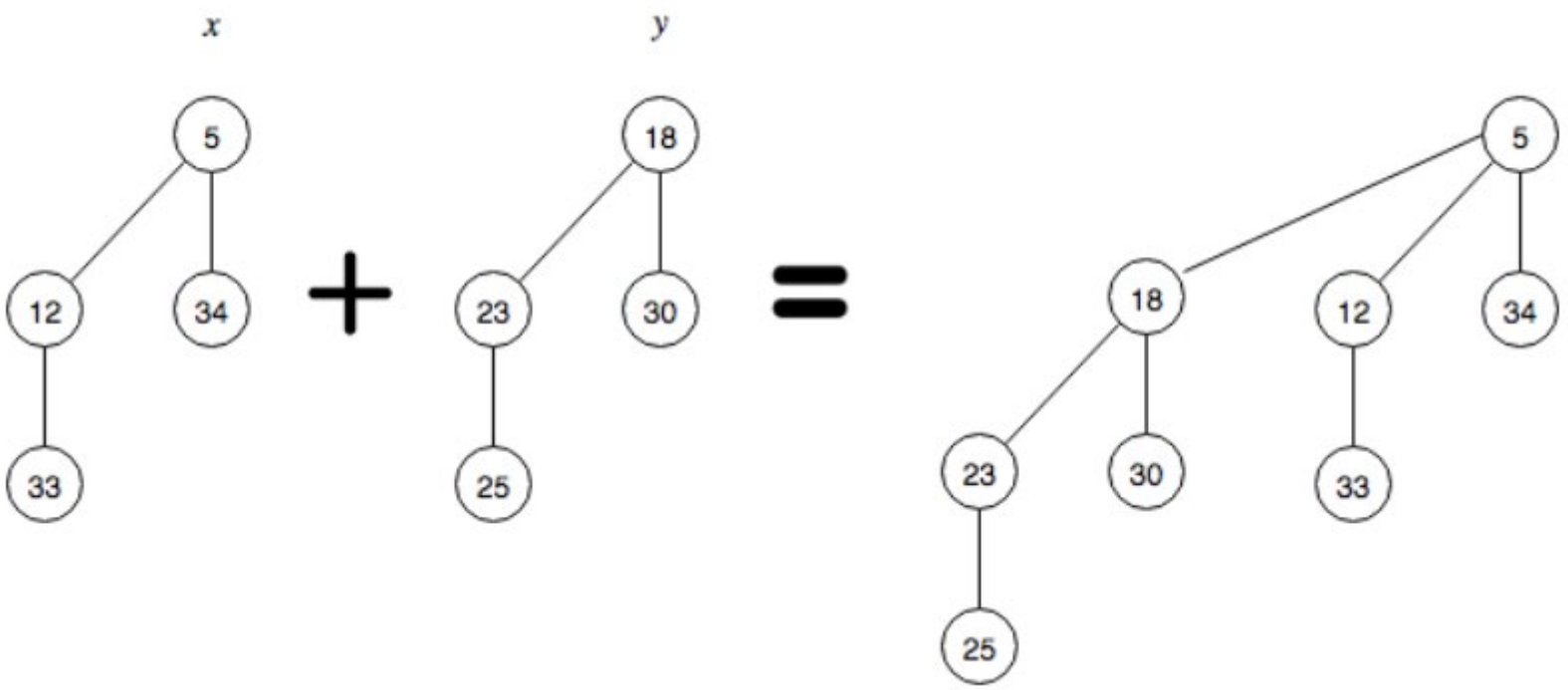
- An

Operations	Binary Heap	Binomial Heap	Fibonacci Heap
Procedure	**Worst-case**	**Worst-case**	**Amortized**
Making Heap	Θ(1)	Θ(1)	Θ(1)
Inserting a node	Θ(log(n))	O(log(n))	Θ(1)
Finding Minimum key	Θ(1)	O(log(n))	O(1)
Extract-Minimum key	Θ(log(n))	Θ(log(n))	O(log(n))
Union or merging	Θ(n)	O(log(n))	Θ(1)
Decreasing a Key	Θ(log(n))	Θ(log(n))	Θ(1)
Deleting a node	Θ(log(n))	Θ(log(n))	O(log(n))

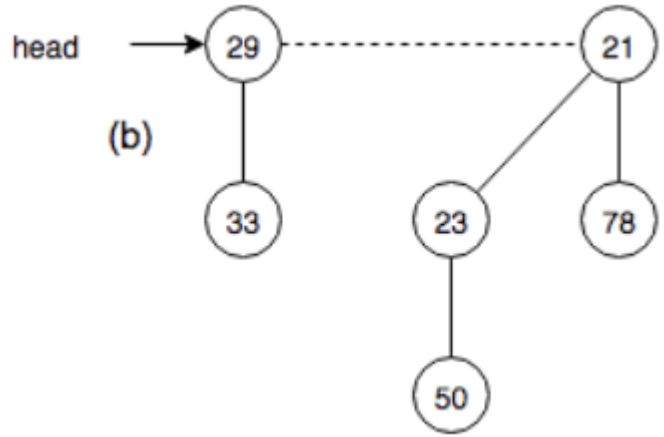
Binomial Heap – Union/Merge

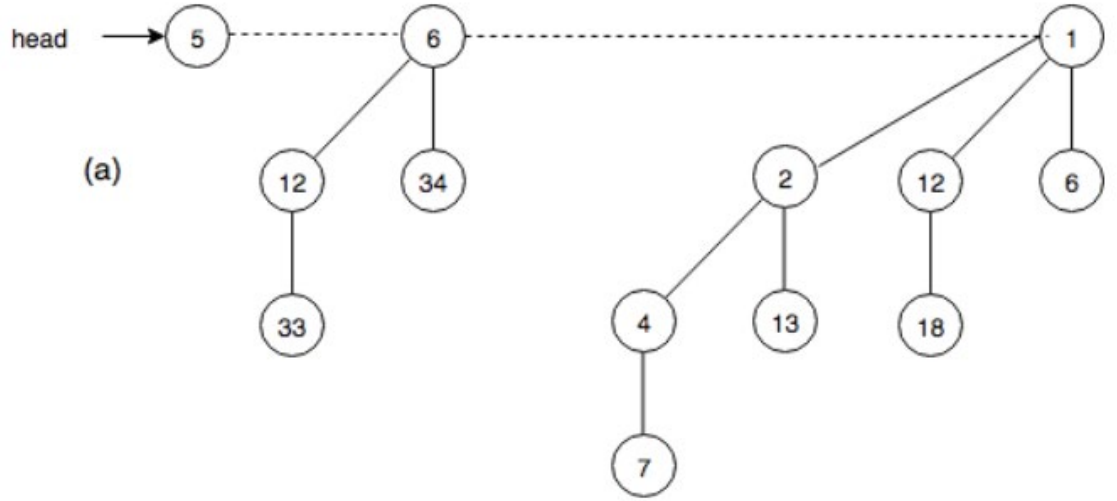
We repeatedly merge two binomial trees of the same degree until all the binomial trees have a unique degree. To merge two binomial trees of the same degree, we do the following.

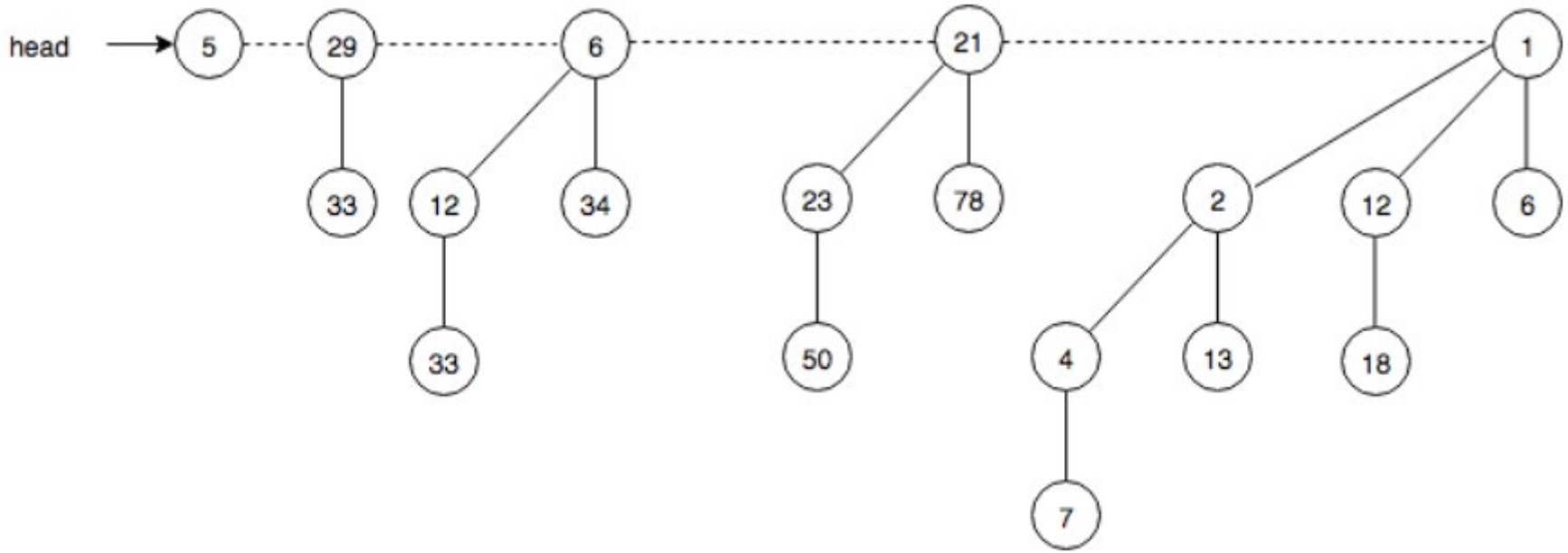
1. Compare the roots of two trees (x and y). Find the smallest root. Let x is the tree with the smallest root.
2. Make the x's root parent of y's root.



