Priority Queues and Heaps
Tom Przybylinski

Maps

- We have (key,value) pairs, called entries
- We want to store and find/remove arbitrary entries (random access)
- Only one entry in the map ever has the same key.

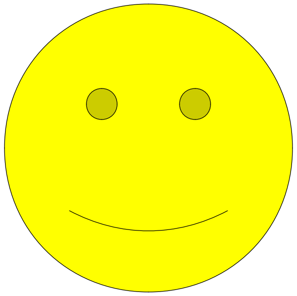
Priority Queues

- Only care about an entry with a maximum xor minimum key, and no others.
- We can assume we want the minimum key (a min-queue)
  - We may say that the entry with the minimum key has the highest priority
  - Everything I say will hold for max-queues as well.
-

Priority Queues

- One way to think of a priority queue is as a generalization of Queues and Stacks.
- Queue is FIFO, so the first in has the highest priority
- Stack is FILO so the last has highest priority

Example

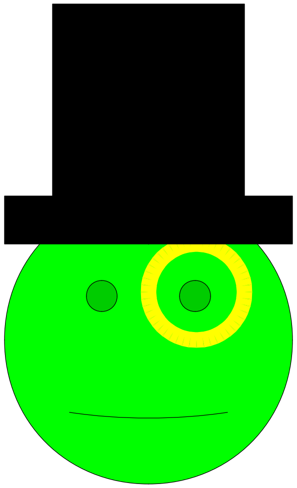


Coworker:

Wants to talk
about bottle caps

Boss:

Is going to tell you
the server is
down, and you
need to fix it.



Rich Long Lost
Uncle:

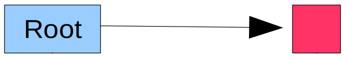
Wants to tell you
that you that he is
your long lost
uncle and you are
the sole heir to his
vast fortune.

Priority Queue Functionality

- size(): the number of entries in the queue
- isEmtpy(): tests whether the queue is empty
- peek(): Returns the entry with the minimum key in the queue without removing it.
- poll(): returns the entry with the minimum key in the queue, removing it.
- insert(key k, value v): inserts the entry with key k and value v into the queue

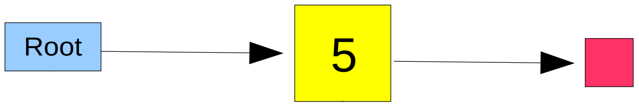
Implementation 1

- Use an unsorted list, it works best if you have a doubly linked list.
- Insertion just appends the entry to the end of the list.
- size() and isEmpty() use the list size
- peak() or poll() search the list for the smallest element, removing it if we are using poll()



Root

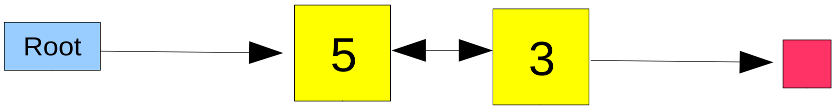
Insert 5



Root

5

Insert 3

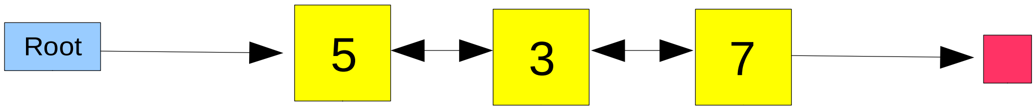


Root

5

3

Insert 7



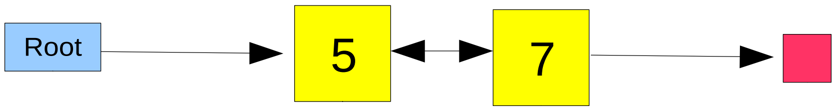
Root

5

3

7

Poll



Root

5

7

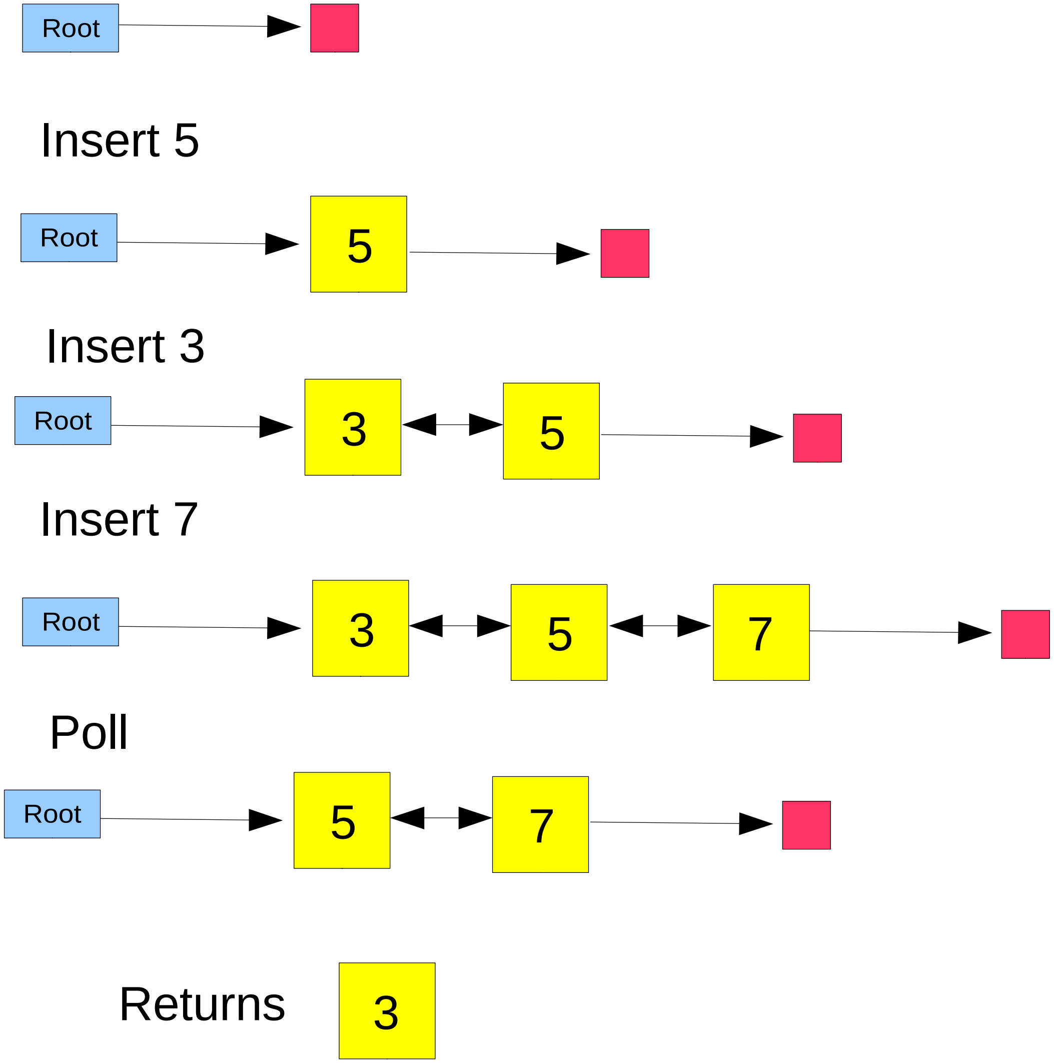
Returns 3

Implementation 1

- With a pointer to the end of the list, insertion is O(1)
- size() and isEmpty() are O(1)
- peak() and poll() are O(n)

Implementation 2

- Use a list sorted in non-decreasing order, using a doubly linked list again.
- When inserting an element, go up the list until you find the right place to insert it.
- When we peak or poll, just have to return/remove the head of the list.



Insert 5

Insert 3

Insert 7

Poll

Returns

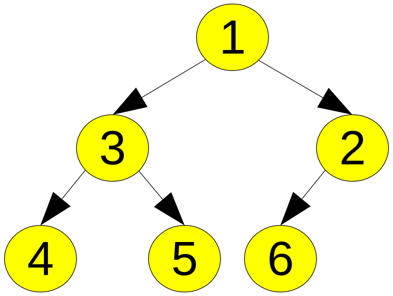
Implementation 2

- This time insert is O(n)
- While peek and poll are O(1)

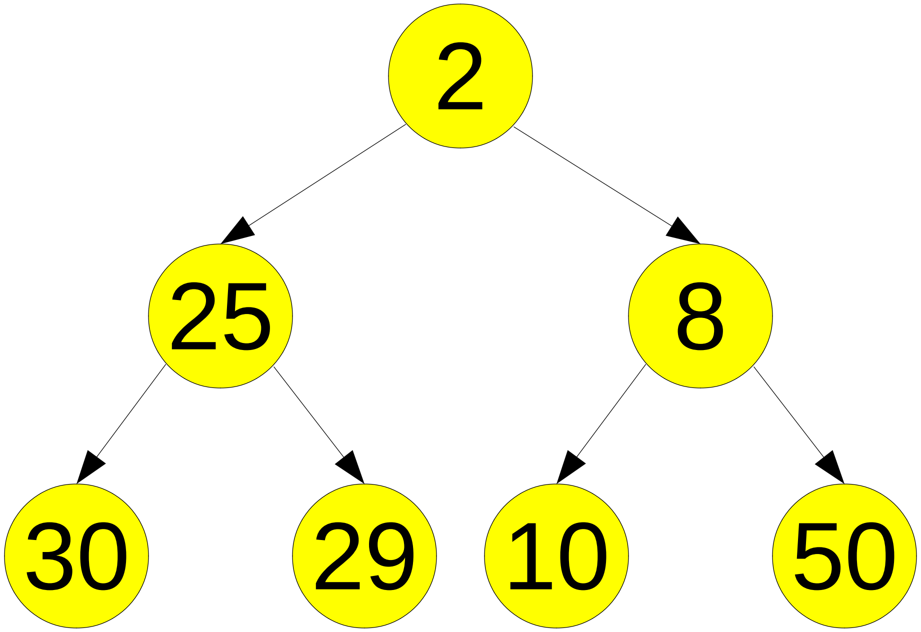
- So the unsorted list is good for when we are *frequently* inserting entries but *rarely* getting them out.
- The sort list implementation is better for when we are *rarely* inserting entries but *frequently* getting them out.

