<u>Priority queues</u>

- Heapsort is an excellent algorithm, but a good implementation of quicksort usually beats it in practice. Nevertheless, the heap data structure itself has enormous utility.
- Here, we will discuss one of the most popular applications of a heap: its use as an efficient priority queue.
- As with heaps, there are two kinds of priority queues: max-priority queues and min-priority queues. We will focus on how to implement max-priority queues, which are in turn based on max-heaps (the procedures for min-priority queues can be written in analogous manner.)
- A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key. A max-priority queue supports the following operations:
    - MAXIMUM(S) returns the element of S with the largest key.
    - EXTRACT-MAX(S) removes and returns the element of S with the largest key.
    - INCREASE-KEY(S, x, k) increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.
    - INSERT(S, x) inserts the element x into the set S. This operation could be written as $S \leftarrow S \cup \{x\}$.

- One application of max-priority queues is to schedule jobs on a shared computer. The max-priority queue keeps track of the jobs to be performed and their relative priorities. When a job is finished or interrupted, the highest-priority job is selected from those pending using EXTRACT-MAX. A new job can be added to the queue at any time using INSERT.

- we can use a heap to implement a priority queue.
    - In a given application elements of a priority queue correspond to objects in the application (such as jobs in scheduling)
    - When a heap is used to implement a priority queue, therefore, we often need to store a handle to the corresponding application object in each heap element. The exact makeup of the handle (i.e., a pointer, an integer, etc.) depends on the application. Similarly, we need to store a handle to the corresponding heap element in each application object. Here, the handle would typically be an array index

<u>The operations of a max-priority queue:</u>

[1] Following procedure implements the MAXIMUM operation in $\Theta(1)$ time.

```
HEAP-MAXIMUM(A)
1  return A[1]
```

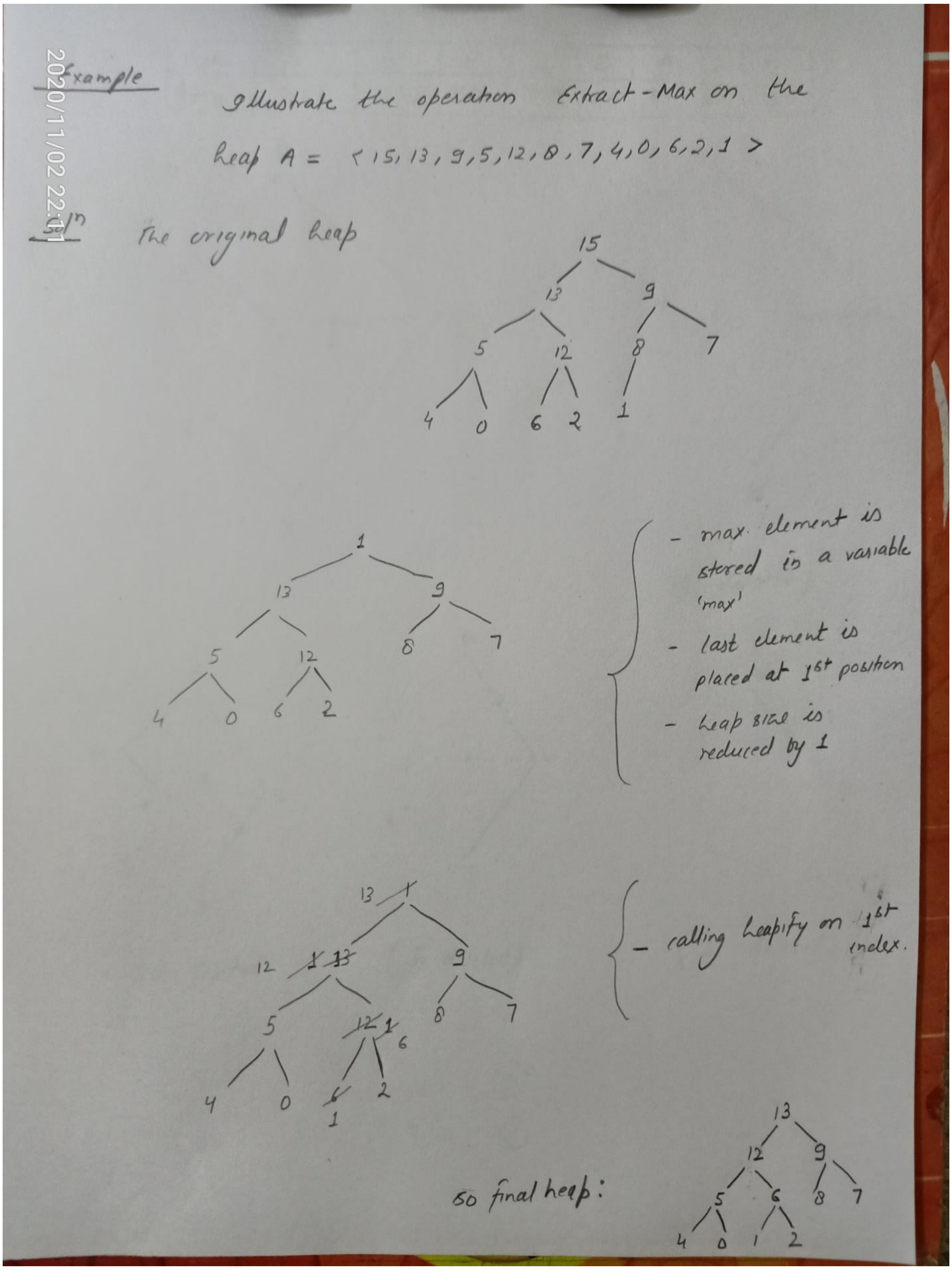
[2] The procedure given below implements the EXTRACT-MAX operation.

```
HEAP-EXTRACT-MAX(A)
1  if heap-size[A] < 1
2      then error "heap underflow"
3  max ← A[1]
4  A[1] ← A[heap-size[A]]
5  heap-size[A] ← heap-size[A] − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

// note the similarity to the for-loop body (lines 3–5) of the HEAPSORT procedure.

The **<u>running time</u>** of HEAP-EXTRACT-MAX is O(lg n), since it performs only a constant amount of work on top of the O(lg n) time for MAX-HEAPIFY.

Example (Extract-Max):



Example

Illustrate the operation Extract-Max on the heap A = < 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 >

Sol^n The original heap

15
13
9
5
12
8
7
4
0
6
2
1

1
13
9
5
12
8
7
4
0
6
2

- max. element is stored in a variable 'max'
- last element is placed at 1st position
- heap size is reduced by 1

13 ~~1~~
12 ~~1~~ ~~13~~
9
5
~~12~~ ~~1~~ 6
8
7
4
0
~~6~~ 1
2

- calling heapify on 1st index.

so final heap:

13
12
9
5
6
8
7
4
0
1
2

[3] The procedure given below implements the INCREASE-KEY operation. The priority-queue element whose key is to be increased is identified by an index *i* into the array.

