NO:
DATE:

-PRIORITY QUEUE-

* Priority Queue

- set ADT (unique elements) with operations (INS, DM)
- type of queue that arranges elements based on their priority values
- elements w/ higher priority values are typically retrieved first

* Operations

1. INSERT (x, A) - adds element x in set A IF elem $\notin$ A
2. DELETEMIN (A) - removes & returns smallest elem OR elem w/ smallest priority num.
   if set is not empty
   - will never use BOTH (smallest elem / smallest PN)
   - if each elem has o priority num. base the function on PN
     ↳ smallest PN = highest priority
   - if no PN, primary key is used to determine smallest elem
3. INIT(A)
4. MAKENULL (A)

* Implementations

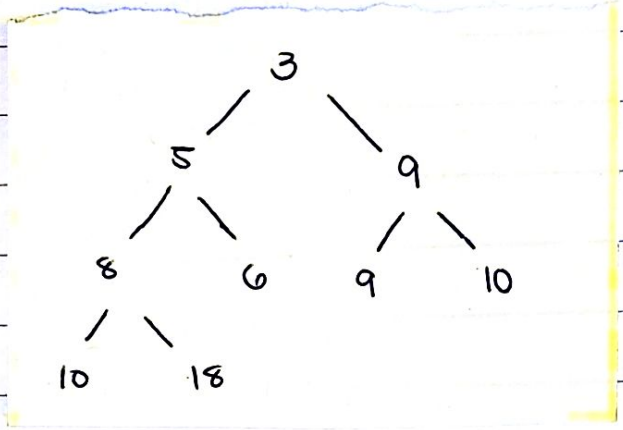
1. Bit-vector - first elem is SMALLEST
2. Linked List - DEAD O(N)
3. Array
3. CB
4. Partially Ordered Tree (P.O.T)

NO.

DATE:

-PARTIALLY ORDERED TREE-

*Partially Ordered Tree



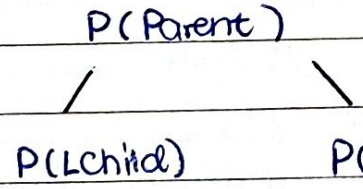
*Characteristics

1.) Binary Tree - each node has R/L child

2.) Balanced Tree - height difference of L subtree & R subtree is AT MOST 1.

3.) At the lowest level, missing leaves should be TO THE RIGHT of all leaves present at lowest level.

4.) POT Property - priority of parent is <= to priority of children



P(Parent) <= P(children)

* Right subtree cant be bigger height than Left

* Unique elements can have SAME PN..

*When current level is full, move to next level. Left → Right

NO:

DATE:

-P.O.T OPERATIONS-

* Insert

1.) Start at LEAF LEVEL.

Insert new element x at the right of leaves present.

OR

Insert at next level if current level is full.

2.) compare x to parent and SWAP until POT property is satisfied OR x is NOT ROOT.

3.) Child node needs to know where parent is to compare (formula).

Running Time : worst $O(\log_2 N)$ best $< O(\log_2 N)$

* Deletemin

1.) Set minimum = root node bcs its the SMALLEST.

Return FIRST, then "remove".

2.) Don't delete nodes as it will create a FOREST.

Just replace root with element x found at lowest level, far right.

- happens in each subtree SO parent must KNOW L/R to determine smaller child.

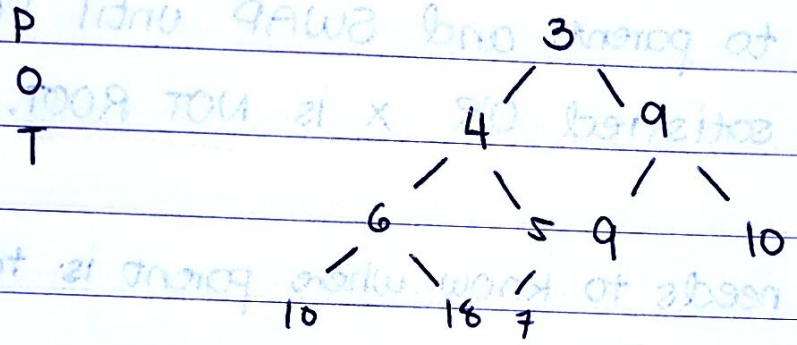
3.) SWAP until POT property is satisfied AND x is a leaf.

