NO.

DATE April 19, 2023

PRIORITY QUEUES

> A set ADT with operations:
1. insert
2. DeleteMin
   ↳ Note: the function itself may vary depending on type of ordering e.g. in a BST, the MIN is the leftmost node. But in Priority Queue, min may be the root or elsewhere.

> includes initialize & MakeNull

> APPLICATIONS: Process scheduling in a time-shared computing system.

• Operations •

1. Insert(x, A) - adds element X in set A, if the element X is not a member of the set.
2. Deletemin(A) - removes and returns the smallest in set A if the set is not empty; otherwise the operation fails.

// Note: For basis of deleting, either the priority # or key ID of the element will be used.

* IMPLEMENTATIONS *

1.) BitVector - the first elem to be seen is the smallest
2.) Linked List - O(N) for both operations
3.) Array
4.) Cursor-Based
5.) Partially Ordered Tree (P.O.T)

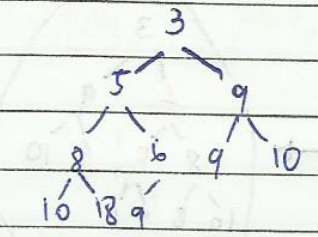
Sterling

NO.

DATE

★ Partially Ordered Tree ★

• Illustration:



• Observations:

> is Height Balanced

> There are duplicate elements - The numbers seen are not elements, but the priority number of elements

∴ Unique elements can share the same priority #

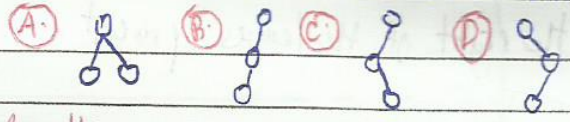
> Parents have smaller or equal to their children

★ Characteristics of P.O.T ★

1) Binary Tree - a tree in which each node has either a left child, or a right child or both left and right children

2) Balanced Tree - a tree in which the height is a minimum for the # of nodes.

↳ for instance, if we have 3 nodes, it can be:



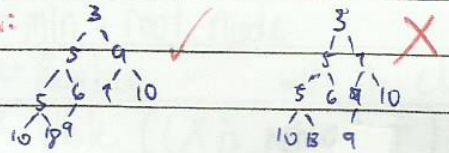
// Recall: a tree is height balanced when: the max diff. from the left subtree to the right subtree is 1.

OR

(height of left subtree - height of right) ≤ 1

3.) At the lowest level, where the leaves may be missing, we require that all missing leaves are to the right of all the leaves present in the lowest level.

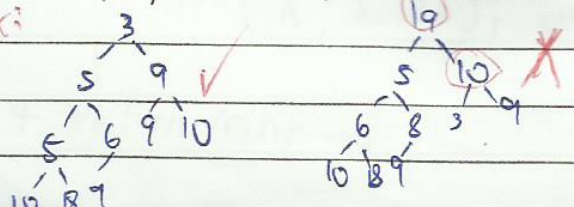
ex:



4.) POT Property - the priority of the parent is less than or equal to the priority of the children

Priority (Parent) <= Priority (child)

ex:

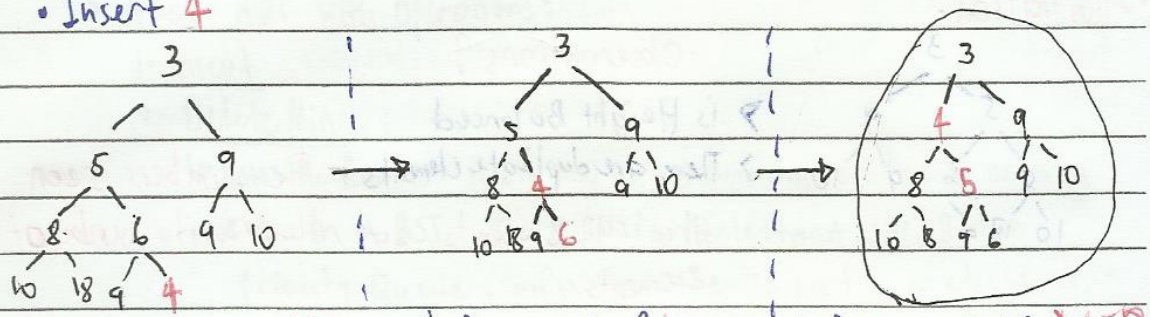


NO.