```
HEAP-INCREASE-KEY($A, i, key$)
1  if $key < A[i]$
2      then error "new key is smaller than current key"
3  $A[i] \leftarrow key$
4  while $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
5          do exchange $A[i] \leftrightarrow A[\text{PARENT}(i)]$
6             $i \leftarrow \text{PARENT}(i)$
```

The procedure first updates the key of element A[i] to its new value.

- Because increasing the key of A[i] may violate the max-heap property, the procedure then, traverses a path from this node toward the root to find a proper place for the newly increased key.
  // Please note the similarity to the inner loop (lines 5–7) of INSERTION-SORT.
- During this traversal, it repeatedly compares an element to its parent, exchanging their keys and continuing if the element's key is larger, and terminating if the element's key is smaller, since the max-heap property now holds.
- **Loop-Invariant:**
  "At the start of each iteration of the while loop of lines 4–6, the array A[1 . . heap-size[A]] satisfies the max-heap property, except that there may be one violation: A[i] may be larger than A[PARENT(i)]."

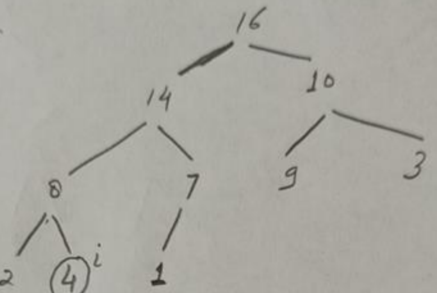
- The **running time** of HEAP-INCREASE-KEY on an n-element heap is O(lg n), since the path traced from the node updated in line 3 to the root has length O(lg n).

Example (Increase-Key):

Example

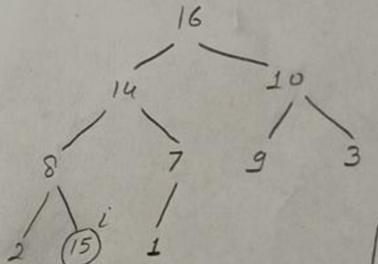
Illustrate the operation of Increase key on the heap A = < 16, 14, 10, 8, 7, 9, 3, 2, 4, 1 > for (A, i, key) ≡ (A, 9, 15)

