

Binary Heap

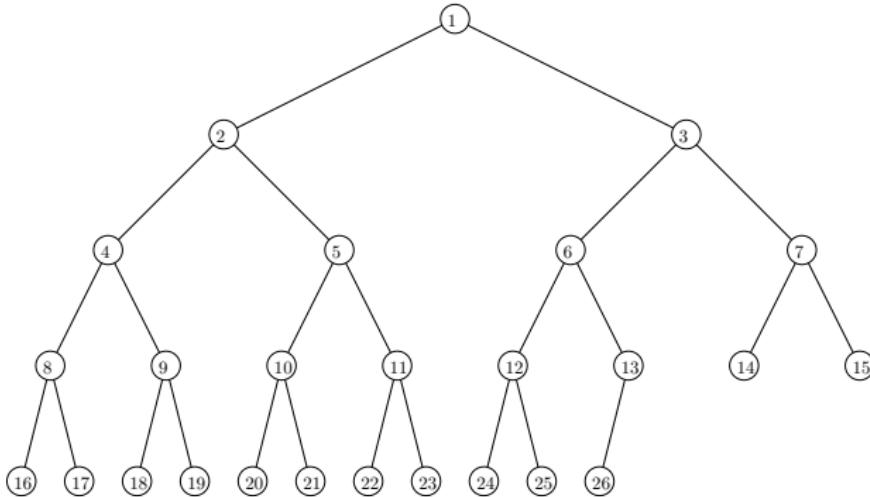
Data Structures and Algorithms (094224)

Tutorial 7

Winter 2022/23

Nearly Complete Binary Tree — Array Representation

- Representing an n -node nearly complete binary tree in array $A[1 \dots n]$
 - The root is identified with $A[1]$
 - Left child of internal node $A[i]$ is identified with $A[2i]$
 - Right child of internal node $A[i]$ is identified with $A[2i + 1]$
 - Parent of non-root node $A[i]$ is identified with $A[\lfloor i/2 \rfloor]$
 - Define $\text{Left}(i) = 2i$, $\text{Right}(i) = 2i + 1$, $\text{Parent}(i) = \lfloor i/2 \rfloor$



Binary heap

binary heap \langle ערמה בינארית \rangle

(sometimes simply heap)

- Objects associated with the nodes of a nearly complete binary tree T
- T represented in an array $A[1, \dots, A.length]$
 - $A.length$ is an upper bound on #objects in the dynamic set
 - Known in advance
- Data structure's attribute (on top of array's attributes):
 - $heap-size = \#$ currently stored objects (n)
- The binary heap property:
 $A[\text{Parent}(i)].key < A[i].key$ for every non-root $A[i]$
- Sometimes referred to as a *priority queue* \langle תור עדיפויות \rangle or *minimum heap* \langle ערמת מינימום \rangle
 - Switching the relevant operations/inequalities yields a *maximum heap* \langle ערמת מקסימום \rangle

Question 1

Prove/Disprove: given a heap, the second smallest key is a child of the root

Solution (the claim is true):

- Based on the heap property, the smallest key resides in the root of the heap
- For a heap of size 1 the claim holds (vacuously)
- For a heap of size 2 the claim holds
- For a heap of size ≥ 3 let r , ℓ and h be the key values of the root, root's left child and root's right child, respectively
- Let L be the set of keys in the subtree rooted at the root's left child (without ℓ)
- Let H be the set of keys in the subtree rooted at the root's right child (without h)

Solution — cont.

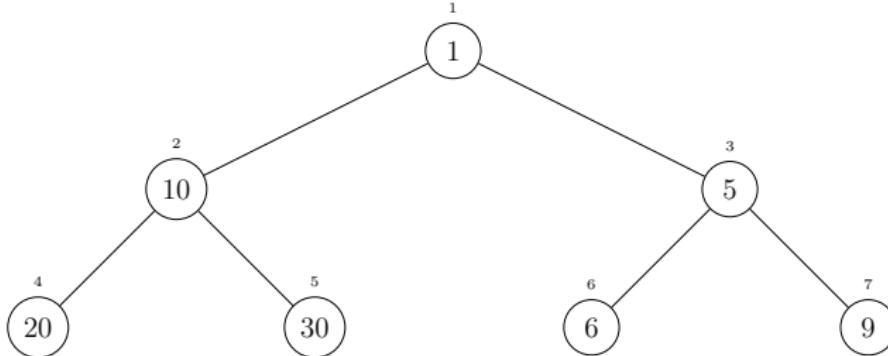
- $r < \ell \wedge \forall x \in L, \ell < x$
 - By the heap property
- $r < h \wedge \forall x \in H, h < x$
 - By the heap property
- Case 1: $\ell < h$
 - $r < \ell < h < x$ for all $x \in H$ and $r < \ell < y$ for all $y \in L$
 - $\Rightarrow \ell$ is the second smallest
- Case 2: $h < \ell$
 - $r < h < \ell < x$ for all $x \in L$ and $r < h < y$ for all $y \in H$
 - $\Rightarrow h$ is the second smallest