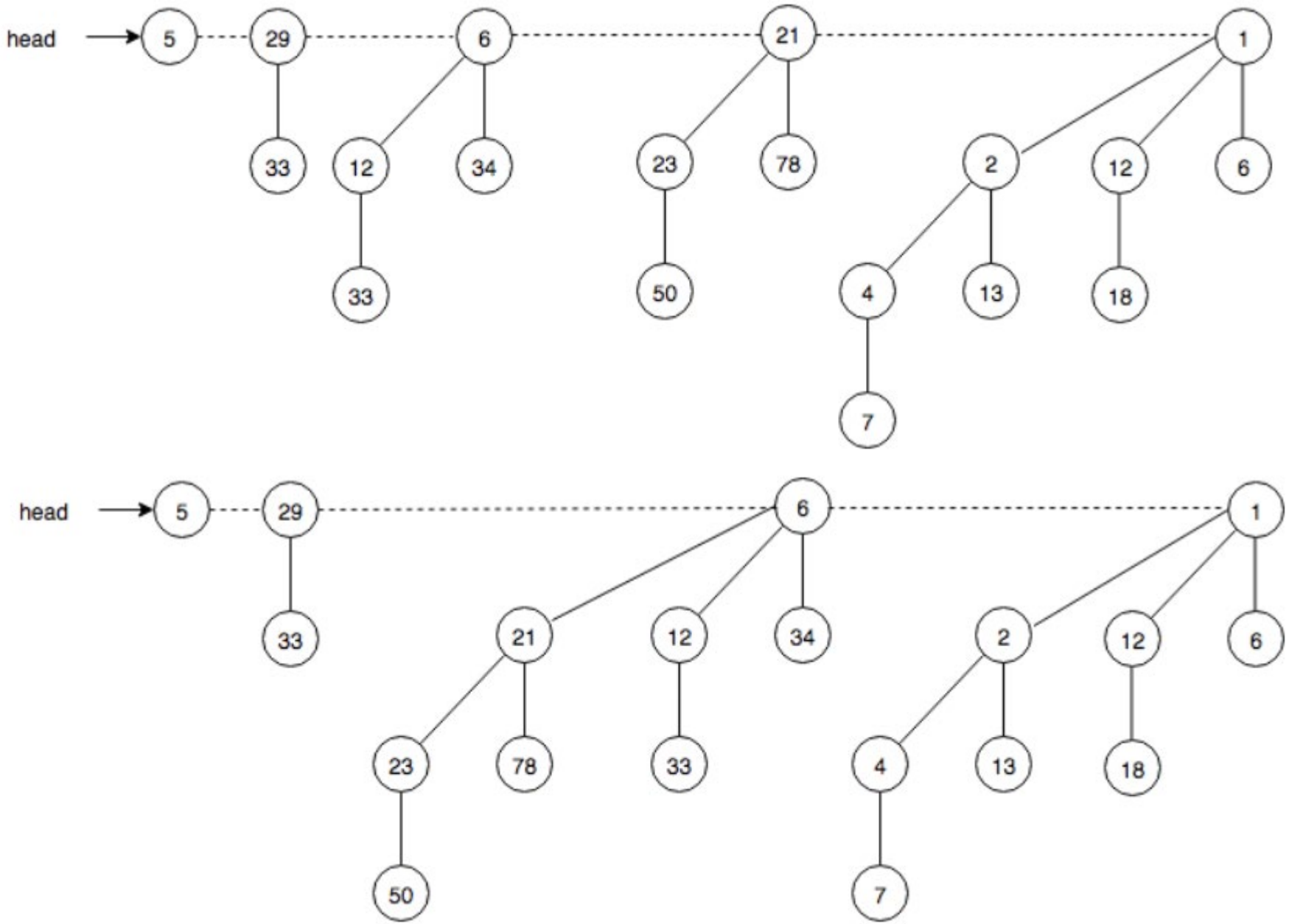
Binomial Heap – Union/Merge



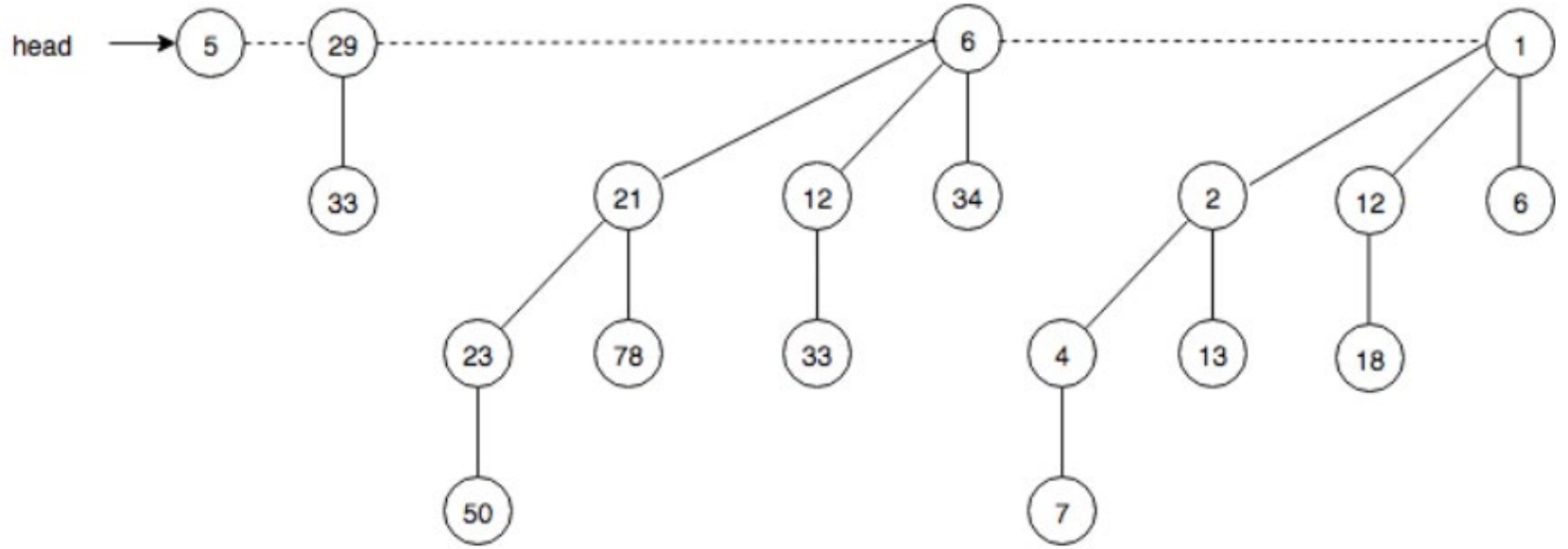


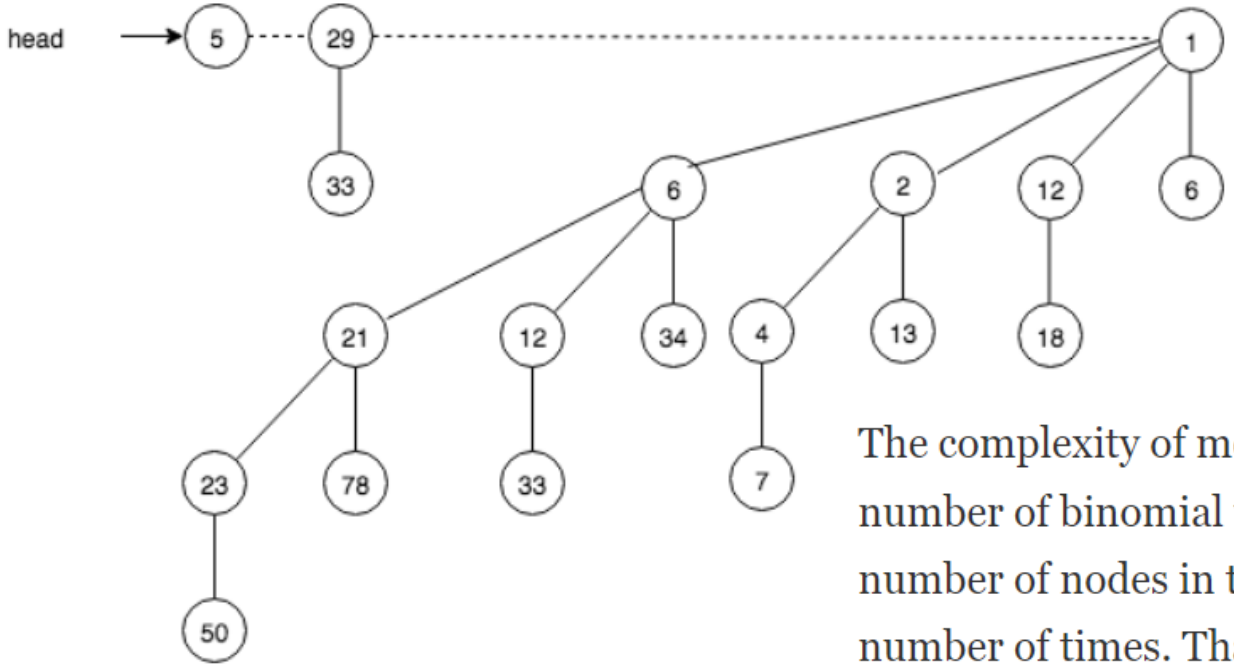


Binomial Heap – Union/Merge



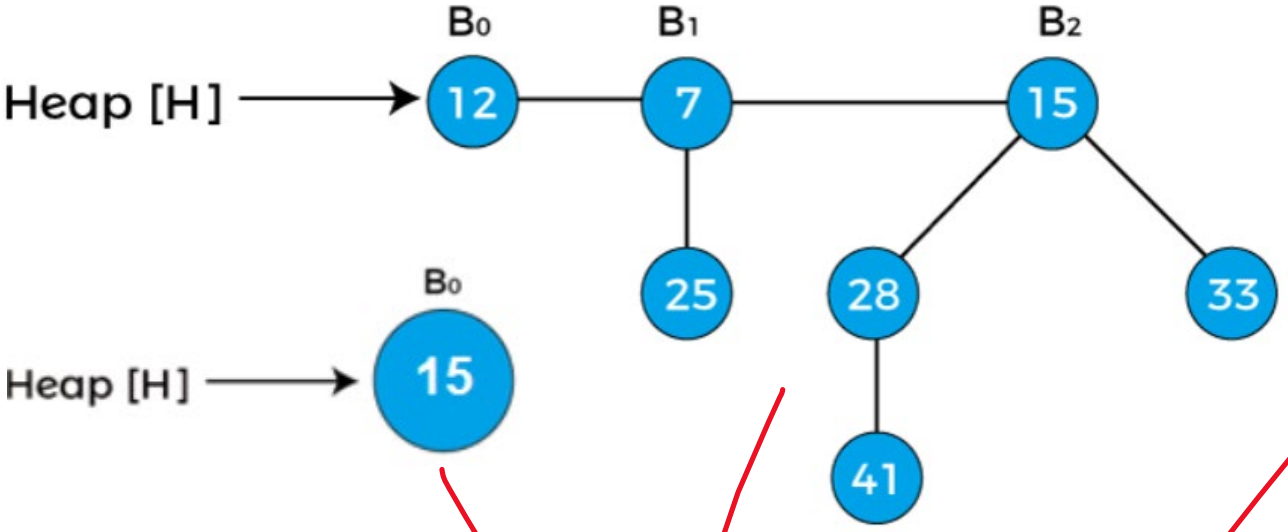
Binomial Heap – Union/Merge

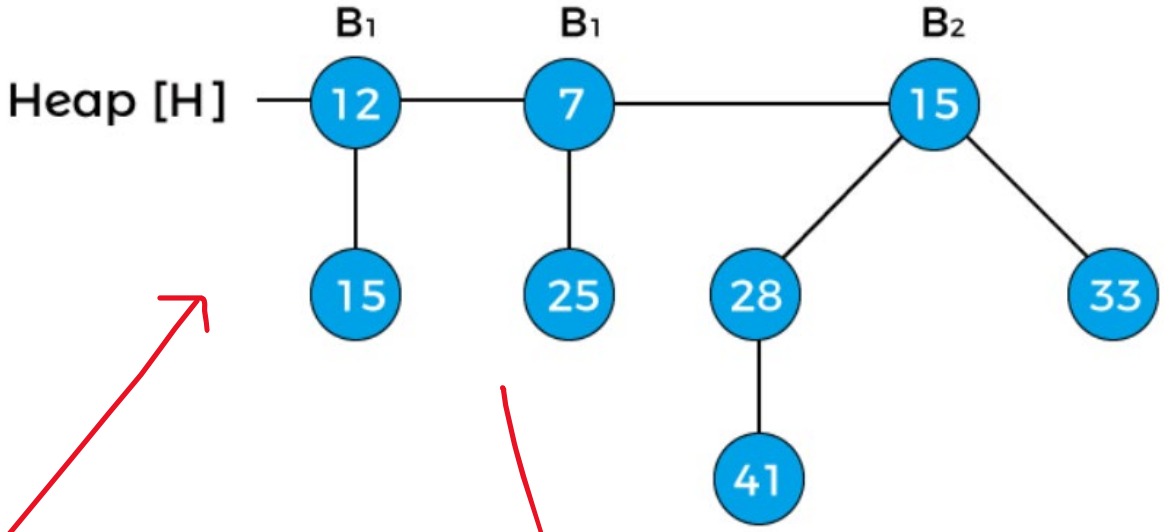


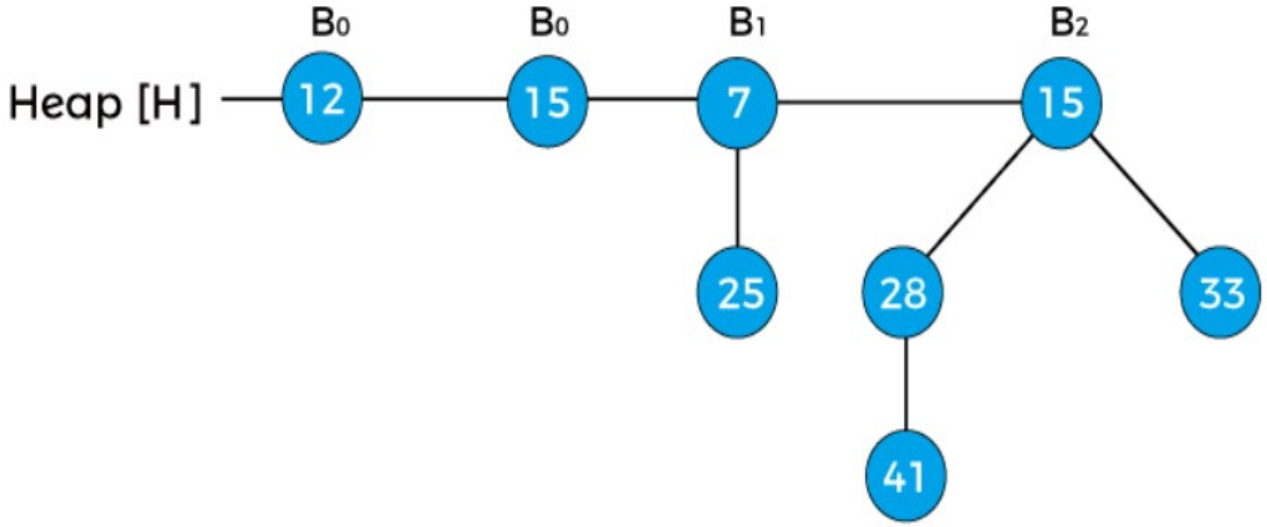


