Outline

Priority Queues

Priority Queues

- A Priority Queue is a container for items with different **priorities**.
- The interface of a Priority queue resembles that of a queue, since an item can be put into the priority queue (**enqueued**) at any time.
- The item with the highest priority is the first one to be removed from the priority queue (**dequeued**). Rather than first-in-first-out, as a normal queue, a priority queue is **best-in-first-out**.

Priority Queues

Priority Queues

Priority Queue ADT:

```
class PQueue(object):
   def enqueue(self, item, priority):
      '''post:  item is inserted with specified priority'''

   def first(self):
      ''''''post:  returns, but does not remove, highest priority
item'''

   def dequeue(self):
      ''''''post:  removes and returns the highest priority
item'''

   def size(self):
      '''post:  returns the number of items'''
```

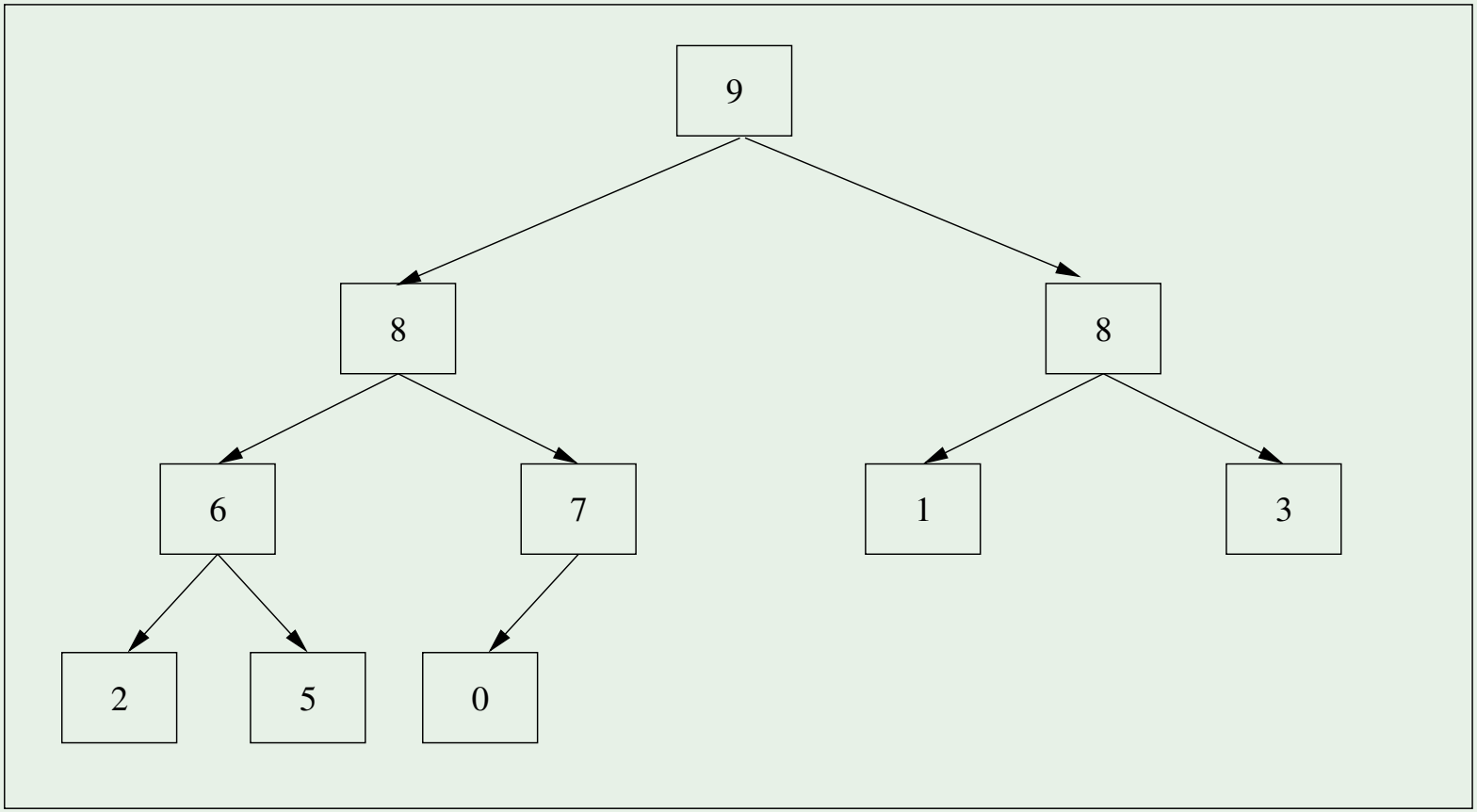
Heaps

For better performance, we use a Binary Heap:

- A complete binary tree, whose nodes are labeled with integer values (**priorities**).
- Has the Heap property:
  For any node, no node below it has a higher priority.
- The node with the highest priority is at the top of the heap!
- The `enqueue` method is called the `insert` method for the `Heap` class.
- The `dequeue` method is called the `delete_max` method for the `Heap` class.