DATE

★ Illustrations of P.O.T Operations ★

- Insert 4



- Is P.O.T satisfied? NO
  (4 is not root && 6 > 4)
  - SWAP 6 & 4

6
temp

- is P.O.T satisfied? NO
  (5 > 4 & 4 is not the root)
  - SWAP 5 & 4

5
temp

- is P.O.T satisfied? YES

// worst case is that we reach the root Node = $O(\log_2 N)$

Summary: Steps in INSERT Operation

1. Place new element X at
   the lowest level to the right of the leaves present

   OR

   next level if current level is full.
2. while (X is not root and POT property is not satisfied)
   SWAP (parent, X);

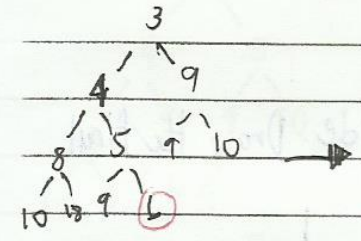
Sterling

NO.

DATE

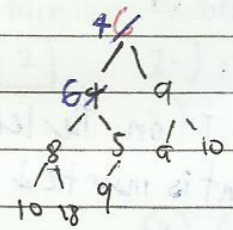
• Delete Min

// Note: we will be instead replacing the root node, deleting results in a forest



• To be returned [3] min

• 6 is to replace 3



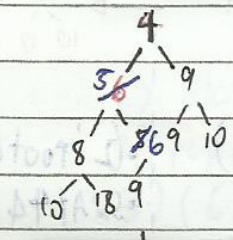
• is P.O.T satisfied? NO

1.) Determine the smaller child (schild)

schild = 4

2.) swap parent & schild

• To be returned [3] min [4] temp



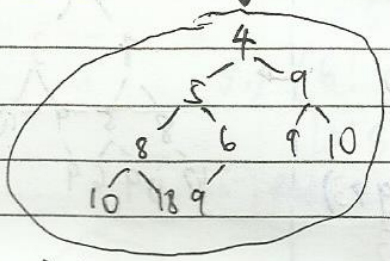
• is P.O.T satisfied? NO

1.) Determine the smaller child (schild)

schild = 5

2.) Swap parent & schild

• To be returned [3] min [5] temp



• is P.O.T property satisfied? YES

• Return minimum priority 3

*Summary: steps in Delete min operation

1. min = root node
2. Replace root with the element X found at the lowest level far right.
3. While ((X is not a leaf) and NOT POT) & ← parent node has a bigger value than its children

```
    Schild = smaller child of parent X;
    SWAP = (X, schild);
}
```

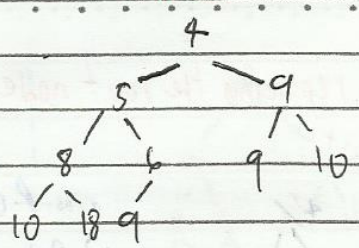
4. return min;

Sterling

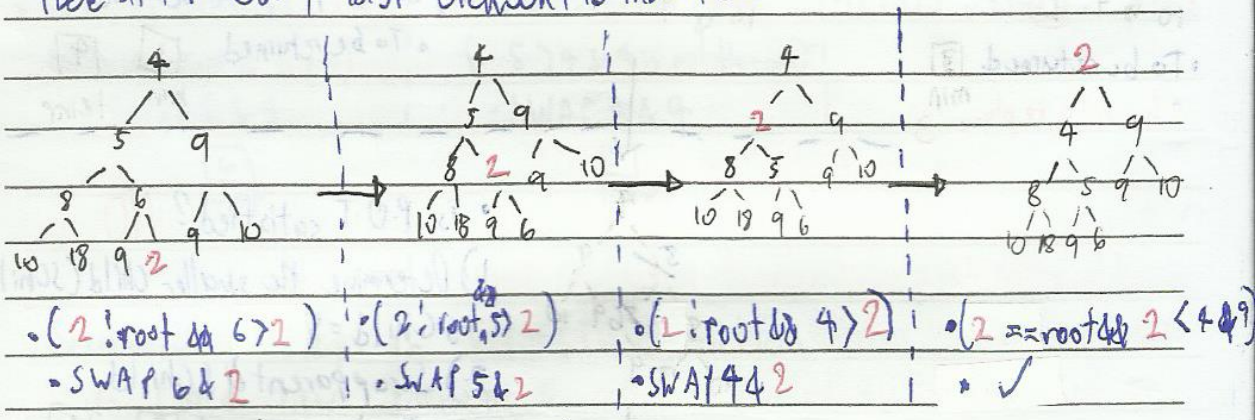
NO.