Max path : $1 + (\log_2 N)$ nodes

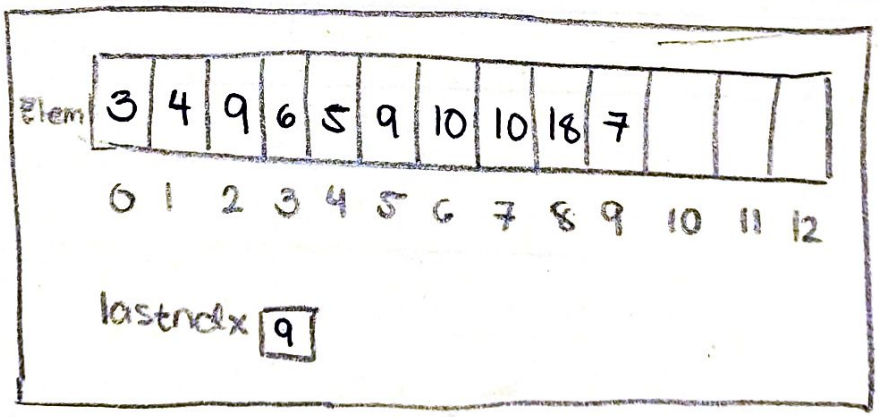
* Implementations

• Heap

- Array Implementation of P.O.T
- data structure



HEAP



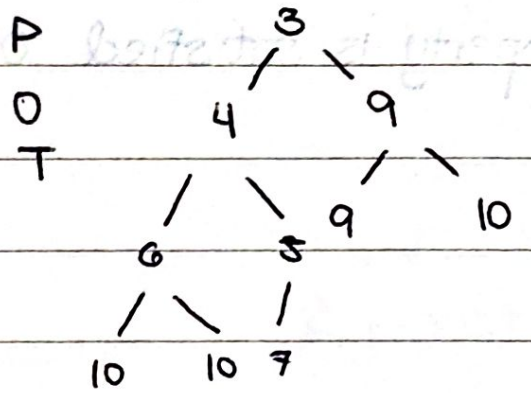
* Input elems in HEAP by LEVEL. Root is at index [0].
Last index is -1 if list is empty.

* Pattern: index of Parent: increment of 1

index of LChild: parent +1

index of RChild: parent +2

* Formula to get Parent, LChild, RChild



Parent Node	Parent Index	LChild Index	RChild Index
3	0	1	2
4	1	3	4
9	2	5	6
6	3	7	8
5	4	9	-

PARENT		CHILDREN	
using LChild	using RChild	LChild	RChild
(LC-1)/2	(RC-2)/2	2P+1	2P+2

* make it generic:

$$PARENT = \frac{child - 1}{2}$$

* depends on array starting with index [0]

* Inserting in Heap

- same steps as before, just manipulating elems using index until POT property is satisfied OR elem x is at root now.

NO:
DATE:

- HEAPSORT -

* Heapsort

- sorting technique that uses the concept of POT implemented using heap
- comparison-based sorting technique
- find minimum element & place it at the beginning & repeat for rest of elems

* Advantages

1. Efficiency - $O(\log_2 N)$
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

* Two Versions

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

DATE:

* steps

1.) INSERT all elements to be sorted in an initially empty P.O.T.