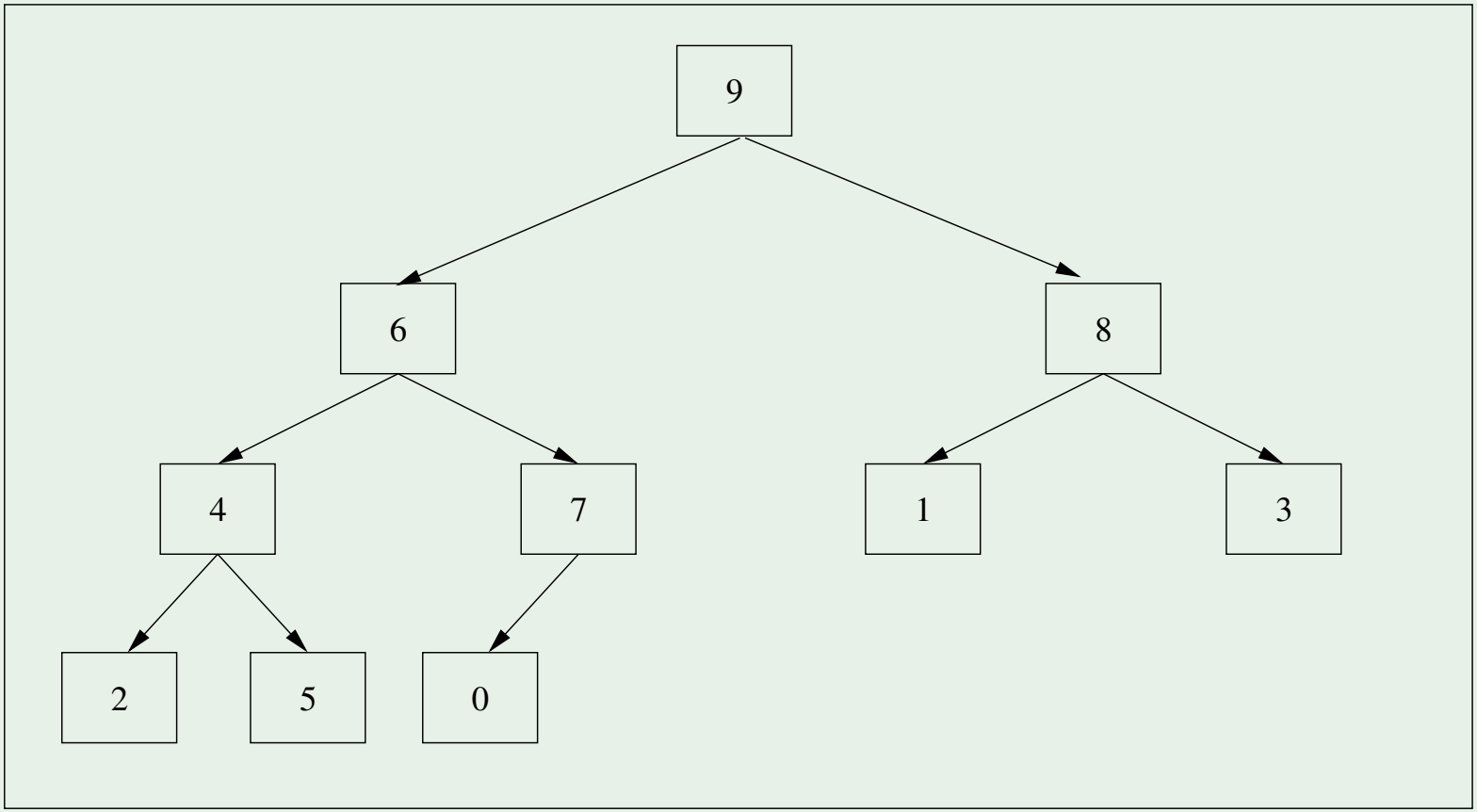
Heaps

A tree with the heap property



Heaps

A tree without the heap property



Heaps

Implementation:

- The `enqueue` and `dequeue` methods are implemented so they preserve the heap property.

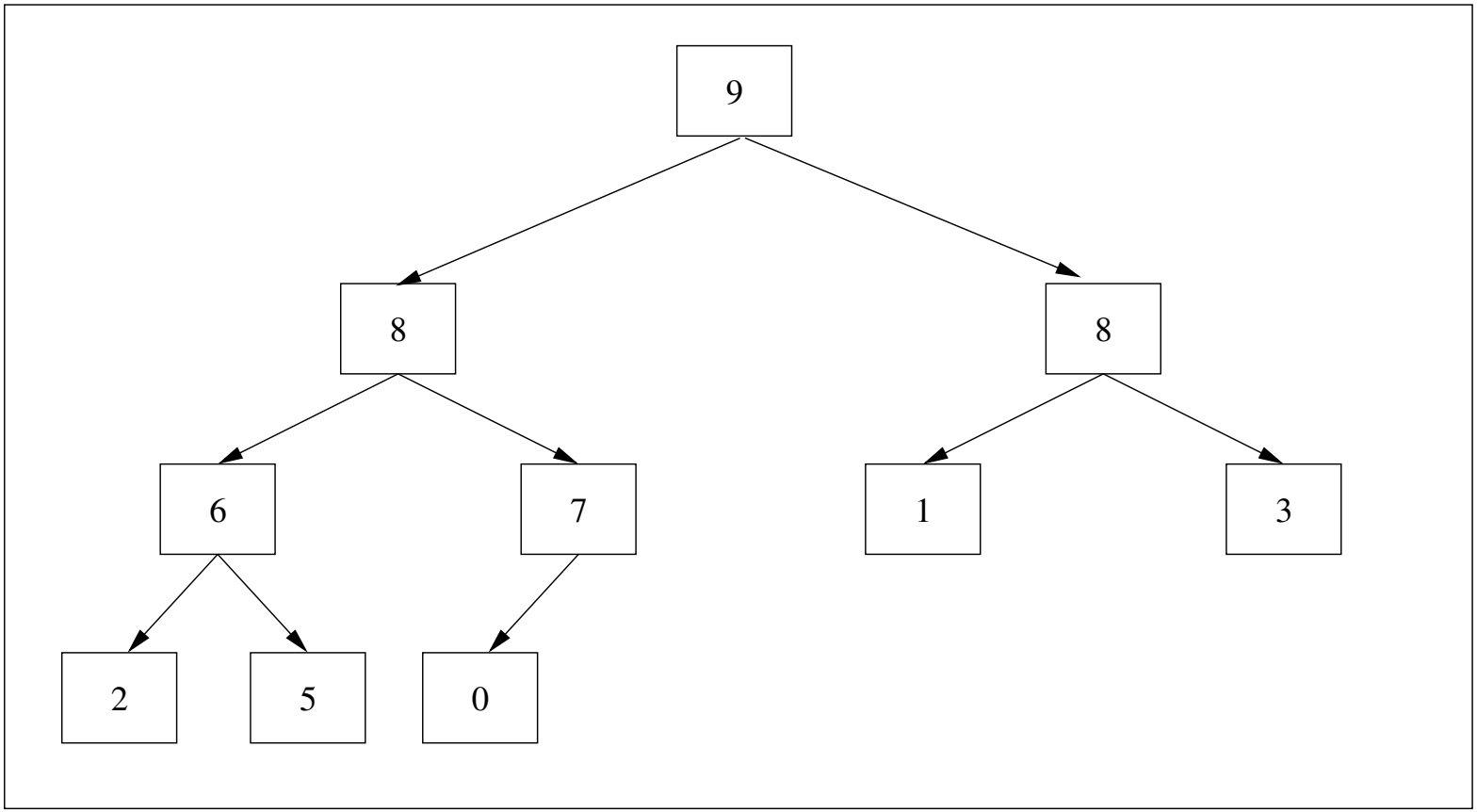
- To save space, the complete binary tree is implemented as an array.
  The root is at index 1.
  The children of the node at index `i` are at indexes `2 * i` and `2 * i + 1`.