Question 2

Prove/Disprove: given a heap, the third smallest key is a child of the root

Solution (the claim is false):

- Consider the following heap where the third smallest key, 6, is not a child of the root
- Number inside node are keys, numbers above nodes are array indices



Question 3

Given an n element minimum heap A , what is the time complexity of an algorithm that returns the maximum key?

Solution:

- In a minimum heap we cannot use the heap property to search for the largest key in $O(\log n)$
- Maximum key must reside in a leaf
- In array A , leaves are at indices $\lfloor \frac{n}{2} \rfloor + 1, \dots, n$
 - Only the above indices have no left or right child thus, are leaves
- $O(n)$ time to return the maximum via exhaustive search
- Is there a faster algorithm?
 - Intuitively, it seems hopeless
 - Finding maximum requires querying every leaf
 - Leaves of minimum heap are ordered arbitrarily
 - Not a formal proof

Binary Heap — Operations

- **Heapify(A, i)** – Precondition: subtrees rooted at $A[Left(i)]$ and $A[Right(i)]$ are heaps; modify A so that the subtree rooted at $A[i]$ is a heap
 - Run time – $O(\log n)$
- **Build_Heap(A)** – Build a heap from the objects in $A[1 \dots A.length]$
 - Run time – $O(n)$
- **Heap_Extract_Min(A)** – Delete the minimum object from heap A and return it
 - Run time – $O(\log n)$
- **Heap_Decrease_Key(A, i, k)** – Precondition: $k < A[i].key$; change the key of $A[i]$ to k
 - Run time – $O(\log n)$
- **Heap_Insert(A, x)** – Insert the new heap node x into heap A
 - Run time – $O(\log n)$

Question 4

Given an n element heap A , show how to implement the following operations in $O(\log n)$

- `Heap_Delete(A, i)` – deletes element i from heap A
- `Heap_Increase_Key(A, i, k)` – Precondition: $k > A[i].key$; change the key of $A[i]$ to k

Solution

deletes element i from heap A

`Heap_Delete(A, i)`

- 1: `Heap_Decrease_Key($A, i, -\infty$)`
- 2: `Heap_Extract_Min(A)`

Run time analysis:

- Line 1 – $O(\log n)$
- Line 2 – $O(\log n)$
- In total – $O(\log n)$

Correctness:

- Follows directly from the correctness of heap operations

Solution — cont.

Precondition: $k > A[i].key$; change the key of $A[i]$ to k

Heap_Increase_Key(A, i, k)

- 1: **if** $A[i].key > k$ **then**
- 2: error "new key is smaller than current key"
- 3: $A[i].key = k$
- 4: Heapify(A, i)

Run time analysis:

- $O(\log n)$

Correctness:

- The left and right children of $A[i]$ are heaps. Correctness follows directly from $\text{Heapify}(A, i)$ correctness

Question 5

Give an $O(n \log k)$ -time algorithm to merge $k \geq 2$ sorted lists into one sorted list, where n is the total number of elements in all the input lists. Assume all elements are distinct.

Definition:

- The input to the algorithm is an array A (of size k) which holds the head of the k lists

Solution — first attempt

High level idea:

- ① Find x , the element with minimum key pointed by A
- ② Output $x.key$
- ③ Replace x in array A with $x.next$ if $x.next \neq NIL$
- ④ Repeat until output of all elements

Time Complexity:

- Finding minimum element out of k elements – $O(k)$
- n iteration of minimum finding. In total – $O(nk)$

Solution

High level idea:

- Maintain a heap of size k to extract the next smallest element to be output
- Initialize the heap with first element in each list

Solution — cont.

