LET’S GO TO WORK: Build a minheap

MinHeap

```python
from typing import TypeVar, List

T = TypeVar('T')


class MinHeap:

    def __init__(self):
        """
        Initializes the priority heap
        """
        self.data = []

    def __len__(self) -> int:
        """
        Length override function
        :return: Length of the  heap
        """
        return len(self.data)

    def _swap(self, i, j):
        """Swap the elements at indices i and j of array."""
        self.data[i], self.data[j] = self.data[j], self.data[i]

    def empty(self) -> bool:
        """
        Checks if the heap is empty
        :return: True if heap is empty, False otherwise
        """
        return len(self) == 0
```

top, left_child, right_child

```python
    def top(self) -> T:
        """
        Gets the heap root value
        :return: None if heap is empty, the root value otherwise
        """
        if not self.empty():
            return self.data[0]

    def left_child_index(self, index: int) -> int:
        """
        Returns the index of the left child of the node at index
        :param index: index to find the left child index of
        """
        if 2 * index + 1 < len(self.data):
            return 2 * index + 1

    def right_child_index(self, index: int) -> int:
        """
        Returns the index of the right child of the node at index
        :param index: index to find the right child index of
        """
        if 2 * index + 2 < len(self.data):
            return 2 * index + 2
```

parent
min_child_index

```python
def parent_index(self, index: int) -> int:
    """
    Returns the index of the parent of the node at index
    :param index: index to find the parent of
    """
    parent = (index - 1) // 2
    if parent >= 0:
        return parent
```

```python
def min_child_index(self, index: int) -> int:
    """
    Finds the minimum child at the specified index
    :param index: the index of the node where the minimum child needs to be found
    :return: The minimum child, or None if there are no children
    """
    left_i = self.left_child_index(index)
    right_i = self.right_child_index(index)
    if right_i is not None and left_i is not None:  # FULL  node
        if self.data[left_i] < self.data[right_i]:
            return left_i
        else:
            return right_i
    elif right_i is None and left_i is None:  # node without children
        return
    return left_i  # must be a complete tree right??? it can not have right if it does not have a left...
```

MinHeap

```
def percolate_up(self, index: int) -> None:
"""
Moves a node up the heap to its desired position
:param index: The index of the node to be
percolated up
"""
pass
```

You have the full implementation of perc_down and perc_up in lecture source code, MAKE IT WORK FOR THIS!

```
def _upheap(self, j):
    parent = self._parent(j)
    if j > 0 and self._data[j] < self._data[parent]:
        self._data[j], self._data[parent] = self._data[parent], self._data[j]
        self._upheap(parent)
```

MinHeap

```
def percolate_down(self, index: int) -> None:
"""
Moves a node down the heap to its desired position
:param index: The index of the node to be
percolated down
"""
pass
```

You have the full implementation of perc_down and perc_up in lecture source code, MAKE IT WORK FOR THIS!

```
def _downheap(self, j):
  if self._has_left(j):
    left = self._left(j)
    small_child = left              # although right may be smaller
    if self._has_right(j):
      right = self._right(j)
      if self._data[right] < self._data[left]:
        small_child = right
    if self._data[small_child] < self._data[j]:
      self._swap(j, small_child)
      self._downheap(small_child)    # recur at position of small child
```

MinHeap

You have the full implementation of add and remove in lecture source code
MAKE IT WORK FOR THIS!

```
def add(self, key: T) -> None:
"""
Creates a node and adds a new element to the heap
:param key: key of the added node
:param val: value of the new node
"""
pass
```

```
def add(self, key, value):
    """Add a key-value pair to the priority queue."""
    self._data.append(self._Item(key, value))
    self._upheap(len(self._data) - 1)            # upheap newly added position
```

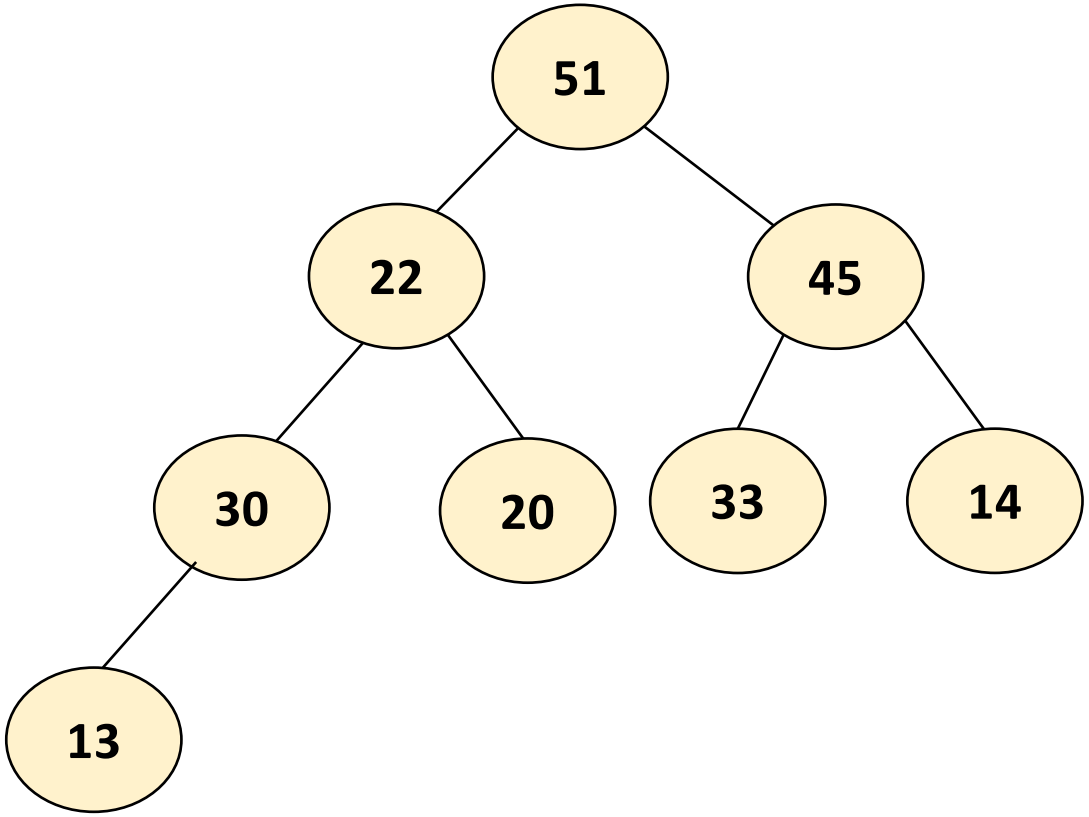
```
def remove(self) -> T:
"""
Removes the smallest element from the heap
:return: the root of the heap
"""
pass
```

```
def remove_min(self):
  """Remove and return (k,v) tuple with minimum key.

  Raise Empty exception if empty.
  """
  if self.is_empty():
    raise Empty('Priority queue is empty.')
  self._swap(0, len(self._data) - 1)          # put minimum item at the end
  item = self._data.pop()                     # and remove it from the list;
  self._downheap(0)                           # then fix new root
  return (item._key, item._value)
```