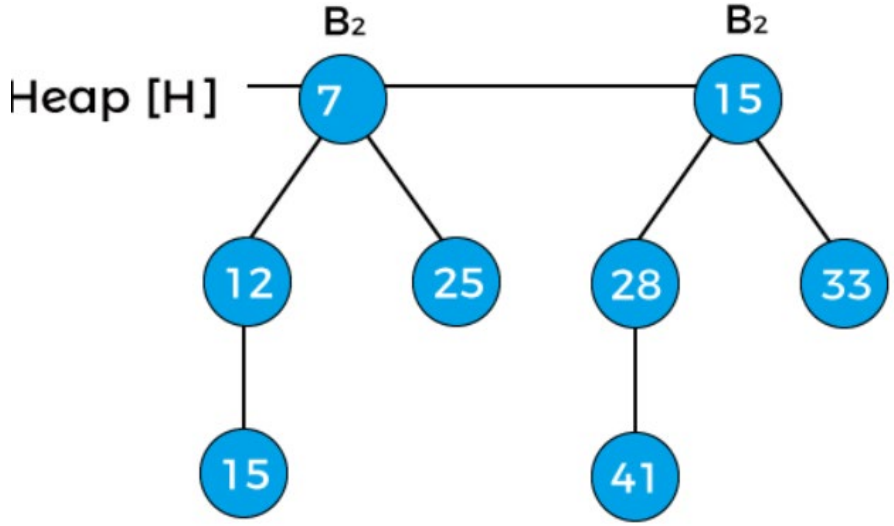
The complexity of merge operation is $O(\log n)$. There are at most $\lfloor \log n_1 + \log n_2 \rfloor + 2$ number of binomial trees (where n_1 is the number of nodes in the first heap and n_2 is the number of nodes in the second tree). We traverse the roots of these trees constant number of times. That gives the complexity of $O(\log n)$.