Heaps

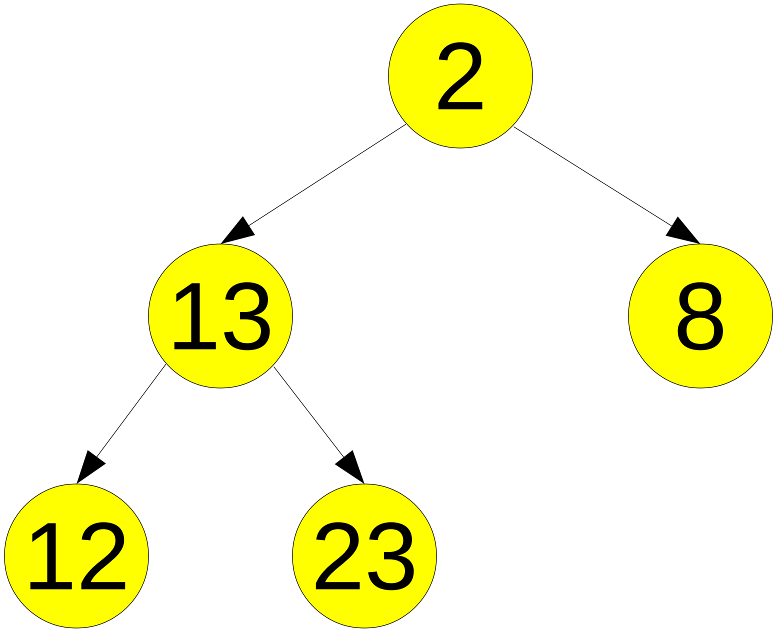
- We'd like something more balanced for a general-purpose priority queue.
- A heap is a binary tree with the following properties:
  - For every node N, every child of N has a key greater than or equal to the key of N
  - It is complete. This means that it is as balanced as possible, i.e all levels except for the last are completely filled. In addition, on the last level, it is packed to the left.



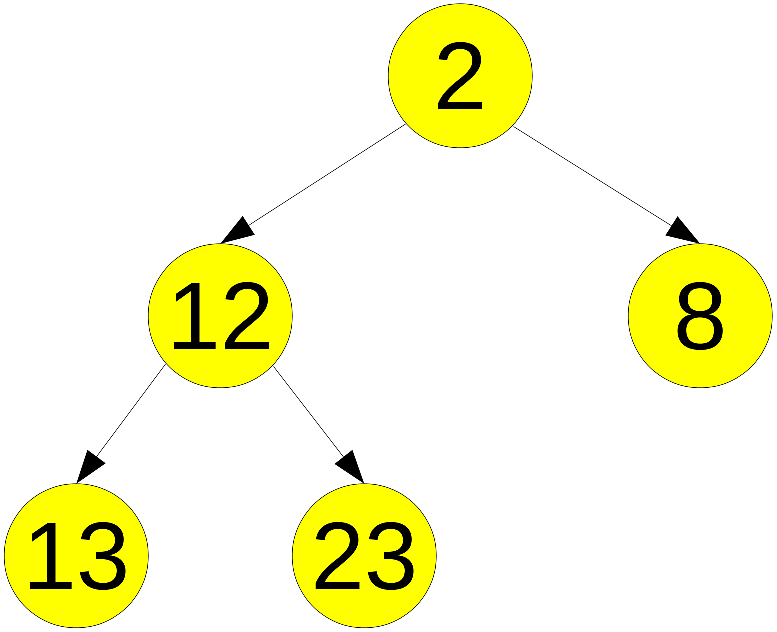
Good Heap



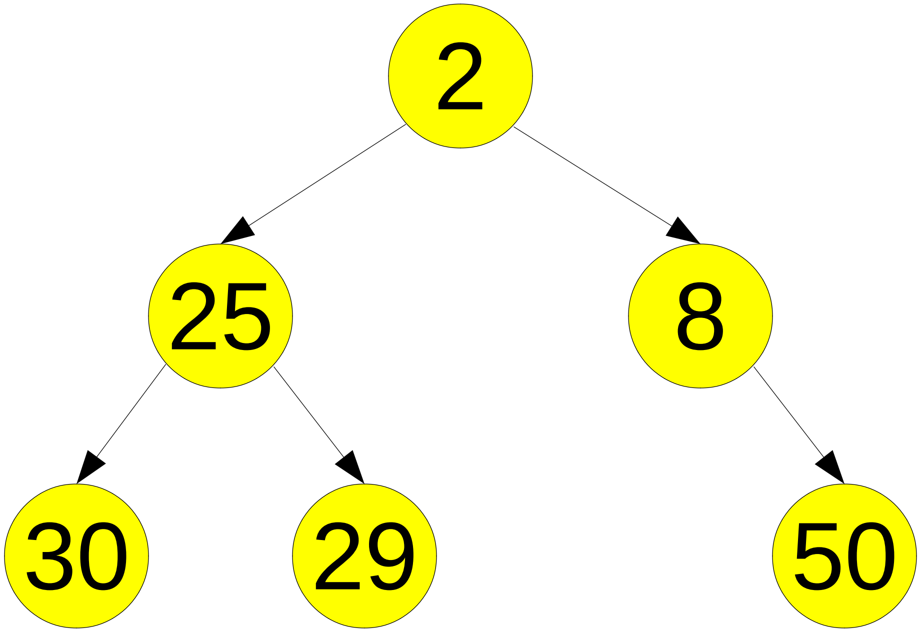
Bad Heap



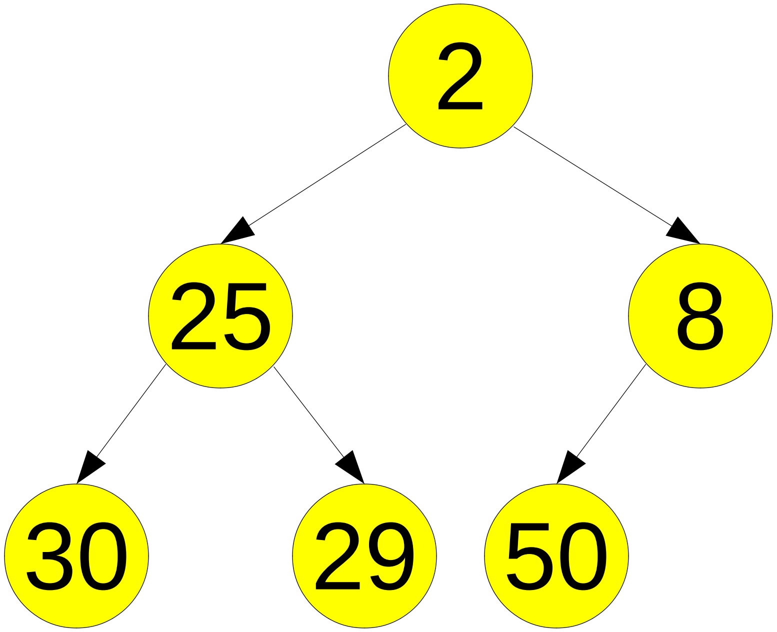
Fixed



Bad Heap



Fixed



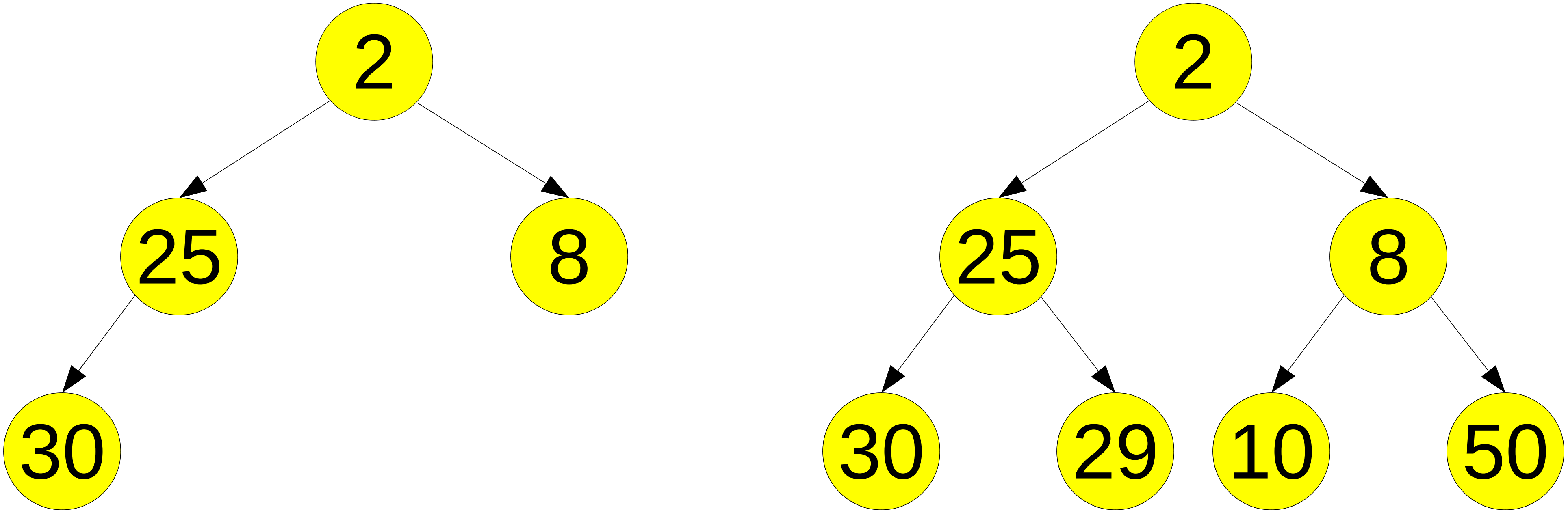
Binary Search Trees

- Why use heaps when we can use BSTs?
- Binary search trees are more strict to have fast searches.
- They could technically be used:
  - We insert entries into the tree as normally.
  - When peeking or polling, we return the leftmost child and/or delete it.
  - So insertion is O(log(n)) and peek and poll are both O(log(n))

Binary Search Trees

- However, balancing a tree can get complicated.
- Heaps are much simpler, and so are faster. Their performance for peek() is also better.
- Heaps also lend themselves to an alternate implementation.