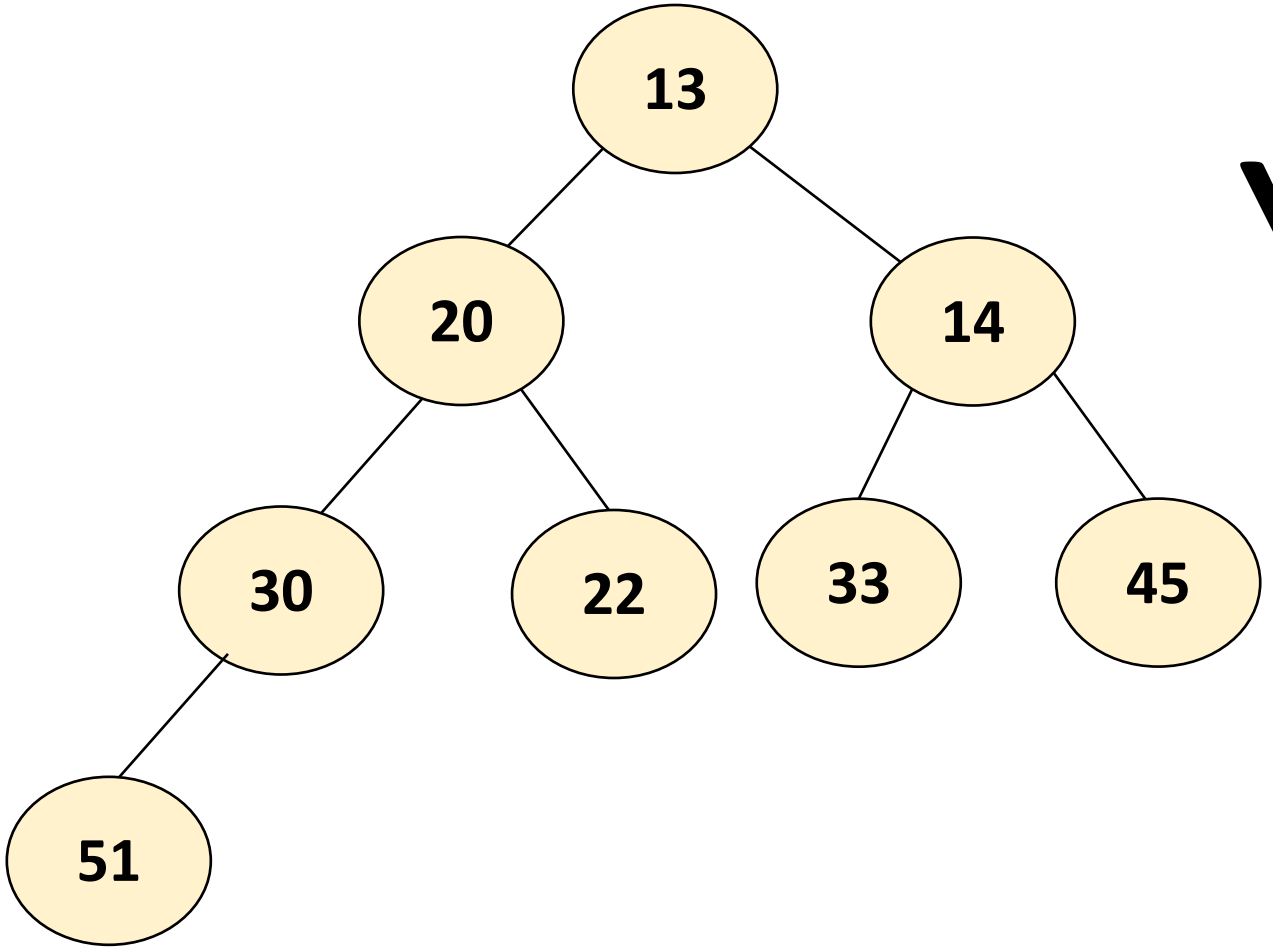
Application Problem: Is this a “Min Heap”?

Application problem: is this a Min Heap?



No!

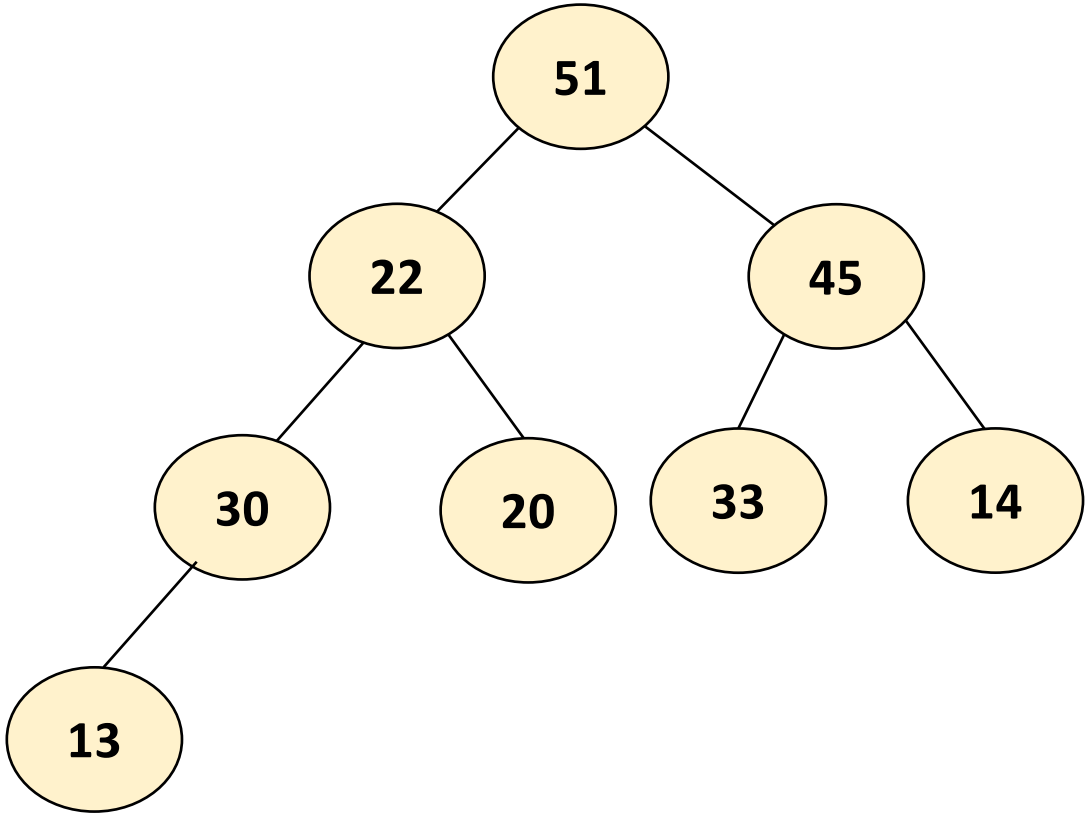
Application problem: is this a Min Heap?