Binomial Heap - Insert

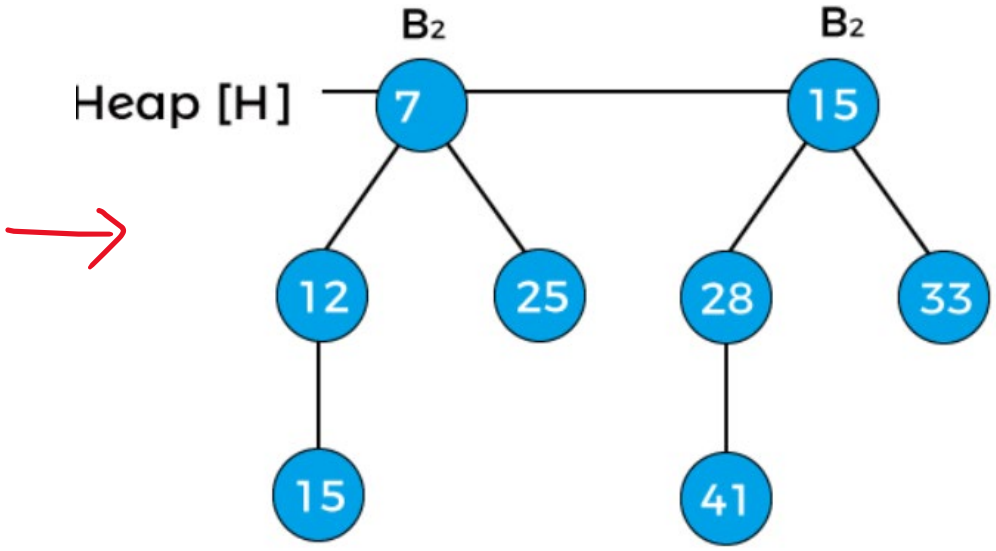


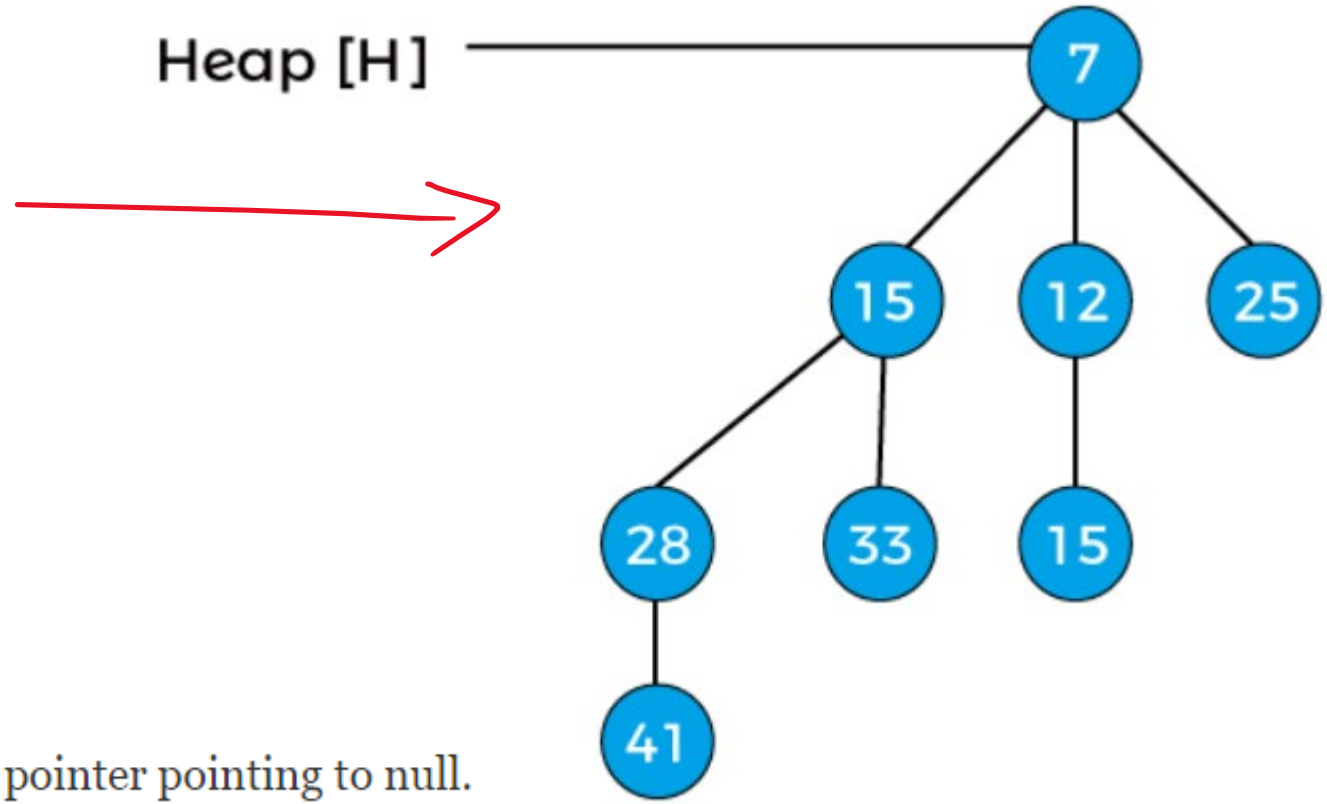






Binomial Heap - Insert



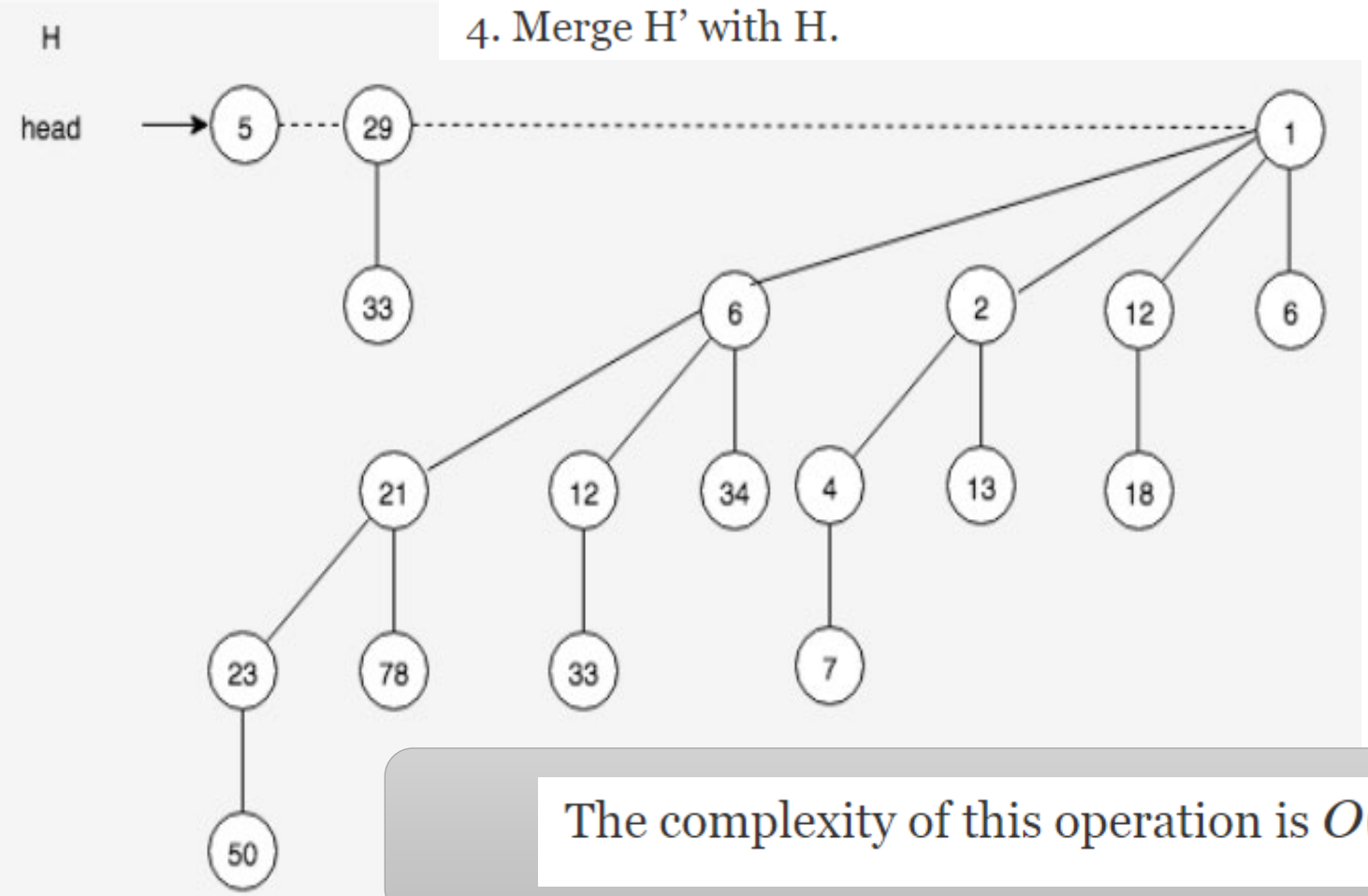


1. Create a new empty binomial heap H'. It has a head pointer pointing to null.
2. Create a new node x with all the necessary fields. Point the head of the heap to x.
3. Merge this new heap H' with the existing heap H.

The complexity of inserting a new node is $O(\log n)$. Creating new heap takes only a constant amount of time and merging two heaps takes $O(\log n)$.