Heap Height

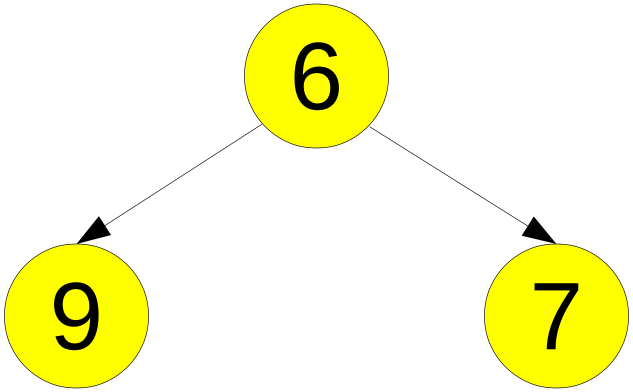


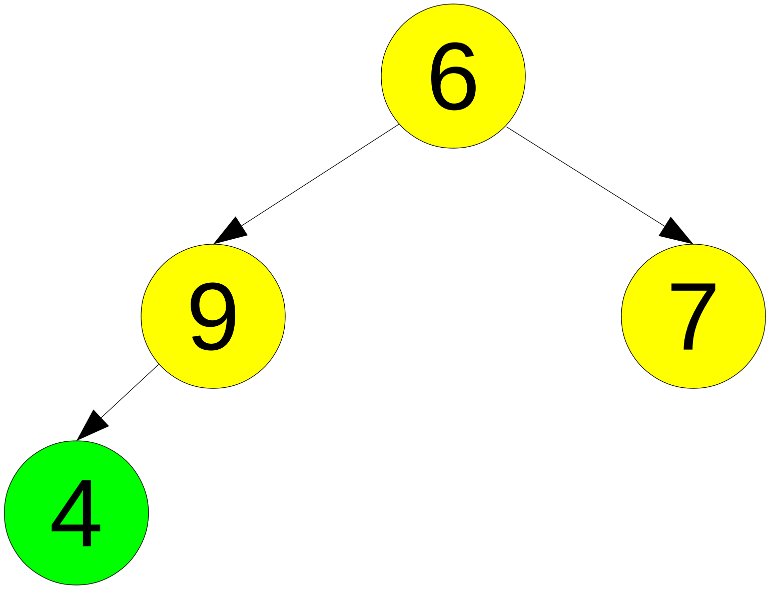
Priority Queues as Heap

- We can keep track of the heap's size for size() and isEmpty()
- When we call peek(), we can just return the root entry.

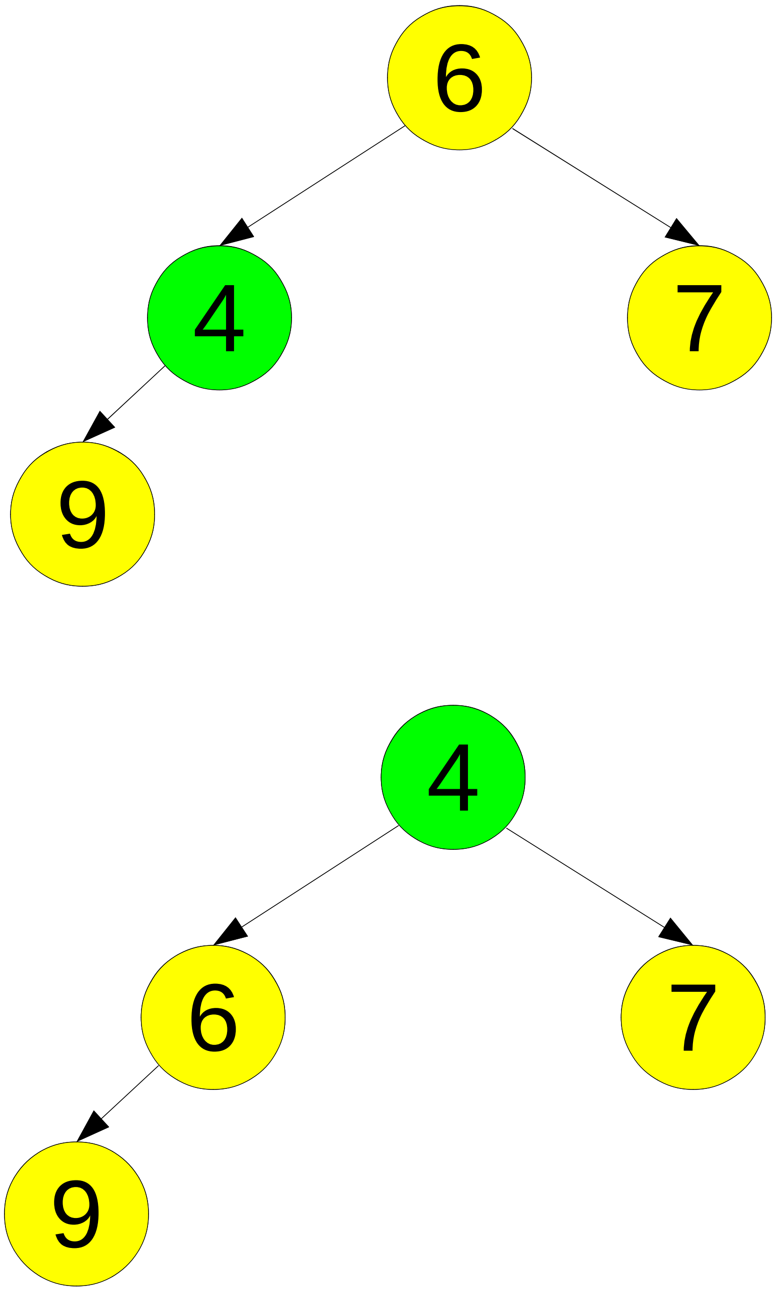
Insertion

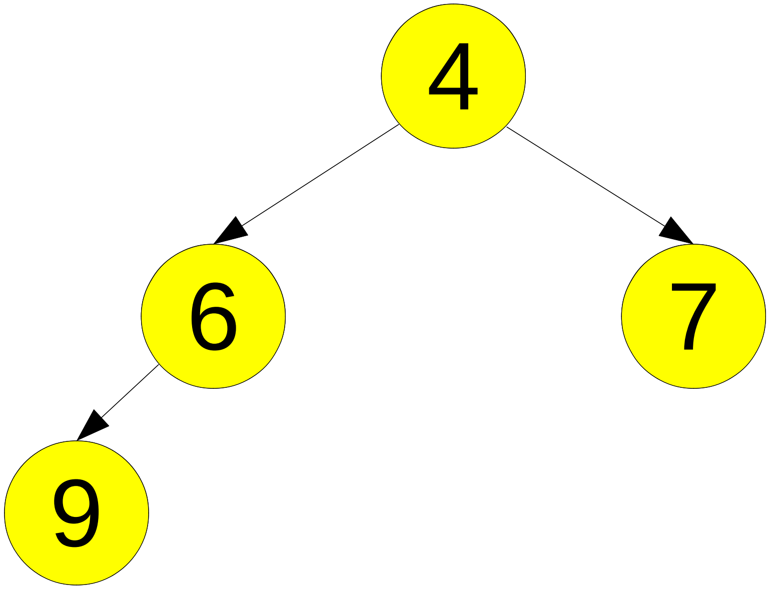
- Add entry to leftmost free node to preserve completeness.
- While your current entry is less than the entry in its parent, swap entries. We call this operation a “bubble up”