Yes!

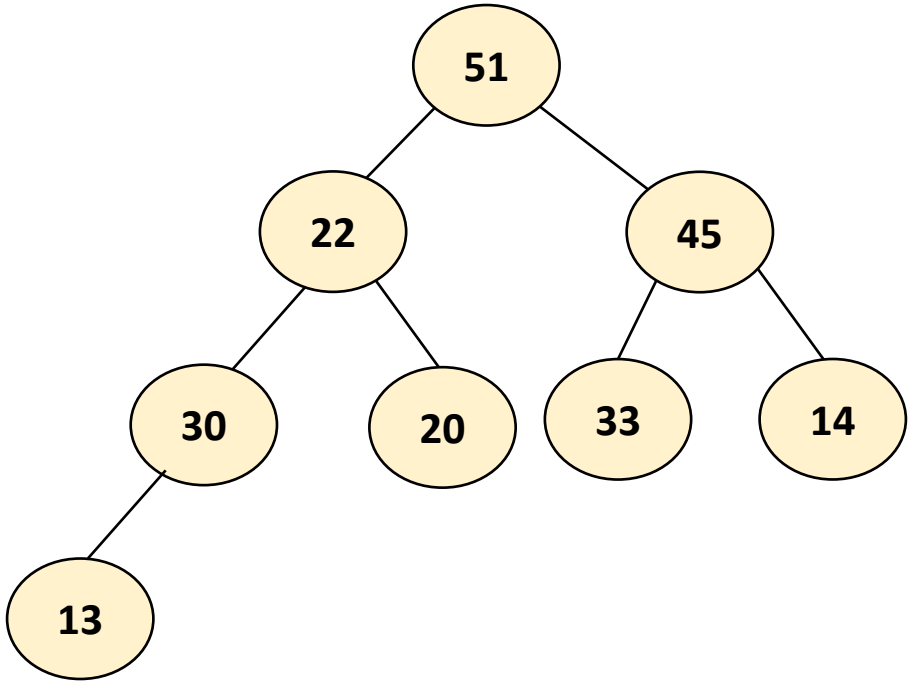
Let’s work on an example.
Imagine we have the following array

51	22	45	30	20	33	14	13



Write the build heap function to turn this into a minHeap

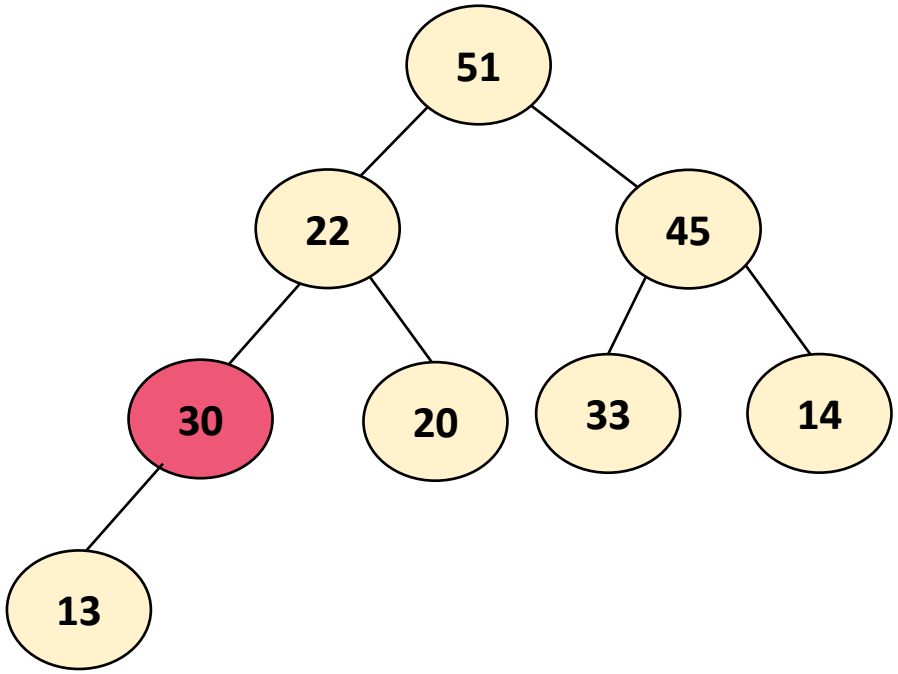
51	22	45	30	20	33	14	13




```
def build_heap(self, array):
    # consider using percolate down…
    return array
```

Expected behavior of build_heap method:

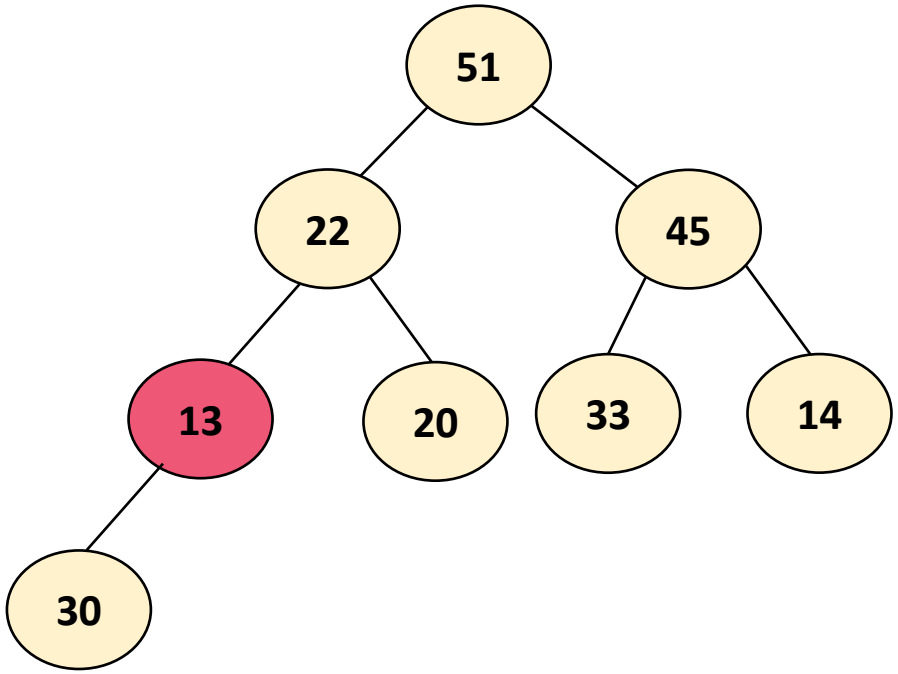
51	22	45	30	20	33	14	13




```
def build_heap(self, array):
    # consider using percolate down…
    return array
```

