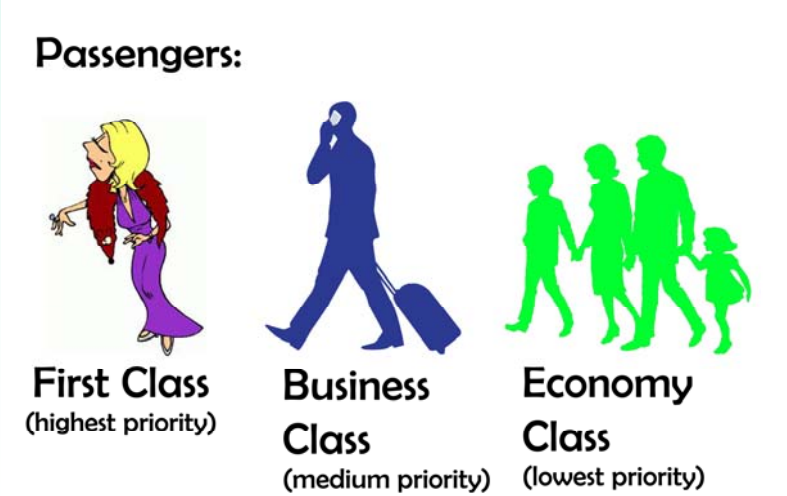
CompSci 105
Lecture 33 Content

Heaps – using a binary tree structure to store a priority queue

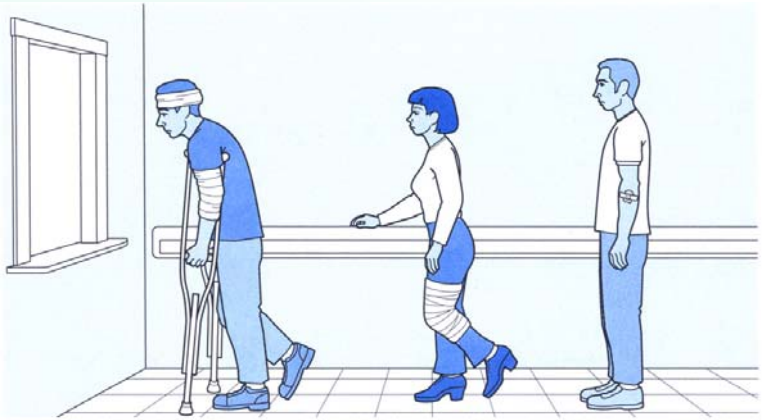
Textbook: Chapter 6



Priority Queues - ADT

A priority queue is a 'queue' in which each item has a priority and items with the highest priority are removed before those with lower priority irrespective of when they are added to the queue.

In this discussion we assume that each item has a unique priority.



Priority Queues - ADT

The things we want to do with a priority queue are:

- Op 1: create the data structure
- Op 2: add an items
- Op 3: remove the item with the highest priority
- Op 4: get the size
- Op 5: find out if the structure is empty

We can implement this structure using a **sorted** list:

Op 1: O(1) Op 4: O(1)

Op 2: **O(n)** Op 5: O(1)

Op 3: O(1)

We can do better than this!

We can implement this structure using an **unsorted** list:

Op 1: O(1) Op 4: O(1)

Op 2: O(1) Op 5: O(1)

Op 3: **O(n)**

Priority Queue – using a binary heap

We will implement an efficient priority queue using a so-called binary heap - a complete binary tree, which can be stored in a list.

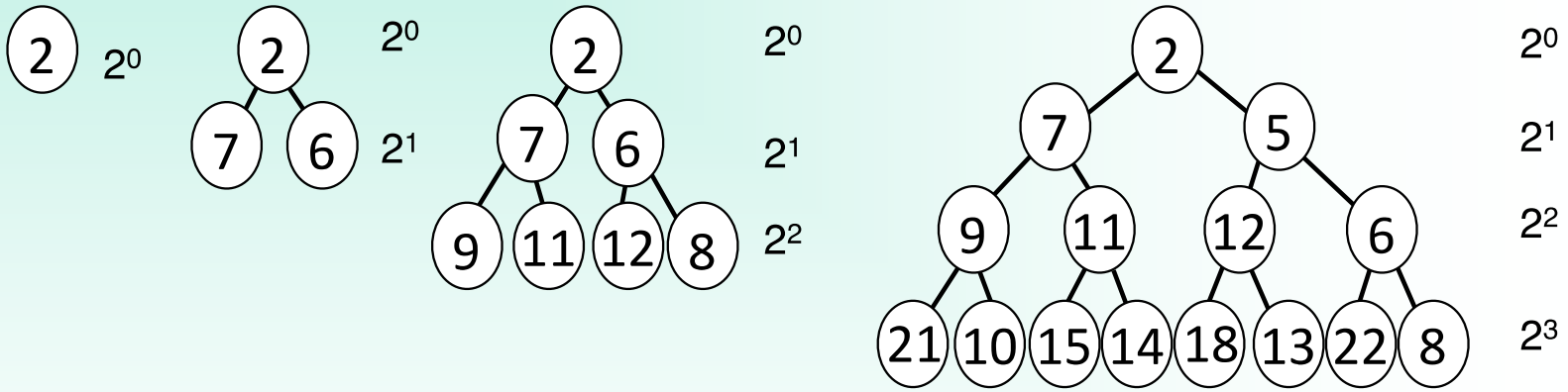
For simplicity, in these examples (and in the text) the heap only contains the priority number (there is no attached item – the payload).

```
class BinHeap:
    def __init__(self):
        …
def main():
    heap = BinHeap()
    heap.insert(5)
    heap.insert(7)
    heap.insert(3)
    heap.insert(11)

    print(heap.del_min())
    print(heap.del_min())
    print(heap.del_min())
    print(heap.del_min())
main()
```

```
3
5
7
11
```