6
9
7

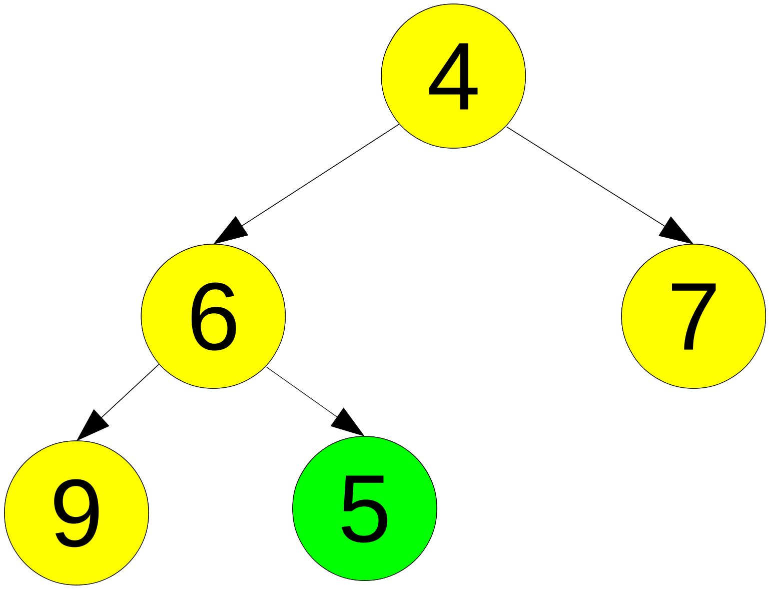

6
9
7
4

Bubble Up

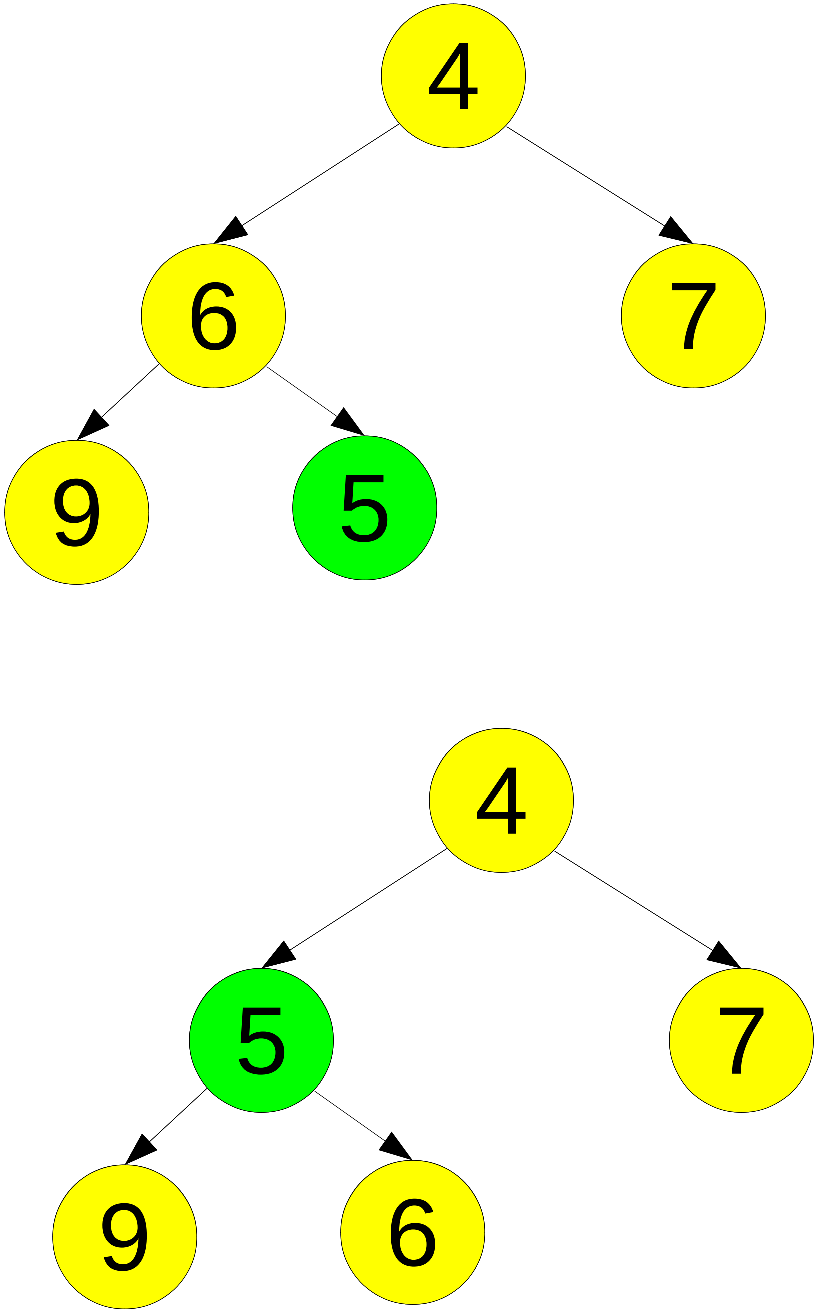




Insert 5

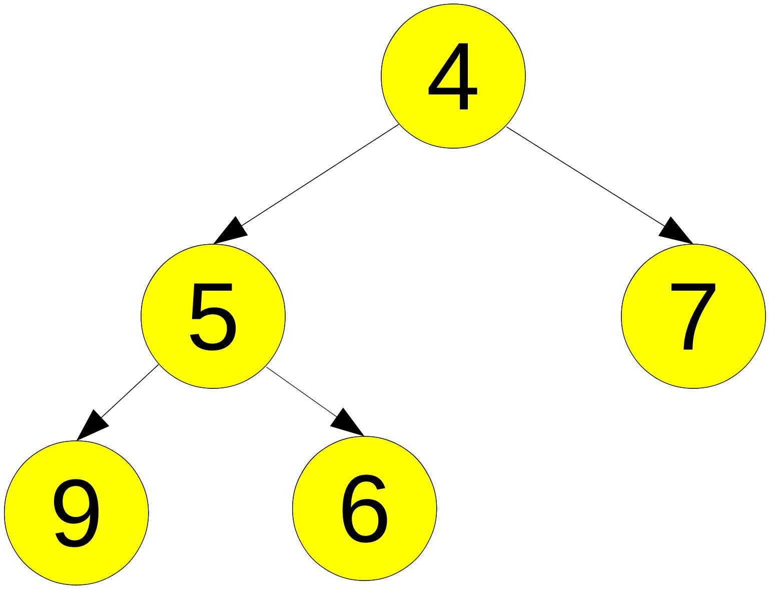


Bubble Up

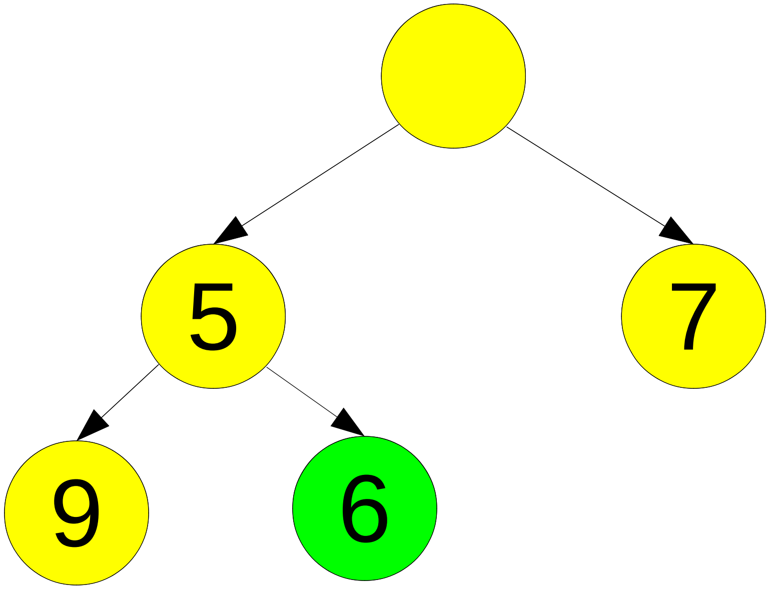


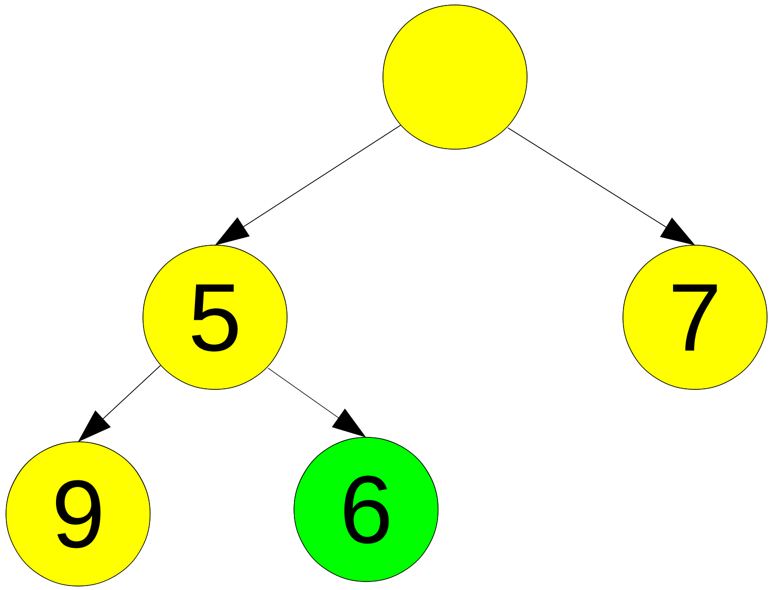
Deletion

- Since we only delete an entry on a poll(), we only ever delete the root.
- Procedure:
    - Remove the root
    - Replace the root with the leftmost node in the last level to preserve completeness.
    - While that node is greater than either of its children, swap with the smallest of the two children. This is known as a “bubble down”