- In Python: list class is used to implement binary heaps, so resizing will not be a problem when items are enqueued.

- in C++: dinamic arrays are used, and the class is defined as template class.

Heaps representation

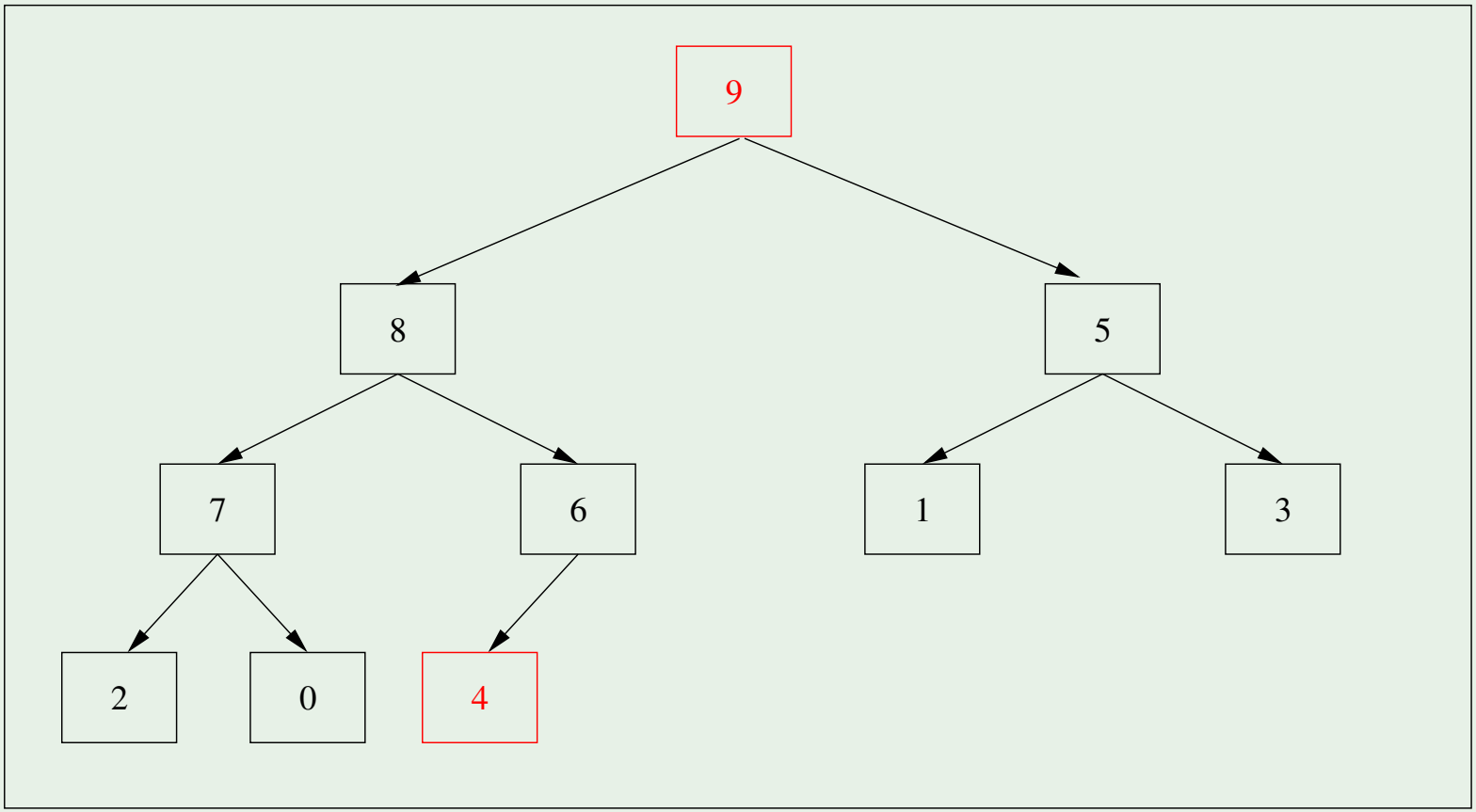
Heaps will be represented by arrays, with the root at index 1.