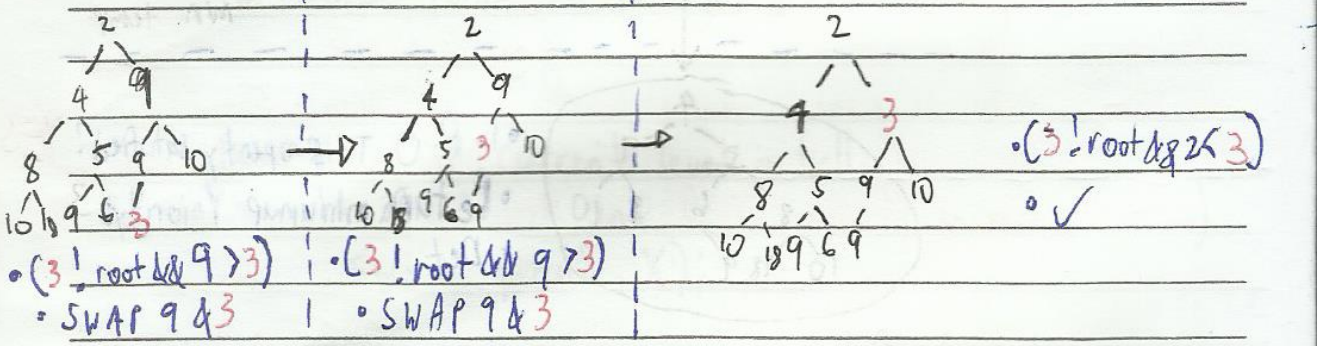
DATE

Exercise 1.

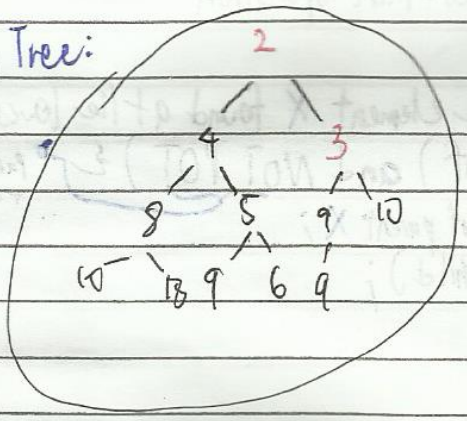


1. Insert 2 and 3 in the P.O.T on the left side. Draw the final tree after every last element is inserted





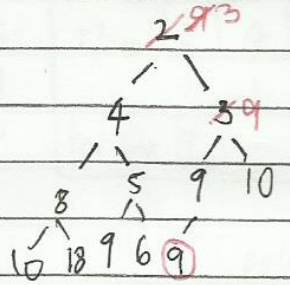
Final Tree:



NO.

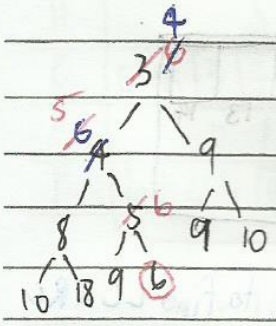
DATE

2. Perform DeleteMin 2 times using the final tree in #1. Draw the final tree after every DeleteMin operation



min
[2]

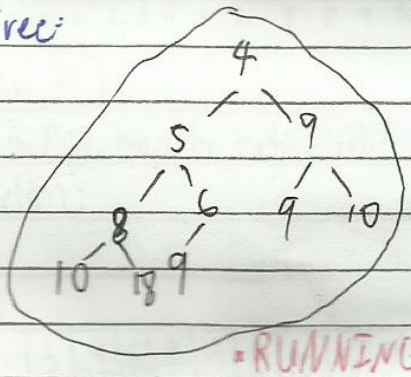
1.) Store 2 in min
2.) replace 2 with 9
3.) (9! = leaf && POT ! satisfied)
↳ SWAP 9 & 3
4.) (9! = leaf && POT == satisfied
↳ STOP
5.) return min.



min
[3]

1.) Store 3 in min
2.) replace 3 w/ 6
3.) (6! = leaf && POT ! satisfied)
↳ SWAP 6 & 4
4.) (6! = leaf && POT ! satisfied
↳ SWAP 6 & 5
5.) (6! = leaf && POT == satisfied)
↳ STOP
6.) Return min

Final Tree:



•RUNNING TIME•

- The running time for both Insert & DeleteMin operations is $O(\log_2 N)$
- The maximum path of the tree has $1 + \log_2 N$ nodes.