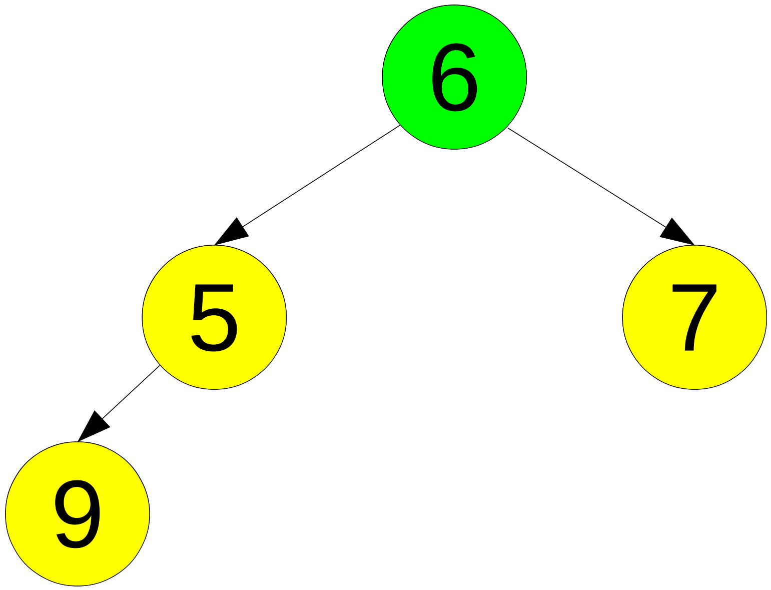


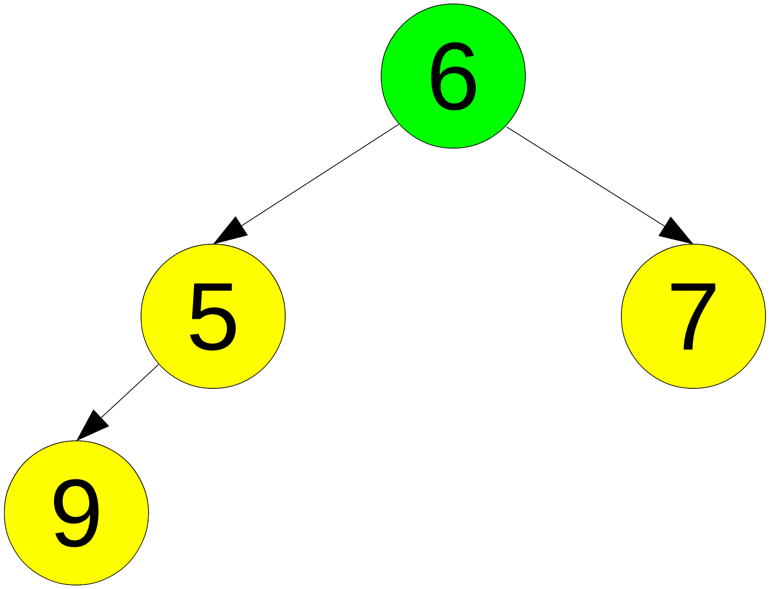
Poll



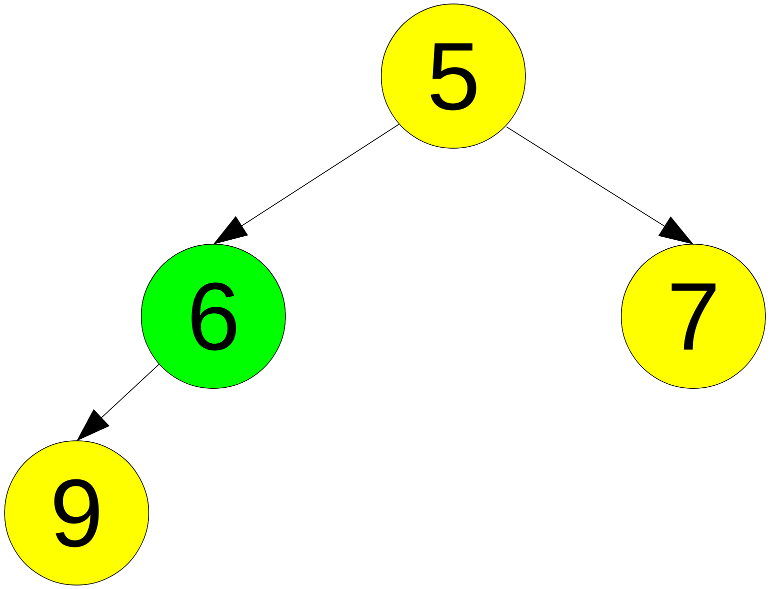


Replace Root





Bubble Down



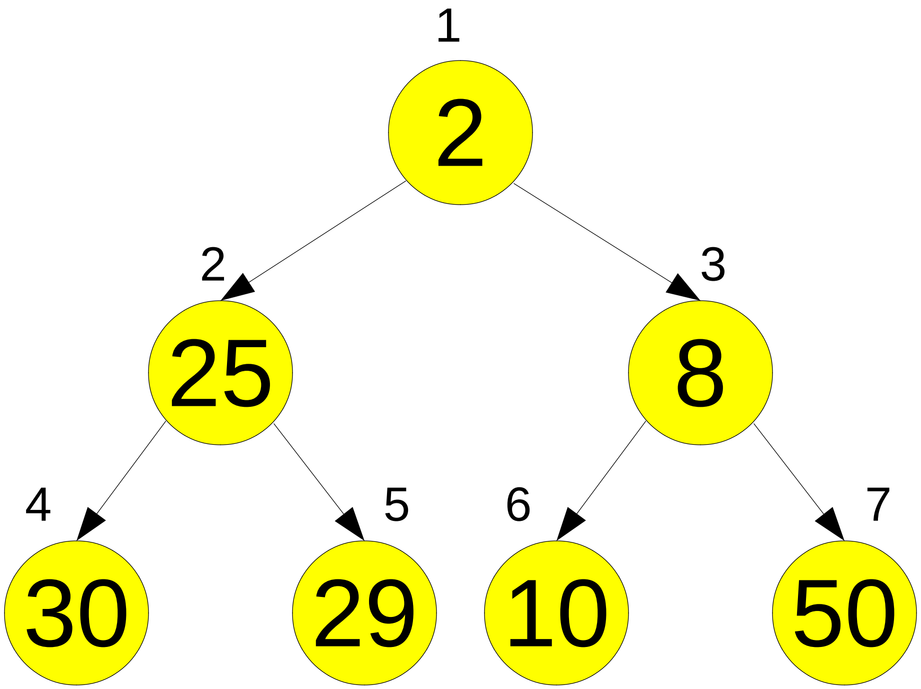
Efficiency

- Insertion is O(log(n)) since each bubble up moves the entry up one level.
- All of size(), isEmpty(), and peek() are O(1)
- Deletion is O(log(n)) for similar reasons to insertion.

Array Implementation of the Heap

- Because of completeness, we always add and remove nodes in a predictable way.
- This allows us to assign each node an index

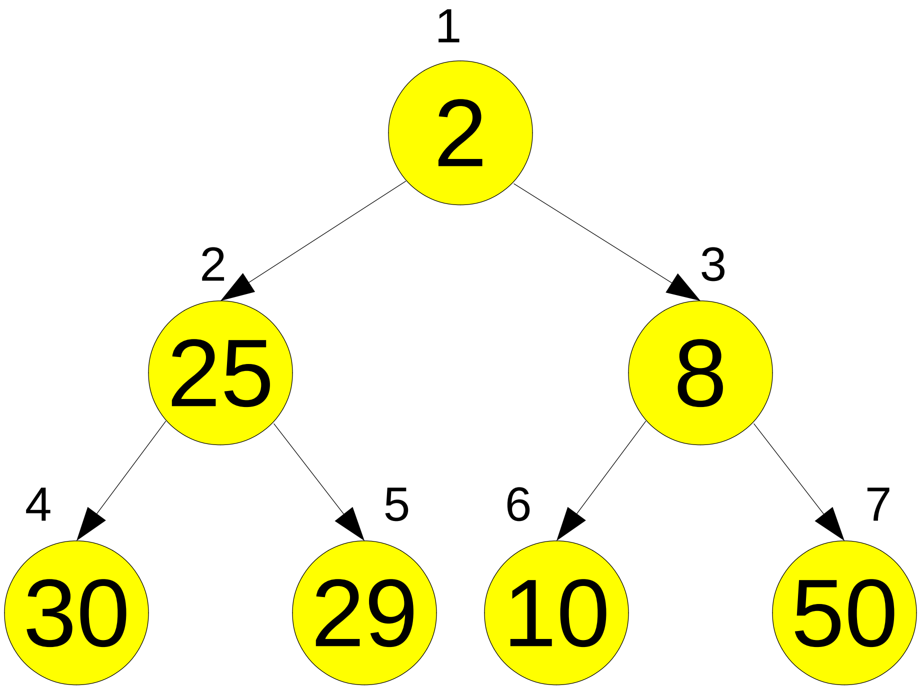
Array Implementation of the Heap



2	25	8	30	29	10	50
1	2	3	4	5	6	7

Array Implementation of the Heap

- For a parent node at index i, its children are at index 2i and 2i+1
- For a child node at index i, its parent is at i/2 (truncated) unless it is the root.



Priority Queue Sorting

- We can sort using a priority queue by:
  - Adding all elements to priority queue
  - while(priorityQueue not empty)
    - Append the result of poll() to the list
- Since insertion and deletion are O(log(n)), this sorting is O(n*log(n))

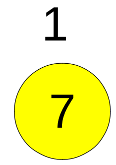
Array Heap Sorting

- We can do a better, *in place* sort using the array representation of a heap.
- An arbitrary array is not a heap representation, however, so we must first “heapify” the array.
- We can do this by starting our “heap” at the first index (for clarity, index 1). We then “add” elements to the heap by increasing index, bubbling up when necessary.

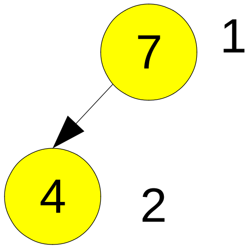
Array Heap Sorting

- We then iteratively “remove” the root, swapping the root with the last valid index of the heap.
- Note that the array will be sorted in the opposite direction the heap sorted them.