Full Binary Tree

A binary tree is **full** if all its leaves are on the same level.
The number of nodes in level *k* of a full binary tree is 2^k

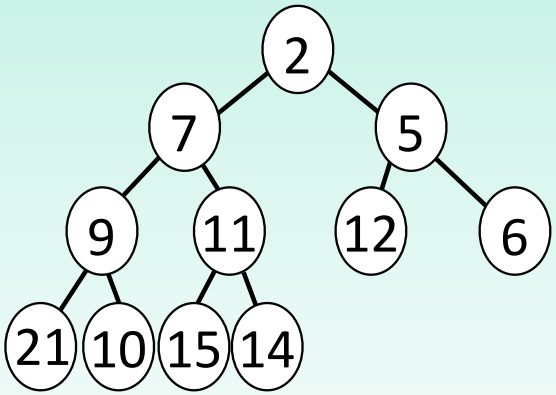


QUESTION:

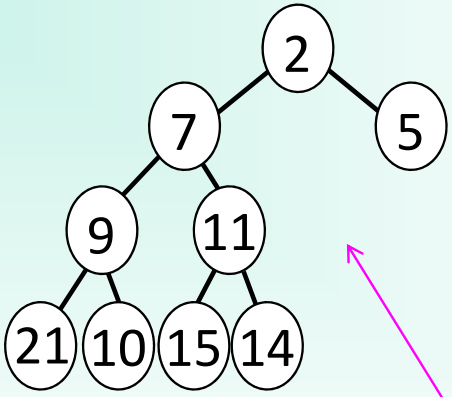
How many nodes does a full binary tree of height *h* have? $2^{h+1} - 1$

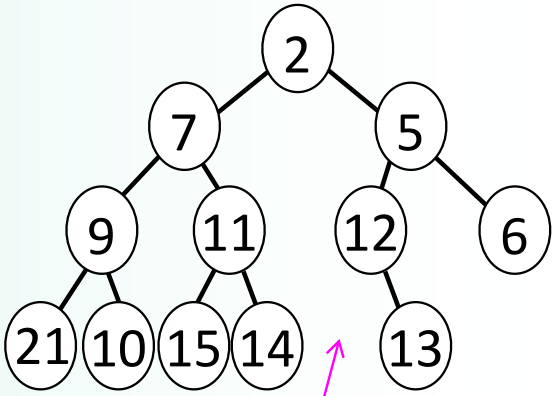
Complete Binary Tree

A **complete** binary tree has all levels full except the last one.
The last level is filled from the left.



Complete



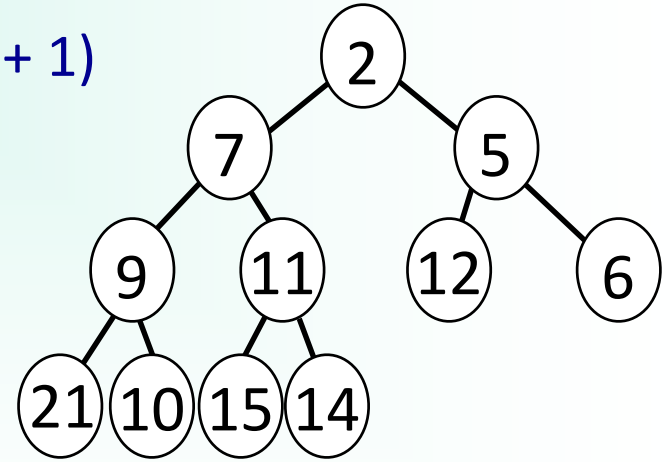


Not complete

Complete Binary Tree

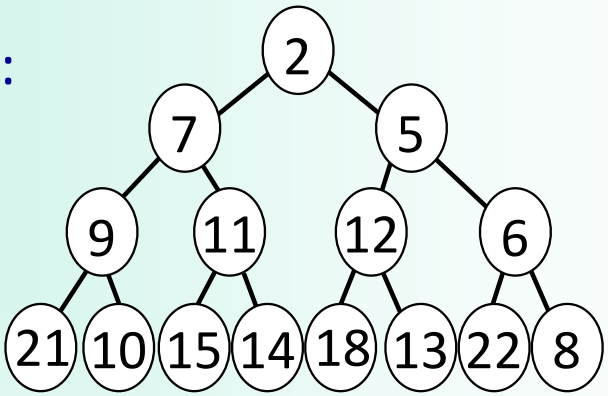
Things to note about complete binary trees:

- they are balanced (i.e. the height of the left and right subtree differs by at most 1)
- half the nodes are leaves (or half + 1)
- the left subtree always has more or the same number of nodes as the right subtree.



