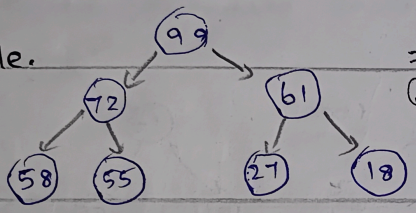
heaps.

example.



⇒ its seems like a binary search tree.

① its a binary search tree

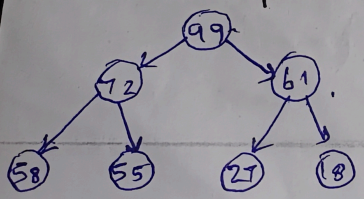
But the numbers are not distributed in the same way with a heap.

highest value always be at the top

key characteristic : is a complete tree

, one of the diff about heap from binary search tree is that have a duplicates.

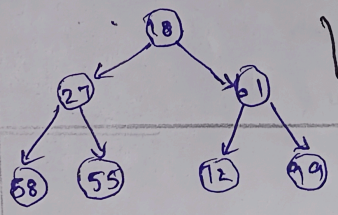
max heap



Root value largest

parent > children.

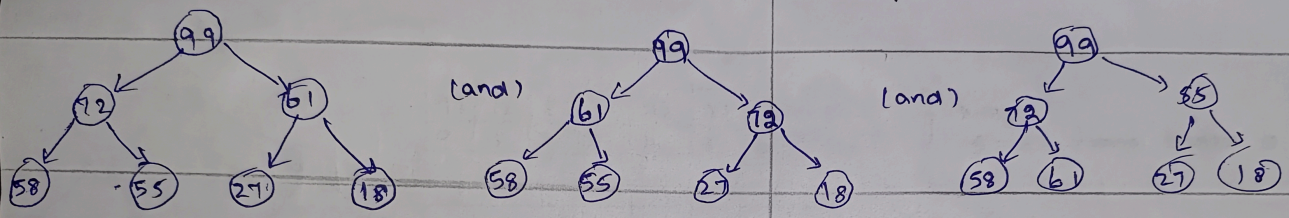
min heap.



Root value smallest

parent < children.

There is no guarantee to the order in a heap.



all are valid max heap there is no particular order other than all of the decendants are going to be less than or equal to item at the top.

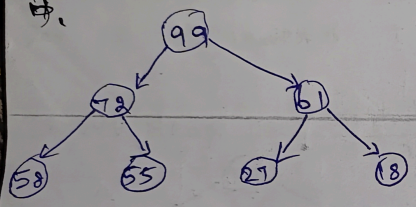
(*). so the heap is not good for searching only thing you use a heap for is being able to keep track of the largest item at the top and able to quickly remove it.

also a huge diff how we store this tree versus how we stored our binary search tree. we store the tree in a list and we will not be creating a node class.

The list only going to store integers.

Storing into list 2 ways,

#.



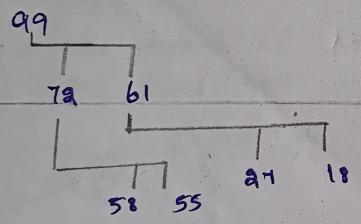
99	72	61	58	55	27	18	
0	1	2	3	4	5	6	7

} way
start from index 0.

X	99	72	61	58	55	27	18
0	1	2	3	4	5	6	7

} way 2
start from index 1
easier to do Calculations.

math is looks easier and get the concept easier.



think a tree like this,
it go in a list like this

1. find child for 99 => left_child = 2 * parent_index
   2 * 1 => 2 => list[2] => 72.
   right = 2 * parent_index + 1 => => 72 61

2). find child for 72 =>

X	99	72	61	58	55	27	18
0	1	2	3	4	5	6	7

⋮

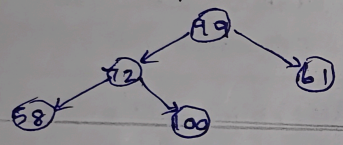
want to find a parent for 27 and 18.

to do interger division, divide the number by 2, so.

for 27 the index is 6 => 6/2 => 3 => list[3] => 61

for 18 the index is 9 => 7/2 => 3.5 => 3 => list[3] => 61

Insert Heap. here we adding 100 to the tree.