Merge k sorted lists into one sorted list

k-way_Merge(A)

```
1: new list  $O$ ,  $k = A.length$ , new heap  $B[1 \dots k]$ 
2: for  $i = 1$  to  $k$  do
3:    $B[i] = A[i].head$ 
4: Build_Heap( $B$ )
5: while  $B.heap-size > 0$  do
6:    $x = \text{Heap\_Extract\_Min}(B)$ 
7:   List_Insert_Tail( $O, x$ )       $\triangleright$  insert the element to the tail of  $O$ 
8:   if  $x.next \neq NIL$  then
9:     Heap_Insert( $B, x.next$ )
10: return  $O$ 
```

Solution — cont.

Run Time Analysis:

- Line 1 – $O(1)$
- For loop in Line 2 – $O(k)$
- Line 4 – $O(k)$
- While loop in Line 5 – $O(n \log k)$
 - Insert an element to a list – $O(1)$
 - Every element of the input lists is inserted and extracted exactly once – $O(n \log k)$
- In total – $O(k) + O(n \log k)$
- Since $k \leq n \implies O(n \log k)$

Question 6

Show how to implement a heap without apriori bound on the number of elements without changing heap operations run time.

Solution

- We can use a rooted tree with bounded degree (of 2) representation to represent a nearly complete binary tree
- We will use this representation to implement a heap
- Data structure's attribute:
 - *root* = pointer to root
 - *heap-size* = # of nodes (= # currently stored elements)
- Node attributes (additional):
 - *left* = pointer to left child
 - *right* = pointer to right child
 - *p* = pointer to parent
- Denote this data structure as *Tree_Heap*

Solution — cont.

- We will show (high level) heap operations implemented using *Tree_Heap*
 - $\text{Heapify_T}(T, x)$ – Precondition: subtrees rooted at $x.\text{left}$ and $x.\text{right}$ are heaps; modify T so that the subtree rooted at x is a heap
 - $\text{Heap_Decrease_Key_T}(T, x, k)$ – Precondition: $k < x.\text{key}$; change the key of x to k
 - $\text{Build_Heap_T}(A)$ – A is an array of objects, $\text{Build_Heap_T}(A)$ returns T , a *Tree_Heap*
 - $\text{Heap_Extract_Min_T}(T)$ – Delete the minimum object from heap T and return it
 - $\text{Heap_Insert_T}(T, x)$ – Insert the new heap node x into heap T

Solution — cont.

A is an array of objects, $\text{Build_Heap_T}(A)$ returns T , a Tree_Heap

$\text{Build_Heap_T}(A)$:

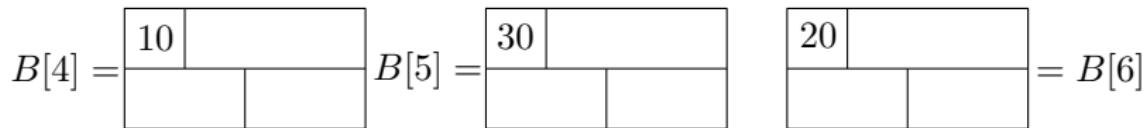
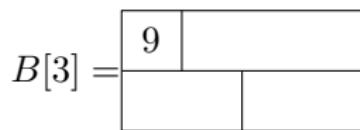
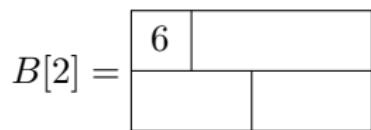
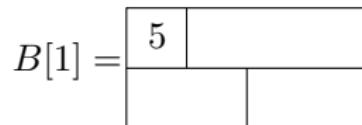
```
1: Build_Heap(A)
2: new array  $B[0 \dots A.length]$ 
3:  $B[0] = NIL$ 
4: for  $i = 1$  to  $A.length$  do
5:   new tree node  $x$                       ▷ node attributes initialized to  $NIL$ 
6:    $x.key = A[i].key$ 
7:    $B[i] = x$ 
8: for  $i = 1$  to  $A.length$  do
9:    $B[i].parent = B[\lfloor i/2 \rfloor]$ 
10:  if  $2i \leq A.length$  then
11:     $B[i].left = B[2i]$                       ▷ pointer to left child
12:    if  $2i + 1 \leq A.length$  then
13:       $B[i].right = B[2i + 1]$                   ▷ pointer to right child
14: new tree object  $T$ 
15:  $T.root = B[1]$ 
16:  $T.heap-size = A.length$ 
17: return  $T$ 
```