Complete Binary Tree

A complete binary tree is efficiently stored using a list:



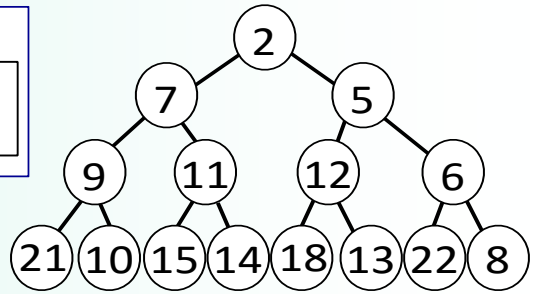
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	7	5	9	11	12	6	21	10	15	14	18	13	22	8

For convenience we are going to leave the first element blank and store the root element in position 1:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	2	7	5	9	11	12	6	21	10	15	14	18	13	22	8

Complete Binary Tree

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	2	7	5	9	11	12	6	21	10	15	14	18	13	22	8



QUESTIONS

What indices are the children of node at index 6 in the list?
What indices are the children of node at index i in the list?
What is the index of the parent of the node at index 6?

Children of node L[i] are L[2i] and L[2i+1]
Parent of node L[i] is L[i // 2]

Priority Queue – using a binary heap

We can improve on the performance of the sorted/unsorted list implementation of a priority queue (see slide 3) by using a binary heap.

A binary heap can be implemented using a complete binary tree. This means that the elements of the heap implementation can be stored using a Python list.

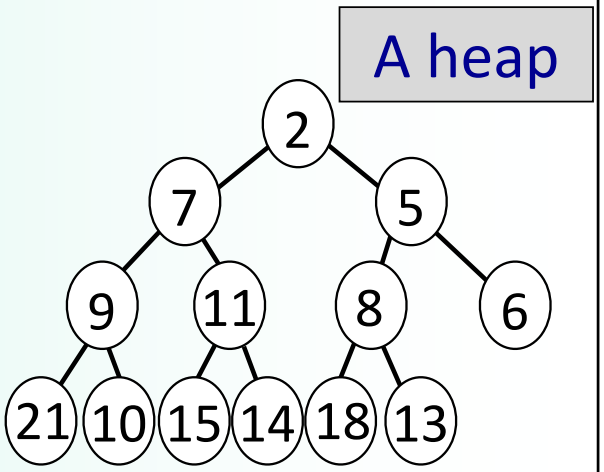
No links needed to store the tree

☺

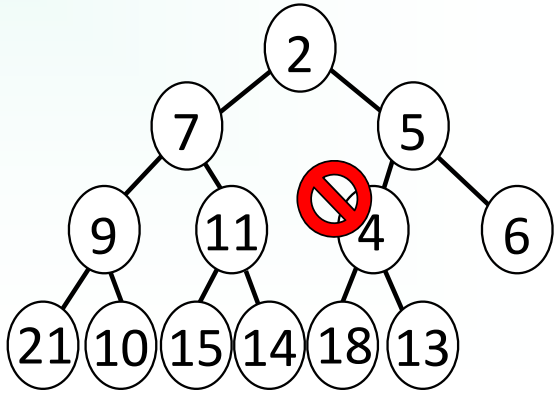
Heap Properties

Below right is an example of a min heap structure:

The heap property: in a min heap, the parent is always smaller than or equal (we are using unique priorities in these example) to both its children.



Note: we are studying min heaps, but max heaps where the parent is always greater than both children can also be used if the highest priority is the larger priority number.



Binary heap – implementation

The things we want to do with a heap (a priority queue implementation) are:

1. create the heap
2. add items
3. remove the item with the highest priority
4. get the size
5. find out if the structure is empty

Must always maintain the heap property!

Binary heap – create the heap


```
class BinHeap:
    def __init__(self):
        self.heap_list = [0]
```

Remember we are ignoring the first element of the list.

```
from BinHeap import BinHeap

def main():
    heap = BinHeap()

main()
```

Binary heap – create the heap


```
class BinHeap:
    def __init__(self, a_list = []):
        self.heap_list = [0] + a_list
        #see later slides
```

Better to use optional named arguments.

```
from BinHeap import BinHeap

def main():
    heap = BinHeap()
    heap = BinHeap([9, 5, 8, 6, 3, 2])
main()
```

Two ways to create a BinHeap

Binary heap – size(), is_empty()


```
class BinHeap:
    def __init__(self, a_list = []):
        self.heap_list = [0] + a_list
        #see later slides

    def size(self):
        return len(self.heap_list) – 1

    def is_empty(self):
        return self.size() == 0
```

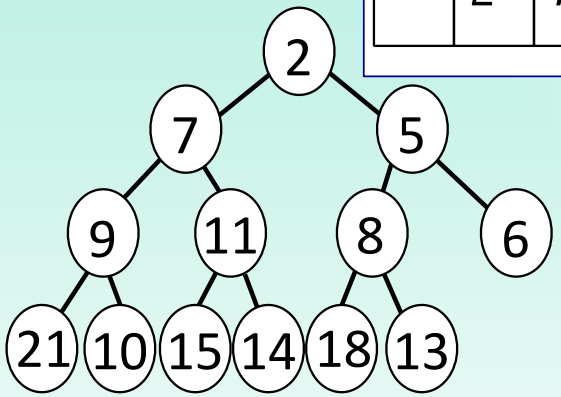
Remember we are ignoring the first element of the list.

```
from BinHeap import BinHeap

def main():
    heap = BinHeap()
    print(heap.is_empty())
    print(heap.size())

main()
```