Binomial Heap – Extracting the minimum key (or Delete min)

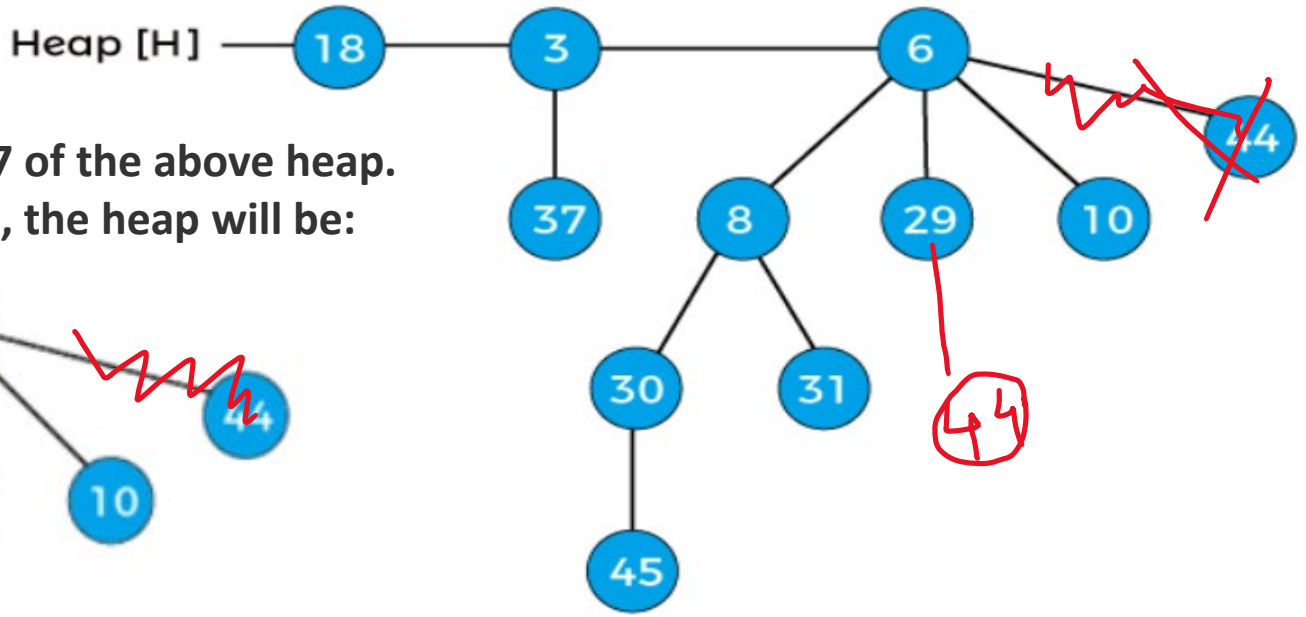
1. Search through the roots of binomial trees and find the root with the smallest key and call it x. Remove the x from the tree.
2. Create a new empty heap H'.
3. Reverse the order of x's children and set the head of H' to point to the head of the resulting list.
4. Merge H' with H.



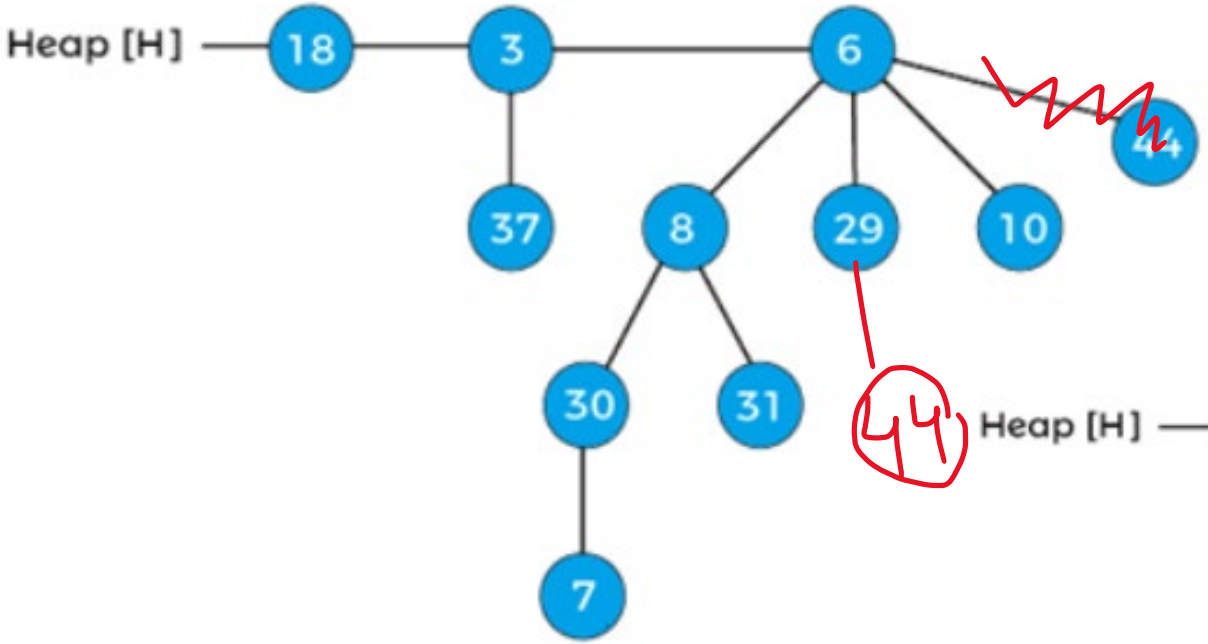
In min-heap, the root element contains the minimum key value. So, we have to compare **the key value of the root node of all the binomial trees.**