- we build the P.O.T. (heap)
- heapify the list (make list into heap)

2.) Perform DELETEMIN and store deleted minimum element in the position of element which has replaced root UNTIL POT is empty.

unsorted list → P.O.T heap → sorted

SAME ARRAY

Example

oldlast [10]

Elem last [10] temp : 2

* Root will always be SMALLEST. So, remove/replace first.
* old Last var to hold OG value of Last as we will manipulate last during sorting.

0	2	5 3 7 4 1ST Deletemin(2)
1	3	5 4 7 5 temp = 2
2	9	last = 9
3	6	parent : 5 [0]
4	4	5 7 Lchild : 9 [1]
5	9	Rchild : 9 [2]
6	10	Small Child : 3 [1]
7	10	SWAP ()
8	18	temp : 5
9	7	3 Parent = 5 [1]
10	5	2 LC = 6 [3] SC = 4 [4]
		RC = 4 [4]

2ND Deletemin (3)
temp: 3
SWAP ()
temp: 7
last = 8
Parent : 7 [0]
LC : 4 [1]
RC : 9 [2]
SC : 4 [1]

SWAP ()
parent = 5 [4]
LC : 7 [9]
auto SC

SWAP ()
temp:
parent: 7 [1]
LC: 6 [9]
RC: 5 [4]
SC : 5 [4]

SWAP ()
parent = 7 [4]
LC = none so its LEAF

END

END

IDEAS COME FROM THESE

NO:

DATE:

Elem

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

3RD

Deletemin (4)

temp = 4

Parent = 18 [0]

LC = 5 [1]

RC = 9 [2]

SC = 5 [1]

last = 7

SWAP()

temp = 18

Parent = 18 [1]

LC = 6 [3]

RC = 7 [4]

SC = 6 [3]

SWAP ()

parent = 18 [3]

LC = 10 [7]

RC = none

SWAP ()

parent = 18 [7]

LEAF

STOP

* keep deletemin () until last = -1

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

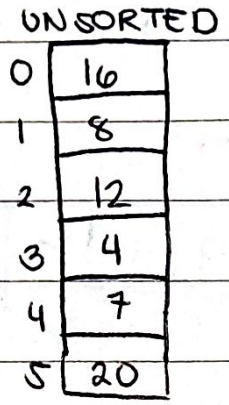
* After heapsort, restore original valre of last using oldLast variable

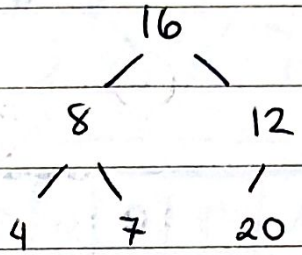
NO:

DATE:

Example 2

unsorted list → P.O.T. heap → sorted

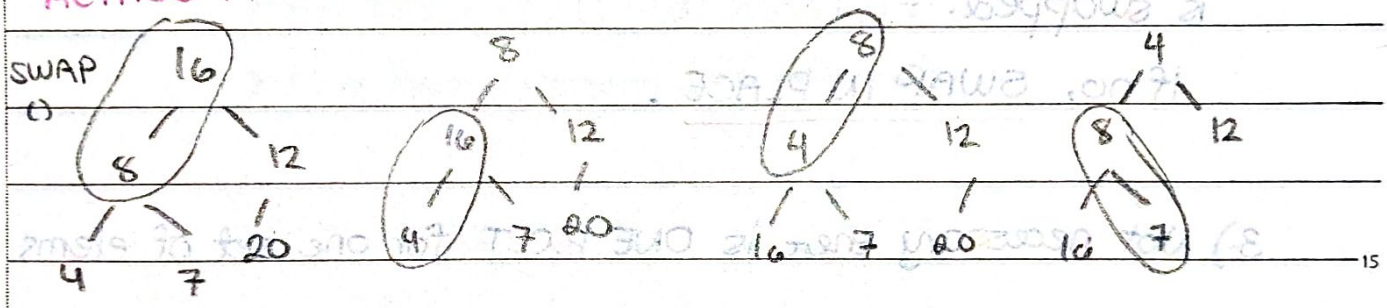


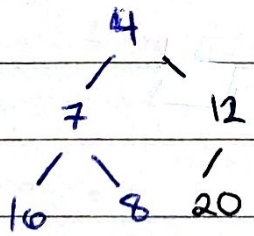


(1.) Check if list satisfies P.O.T property. (METHODS: • Initially empty • Heapify)

Create P.O.T heap in SAME ARRAY.