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

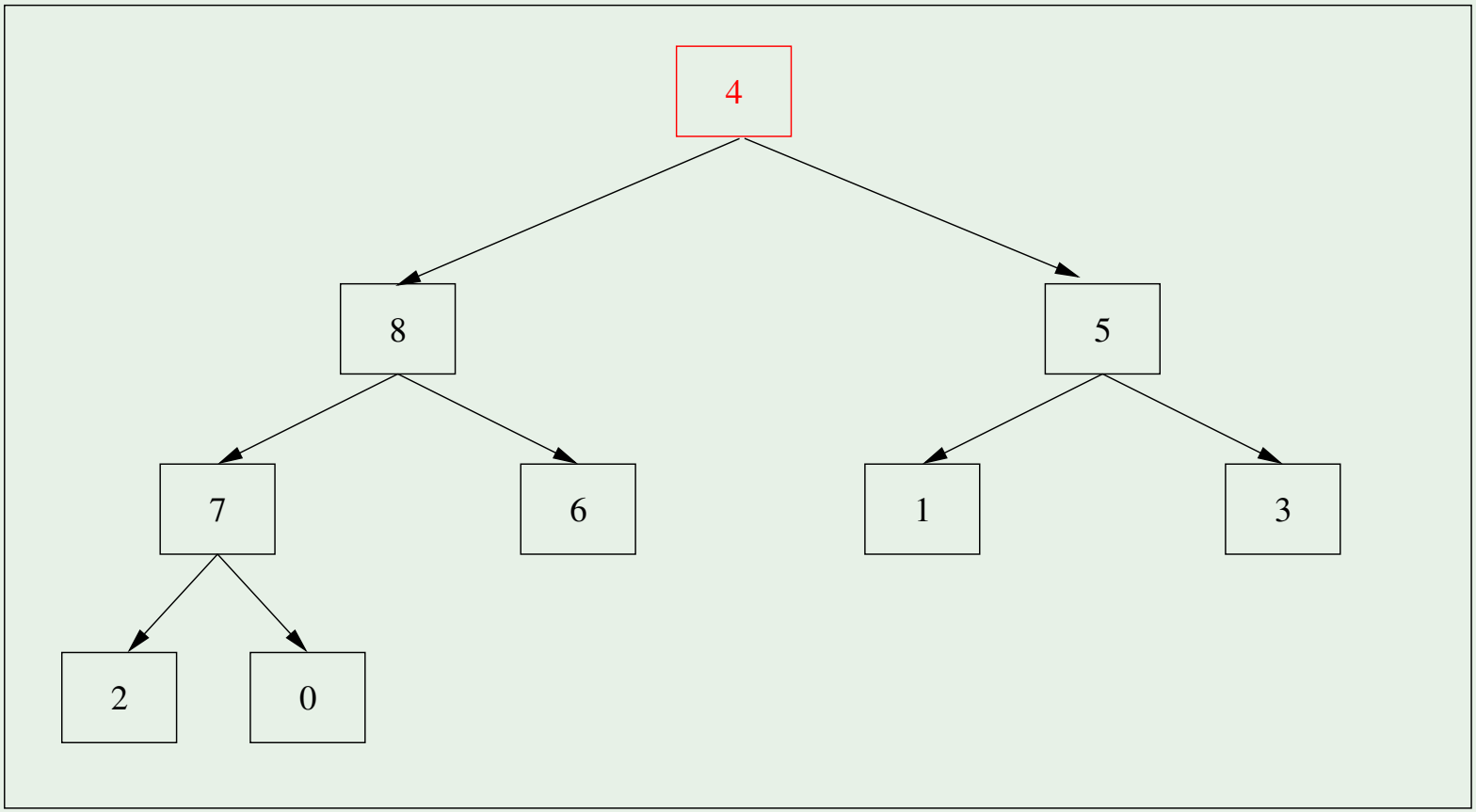
Heaps: operations delete_max and _heapify

Want to remove the highest priority item



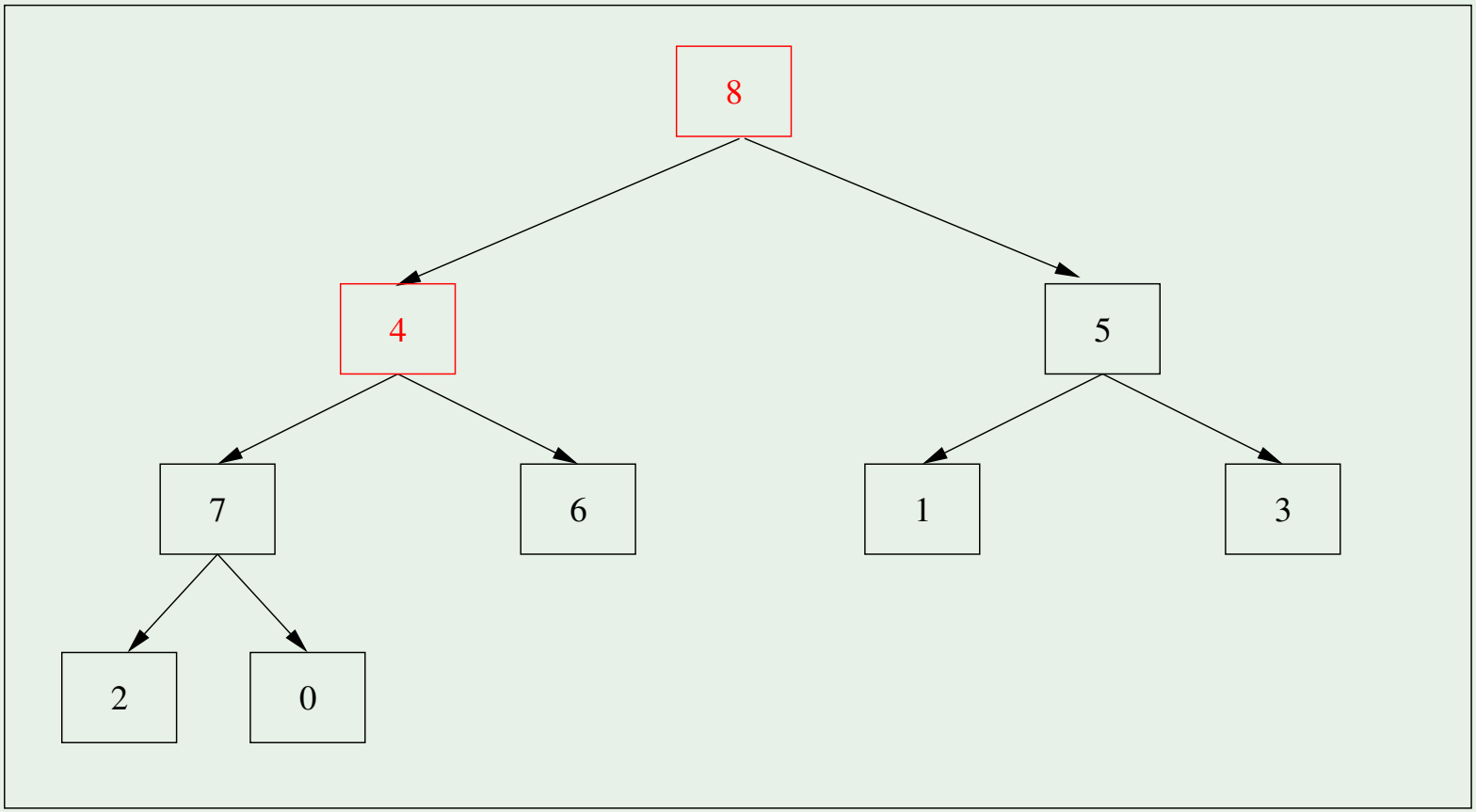
Heaps: operations delete_max and _heapify