Build_Heap_T(A) – example

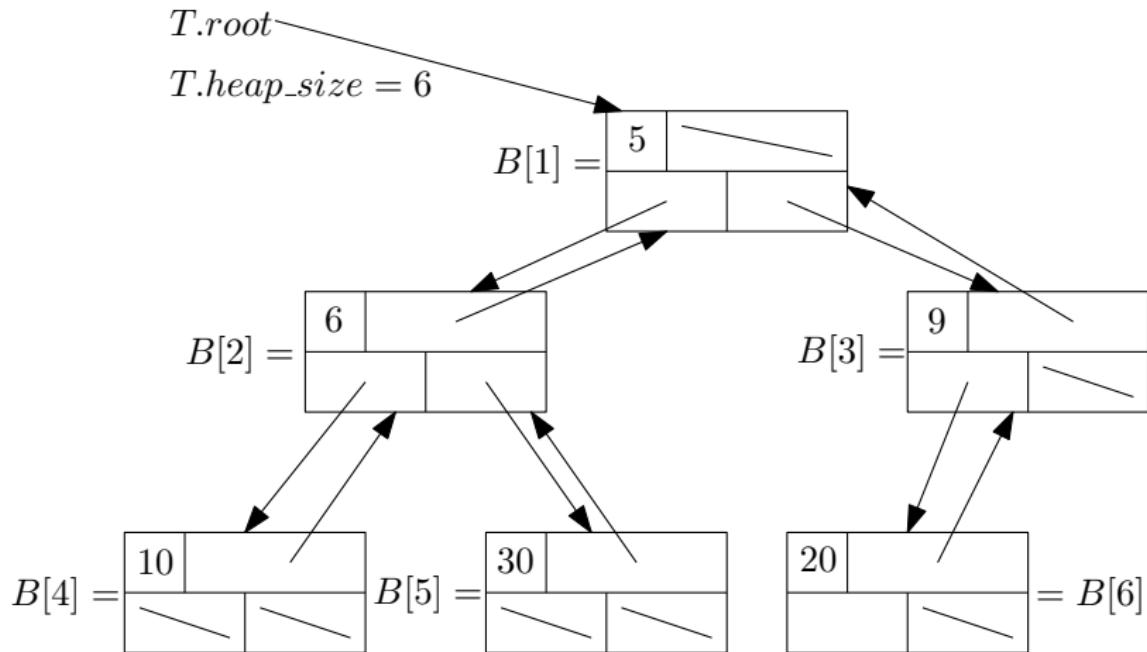
$$A = \boxed{\begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 20 & 10 & 30 & 5 & 6 & 9 \end{matrix}}$$

$$\text{Build_Heap}(A) = \boxed{\begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 5 & 6 & 9 & 10 & 30 & 20 \end{matrix}}$$

Build_Heap_T(A) – example cont.



Build_Heap_T(A) – example cont.



Solution — cont.

Run time analysis:

- `Build_Heap(A)` – $O(n)$
- For loop in line 4 – $O(n)$
- For loop in line 8 – $O(n)$
- Total – $O(n)$

Implementing `Heapify_T(T, x)`, `Heap_Decrease_Key_T(T, x, k)`:

- Similar to heap implementation using an array
- Changes associated with representing the heap with a bounded degree rooted tree representation instead of an array
- $O(\log n)$ time

Solution — cont.

Implementing `Heap_Extract_Min_T(T)`, `Heap_Insert_T(T, x)`:

- Similar to heap implementation using an array
- Changes associated with representing the heap with a bounded degree rooted tree representation instead of an array
- `Heap_Extract_Min_T(T)` require the location of the "last" leaf in T
 - The right most leaf with the highest depth
- `Heap_Insert_T(T, x)` require the parent of the leaf to be inserted
- By using the properties of a nearly complete binary tree we can reach any node with "index" i from $T.root$ in $O(\log n)$ time
 - Node with index i in a nearly complete binary tree is the node that would reside in index i if an array was used to represent the nearly complete binary tree

Solution — cont.