Sterling

NO.

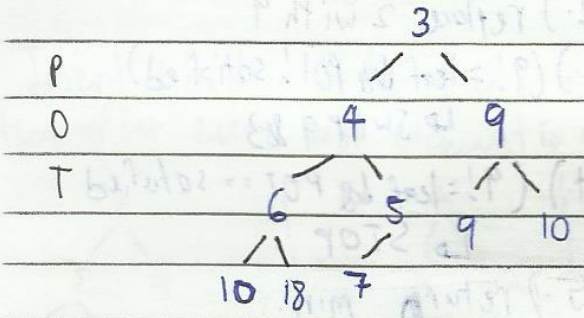
DATE

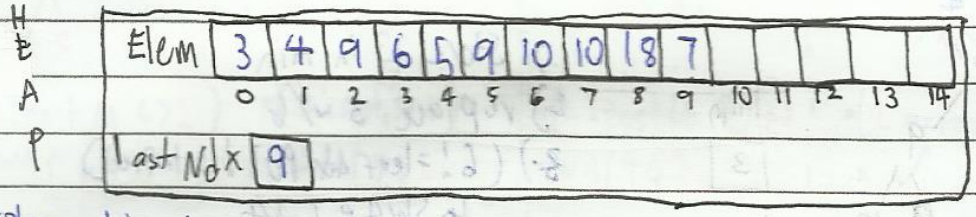
~o~ IMPLEMENTATION of POT ~o~

- Heap → Array Implementation of P.O.T

//Note: This is not the dynamically allocated memory space, they just have the same name

- Illustration/Ex:





- Observations:

Parent Node	P. Index	LC. Index	RC. Index
3	0	1	2
4	1	3	4
9	2	5	6
6	3	7	8

* Formula to find LC & RC

LC = 2 * Parent + 1

RC = 2 * Parent + 2

* Formula to find Parent

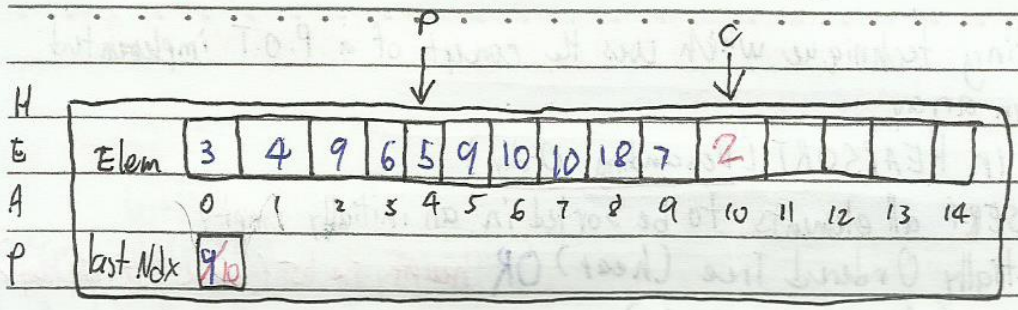
Parent = (child - 1) / 2

Sterling

NO.