Expected behavior of build_heap method:

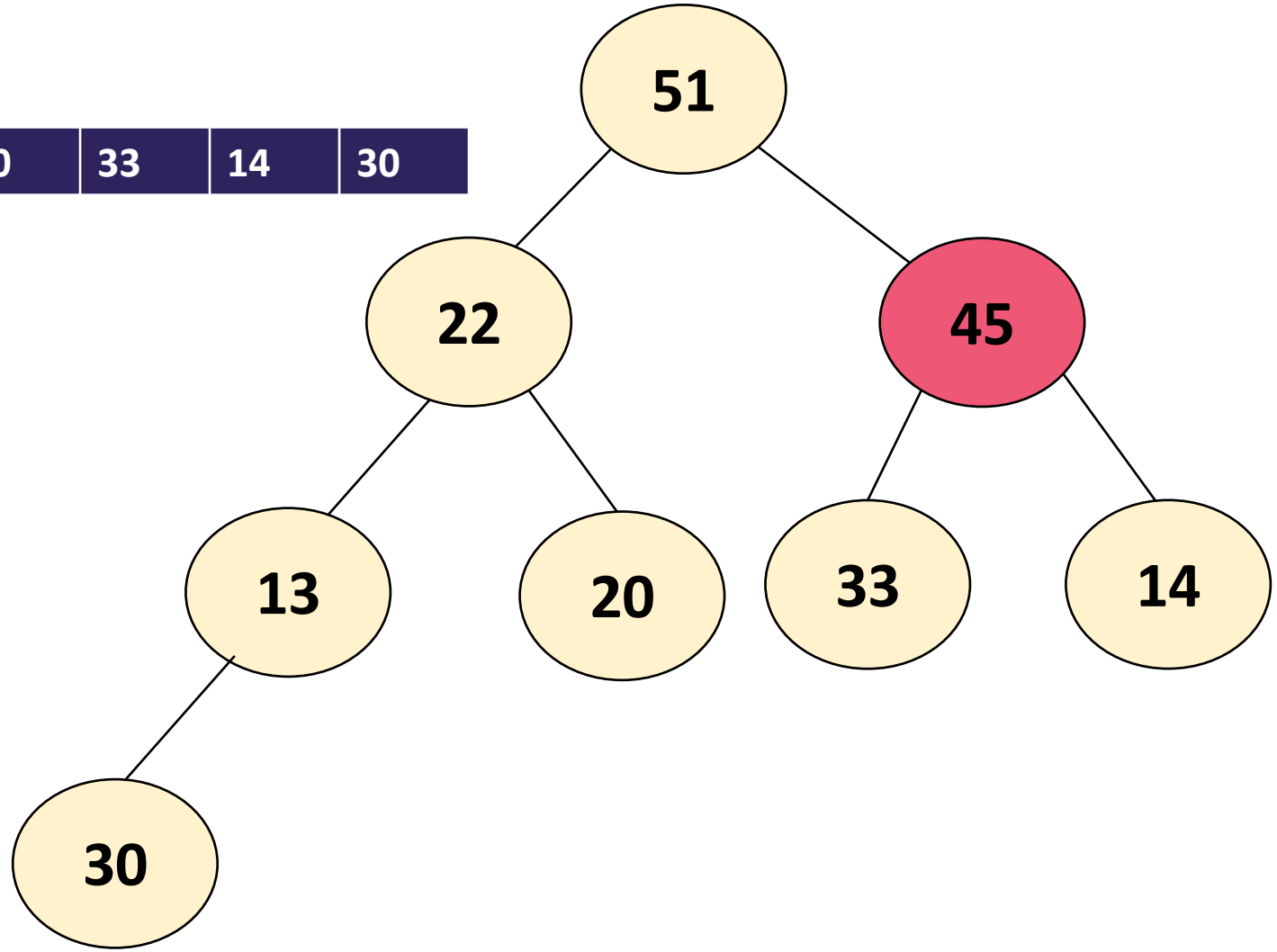
51	22	45	13	20	33	14	30




```
def build_heap(self, array):
    # consider using percolate down...
    return array
```

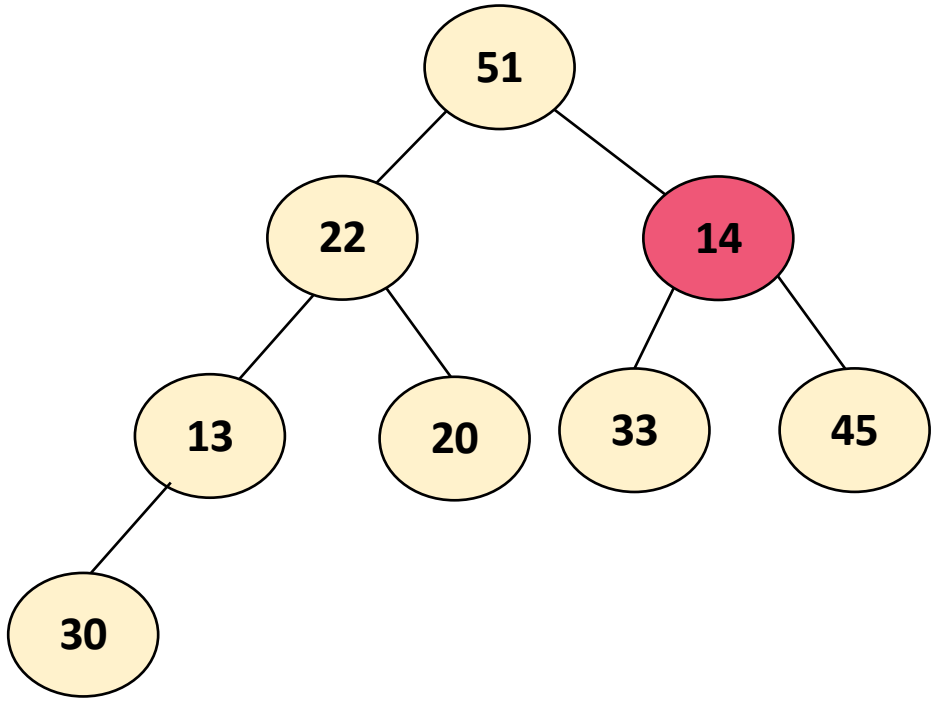
Expected behavior of build_heap method:

51	22	45	13	20	33	14	30



Expected behavior of build_heap method:

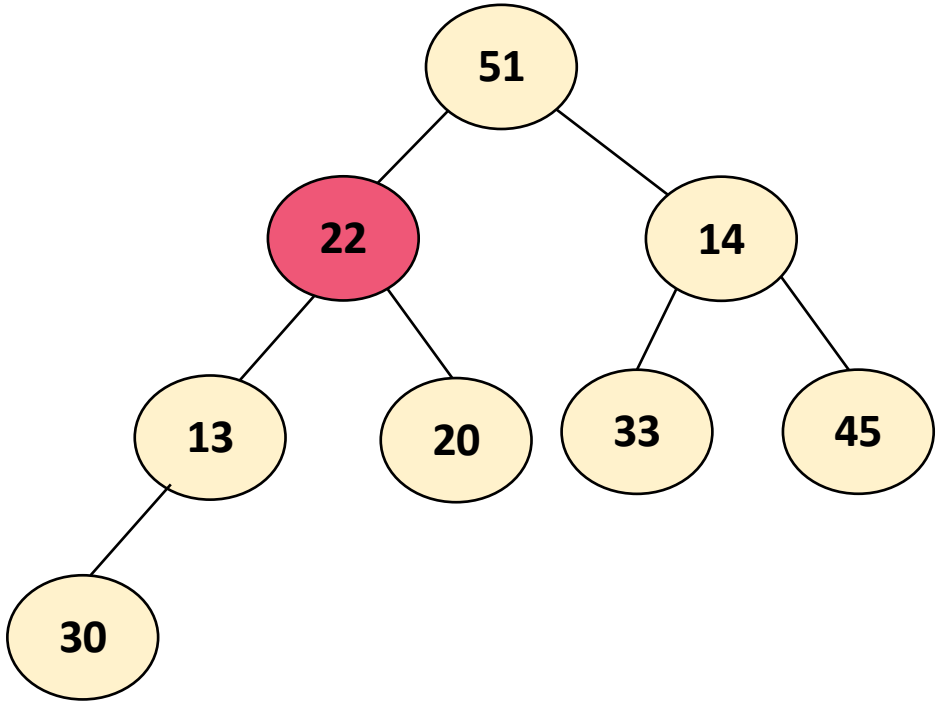
51	22	14	13	20	33	45	30




```
def build_heap(self, array):
    # consider using percolate down…
    return array
```

Expected behavior of build_heap method:

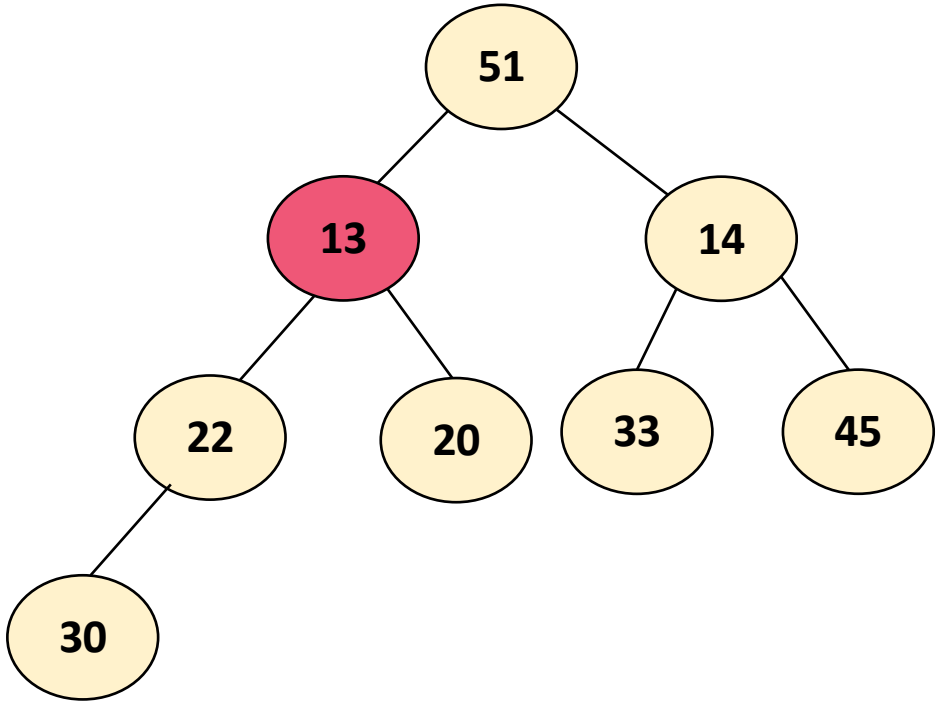
51	22	14	13	20	33	45	30




```
def build_heap(self, array):
    # consider using percolate down…
    return array
```

Expected behavior of build_heap method:

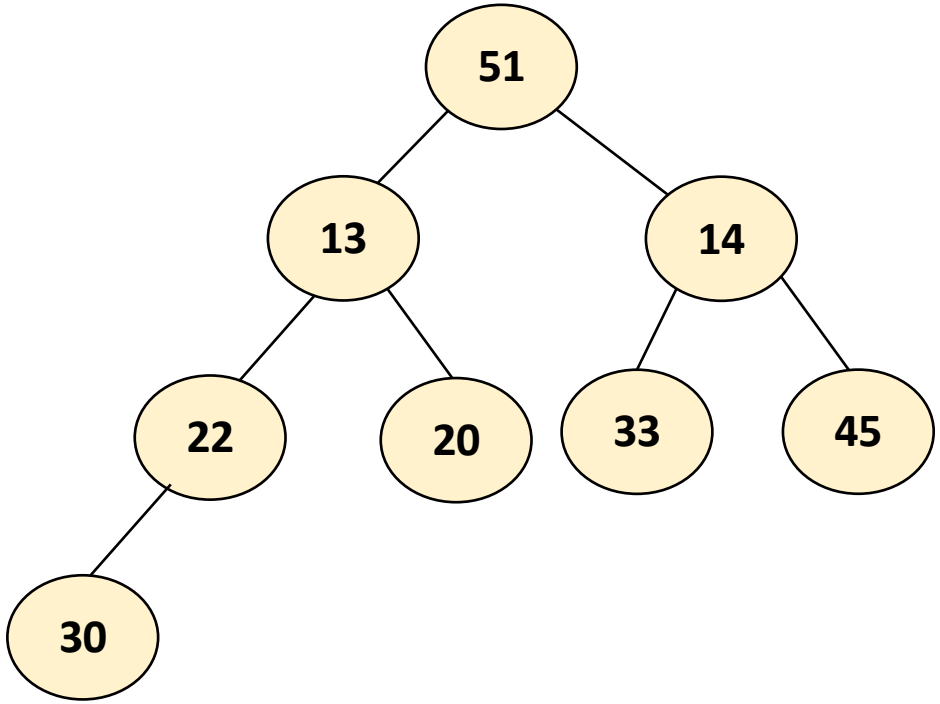
51	13	14	22	20	33	45	30




```
def build_heap(self, array):
    # consider using percolate down…
    return array
```