Binary heap – insert()

my_heap →

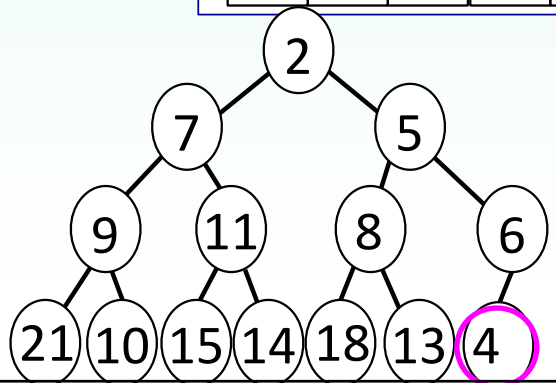
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	2	7	5	9	11	8	6	21	10	15	14	18	13	...	



my_heap.insert(4)

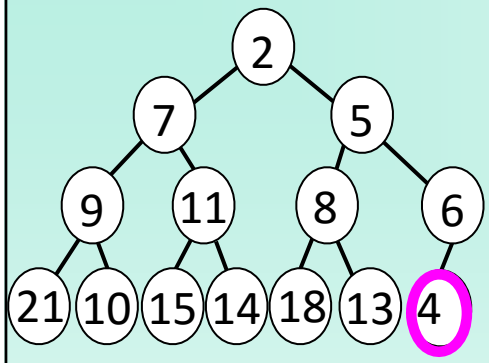
Add 4 to the end of the list. But now the heap property has been violated:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	2	7	5	9	11	8	6	21	10	15	14	18	13	4	



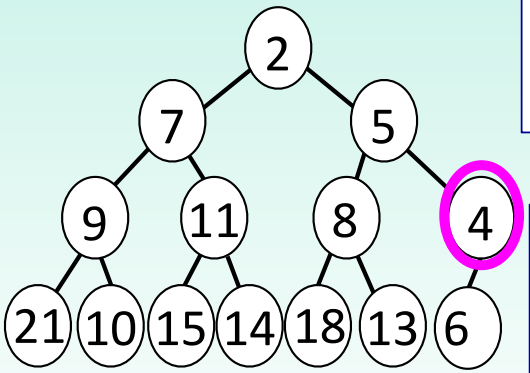
No longer a heap

Binary heap – perc_up()

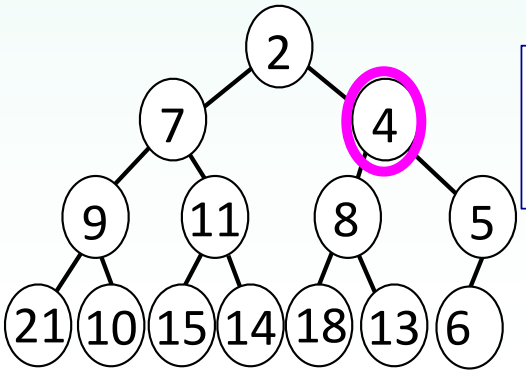


Need to keep comparing the element with its parent until heap propery fulfilled or we reach the root. Swap node and parent if the heap property is violated.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	2	7	5	9	11	8	6	21	10	15	14	18	13	4	



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	2	7	5	9	11	8	4	21	10	15	14	18	13	6	



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	2	7	4	9	11	8	5	21	10	15	14	18	13	6	

Valid heap ☺

Binary heap – insert() and perc_up()


```
def insert(self, item):
    self.heap_list.append(item)
    last_position = len(self.heap_list) - 1
    self.perc_up(last_position)
```

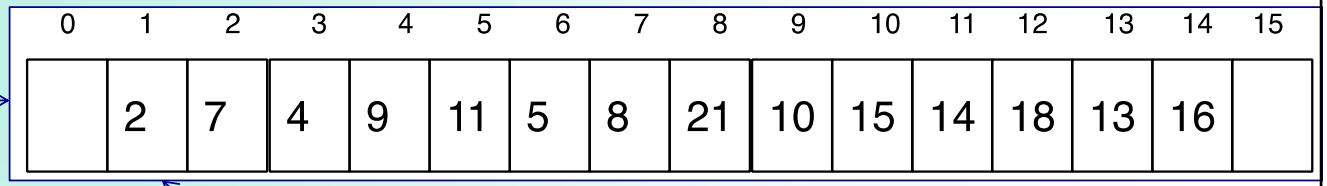
Insert as the last element and then do a perc_up()

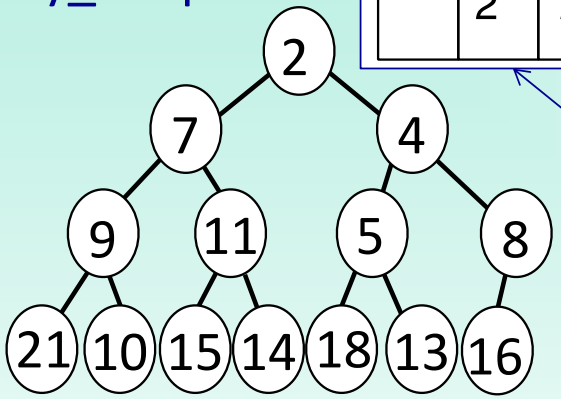
```
def perc_up(self, i):
    parent_index = i // 2
    while parent_index > 0 and
                self.heap_list[i] < self.heap_list[parent_index]:
        self.heap_list[i], self.heap_list[parent_index] =
                    self.heap_list[parent_index], self.heap_list[i]
        i = i // 2
        parent_index = i // 2
```