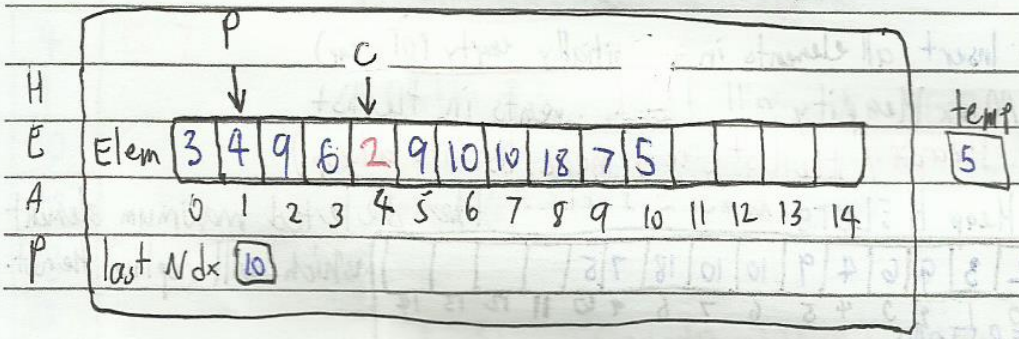
DATE

Insert 2 in the Heap



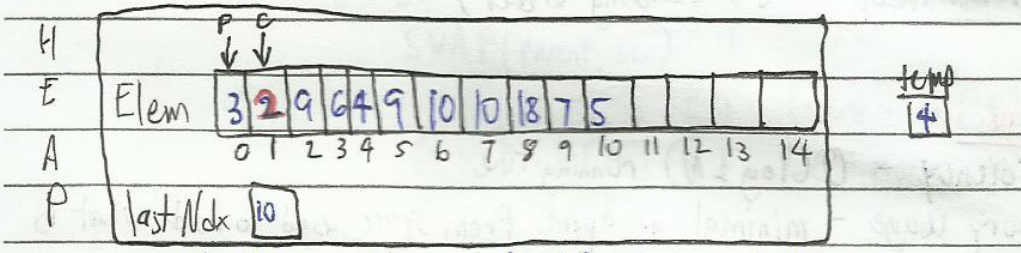
- is POT property satisfied? NO → SWAP 2 & 5

- 1st iteration:



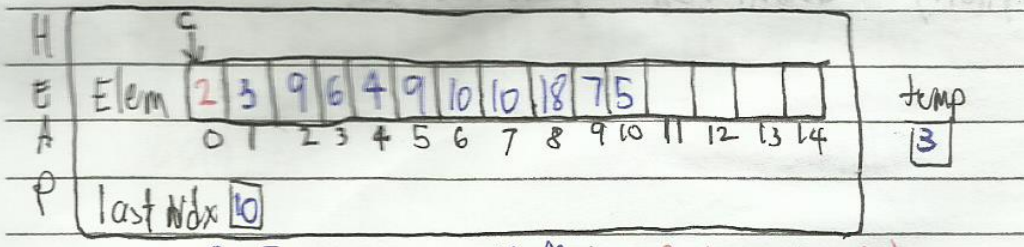
- is POT property satisfied? NO → SWAP 2 & 4

- 2nd iteration:



- is POT property satisfied? NO → SWAP 2 & 3

- 3rd iteration:



- POT property is satisfied → 2 is now inserted

Sterling

NO.

DATE

~ HeapSort ~

* A sorting technique which uses the concept of a P.O.T implemented using an array.

* Steps in HEAPSORT (Descending Order):

1.) INSERT all elements to be sorted in an initially empty Partially Ordered Tree (Heap) OR heapify the list (make list into heap)

2.) While (POT is not empty)

- DELETEMIN and store the deleted minimum element in the position of the element which has replaced the root.

* Steps in HEAPSORT (Ascending Order):

1.) MaxHeapify all the elements in the list

2.) While(POT/Heap != empty)

- DELETE MAX and store the deleted maximum element in the position of the element which will replace the root.

* TWO VERSIONS (as seen above)

- Min Heap - (descending order)
- Max Heap - (ascending order)

* Advantages

1.) Efficiency - O(log2 N) running time

2.) Memory usage - minimal as apart from space used to hold what is necessary (initial list), no additional memory space is needed

3.) Simplicity - doesn't require/use recursion

Sterling

NO.

DATE

// Note for HeapSort: unsorted list → P.O.T heap → Sorted

occur in the same array

• Example:

last 10

	Elem	old last 10
0	~~2~~	~~5 3 4~~
1	~~3~~	~~5 4 5~~
2	9	
3	6	
4	~~4~~	~~5~~ 7
5	9	
6	10	
7	10	
8	18	
9	~~7~~	3
10	~~5~~	2

• 1st iteration:

Deletemin(2)

temp = 2, last = 9

Parent: 5[0]

LChild: 3[1]

RChild: 9[2]

SmallChild: 3[1]

SWAP(Parent, SC)

temp = 5

Parent = 5[1]

LC = 6[3], RC = 4[4]

SC = 4[4]

SWAP(Parent, SC)

Parent = 5[4]

LC = 7[9]

auto SC

END

• 2nd iteration:

Deletemin(3)

temp = 3, last = 8 (?)