Expected behavior of build_heap method:

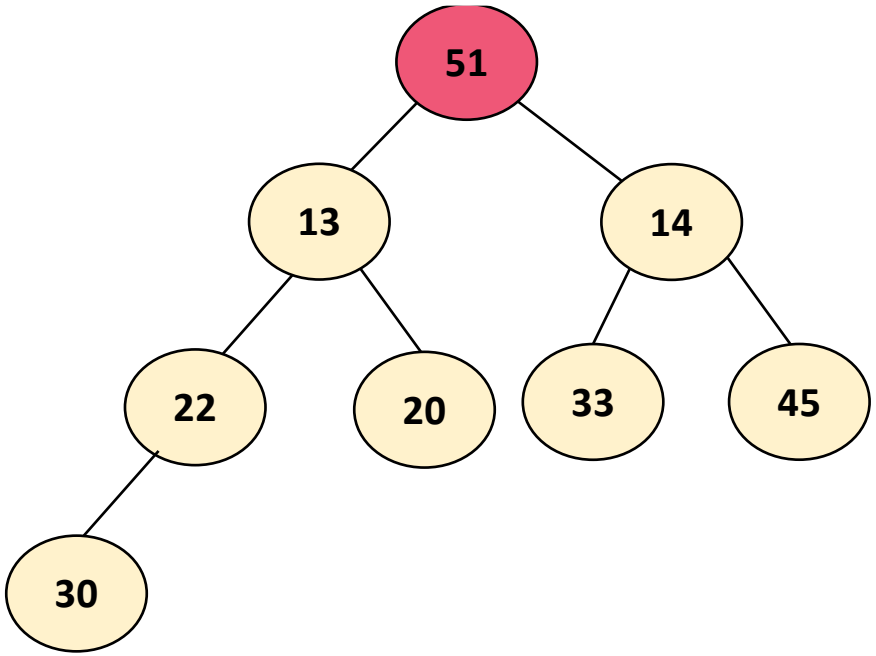
51	13	14	22	20	33	45	30




```
def build_heap(self, array):
    # consider using percolate down…
    return array
```

Last step…

51	13	14	22	20	33	45	30

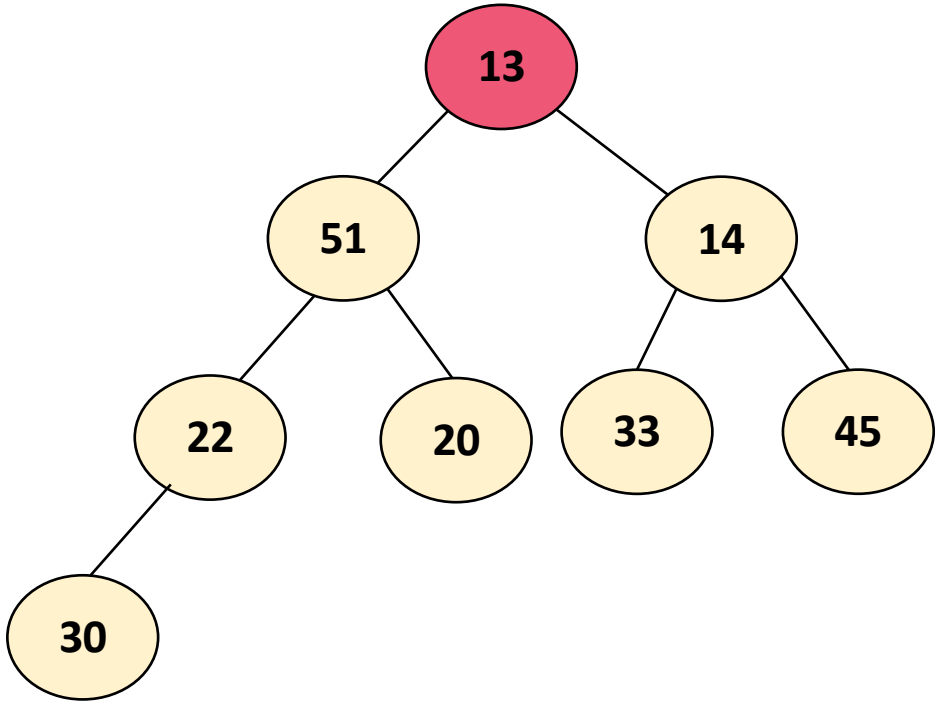



```
def build_heap(self, array):
    # consider using percolate down…

    return array
```