While the child is smaller than the parent, swap them

Binary heap – del_min()

my_heap

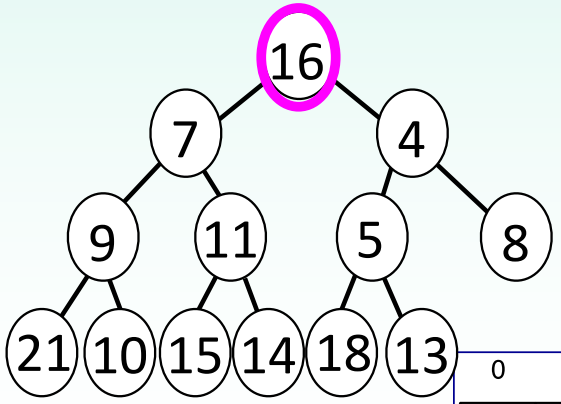




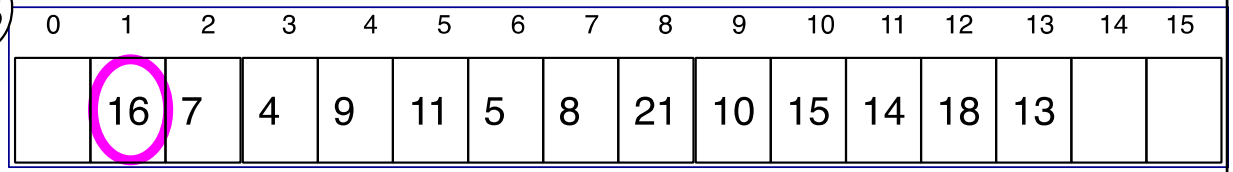
next = my_heap.**del_min()**

The minimum priority item is always the root item (index 1).

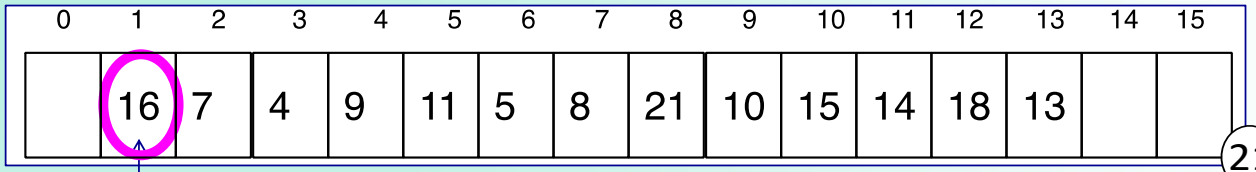
Store the minimum item (index 1), pop the last item from the list, put this item into the root position.

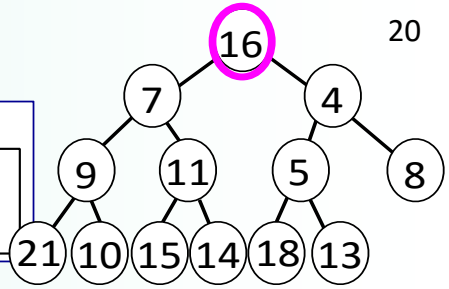


Problem - no longer a heap ☹

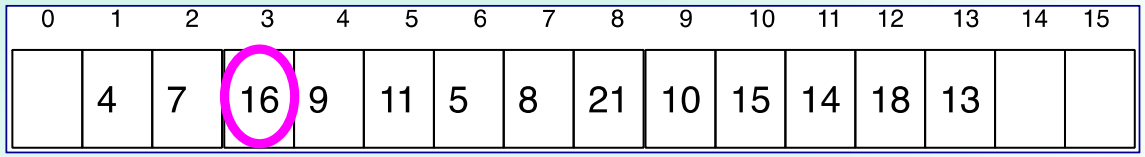


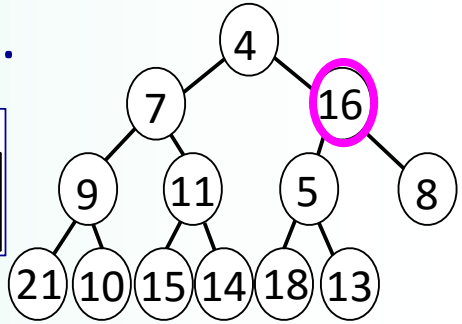
Binary heap – perc_down()