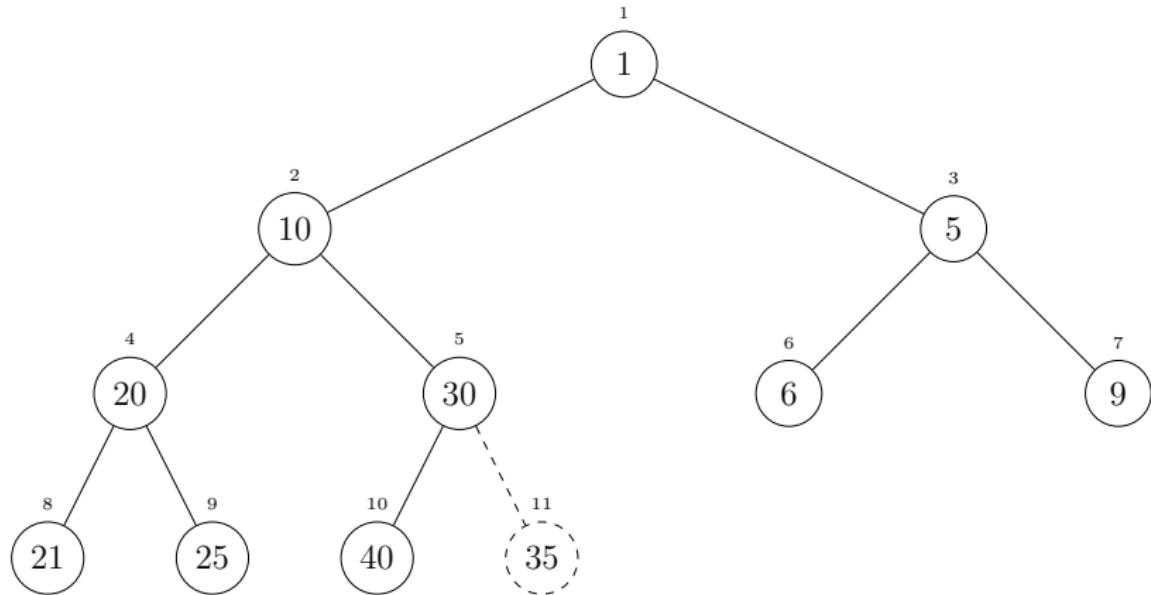
- Let i be some index. In order to locate in $O(\log n)$ the corresponding node from $T.root$ perform:
 - If i is even then the last "hop" in the simple path from $T.root$ to node corresponding to index i is left
 - Otherwise, the last "hop" in the simple path from $T.root$ to node corresponding to index i is right
 - set $i = \lfloor \frac{i}{2} \rfloor$
 - Repeat until $i = 1$

Run time analysis:

- Find leaf – $O(\log n)$
- In total - $O(\log n)$

Solution — Example

$T.heap-size = 10$



Order Statistics and Medians

- The i th *order statistic* סטטיסטי הסדר \langle of a set of n elements is the i th smallest element
 - The minimum is the first order statistic ($i = 1$)
 - The maximum is the n th order statistic ($i = n$)
- A *median* חציון \langle , informally, is the "halfway point" of the set
- If n is odd ,the median is unique, occurring at $i = \frac{n+1}{2}$
- If n is even, there are two medians
 - $i = \frac{n}{2}$ (lower median); and
 - $i = \frac{n}{2} + 1$ (upper median)
- Assumptions:
 - The phrase "the median" refer to lower median
 - Regardless of the parity of n , median occur at $i = \lfloor \frac{n+1}{2} \rfloor$
 - The median of n elements can be found in $O(n)$ -time
 - As we shall see in Tutorial 9

