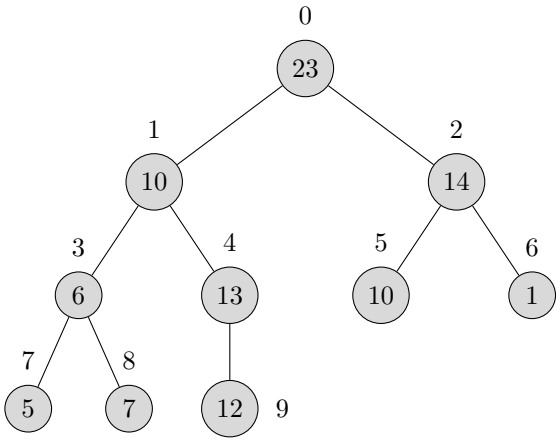
1. **Question 1.1.** (1pt) Is the sequence {23, 10, 14, 6, 13, 10, 1, 5, 7, 12} a max-heap? Can you call MAX-HEAPIFY (A, i) to make it a max-heap for a single i? Why? How to make the sequence to become a heap if your answer to the previous question is no?
Answer : From the given sequence we can draw the heap structure, which is nearly complete binary tree, from left to right and up to down with ordered index

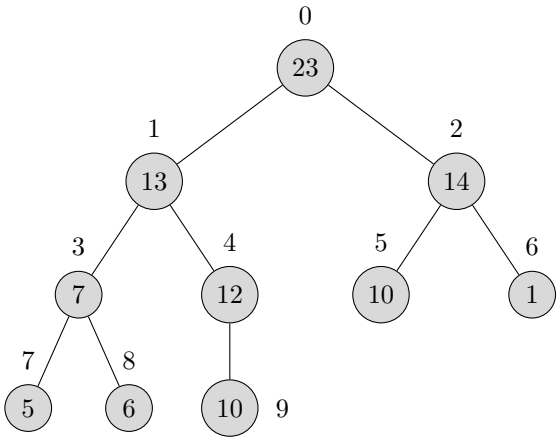


(a) **it's not a max-heap**. since those node (i.e 10, and 6) the right child node is greater than parent node.

(b) We can't only call a single MAX-HEAPIFY(A,i) to create max-heap, to make it become max heap, we need to call MAX-HEAPIFY(A,1) and MAX-HEAPIFY(A,3).

- The MAX-HEAPIFY(A,1) will swap $A[1] = 10$ and $A[4] = 13$, then call again MAX-HEAPIFY(A,4) to swap $A[4] = 10$ and $A[9] = 12$.
- The MAX-HEAPIFY(A,3) will swap A[3] = 6 and A[8]=7

the max-heap is like this :



2. **Question 1.2.** (1pt) What is the effect of calling MAX-HEAPIFY (A, i) for $i > heapsize[A]/2$?

   Answer : the heapsize[A]/2 means the non-leaf on heap structures, then if we call MAX-HEAPIFY(A,i) where $i > heapsize[A]/2$ it may be violating heap-structures where leaf doesn't have child, it's useless to do and might be lead some errors if our code not care about this.

3. **Question 1.3.** (1pt) The operation HEAP-DELETE(A, i) deletes the item in node i from heap A. Give an implementation of HEAP-DELETE that runs in O(log n) time for an nelement max-heap.

   Answer : the code are on python programming languages are given below

   the best case on HEAP-Delete function is $O(1)$ it occurs when we remove directly the last element on an max-heap, and the worst case on HEAP-Delete is $O(log\, n)$ when after we swap the desired deletion element then we do max-heapify where it's have $O(log\, n)$ complexity

```python
def max_heapify(arr, i):
    heapsize = len(arr)
    left = 2 * i + 1
    right = 2 * i + 2
    largest = i

    if left < heapsize and arr[left] > arr[largest]:
        largest = left

    if right < heapsize and arr[right] > arr[largest]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        max_heapify(arr, largest, heapsize)

def heap_delete(arr, i):
    n = len(arr)
    if i < 0 or i >= n:
        print("Error: Index out of bounds")
        return

    # Best case O(1) when i == n
    if i != n :
        # Worst case O(log n)
        # Replace the item at node i with the last leaf
        arr[i] = arr[n - 1]
        # Bubble down or bubble up to maintain the max-heap property
        max_heapify(arr, i)

    # Remove the last leaf
    arr.pop()

# Example Usage:
max_heap = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
print("Original Max-Heap:", max_heap)

# Delete the item at index 4 (value 7)
heap_delete(max_heap, 4)
print("Max-Heap after deletion:", max_heap)
```

the output is given by :

```
Original Max-Heap: [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
Max-Heap after deletion: [16, 14, 10, 8, 1, 9, 3, 2, 4]
```

4. **Question 1.4.** (1pt) In our lectures, we conclude that the meaning of HEAPDECREASE-KEY(A, i, key) is not clear. However, if we have to create such function anyway, how?
   Answer : the step is quite simple

   (a) first change the desired node.

   (b) max-hapify node, this done by checking whether the parents on changed nodes are less or not, if less swap it.