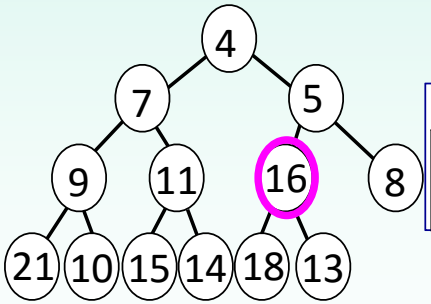


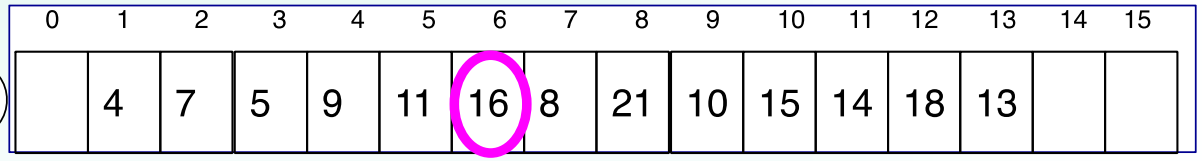


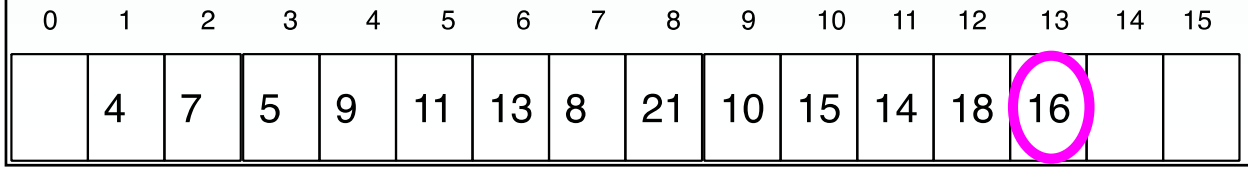
Need to keep comparing the element, swapping it with its smallest child if the heap property is violated.

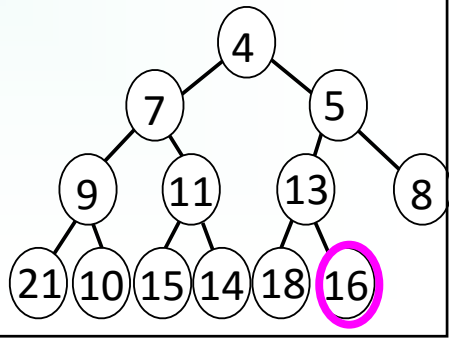












Binary heap – min_child()

Return the index of the smaller of the two children.

```
def min_child(self, i):
    left_child_index = i * 2
    right_child_index = left_child_index + 1
    if right_child_index > self.size():
        return left_child_index
    if self.heap_list[left_child_index] <
                                self.heap_list[right_child_index]:
        return left_child_index
    return right_child_index
```

Binary heap – perc_down()


```
def min_child(self, i): #see previous slide


def perc_down(self, i):
    left_child_index = i * 2
    while left_child_index <= self.size():
        child = self.min_child(i)
        if self.heap_list[i] > self.heap_list[child]:
                self.heap_list[child], self.heap_list[i] =
                                    self.heap_list[i], self.heap_list[child]
        i = child
        left_child_index = i * 2
```

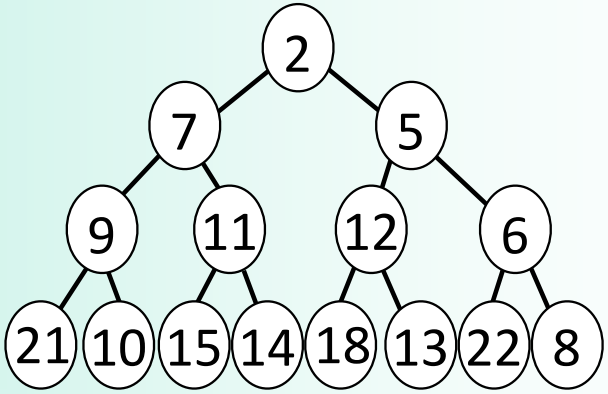
While the smaller child is smaller than the parent, swap them

i is the index of the element being percolated downwards

Binary heap – del_min()


```
class BinHeap:
    def __init__(self, a_list = [0]): #
    def del_min(self):
        return_value = self.heap_list[1]
        replacement = self.heap_list.pop()
        if self.size() > 0:
            self.heap_list[1] = replacement
            self.perc_down(1)
        return return_value
    def min_child(self, i):        # returns index of smaller child
        #see previous slide
    def perc_down(self, i):      # percolates down the heap
        #see previous slide
```

Binary heap operations – cost?



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	2	7	5	9	11	12	6	21	10	15	14	18	13	22	8

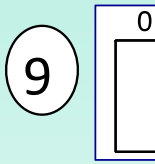
1. add item — **O(log n)**
2. remove the item with the best priority — **O(log n)**
3. get the size — **O(1)**
4. find out if the structure is empty — **O(1)**

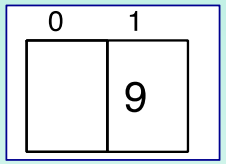
Binary heap – create a heap from list

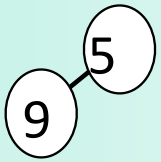
```
heap = BinHeap([9, 5, 8, 6, 3, 2])
```

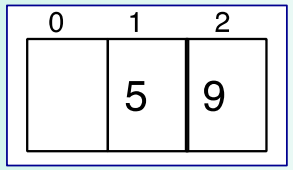
One way to create the heap from a list of elements: add them to an empty heap one by one. What is the cost?