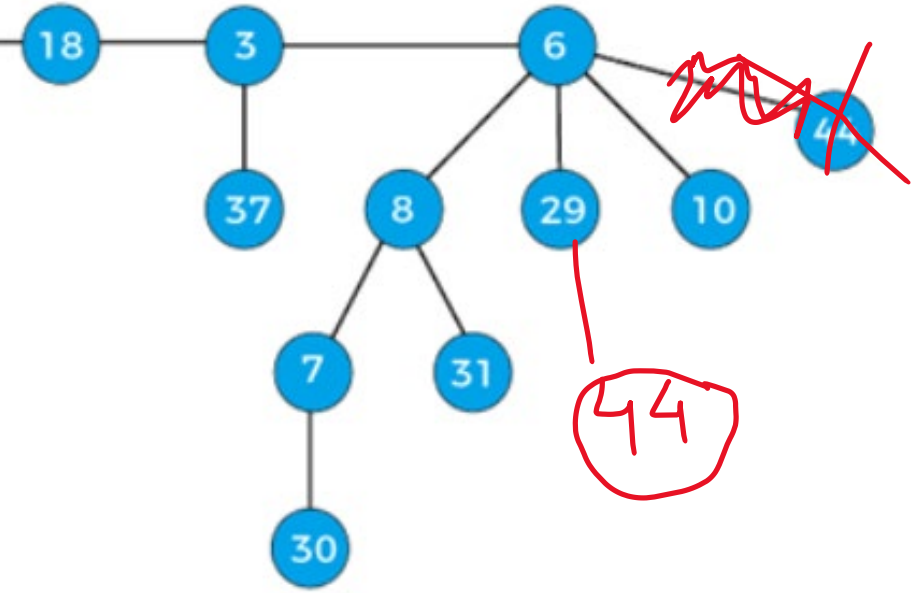
The complexity of this operation is $O(\log n)$.

Binomial Heap – Decreasing the key

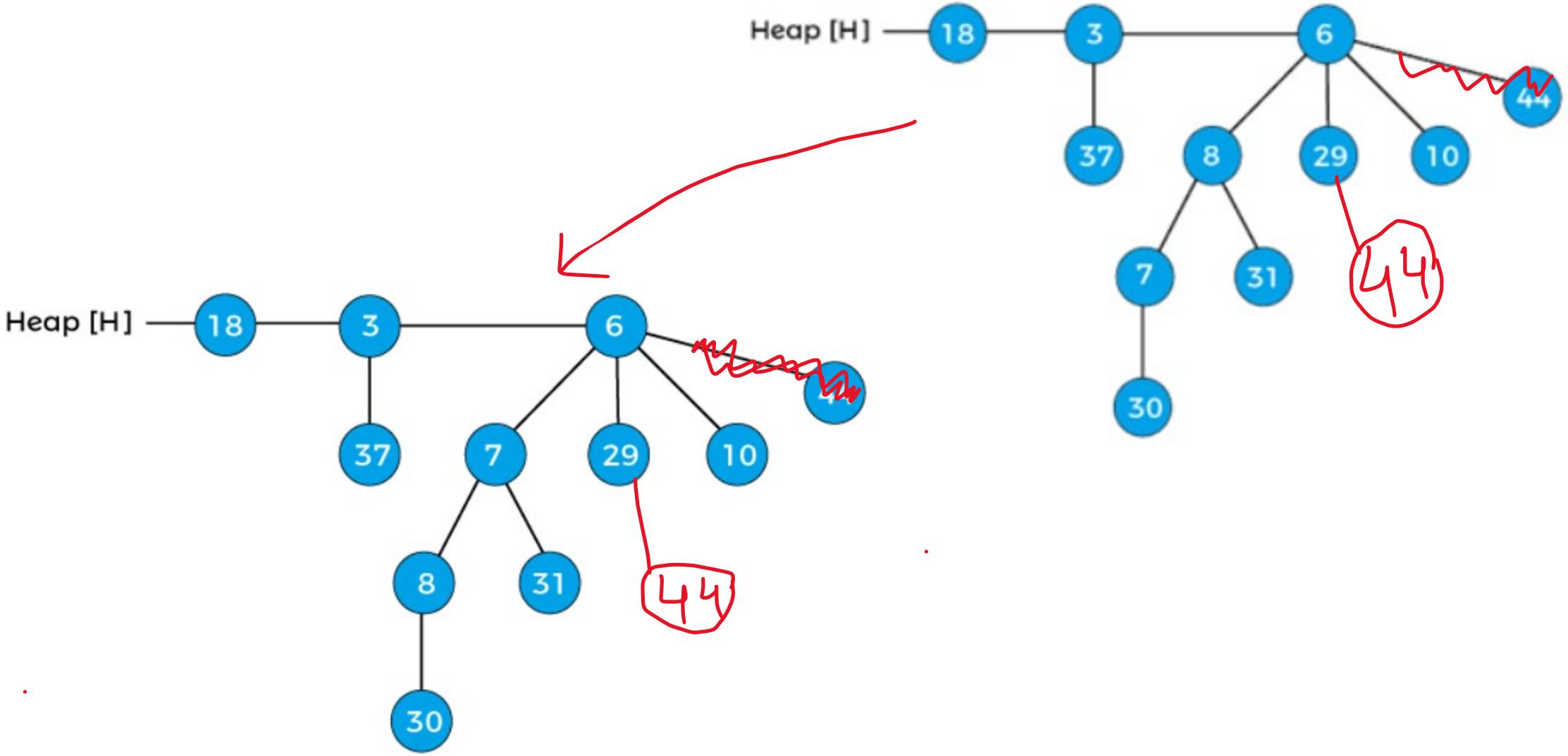


Decrease the key 45 by 7 of the above heap.
After decreasing 45 by 7, the heap will be:

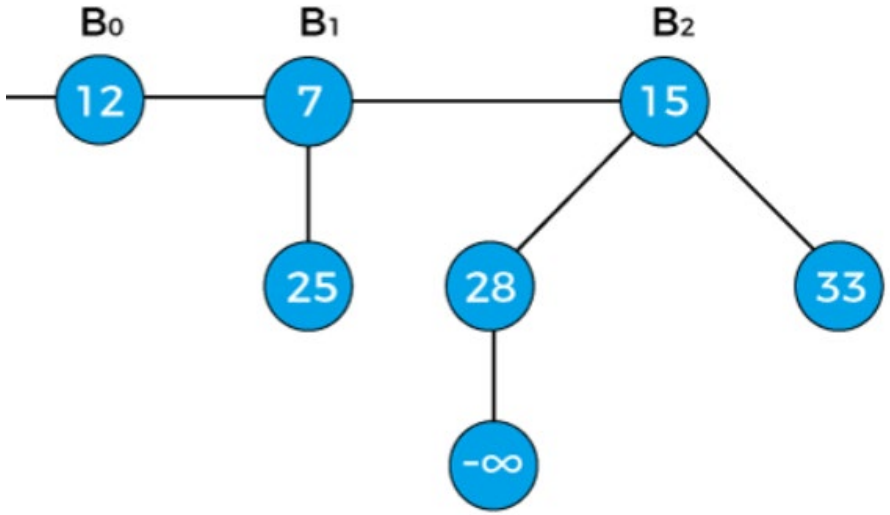


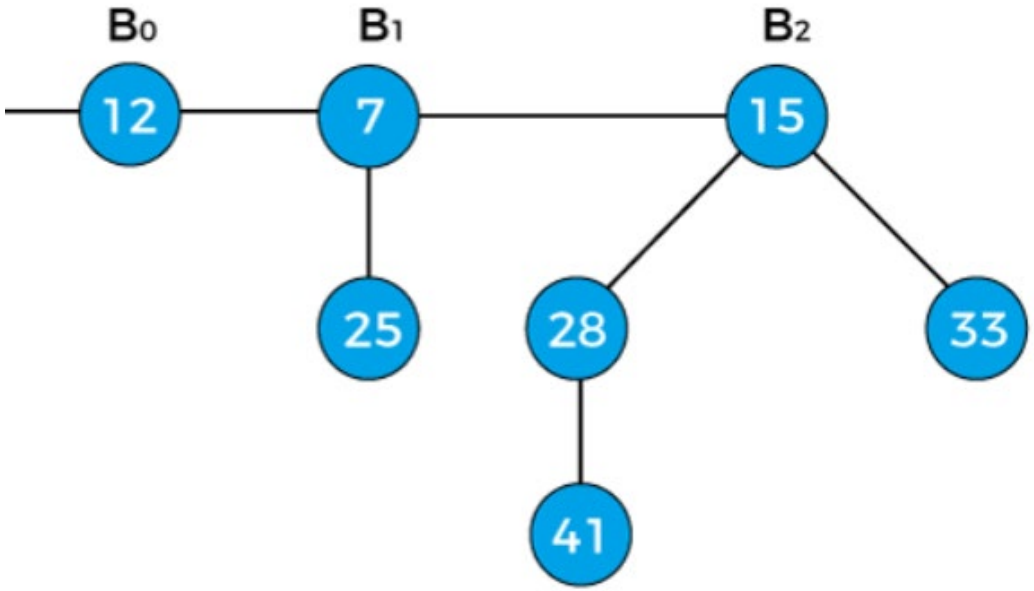


Binomial Heap – Decreasing the key

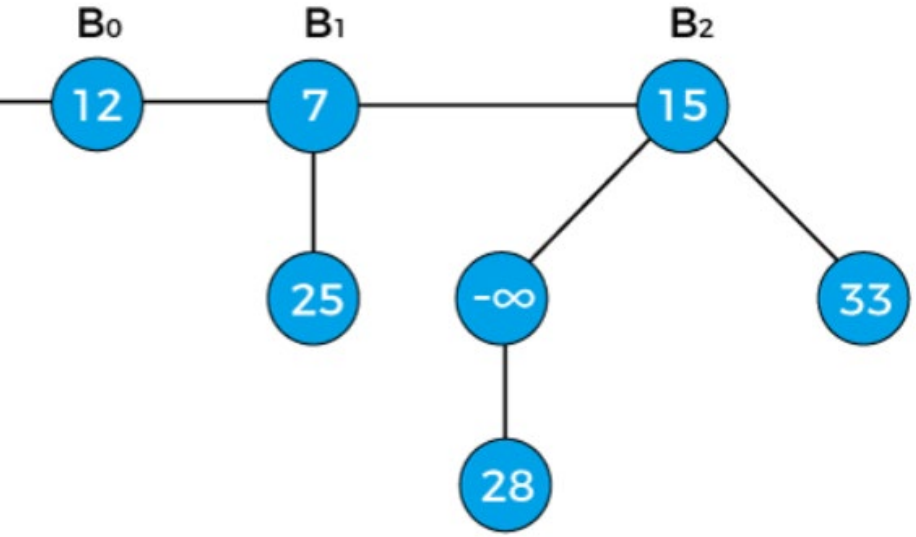


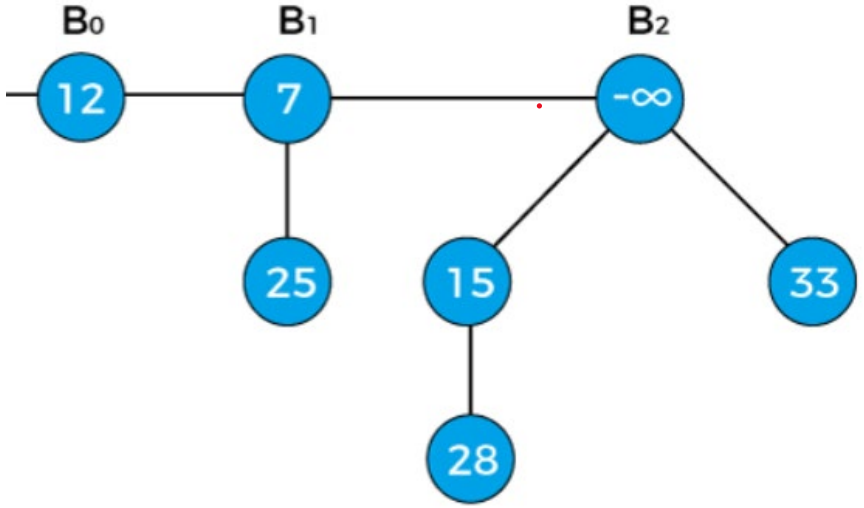
Binomial Heap – Deleting a node from the heap