SWAP()

temp = 7

Parent = 7[0]

LC = 4[1], RC = 9[2], SC = 4[2]

SWAP(Parent, SC)

Parent = 7[1]

LC = 6[3], RC = 5[4]

SC = 5[4]

SWAP(parent, SC)

Parent = 7[4]

LC = none so its a leaf

END

* keep doing deleteMin() until last = -1

// Summary shown next page

// The Root will always be SMALLEST, so remove/replace first.

// The old Last var is there to hold orig. value of Last Ndx as we will manipulate the LastNdx during sorting.

Sterling

NO.

DATE

HeapSort Result of example 1

last	Elem	0	1	2	3	4	5	6	7	8	9	10
10		2	3	9	6	4	9	10	10	18	7	5
9		3	4	9	6	5	9	10	10	18	7	2
8		4	5	9	6	7	9	10	10	18	3	2
7		5	6	9	10	7	9	10	18	4	3	2
6		6	7	9	10	18	9	10	5	4	3	2
5		7	10	9	10	18	9	6	5	4	3	2
4		9	10	9	10	18	7	6	5	4	3	2
3		9	10	18	10	9	7	6	5	4	3	2
2		10	10	18	9	9	7	6	5	4	3	2
1		10	18	10	9	9	7	6	5	4	3	2
0		18	10	10	9	9	7	6	5	4	3	2
-1		18	10	10	9	9	7	6	5	4	3	2

→ Now SORTED LIST

RESULT : SORTED HEAP

0	18
1	10
2	10
3	9
4	9
5	7
6	6
7	5
8	4
9	3
10	2

Descending

last

~~-1~~ 10

* After Heapsort, restore original value of Last Idx using old Last variable

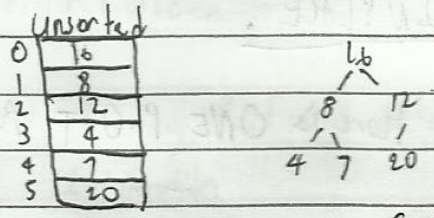
Sterling

NO.

DATE

• Example 2:

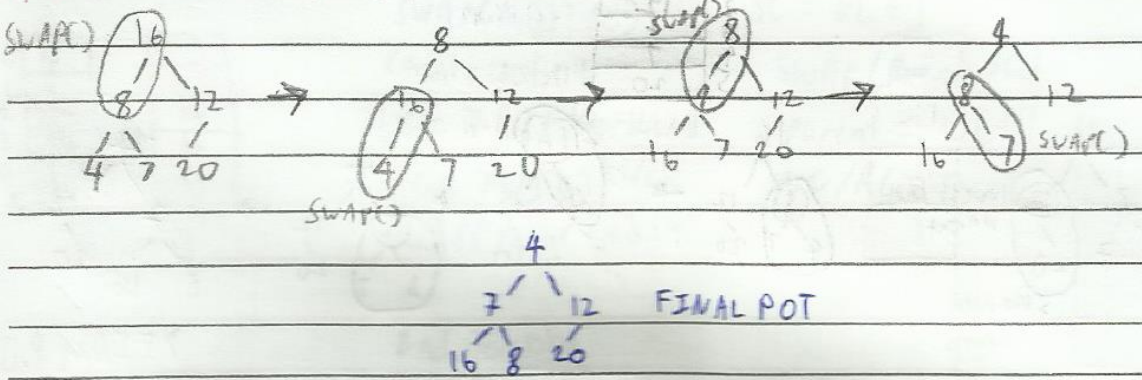
unsorted list → POT heap → sorted



1.) Check if list satisfies P.O.T property. (methods : >Initially empty >Heapify)

Create P.O.T heap in same Array

method 1: Initially Empty

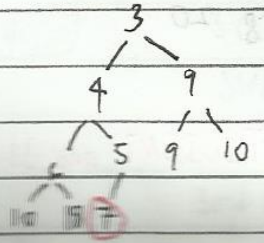


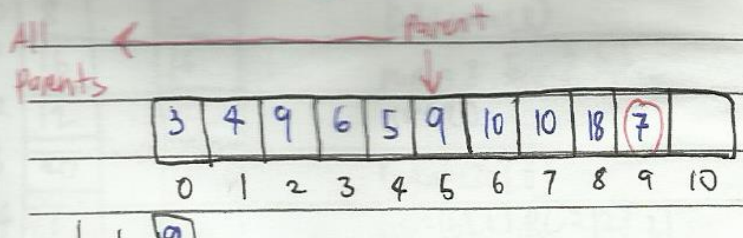
Method 2: HEAPIFY

1.) Start at LOWEST LEVEL PARENT

- Parent of last elem (smallest)
- once found, every elem prior to parent are ALSO PARENTS