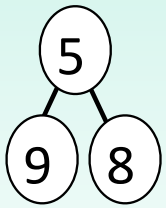
O(n log n)

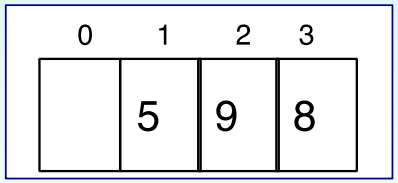


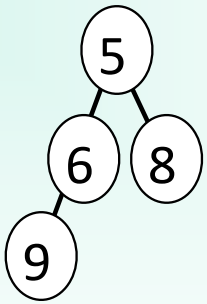


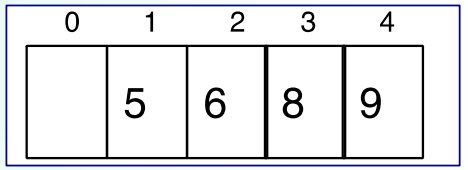


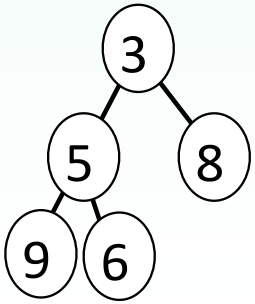


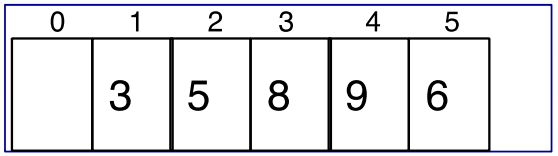


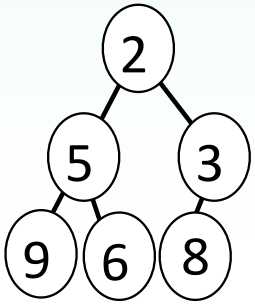


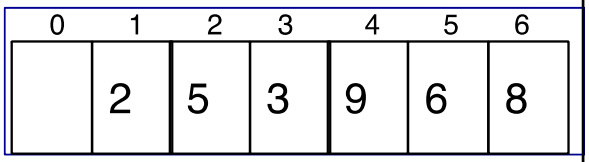






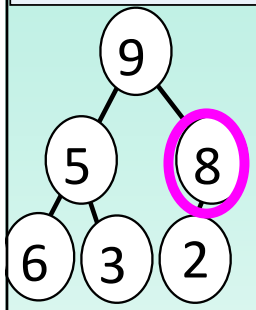


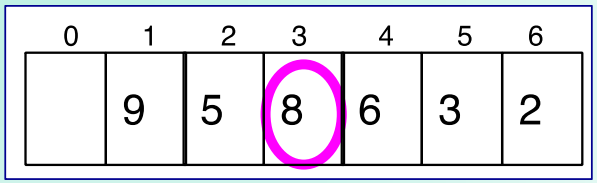


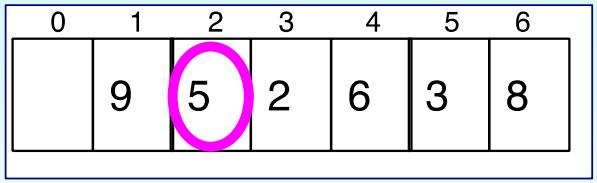


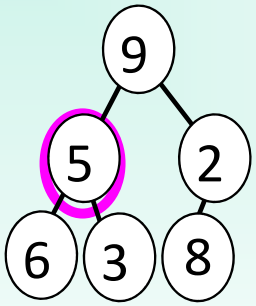
Binary heap – create a heap from list

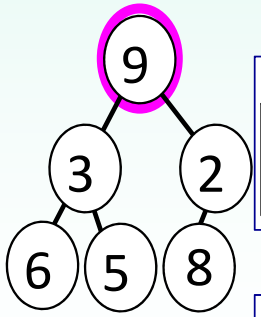
```
heap = BinHeap([9, 5, 8, 6, 3, 2])
```

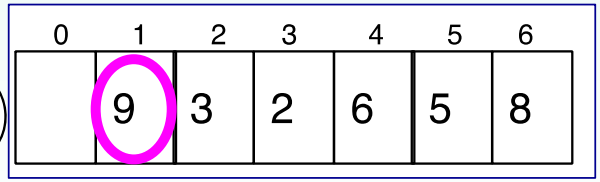


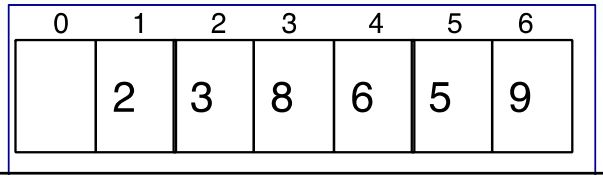


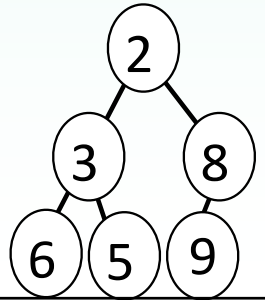












Another way (**BETTER**):

- put elements into the list,
- rearrange the elements starting from **the first element which has a child** (size // 2 – if the heap elements start at index 1).

Keep rearranging (i.e., perc_down()) working backwards towards the first element.