Save top item and replace with last



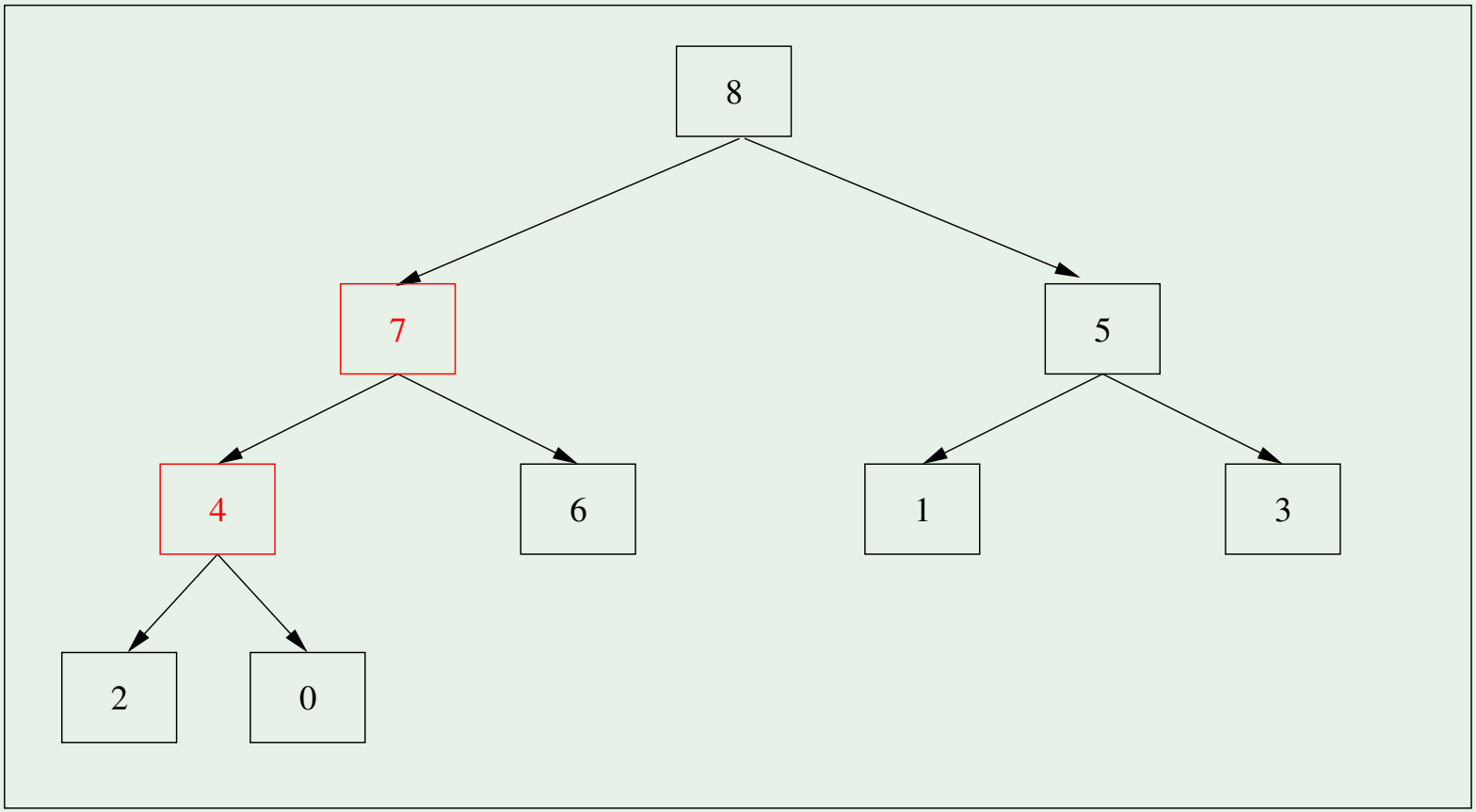
Heaps: operations delete_max and _heapify

Percolate down until...



Heaps: operations delete_max and _heapify

The heap property is restored



Heaps: operations delete_max and _heapify

```
def delete_max(self):
    '''pre:  heap property is satisfied, self.heap is the list of
items with top element at index 1.
    post:  maximum element in heap is removed and returned'''

    if self.heap_size > 0:
        max_item = self.heap[1]
        self.heap[1] = self.heap[self.heap_size]
        self.heap_size -= 1
        self.heap.pop()
        if self.heap_size > 0:
            self._heapify(1)    # brings the swapped element into
the appropriate position
        return max_item
```

Heaps: operations delete_max and _heapify

Heapify

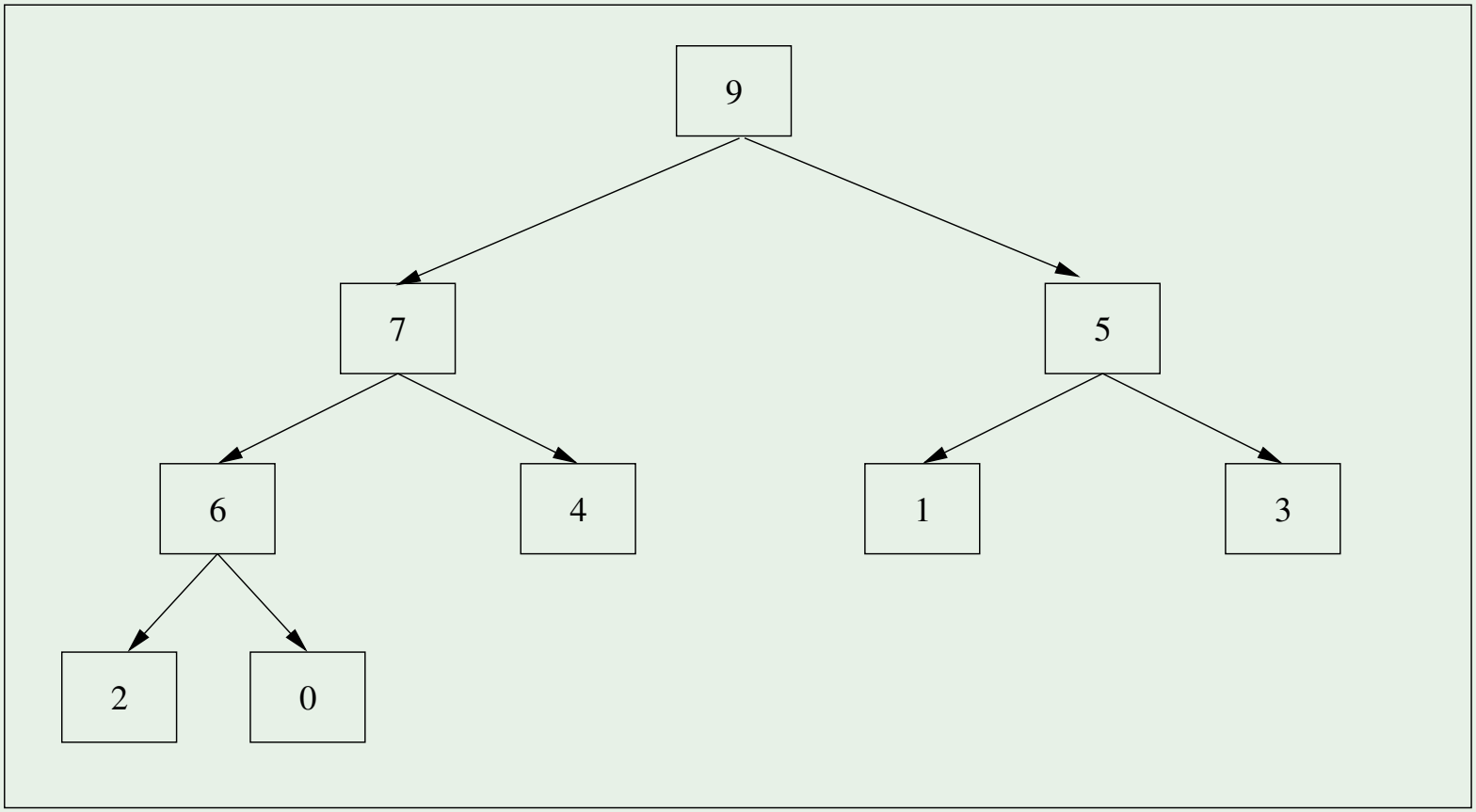
_heapify starts at the node and moves its value down the tree by swapping it with the higher priority child.