   The code are given below

```python
# i = what nodes we want to decrease the value
# key = the desired value
def max_heapify(arr, i):
    heapsize = len(arr)
    left = 2 * i + 1
    right = 2 * i + 2
    largest = i

    if left < heapsize and arr[left] > arr[largest]:
        largest = left

    if right < heapsize and arr[right] > arr[largest]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        max_heapify(arr, largest)

def heap_decrease_key(arr, i, key):
    if key > arr[i]:
        print("Error: New key is greater than the current key.")
        return

    arr[i] = key
    # maintain the property of max-heap since it might be not a max-heap, so why we
        do.
    max_heapify(arr,i)
# Example Usage:
max_heap = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
print("Original Max-Heap:", max_heap)

# Decrease the key at index 3 (value 8) to 5
heap_decrease_key(max_heap, 3, 0)
print("Max-Heap after decreasing key:", max_heap)
```

   the output is given by :

```
Original Max-Heap: [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]
Max-Heap after decreasing key: [16, 8, 10, 4, 7, 9, 3, 2, 0, 1]
```

5. **Question 1.5.** (3pts) Given three sorting algorithms: (i) insertion sort, (ii) selection sort, (iii) bubble sort, can we use any of them to find the largest three elements from a sequence? How? What is the complexity? You should provide (the portion of the codes if you want to make your statement clear).

   (a) **Insertion Sort**, I think the best to find the 3 largest value on array using Insertion sort, is just do usual insertion sort then, extract the 3 largest number, the code are given below and the complexity is still $O(n^2)$

```python
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
# Example Usage:
arr = [12, 11, 13, 5, 6,20,15,17,2,2,2,3]
insertion_sort(arr)
print("Sorted three largest array:", arr[-3:])
```

the output are given by

```
Sorted three largest array: [15, 17, 20]
```

(b) **Selection sort**, usually we do increasing order right ? but on this case for more efficient rather to be descending, the code are below and the complexity are given like this :

$$T(n) = (n) + (n-1) + (n-2) = 3n - 3 = 3(n-3) = O(n)$$

```python
def find_x_largest_selection_sort(arr, val):
n = len(arr)
if val > n:
    print(f"the second parameter should be lower than {n}")
    return
for i in range(val):
    max_idx = i
    for j in range(i + 1, len(arr)):
        if arr[max_idx] < arr[j]:
            max_idx = j
    arr[i], arr[max_idx] = arr[max_idx], arr[i]
return arr[0:val].copy()
# Example Usage:
arr = [64, 25, 12, 22, 11,100,3,5,8,9]
val = 3
arr = find_x_largest_selection_sort(arr,val)
print(f"Sorted array Descending {val} lartgest value :", arr)
```

the output :

```
Sorted array Descending 3 lartgest value : [100, 64, 25]
```

(c) Bubble Sort, for bubble sort we can sort like usual but we only bubble 3 times to the end of array. the code are given below, then the complexity same with selection sort

$$T(n) = (n-1) + (n-2) + (n-3) = 3n - 6 = O(n)$$

```python
def find_x_largest_bubble_sort(arr,val):
arr = arr.copy()
n = len(arr)
if val > n:
    print(f"the second parameter should be lower than {n}")
    return
for i in range(val):
    for j in range(0, n - i - 1):
        if arr[j] > arr[j + 1]:
            arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

```
return arr[-3:]
# Example Usage:
arr = [64, 34, 25, 12, 22, 11, 90]
three_largest = find_x_largest_bubble_sort(arr,3)
print("Sorted 3 largest array:", three_largest)
```

the output is given by :

```
Sorted 3 largest array: [34, 64, 90]
```

6. **Question 1.6** (3pts) Name the algorithms that can be implemented in a stable way for the given candidates: (i) insertion sort, (ii) merge sort, (iii) selection sort, (iv) bubble sort, (v) heapsort, (vi) quicksort. In your answer (Yes or No) for each of the algorithms, you should provide one or two lines of codes that is the key to make sure the stable property can (or cannot) be ensured.

Answer :

(a) **Insertion Sort**

- Stability ? : yes it's stable, Insertion sort is a stable sort. During the selection sort process, we will only swap the ordering of any two items if the item on the right is less than the item to its left. Therefore, the ordering of two equivalent items will always be preserved in insertion sort.
- Key

```
while j >= 0 and arr[j] > key:
```

(b) **Merge Sort**

- Stability ? : yes it's stable. During the merging process, we preserve the ordering of equivalent objects. If the first element of the left list being merged is equivalent to the first element of the right list, we always insert the element from the left list before the right list. Through this process we always preserve the inital relative ordering of equivalent items in the list, resulting in a stable sorting algorithm.
- Key

```
if left[i] <= right[j]:
    arr[k] = left[i]
```

(c) **Selection Sort**

- Stability ? : No, The swapping process in selection sort is not guaranteed to preserve the original ordering of equivalent items. Try manual of [1,5,2,7,4,9,5,0]
- Key

```
if arr[min_idx] > arr[j]:
    min_idx = j
    arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

(d) **Bubble Sort**

- Stability ? : Yes, it's stable because it only swaps adjacent element when they are stricly differ
- Key

```
if arr[j] > arr[j + 1]:
```

(e) **HeapSort**

- Stability ? : No. Through the insertion and removal process, with all the bubbling up and bubbling down there is no guarantee of stability.

(f) **Quick Sort**

- Stability ? : No, Quicksort is not stable by default as it involves partitioning and swapping elements without considering their initial order.