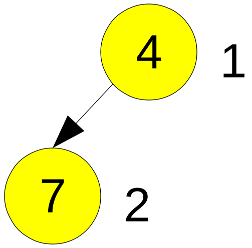
7	4	2	6	1
1	2	3	4	5



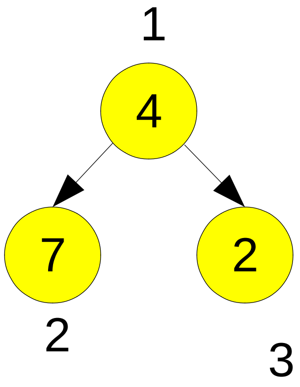
7	4	2	6	1
1	2	3	4	5



7	4	2	6	1
1	2	3	4	5

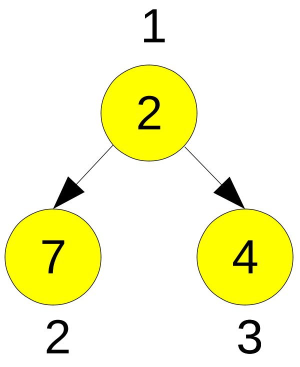


4	7	2	6	1
1	2	3	4	5

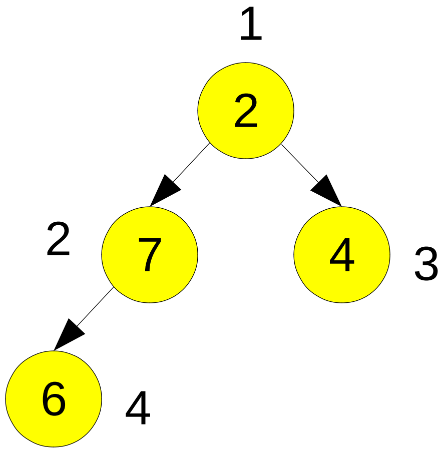


4	7	2	6	1
1	2	3	4	5

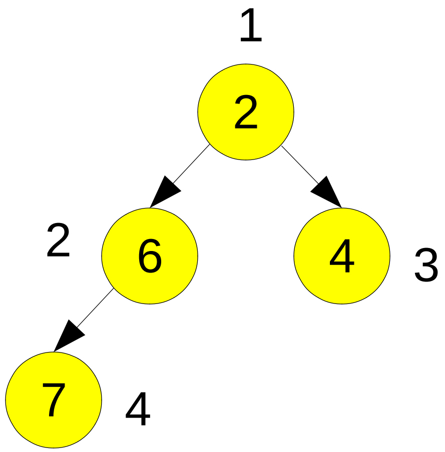
2	7	4	6	1
1	2	3	4	5



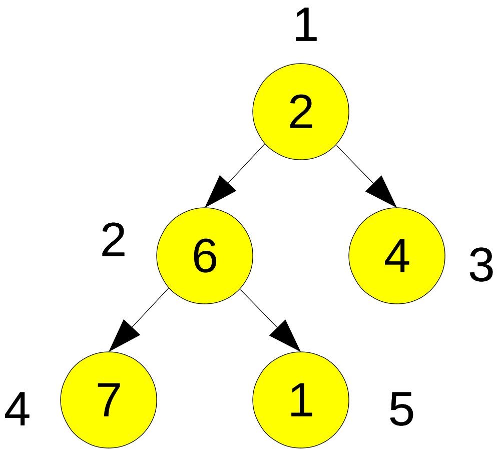
2	7	4	6	1
1	2	3	4	5



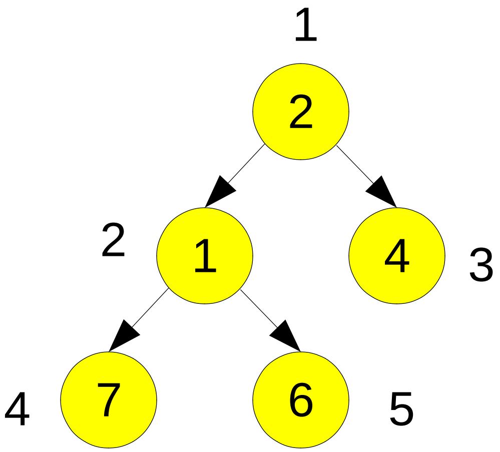
2	6	4	7	1
1	2	3	4	5



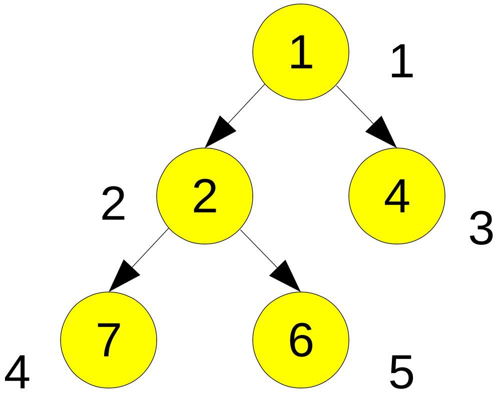
2	6	4	7	1
1	2	3	4	5



2	1	4	7	6
1	2	3	4	5

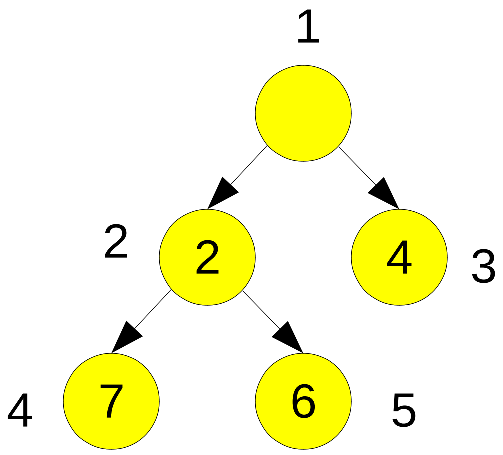


1	2	4	7	6
1	2	3	4	5



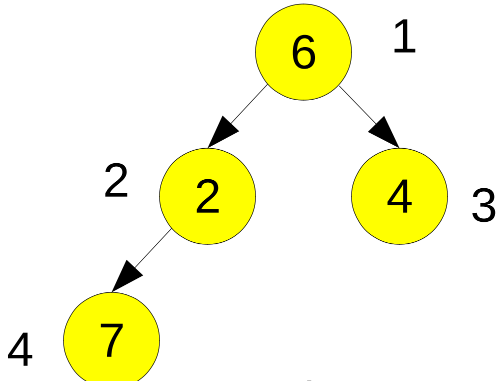
	2	4	7	6
1	2	3	4	5

1

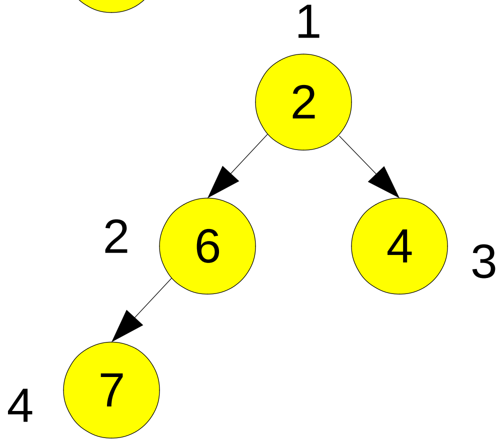


6	2	4	7	
1	2	3	4	5

1

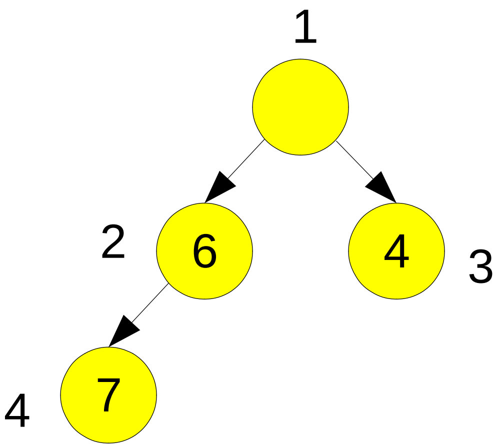


2	6	4	7	1
1	2	3	4	5



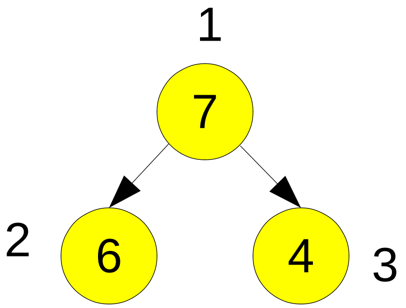
	6	4	7	1
1	2	3	4	5

2

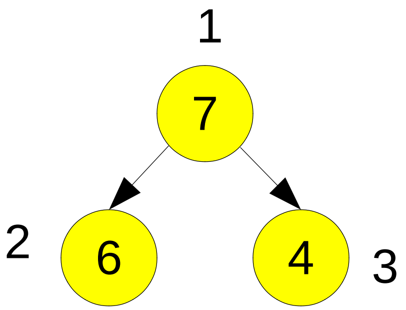


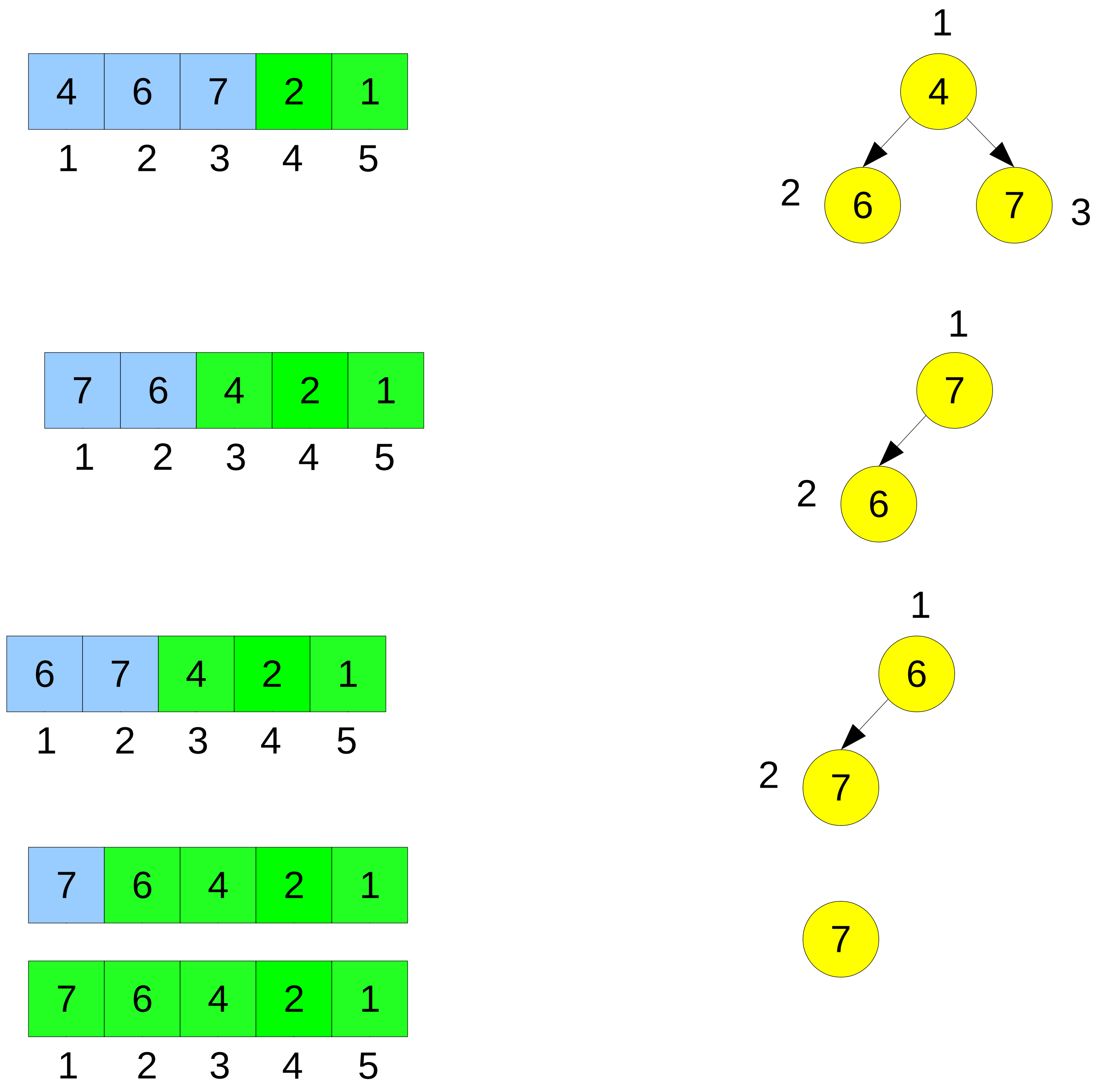
7	6	4		1
1	2	3	4	5

2



7	6	4	2	1
1	2	3	4	5





4 6 7 2 1
1 2 3 4 5
1
4
2
6
7
3
7 6 4 2 1
1 2 3 4 5
1
7
2
6
6 7 4 2 1
1 2 3 4 5
1
6
2
7
7 6 4 2 1
7
7 6 4 2 1
1 2 3 4 5