Heaps: operations delete_max and _heapify

```
def _heapify(self, position):
 '''pre:  heap property is satisfied below position
 post:  heap property is satisfied at and below position'''
 item = self.heap[position]
 while position * 2 <= self.heap_size:
    child = position * 2
    # if right child exists, determine maximum of two children
    if (child != self.heap_size and
       self.heap[child+1] > self.heap[child]):
       child += 1
    if self.heap[child] > item:
       self.heap[position] = self.heap[child]
       position = child
    else:  break
 self.heap[position] = item
```

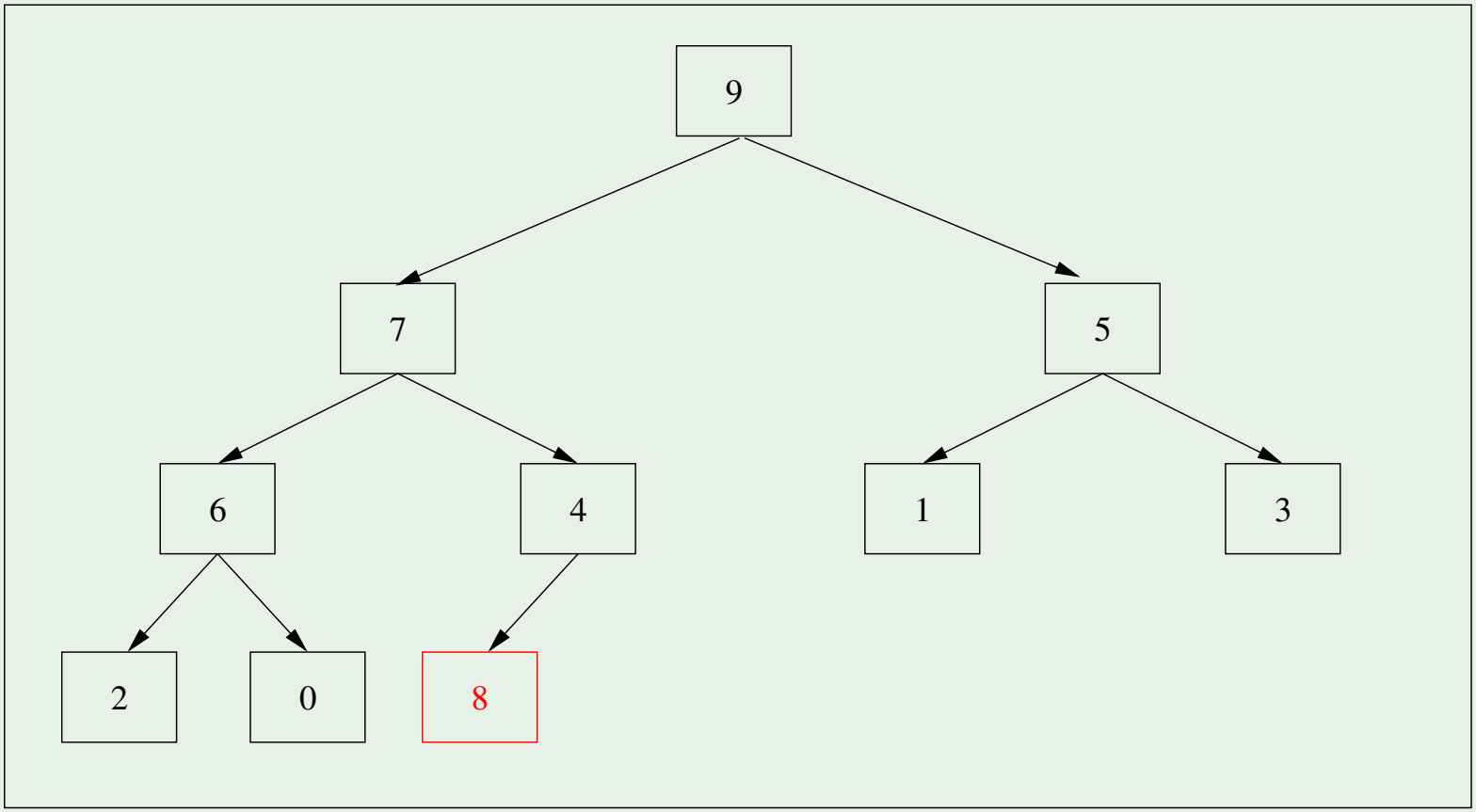
Heap: operation insert

Want to insert item with priority 8



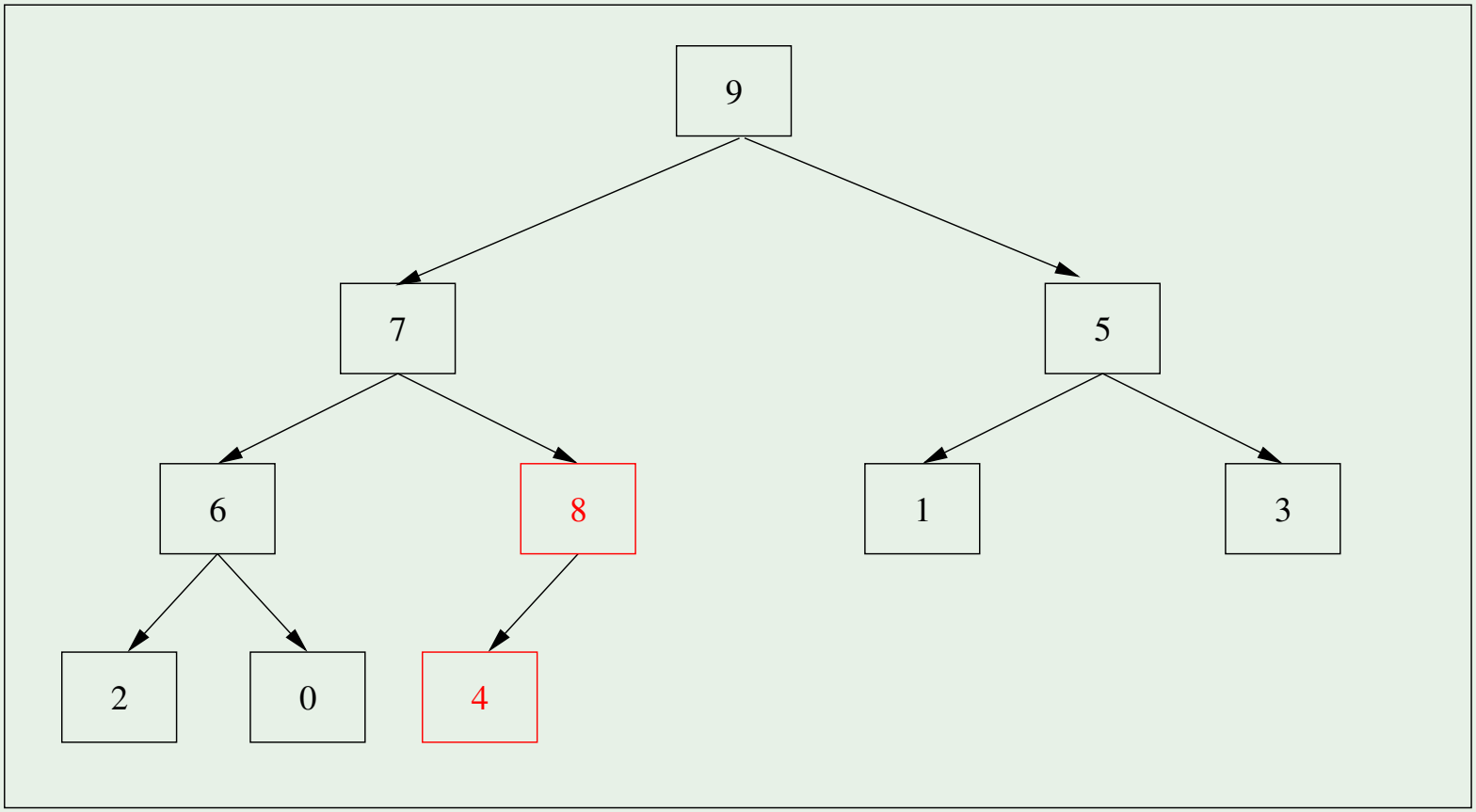
Heap: operation insert

Add the new item at the end



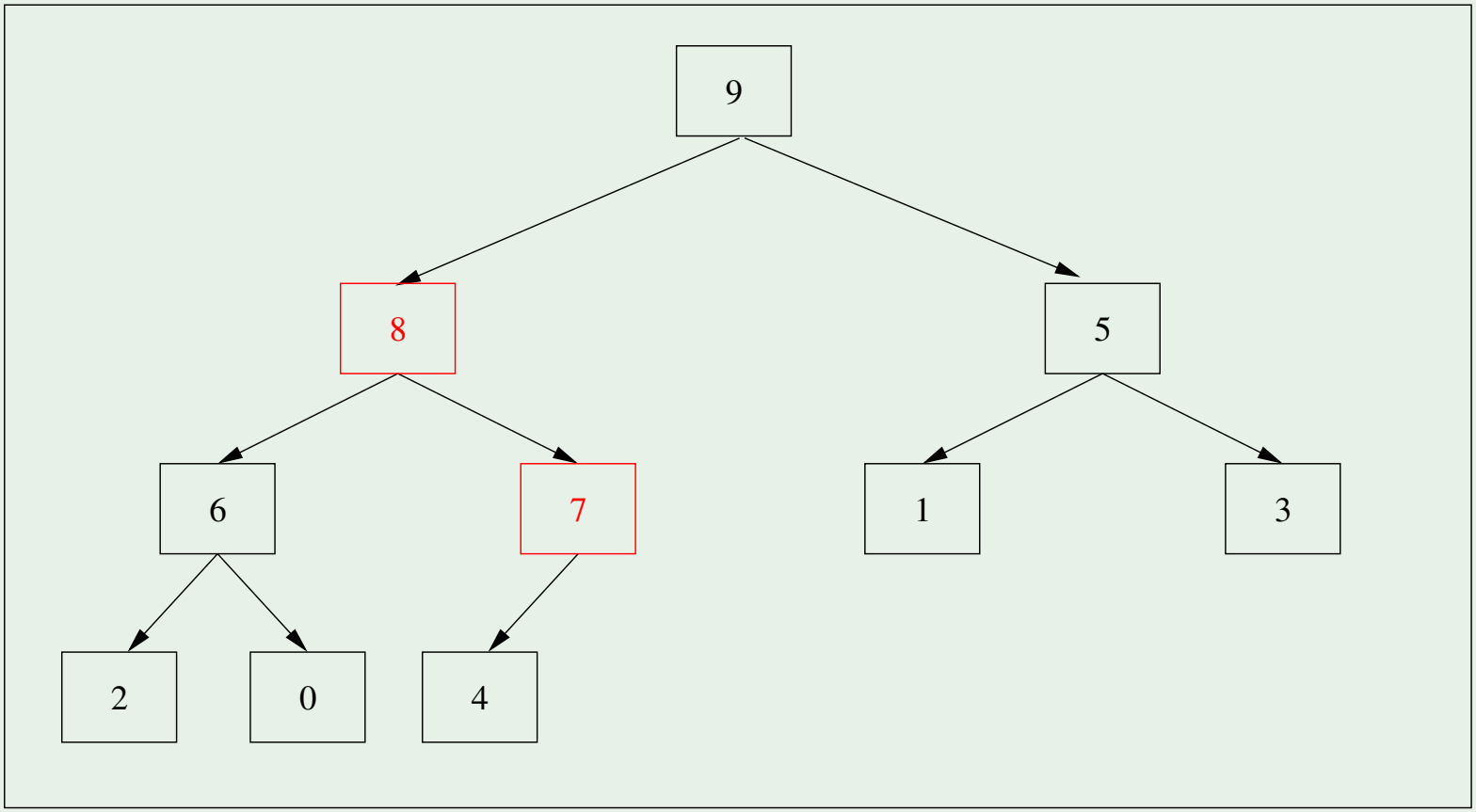
Heap: operation insert

Percolate up until...



Heap: operation insert

The heap property is restored



Heap: operation insert

```
def insert(self, item):
    '''pre:  heap property is satisfied
    post:  item is inserted in proper location in heap'''
    self.heap_size += 1
    self.heap.append(None) # extend the length of the list
    position = self.heap_size
    parent = position // 2
    while parent > 0 and self.heap[parent] < item:
        # move the parent's item down
        self.heap[position] = self.heap[parent]
        position = parent
        parent = position // 2
    self.heap[position] = item # put new item in correct spot
```

__INIT__ AND _BUILD_HEAP

```
def __init__(self, items=None):
    '''post:  a heap is created with specified items'''
    self.heap = [None]
    if items is None:
        self.heap_size = 0
    else:
        self.heap += items
        self.heap_size = len(items)
        self._build_heap()
```

__INIT__ AND _BUILD_HEAP

```
def _build_heap(self):
    '''pre:  self.heap has values in 1 to self.heap_size
    post:  heap property is satisfied for entire heap'''

    # 1 through self.heap_size
    for i in range(self.heap_size // 2, 0, -1):  # stops at 1
        self._heapify(i)
```

__INIT__ AND _BUILD_HEAP

Since the tree is complete, it's height is $\lg n$.

The **insert** and **delete_max** operations are $\Theta(\lg n)$.