• Example:





Sterling

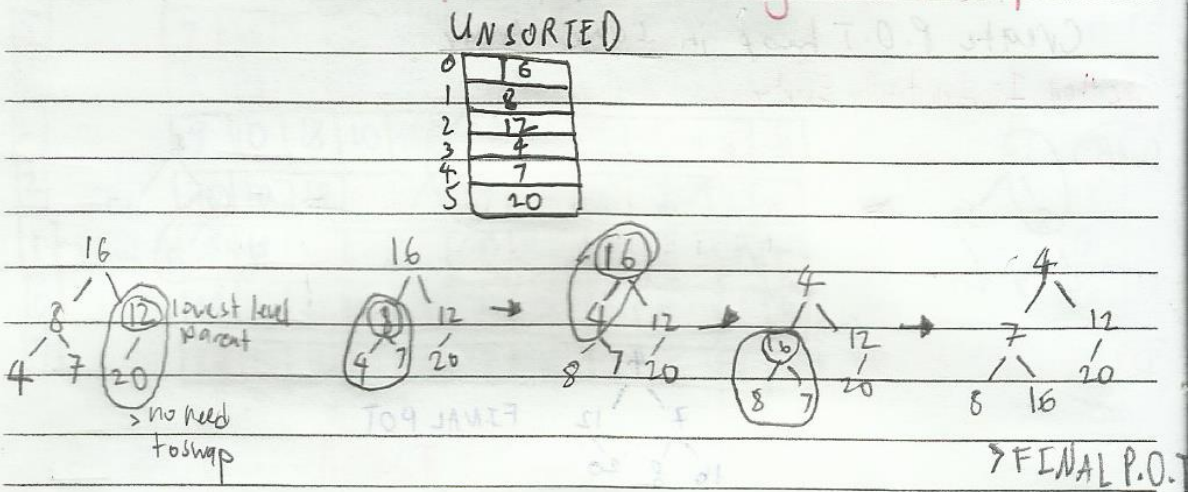
NO.

DATE

2.) Look at each parent & subtree and see if P.O.T property is swapped.
If no, SWAP IN PLACE.

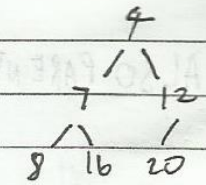
3.) Not necessary there is ONE P.O.T for one set of elems.

HEAPIFY version of creating P.O.T heap:



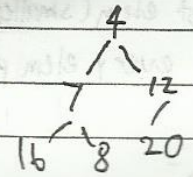
★ Comparison

POT via Heapify



VS

POT via Insert in Initially Empty



// Heapify process is MORE EFFICIENT and shorter.

NO.

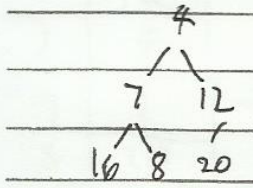
DATE

(2.) After P.O.T heap is created, SORT the HEAP

> min heap order (descending)

> max heap order (ascending)

MIN HEAP - HEAPSORT (using POT via initially empty)



1st iteration

Deletemin(4)

min = 4, last = 4

temp = 4

SWAP(Parent, SC)

Parent = 20[0]

LC = 7[1], RC = 12[2]

SC = 7[1]

SWAP(Parent, SC)

temp = 20

Parent = 20[1]

LC = 16[3], RC = 8[4]

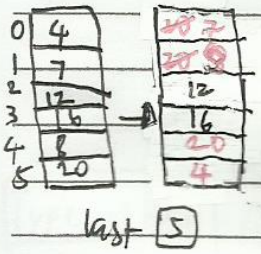
SC = 8[4]

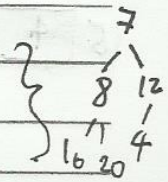
SWAP(Parent, SC)

parent = 20[4]

LC/RC = -

STOP





2nd iteration

Delete min(7)

min = 7, temp = 7, last = 3

SWAP()

parent = 20[0]

LC = 8[1], RC = 12[2]

SC = 8[1]

SWAP(Parent, SC)

Parent = 20[1]