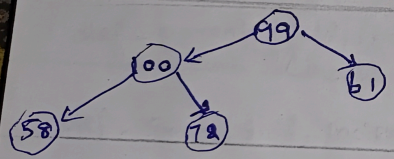


=> while loop check until it's

x	99	72	61	58	~~100~~ 100		
0	1	2	3	4	5	6	7

1. now find the parent of 100 => 5/2 => 2 => List[2] => 72 check the 2 if child > parent then swap here child 100 > parent 72 so swap.

now its looks-



x	99	100	61	58	72		
0	1	2	3	4	5	6	7

1, now find parent of 100 => index of 100 is 2 => 2/2 => 1 => list[1] => 99.

100 > 99 then swap. we reached the top so now its a valid heap.

heap helper methods.

make code cleaner when write insert and remove methods.

* one diff is first we store in a list start from index 1, here start from 0.

```
class MaxHeap:
    def __init__(self):
        self.heap = []

    def _left_child(self, index):
        return 2 * index + 1
    def _right_child(self, index):
        return 2 * index + 2

    def _parent(self, index):
        return (index - 1) // 2

    def _swap(self, index1, index2):
        self.heap[index1], self.heap[index2] = self.heap[index2], self.heap[index1]
```

we start from index 0,
if one it bolcs 2 * index.

to get left child index

to get right child index

to get parent index

swap the 2 index values,

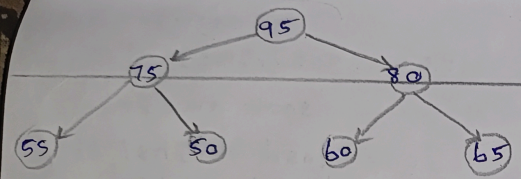
```
def insert(self, value):
    self.heap.append(value)
    current = len(self.heap) - 1
    while current > 0 and self.heap[current] > self.heap[self._parent(current)]:
        self._swap(current, self._parent(current))
        current = self._parent(current)
```

remove: There is only one item can ever remove that at the top doesn't matter its min or max heap ⊗ once we remove the item we don't have a valid heap, The hard part of the remove method is figuring out how to rearrage the tree so its once again a vaild tree

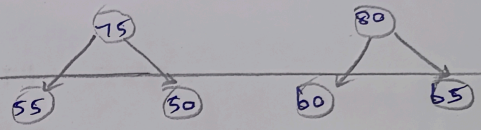
insert (or) remove first step make it complete in one step.

in insert we put a value in bottom and bubbled it up to the appropriate spot.

in remove same as opposite, put something at the top and sink it down to its appropriate spot.



→ after removing the top

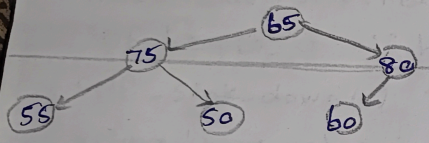


[95, 75, 80, 55, 50, 60, 65]
0 1 2 3 4 5 6

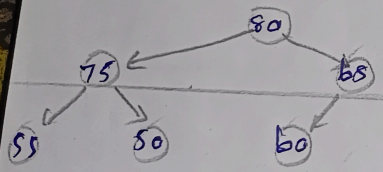
to complete this

[, 75, 80, 55, 50, 60, 65]
0 1 2 3 4 5 6

put the last item to top, sink it down after we need to rearrange appropriate spot.



the way sink down take the top 65 compare with each of the children which children value is high swap with that



[80, 75, 65, 55, 50, 60]
0 1 2 3 4 5

nex step 65 > 60 the parent is > the child so stop right there.

```
def remove(self):
    if len(self.heap) == 0:
        return None
    if len(self.heap) == 1:
        return self.heap.pop()

    max_value = self.heap[0]
    self.heap[0] = self.heap.pop()
    self._sink_down(0)
    return max_value.
```

→ is empty None.

→ if one pop the item and return it.

→ get the first as max_value.
make the first index value as last one and remove the last. and return the maxed removed value.

using this sink_down method need to rearrange the heap.

```
def _sinkdown (self, index):
    max_index = index
    while True:
        left_index = self._left_child( index)
        righ_index = self._right_child(index)
    if (left_index < len(self.heap) and self.heap[left_index] > self.heap[max_
                                                                    index]):
        max_index = left_index
    if (right_index < len(self.heap) and self.heap[right_index] > self.heap[max_index]
                                                                    ):
        max_index = right_index.
    if max_index != index :
        self._swap(index, max_index)
        index = max_index
    else :
        return,
```

heap priority queues. & Big O.

if we say that the highest values was the highest priority ~~the~~ and want always remove the highest value from the queue then heap is really a great way. (because highest value alway on top).

Big O:

insert } O(log n). (very efficient).
remove