Hence, if we use binary heap to implement the priority queue, the **enqueue** and **dequeue** operations will be $\Theta(\lg n)$.

Heapsort

We can use the _heapify and delete_max methods to sort items in $\Theta(n * \log n)$.

The heap size decreases each time an item is removed, - let's use this space.

We delete the max element from the heap, place it at the last spot in the heap before the item was removed. After we have removed all the items except one, the resulting array is sorted.

```
def heapsort(self):
    '''pre:  heap property is satisfied
    post:  items are sorted'''
    sorted_size = self.heap_size
    for i in range(0, sorted_size - 1):
        # Since delete_max calls pop to remove an item,
        # append dummy value to avoid an illegal index.
        self.heap.append(None)
        item = self.delete_max()
        self.heap[sorted_size - i] = item
```

Running times

Running times:

- `insert` is $\Theta(\log\ n)$.
- `delete_max` is $\Theta(\log\ n)$.
- `_heapify` is $\Theta(\log\ n)$.
- `_build_heap` is $\Theta(n)$.
- `heapsort` is $\Theta(n \log\ n)$.

Notes on Heap and Priority Queue Implementations

Using Python

- Use the `Heap` class as defined in this chapter.
- The `enqueue` method is called the `insert` method for the `Heap` class.
- The `dequeue` method is called the `delete_max` method for the `Heap` class.
- Node data will be tuples: `(priority, item)`;
  `(priority1, item1) < (priority2, item2)`
  will be interpreted as
  `priority1 < priority2`

Notes on Heap and Priority Queue Implementations

Using C++

- Write the `Heap` class as a C++ template class with private `priority` and `item` data members.
- Overload `<` and other comparison operators to compare priorities.
- Or just use the Priority Queue template class from the Standard Template Library.

Python Dictionaries

Various names are given to the abstract data type we know as a dictionary in Python:

- Hash (The languages Perl and Ruby use this terminology; implementation is a hash table).
- Map (Microsoft Foundation Classes C/C++ Library; because it maps keys to values).
- Dictionary (Python, Smalltalk; lets you "look up" a value for a key).