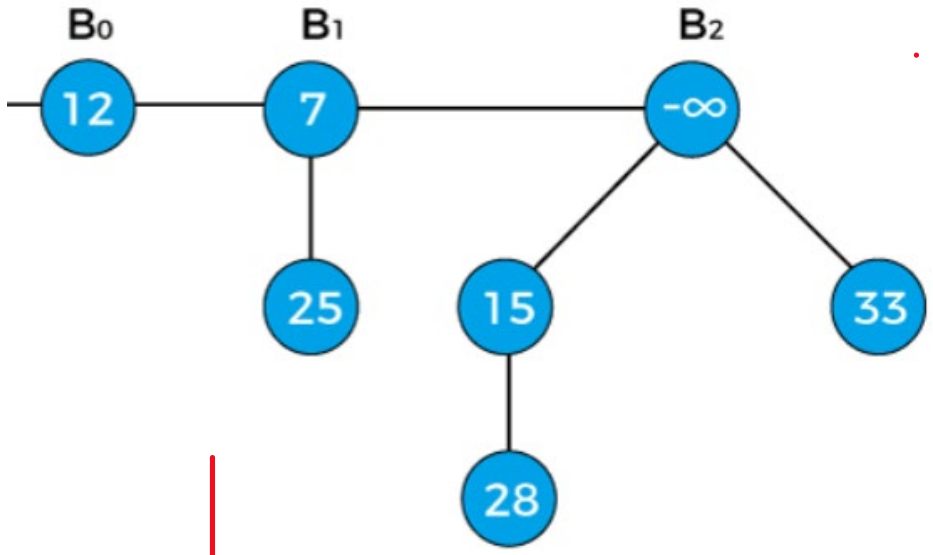


delete the node 41 from the heap

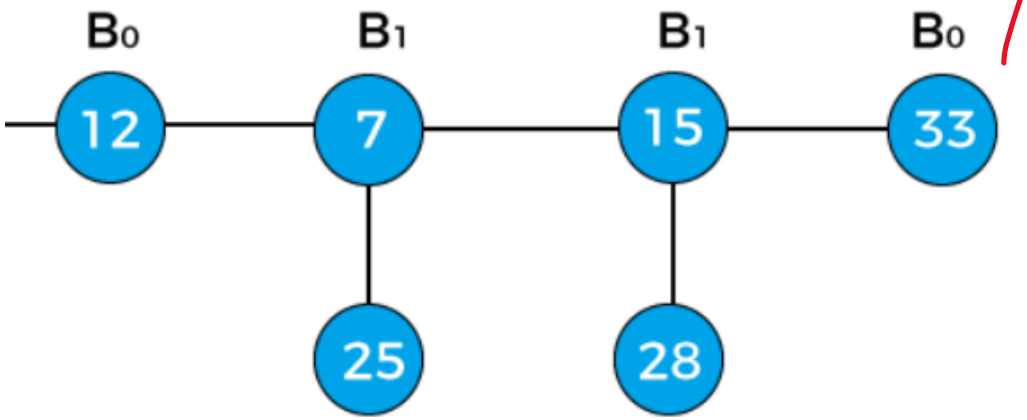


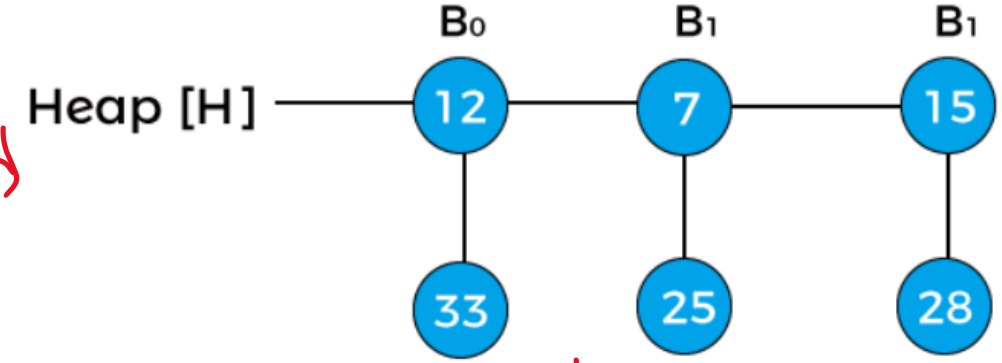


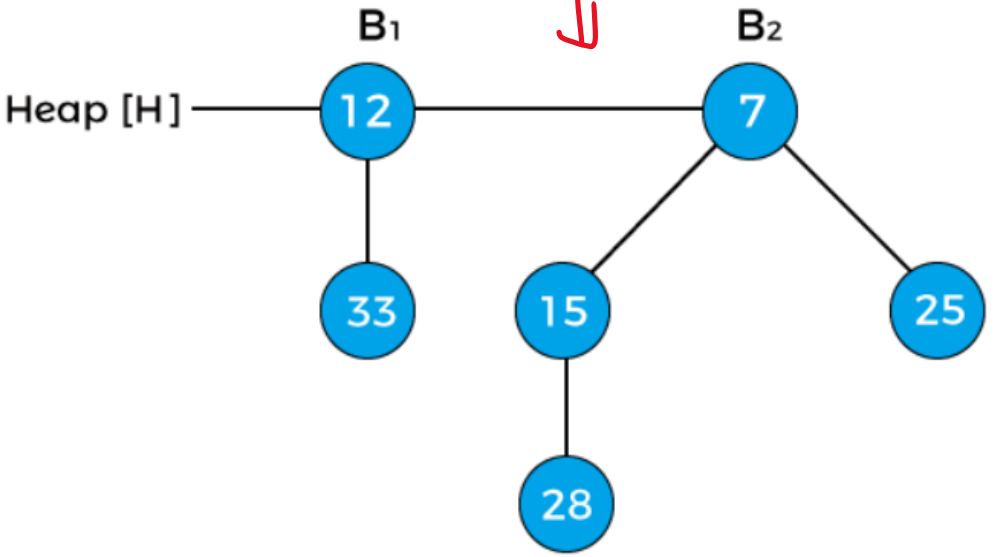
Binomial Heap – Deleting a node from the heap

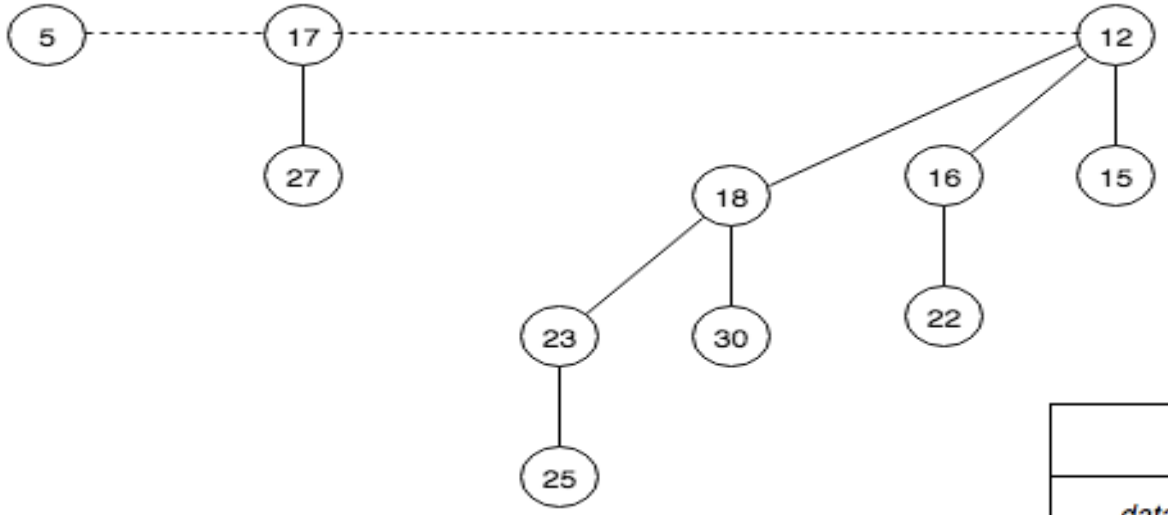


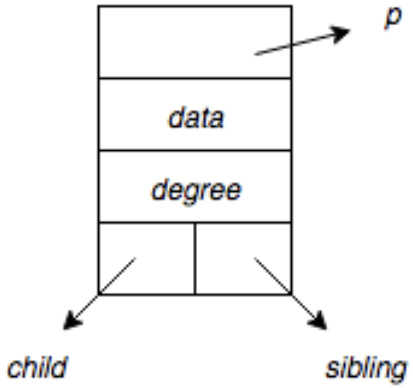
The next step is to extract the minimum key from the heap. Since the minimum key in the above heap is -infinity so we will extract this key, and the heap would be:

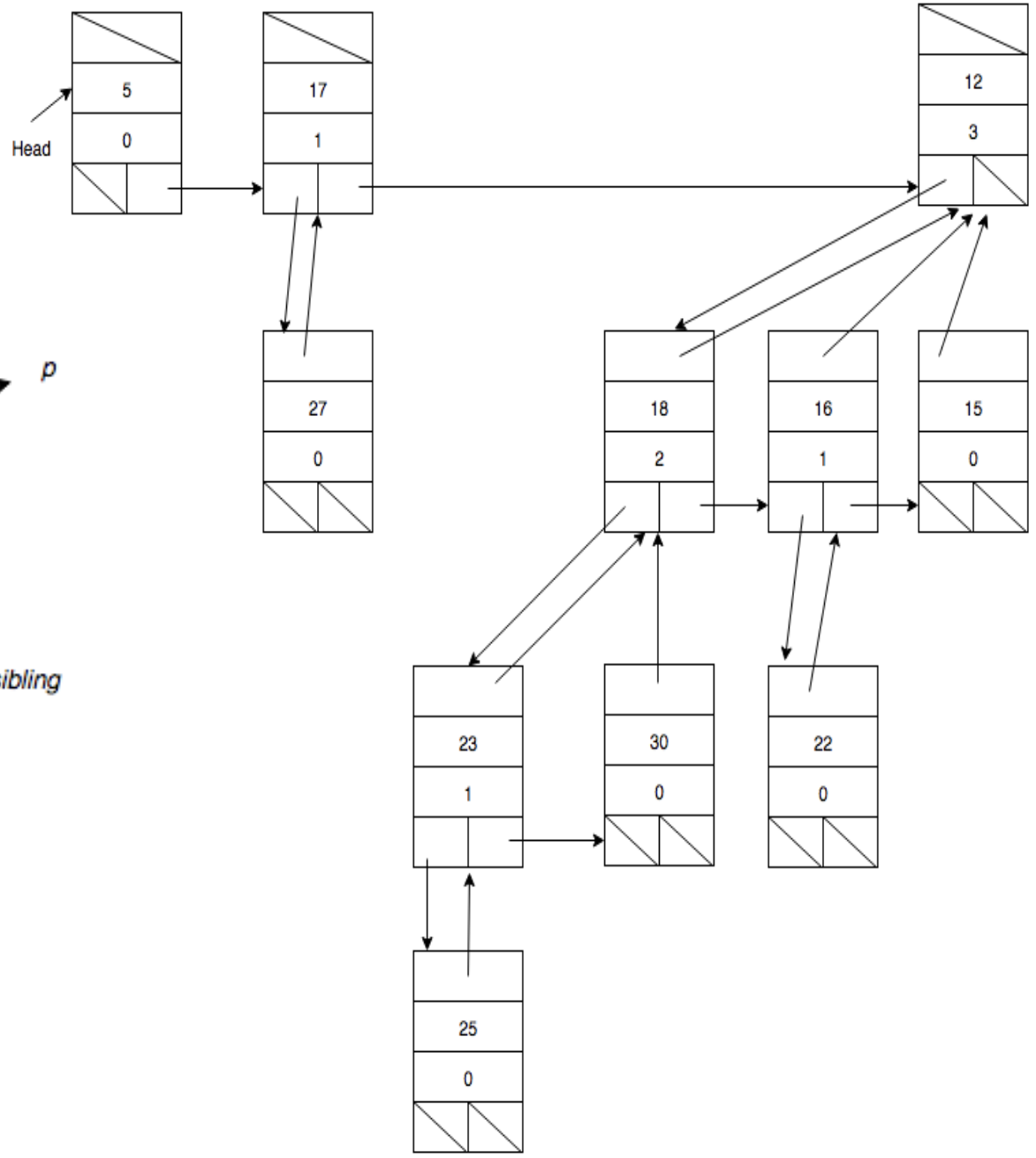












- p: A pointer to the parent node.
- data: The key of the node.
- degree: Number of children.
- child: The pointer to the left-most child of the node.
- sibling: The pointer to the right sibling of the node. In case of a root node, the sibling points to the root of another tree in the right.

Comparison

Operations	Binary Heap	Binomial Heap	Fibonacci Heap
Procedure	**Worst-case**	**Worst-case**	**Amortized**
Making Heap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Inserting a node	$\Theta(\log(n))$	$O(\log(n))$	$\Theta(1)$
Finding Minimum key	$\Theta(1)$	$O(\log(n))$	$O(1)$
Extract-Minimum key	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$
Union or merging	$\Theta(n)$	$O(\log(n))$	$\Theta(1)$
Decreasing a Key	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(1)$
Deleting a node	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$