Comparison To Other Sorts

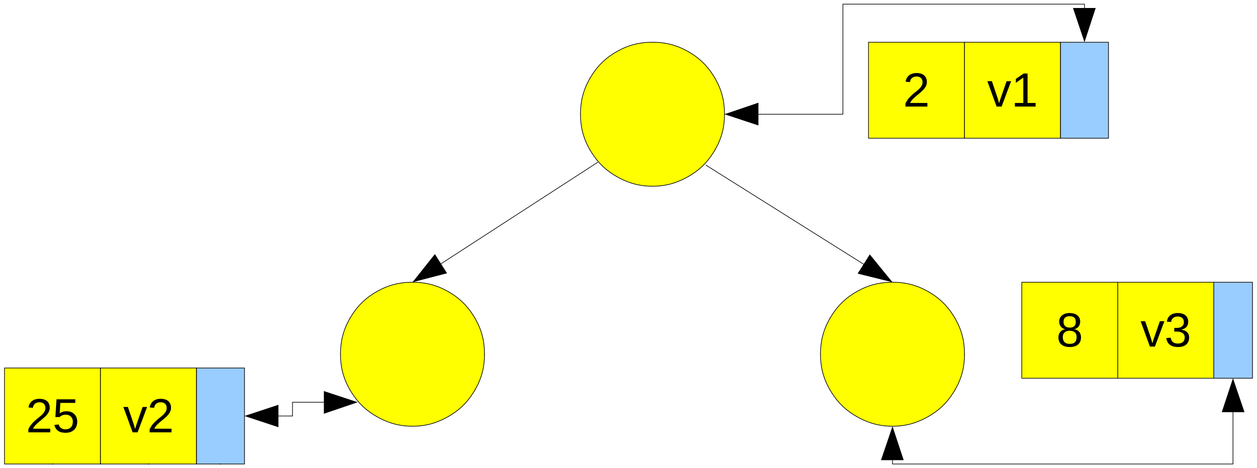
- Unlike Quick Sort, Heap sort is a guaranteed O(log(n))
- Unlike Merge Sort, Heap sort can be done in-place, with little extra memory overhead.
- Tends to be used in secure embedded systems due to low memory usage and guaranteed time, but not often in desktop/server use since Quick Sort is usually faster, and Merge Sort has better parallelism.

Adaptive Priority Queue

- In some scheduling situations, we may want to perform updates on our entries.
- We may want to remove an entry that is no longer needs to be processed.
- We may want to change the value of the entry.
- Some entry's priority may have changed.

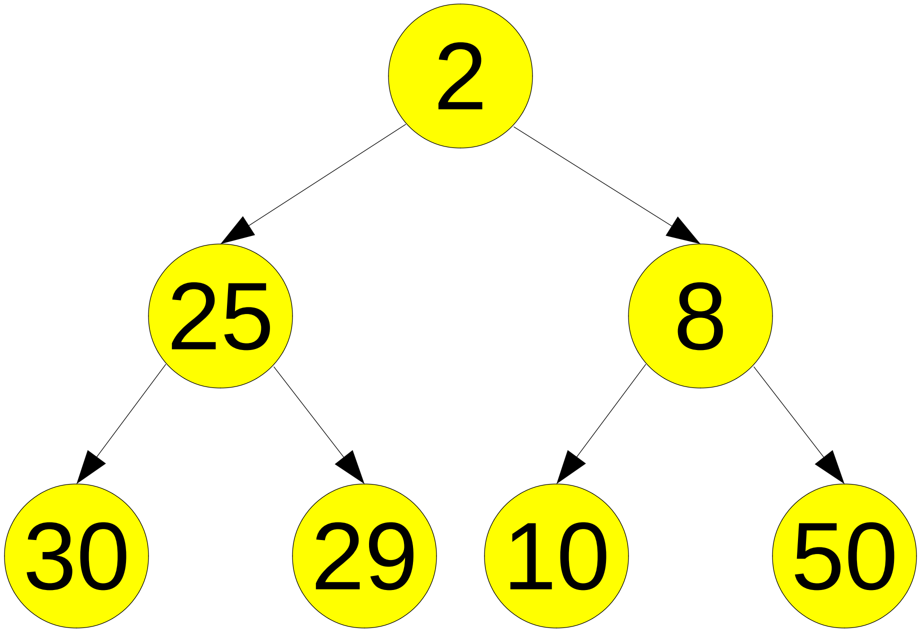
Adaptive Priority Queue

- Those are all operations that require random access to our entries.
- So we may think we need to search through all the nodes of a heap to find the entry.
- We can cheat, however, by modifying our entry so that it points to the node that stores it.

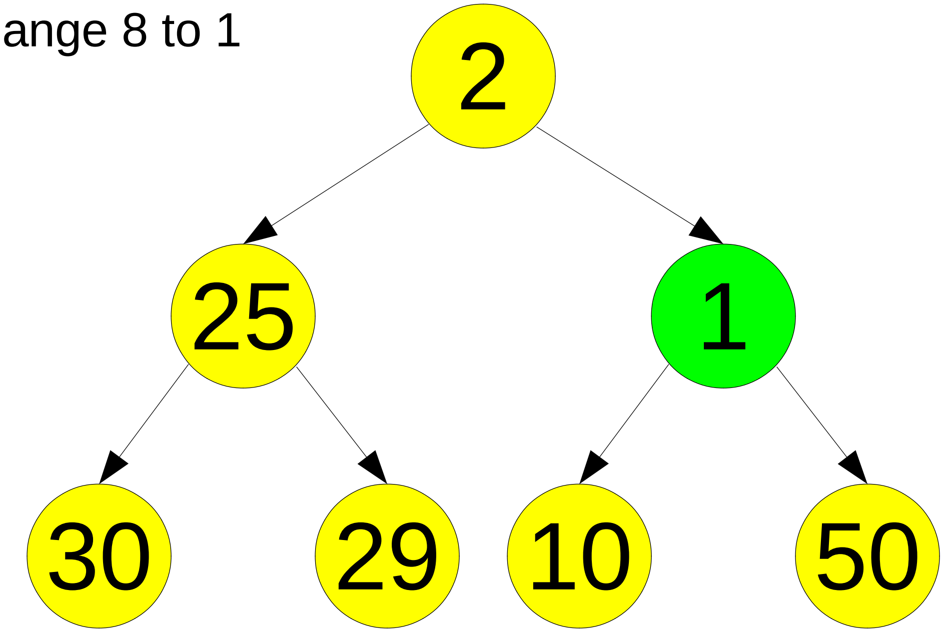

2
v1
25
v2
8
v3

Adaptive PQ Algorithms

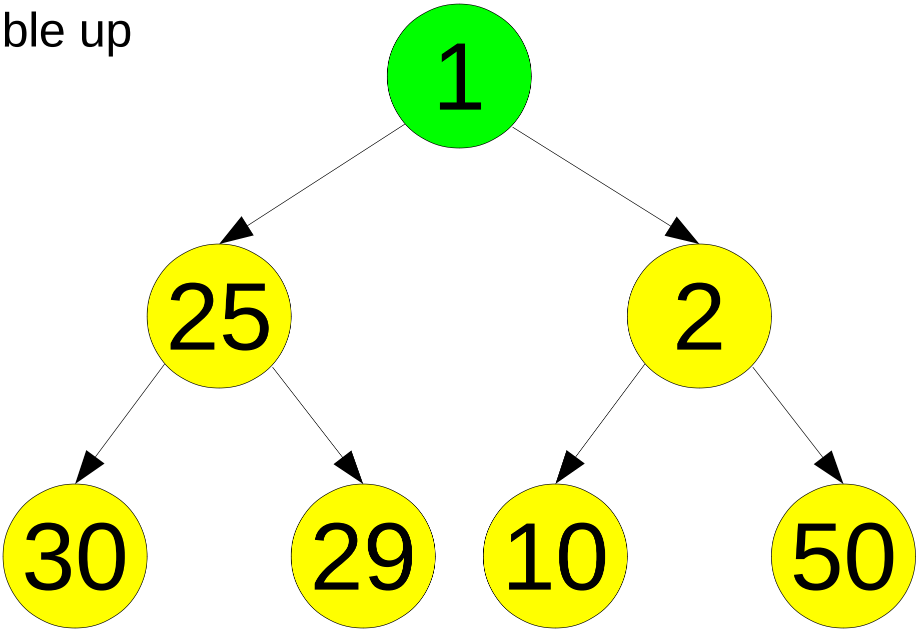
- Changing the value of an entry does not affect the heap, so it's a simple O(1) algorithm.
- Changing the key and deleting an entry are a little more difficult.
- Before, we either “bubbled down” from the root, or “bubbled up” from the leaves. However, this time the node could be on any level.