And that is it!!! No wait ???? What about 51 ???, percolate it down…

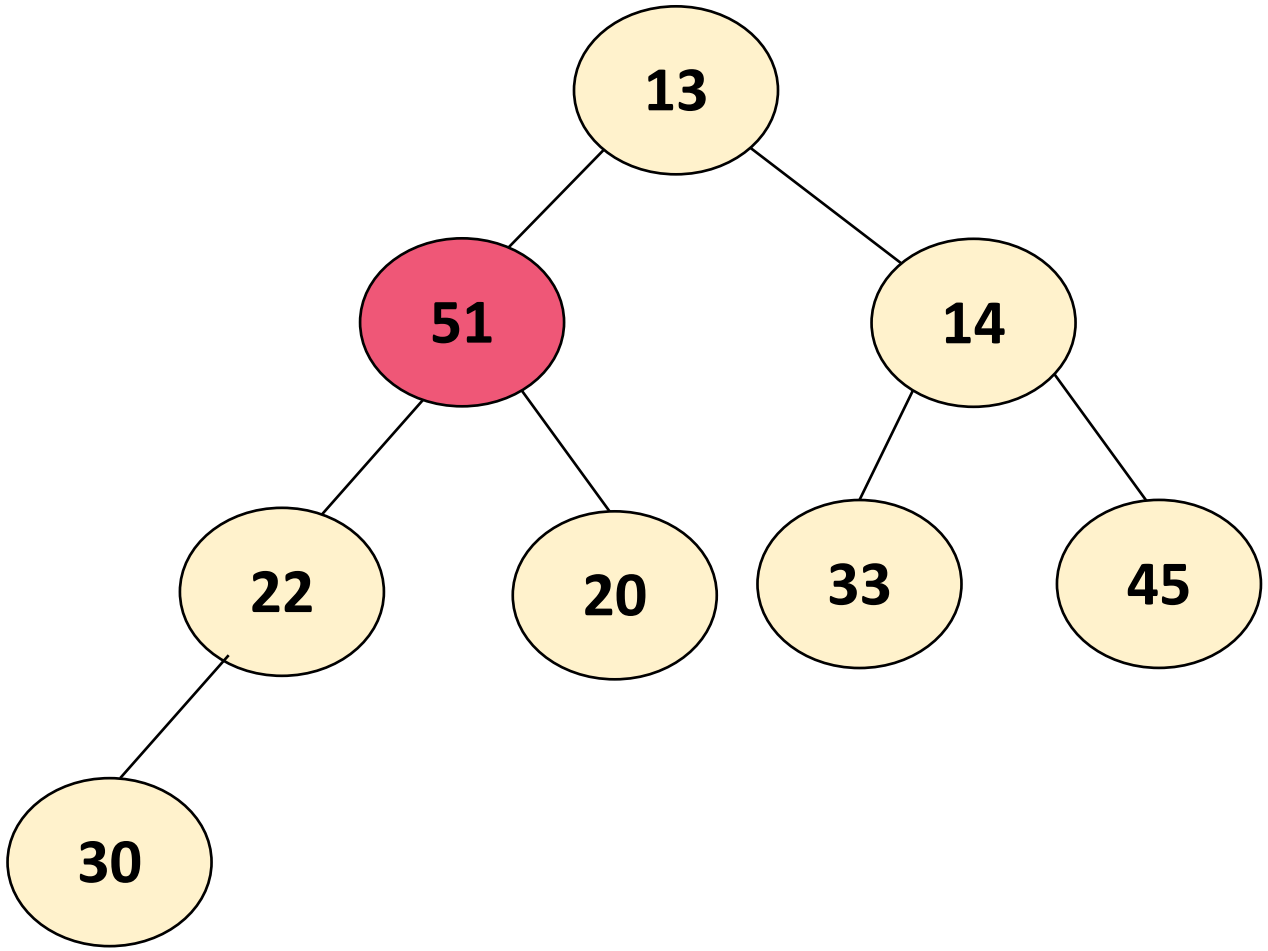
13	51	14	22	20	33	45	30




```
def build_heap(self, array):
    # consider using percolate down…
    return array
```

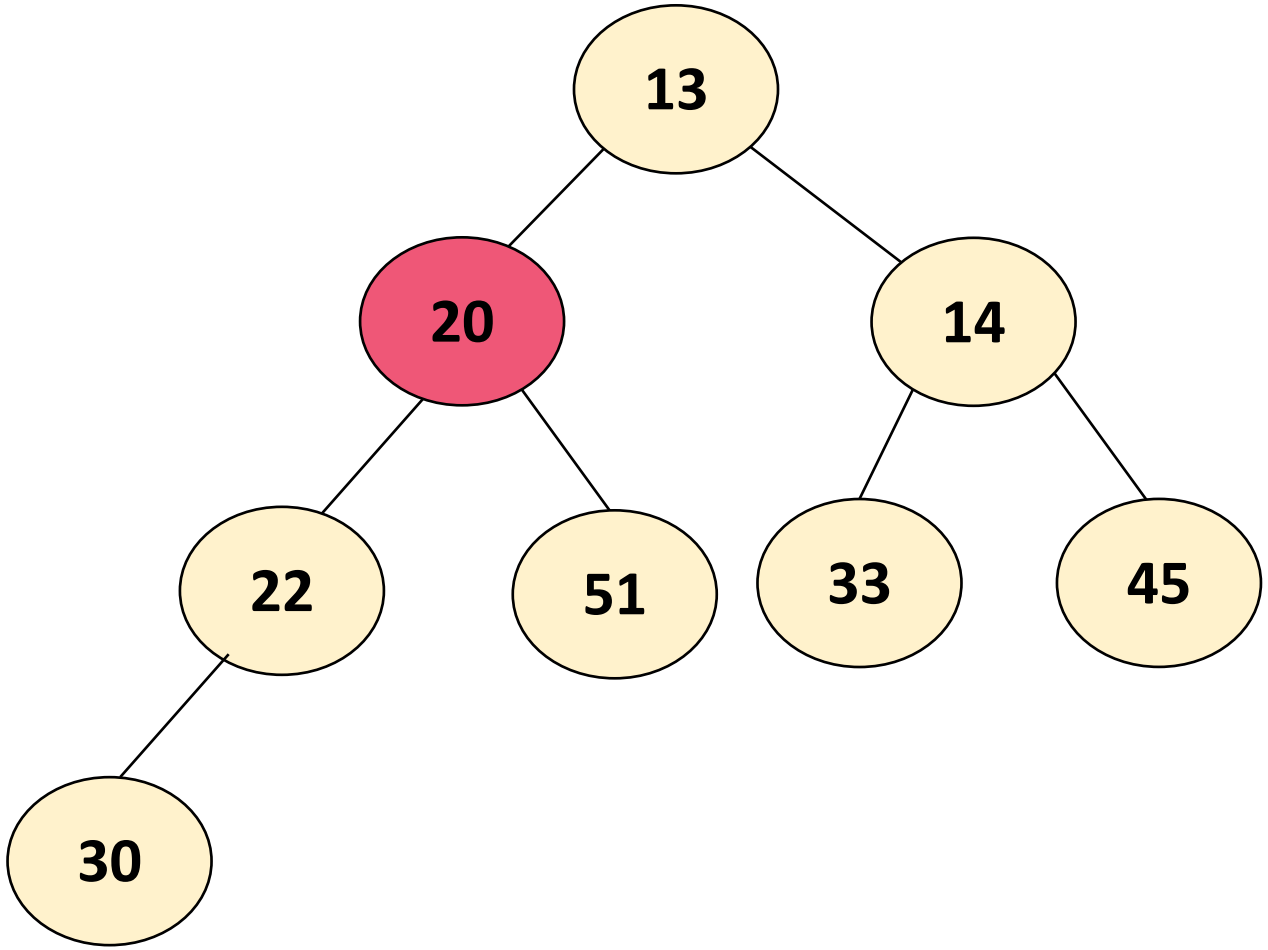
```
def build_heap(self, array):
    # consider using percolate down…
    return array
```