METHOD 1: INITIALLY EMPTY





FINAL P.O.T.

METHOD 2: HEAPIFY

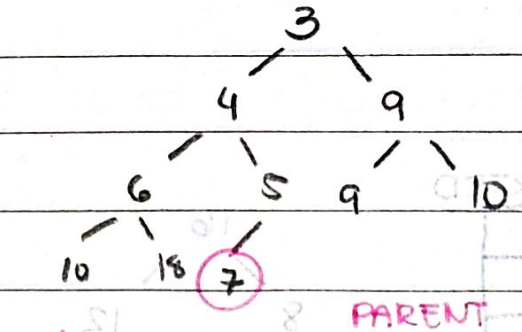
1.) Start at LOWEST LEVEL PARENT.

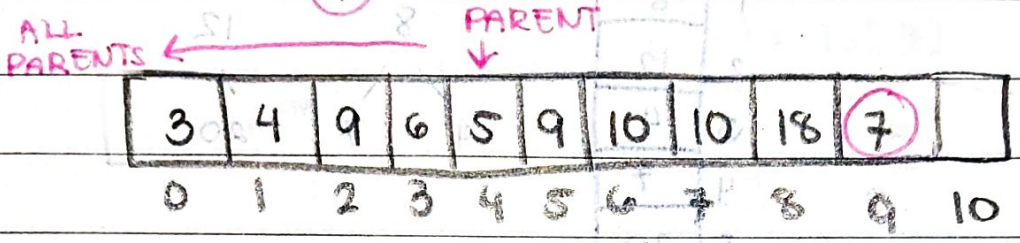
- parent of last elem (smallest)
- once found, every elem prior to parent are ALSO PARENTS

NO:

DATE:

Example





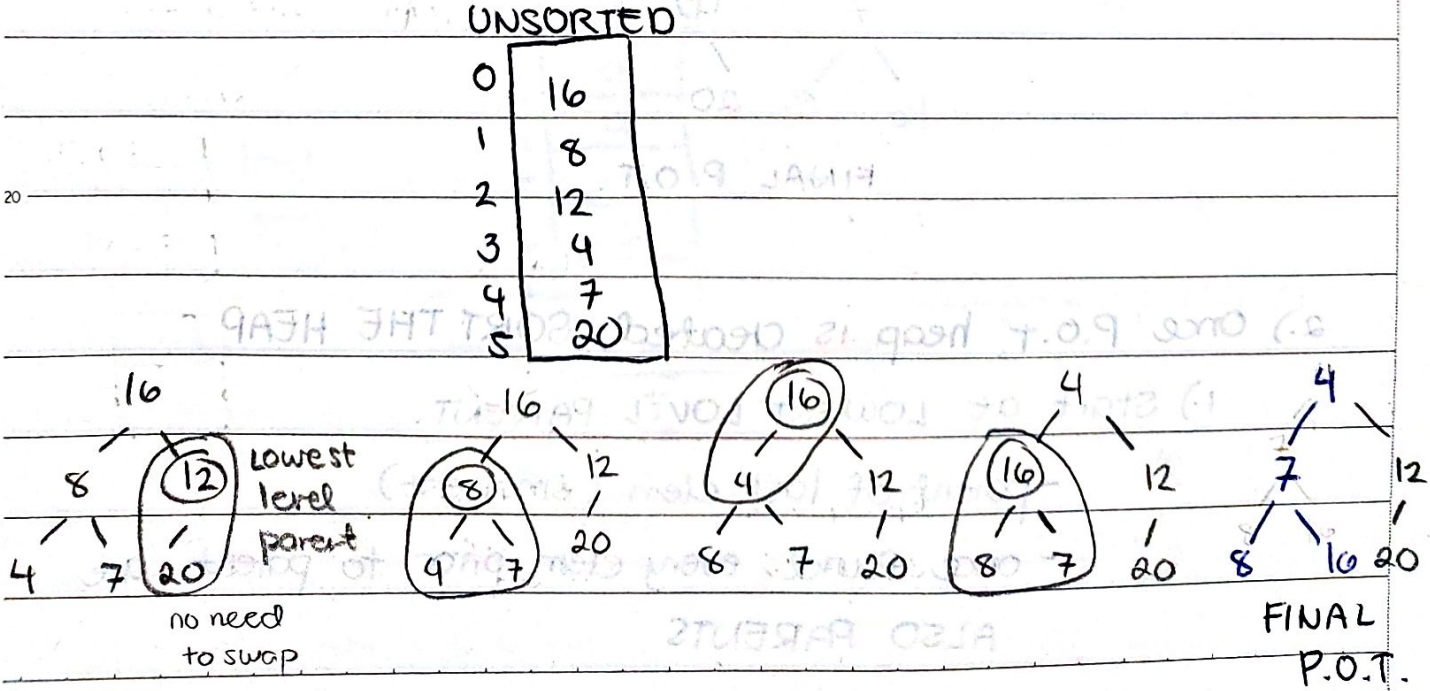
last 9

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.

anyways, HEAPIFY version of creating P.O.T heap:

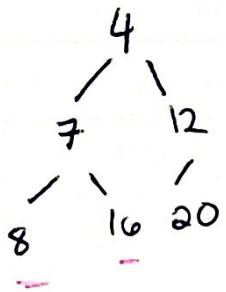


NO:

DATE:

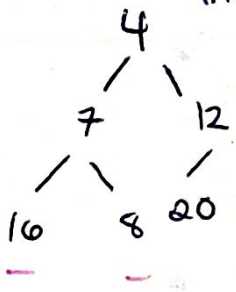
Comparison

POT via Heapify



V.S.

POT via Insert in Initially Empty



* Heapify process is MORE EFFICIENT and shorter.

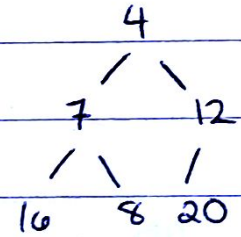
(2.) After P.O.T heap is created, SORT THE HEAP.

> Min heap order (descending)

> Max heap order (ascending)

MIN HEAP - HEAPSORT

(using POT via initially empty)



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

last [5]

1ST

Deletemin (4)

min = 4 last = 4

temp = 4

SWAP

parent = 20 [0]

LC = 7 [1]

RC = 12 [2]

SC = 7 [1]

SWAP ()

temp = 20

parent = 20 [1]

LC = 16 [3]

RC = 8 [4]

SC = 8 [4]

SWAP ()

parent = 20 [4]

LC/RC = —

STOP

2ND

Deletemin (7)

min = 7 last = 3

temp = 7

SWAP

parent = 20 [0]

LC = 8 [1]

RC = 12 [2]

SC = 8 [1]

SWAP ()

parent: 20 [1]

LC = 16 [3]

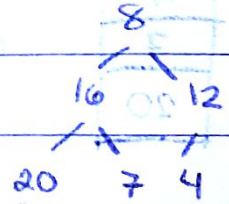
RC = —

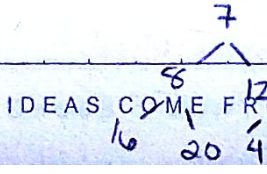
SWAP ()

parent = 20 [3]

LC/RC = —

STOP





NO:

DATE:

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

3RD
Deletemin (8)
min = 8 last = 2
temp = 8
SWAP
parent = 20 [0]
LC = 16 [1]
RC = 12 [2]
SC = 12 [2]
SWAP ()

Parent = 20 [2]
LC/RC = -
STOP

4TH
Deletemin (12)
min = 12
temp = 12 last = 1
SWAP
parent = 20 [0]
LC = 16 [1]
RC = -
SWAP()

parent = 20 [1]
RC/LC = -
STOP

5TH
Deletemin (16)
min = 16 last = 0
temp = 16
SWAP
parent = 20 [0]
LC/RC = -
STOP

RESULT: SORTED MIN HEAP

last
-1 5

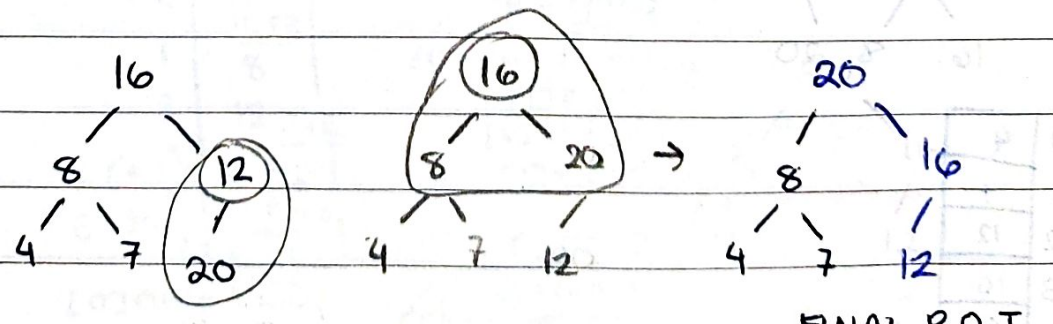
0	20
1	16
2	12
3	8
4	7
5	4

DESCENDING

MAX HEAP - UNSORTED
(using Heapify)

unsorted

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



FINAL P.O.T.

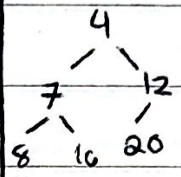
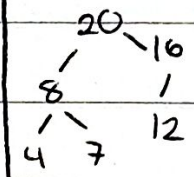
* P.O.T property for Max Heap is
P(parent) >= P(children)

*unsorted bcs i am TIRUD

NO:

DATE:

- SUMMARY -

Type of Heap	Operations	POT Property	Heapify	Root Elem	Heapsort (in place)
Min Heap	• Insert • Deletemin	P(P) <= P(C)		smallest	descending order
Max Heap	• Insert • Deletemax	P(P) >= P(C)		biggest	ascending order

- CODE -

NO:

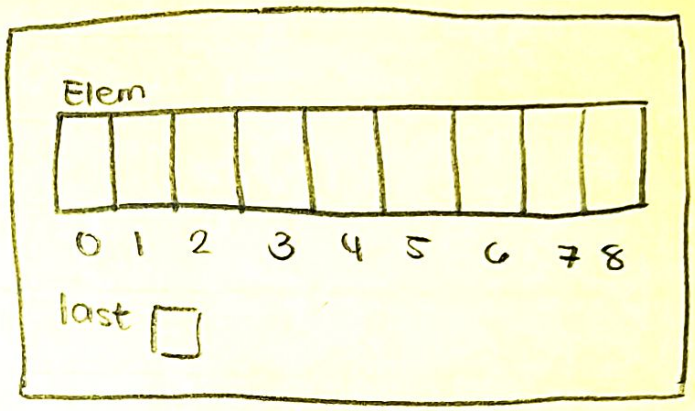
DATE:

//Insert ()

Given List & elem x, write code of fn.
Insert elem in P.O.T heap.

```
#define SIZE 10

typedef struct {
    int Elem[size];
    int lastIndex; //index of last
                   elem
                   //-1 if empty
} List;
```



List (heap)