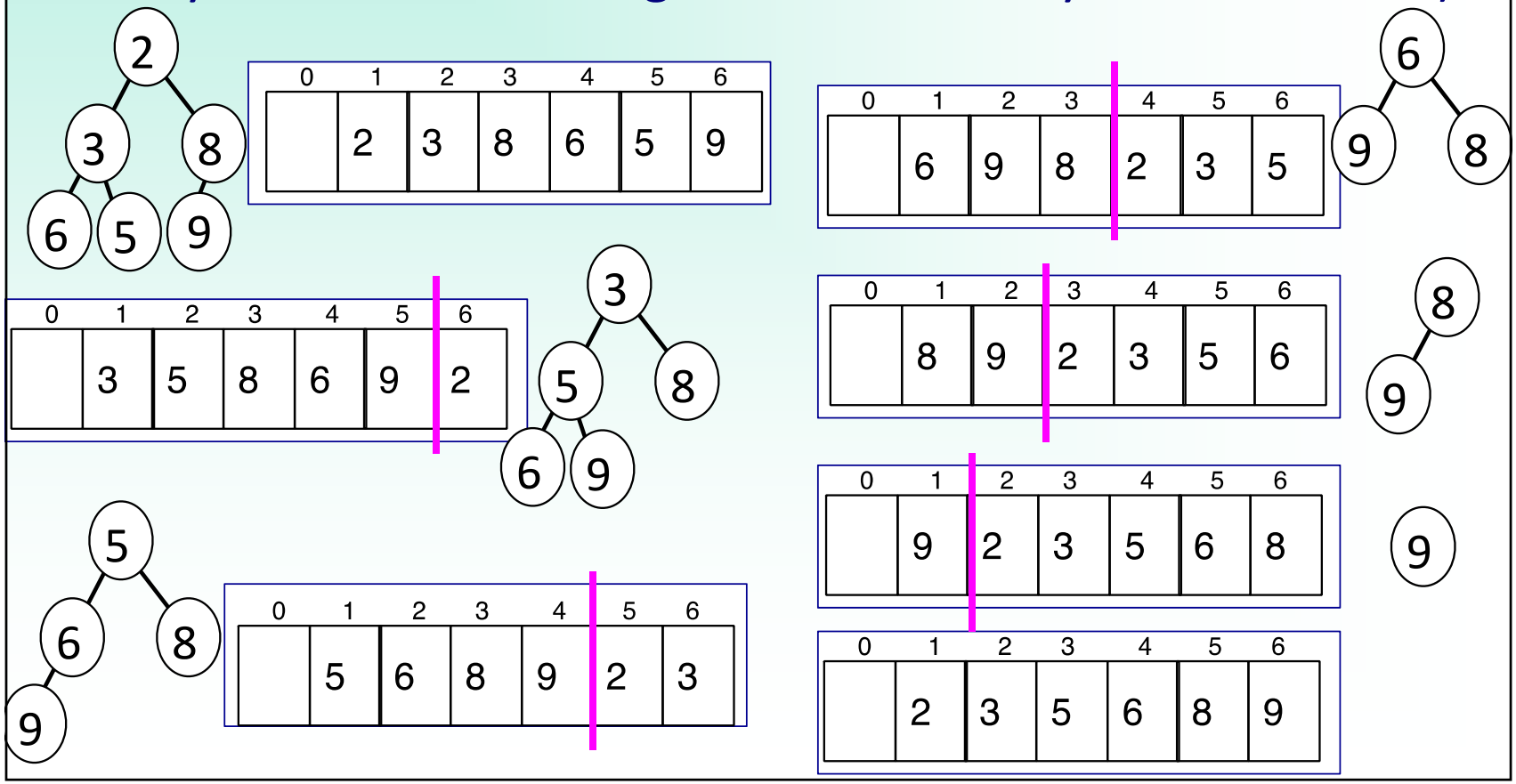
Can be shown to be **O(n)**.

Binary heap – create a heap from list


```
class BinHeap:
    def __init__(self, a_list = []):
        self.heap_list = [0] + a_list
        start_index = self.size() // 2
        for i in range(start_index, 0, -1):
            self.perc_down(i)
```

Heapsort – not part of CompSci 105

Create a heap from a list (O(n)) and remove the minimum element adding it to the end of the list (decreasing the heap size by one and increasing the sorted list by one each time).



Binary heap – past exam question 1

Draw the heap structure after inserting the following integer search key values (in the order given) into an empty min-heap:

15, 23, 42, 12, 91, 75

Show the structure of the heap after EACH insertion. NOTE: show the state of the heap using a tree diagram (not a list).

Binary heap – past exam question 2

This is the BinHeap constructor presented in class.

```
def __init__(self, a_list=[]):
  self.heap_list = [0] + a_list
  for i in range(self.size() // 2, 0, -1):
    self.perc_down(i)
```

The size method returns the size() of the heap and the perc_down() method percolates a value down the heap to its correct place. Using the above algorithm, convert the list

[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

into a binary min heap. Draw the tree representation of the binary heap for each value of i in the for loop.

Binary heap – past exam question 3

a) A heap can be constructed from an unsorted list.
Convert the list with the elements

10, 5, 2, 9, 3, 6

into a min-heap, using the technique shown in lectures.
Show the heap (as a tree) at each step.

b) Draw the heap after one del_min() operation is performed to the heap structure resulting from part a)

c) Draw the heap after another del_min() operation is performed to the heap structure resulting from part b).