13	51	14	22	20	33	45	30



```
def build_heap(self, array):
    # consider using percolate down…
    return array
```

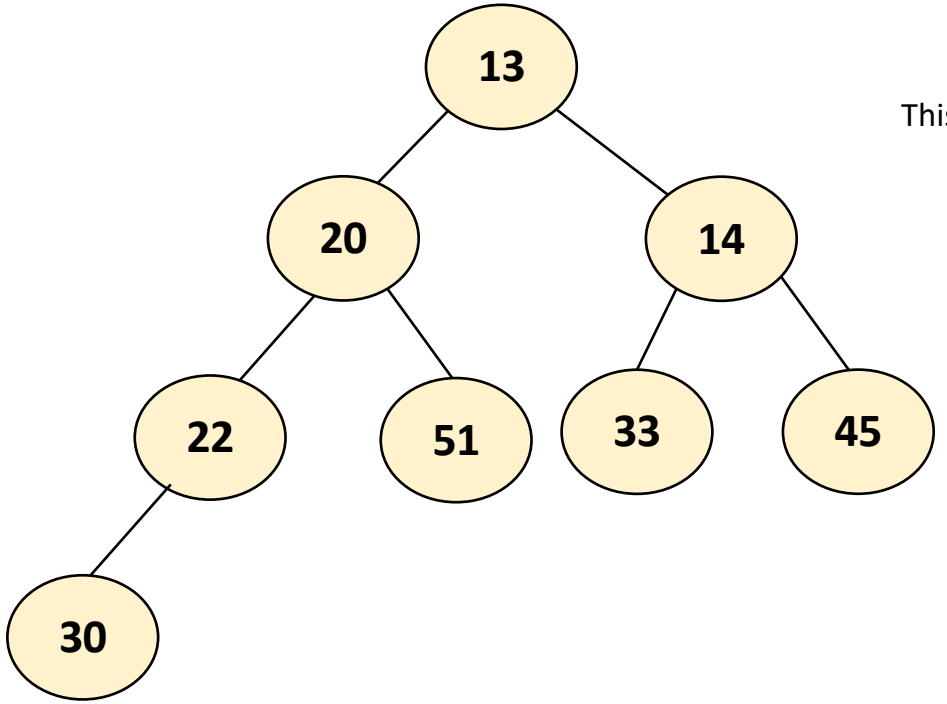
13	20	14	22	51	33	45	30



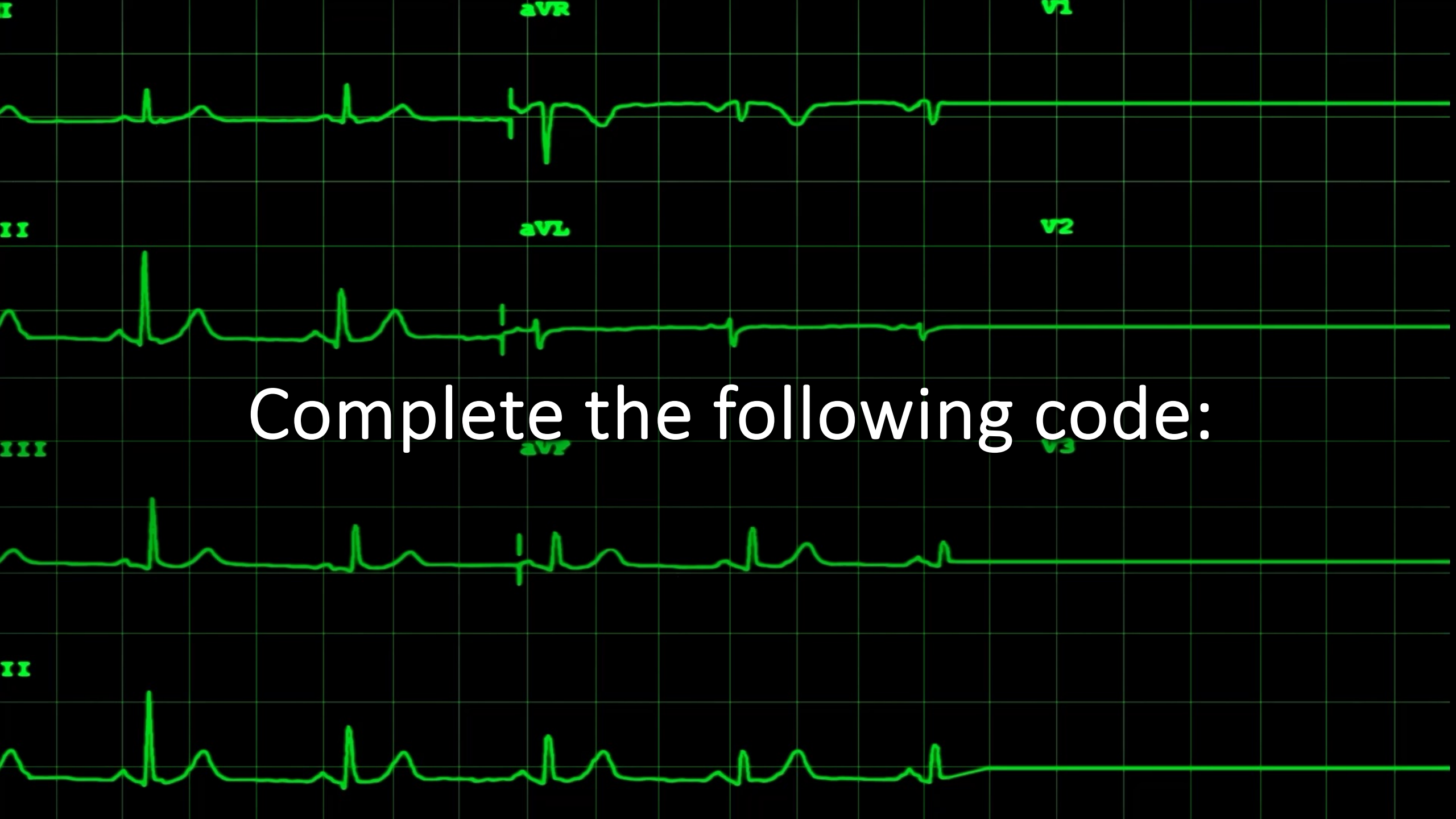
Now it is done !! It is a Min Heap ☺

```
def build_heap(self, array):
    # consider using percolate down…
    return array
```

13	20	14	22	51	33	45	30



This is a min heap!



Complete the following code:

Complete the following code

```
def build_heap(self):
    pass

def is_min_heap(self):
    pass
```