2
25
8
30
29
10
50

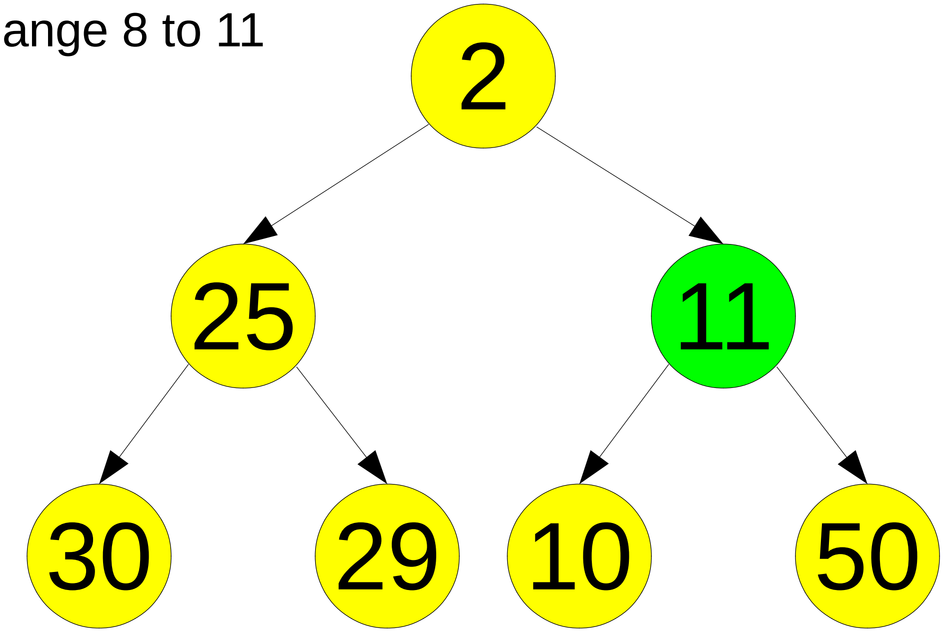
Change 8 to 1



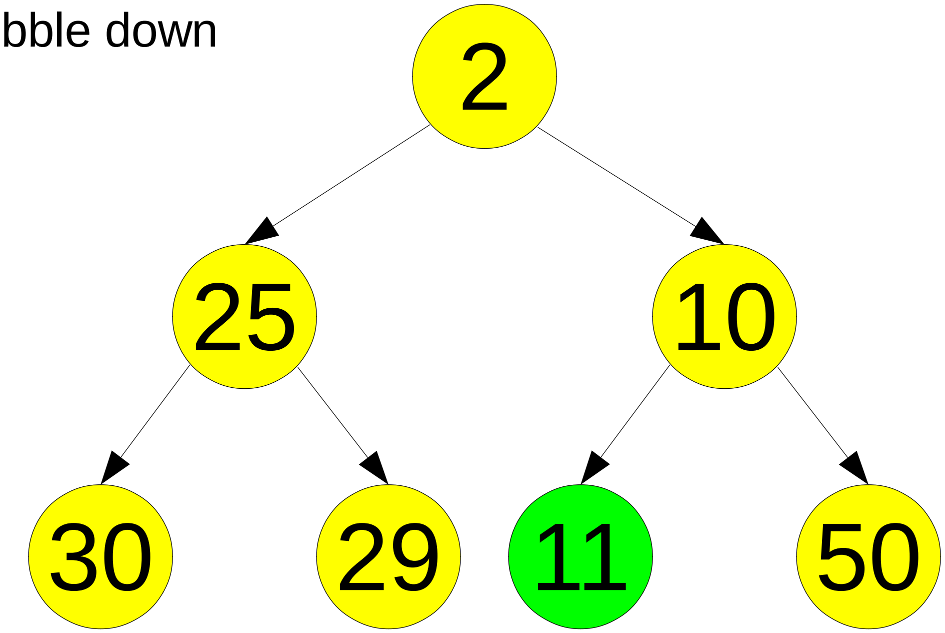
Bubble up



Change 8 to 11



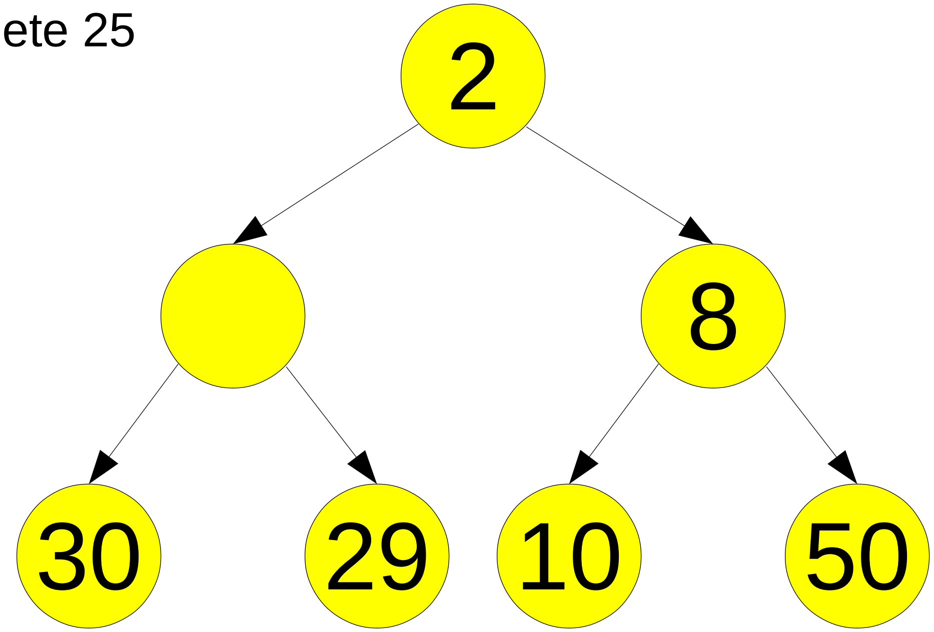
Bubble down



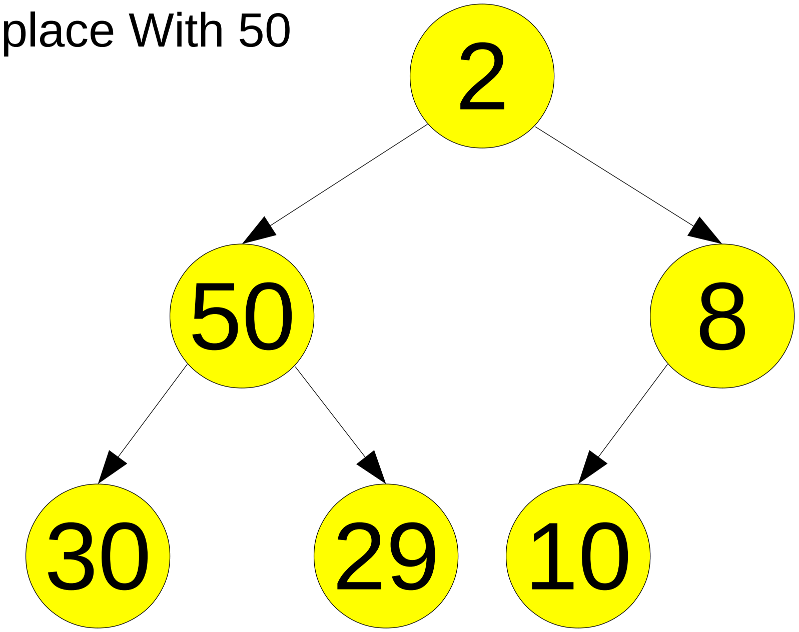
Adaptive PQ Algorithms

- So when changing the key, we need to check both the parent and the children to see if we should bubble up or bubble down.
- When removing an entry, we replace it with the leftmost node of the lowest level (just like with poll()) and check the same nodes to see if we should bubble up or down.

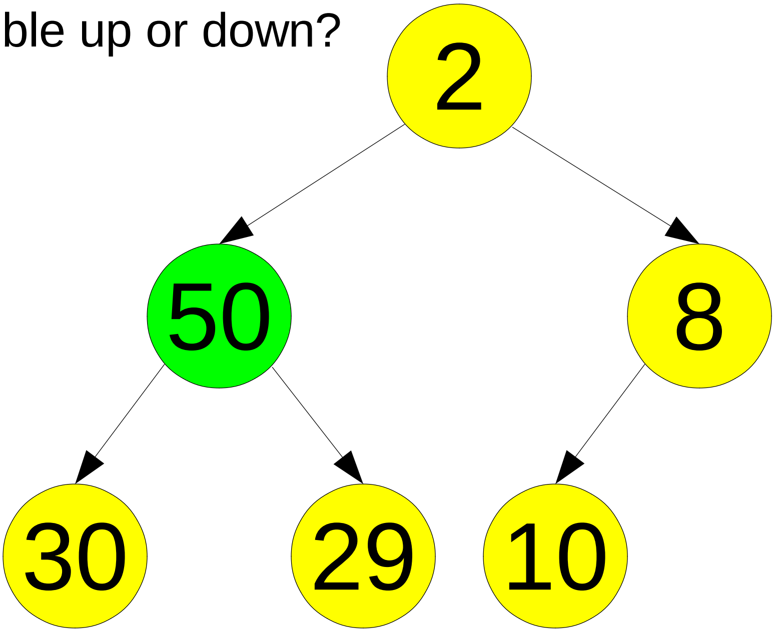
Delete 25



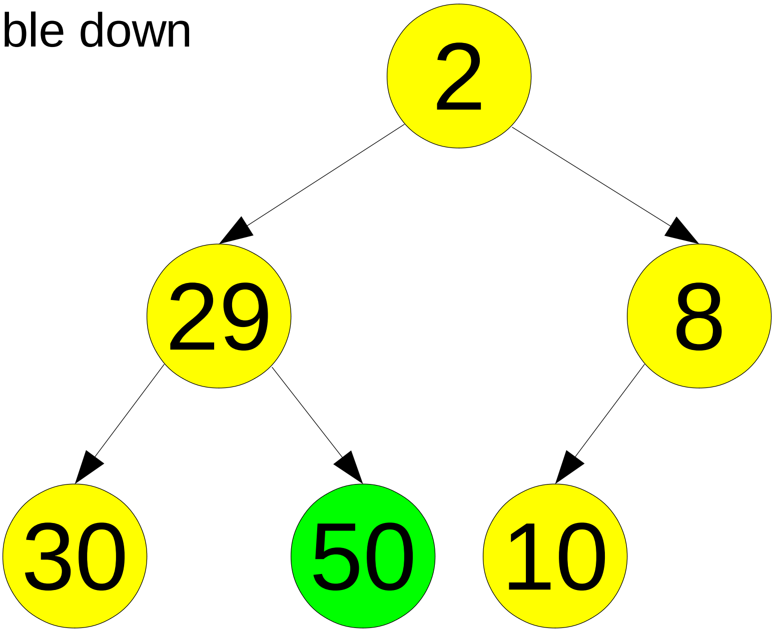
Replace With 50



Bubble up or down?



Bubble down



Adaptive PQ Algorithms

- Note: once we've figured out which bubble operation we need to do, we can continue doing that same operation.
- So both removing an element and changing a key are O(log(n))