Question 7

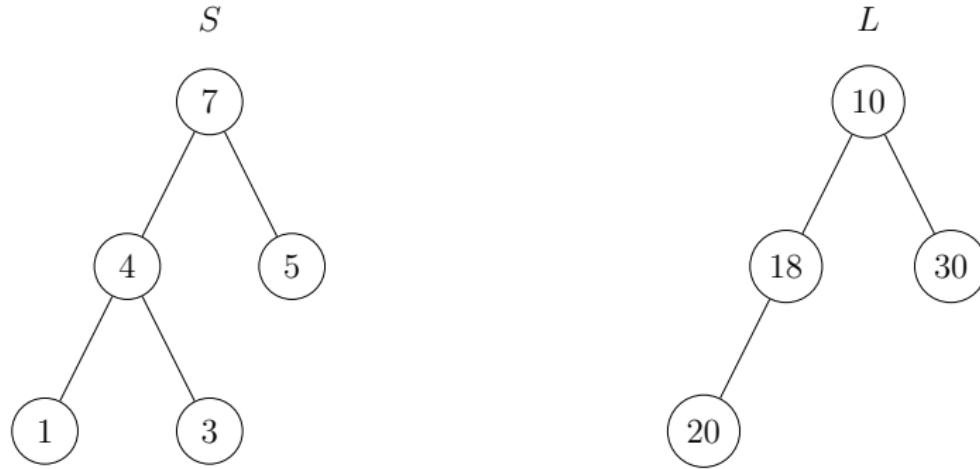
Given a linked list, Q , with distinct key values, implement a dynamic data structure, M , that supports the following operations:

- $\text{Init}(Q)$ - initialization of the dynamic data structure from the linked list with n elements - time complexity $O(n)$
- $\text{Insert}(M, x)$ - adds x to M - time complexity $O(\log n)$
- $\text{Find_Median}(M)$ - returns the median of keys in M - time complexity $O(1)$
- $\text{Del_Median}(M)$ - deletes the median from M - time complexity $O(\log n)$

Solution

Main idea:

- The dynamic data structure will be two heaps implemented as in question 6 since apriori bound is unknown
 - A **maximum heap**, we refer as S , containing $\lfloor \frac{n+1}{2} \rfloor$ smallest elements
 - A **minimum heap**, we refer as L , containing $\lfloor \frac{n}{2} \rfloor$ largest elements
- Example: the keys in Q are $\{1, 3, 4, 5, 7, 10, 18, 20, 30\}$



Solution — cont.

Notice:

- The median is the root of the max heap S
- If n is even, then $\lfloor \frac{n+1}{2} \rfloor = \lfloor \frac{n}{2} \rfloor$
- If n is odd, then $\lfloor \frac{n+1}{2} \rfloor - 1 = \lfloor \frac{n}{2} \rfloor$
- $L.\text{heap-size} = S.\text{heap-size}$, if n is even
- $L.\text{heap-size} = S.\text{heap-size} - 1$, if n is odd

`Init(Q)`

- Find the median of keys in Q — $O(n)$
- Divide the elements of the linked list into two arrays
 - Smaller or equal than the median
 - Greater than the median
 - $O(n)$ time complexity
- Invoke `Build_Min_Heap_T` to build heap L — $O(\frac{n}{2})$ time
- Invoke `Build_Max_Heap_T` to build heap S — $O(\frac{n}{2})$ time
- In total — $O(n)$ time

Solution — cont.

Find_Median(M)

- The median is the root of heap S
- Return root of a heap — $O(1)$

Insert(M, x)

- If $x.\text{key} < \text{Find_Median}(M).\text{key}$, invoke `Heap_Max_Insert_T(S, x)`
- Otherwise, invoke `Heap_Min_Insert_T(L, x)`
- We must maintain the sizes of L and S
 - If, after inserting x , $L.\text{heap-size} = S.\text{heap-size} + 1$ invoke,
 - $r = \text{Heap_Min_Extrac_Min_T}(L)$
 - `Heap_Max_Insert_T(S, r)`
 - If, after inserting x , $L.\text{heap-size} = S.\text{heap-size} - 2$ invoke,
 - $r = \text{Heap_Max_Extrac_Max_T}(S)$
 - `Heap_Min_Insert_T(L, r)`
- Time complexity — $O(\log \frac{n}{2}) + O(\log \frac{n}{2}) + O(\log \frac{n}{2}) = O(\log n)$ time

Solution — cont.

Del_Median(M)

- The median is the root of heap S , invoke `Heap_Max_Extrac_Max_T(S)`
- If $L.heap\text{-}size} = S.heap\text{-}size + 1$ invoke,
 - $r = \text{Heap_Min_Extrac_Min_T}(L)$
 - `Heap_Max_Insert_T(S, r)`
- Time complexity – $O(\log \frac{n}{2}) + O(\log \frac{n}{2}) + O(\log \frac{n}{2}) = O(\log n)$ time