Sol^n The original heap:



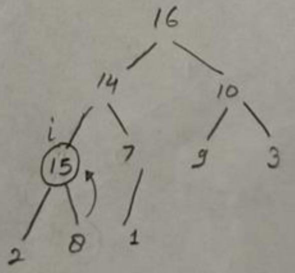
- check the initial condition on the value of key (guard clause)

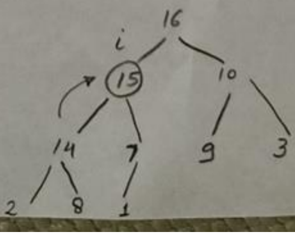
- update to its new value // A[i] ← key



★★ The array A[1.. heapsize] satisfies the max-heap property, except that there may be one violation: A[i] may be larger than A[Parent(i)]

Traverse a path from this node toward the root to find a proper place.



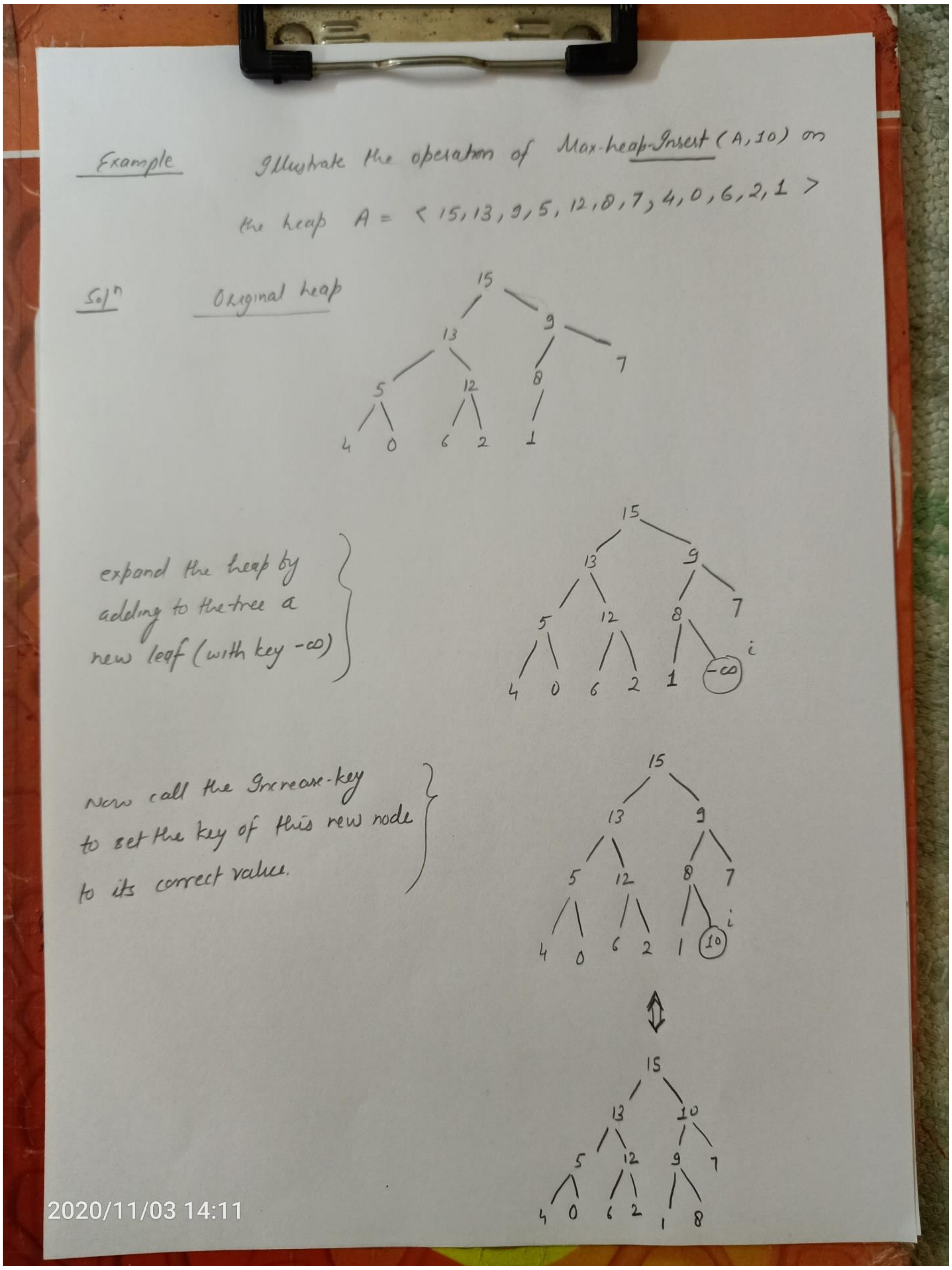


[4] The procedure given below implements the INSERT operation. It takes as an input the key of the new element to be inserted into max-heap A.

```
MAX-HEAP-INSERT($A$, $key$)
1  $heap\text{-}size[A] \leftarrow heap\text{-}size[A] + 1$
2  $A[heap\text{-}size[A]] \leftarrow -\infty$
3  HEAP-INCREASE-KEY($A$, $heap\text{-}size[A]$, $key$)
```

- The procedure first expands the max-heap by adding to the tree a new leaf whose key is $-\infty$.
- Then it calls HEAP-INCREASE-KEY to set the key of this new node to its correct value and maintain the max-heap property.
- The ***running time*** of MAX-HEAP-INSERT on an n-element heap is O(lg n).

Example (Insert):



Example Illustrate the operation of Max-heap-Insert (A, 10) on the heap $A = < 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 >$

Sol^n Original heap

expand the heap by adding to the tree a new leaf (with key $-\infty$)

Now call the Increase-key to set the key of this new node to its correct value.

15
13
9
5
12
8
7
4
0
6
2
1
$-\infty$
i
10
2020/11/03 14:11