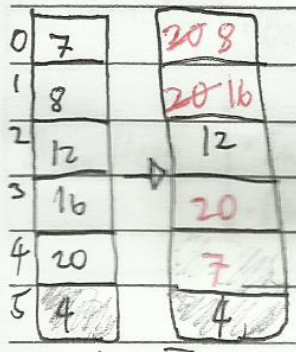
LC = 16[3], RC = -

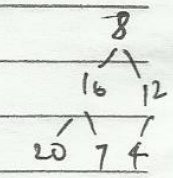
SWAP()

parent = 20[3]

LC/RC = -

STOP





3rd iteration

Deletemin(8)

min = 8, temp = 8, last = 2

SWAP()

Parent = 20[0]

LC = 16[1], RC = 12[2]

SC = 12[2]

SWAP()

Parent = 20[2]

LC/RC = -

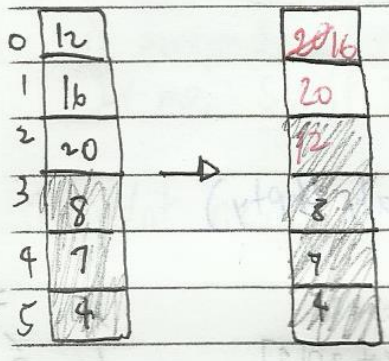
STOP

Sterling

NO.

DATE

4th Iteration



DeleteMin(12)

min=12, temp=12, last=1

SWAP()

Parent = 20[0]

LC = 16[1]

RC = –

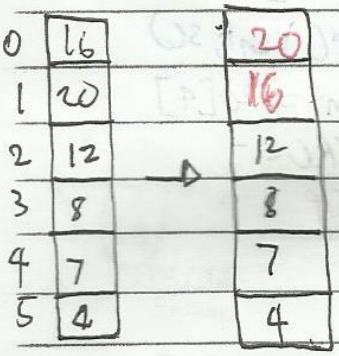
SWAP()

parent = 20[1]

RC/LC = –

STOP

5th iteration



DeleteMin(16)

min=16, temp=16, last=0

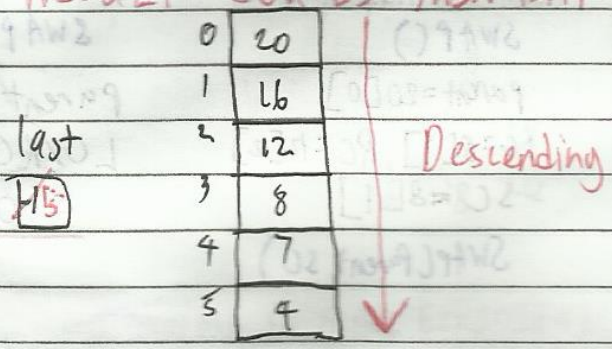
SWAP()

parent = 20[0]

LC/RC = –

STOP

RESULT: SORTED MINHEAP



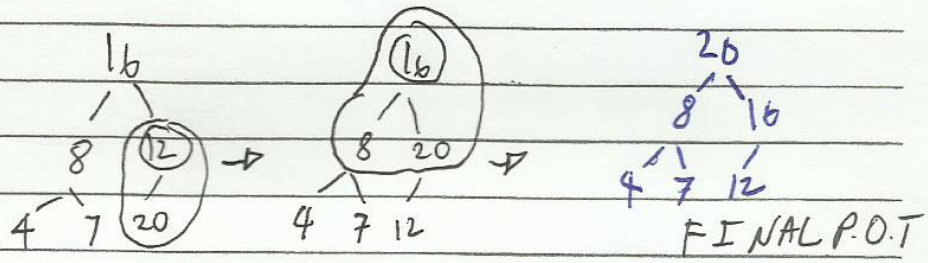
Sterling

NO.

DATE

MAX HEAP - UNSORTED (using Heapify)

	Unsorted
0	16
1	8
2	12
3	4
4	7
5	20



- P.O.T property for max Heap is P(parent) >= P(children)

- SUMMARY -

Type of Heap	Operations	POT Property	Heapify	Root Elem	Heapsort (in place)
Min Heap	• Insert • DeleteMin	P(p) <= P(c)	4 7 12 8 16 20	smallest	descending
Max Heap	• Insert • DeleteMax	P(p) >= P(c)	20 8 16 4 7 12	biggest	ascending