- Association List (LISP—everything in LISP is a list, but this type is implemented by hash table).
- Associative Array (This is the technical name for such a structure because it looks like an array whose 'indexes' in square brackets are key values).

Python Dictionaries

In Python, a dictionary associates a value (item of data) with a unique key (to identify and access the data). The implementation strategy is the same in any language that uses associative arrays. Representing the relation between key and value is a hash table.

Python Dictionaries

The Python implementation uses a hash table since it has the most efficient performance for *insertion*, *deletion*, and *lookup* operations.

The running times of all these operations are better than any we have seen fo far. For a hash table, these are all $\Theta(1)$, taking a constant time to run. (If the table gets larger, the running times do not increase.)

This is done by calculating the address of any item, stored in an array, from its key value. The function used to calculate this address is called a hash function.

Hashing Functions

- For keys which are already numeric (integers), they can be divided by the size of the array. The remainder becomes the index into the fixed array.
- Text keys must be transformed into integers.
  For example: the characters' ASCII codes can be added together, then divided and the remainder is taken.

  Functions like this scatter the items through the array.
  If the function spreads the items 'randomly' over the array, there are few problems until the array starts to fill up (when the load factor—the filled fraction of the array–becomes one-half or greater).

  A collision is when a key having some value gets transformed into the same address as an existing key.
  Where can the value for the new key go, so it can be found later?

Collision Resolution

There are different strategies for resolving collisions.

- Open addressing–Linear Probe
- Open addressing–Quadratic Probe
- Double Hashing
- Separate Chaining

Collision Resolution

Open addressing–Linear Probe

- If a hash function produces a slot address that is already in use, a linear function is used to calculate an alternate location for the new key's data. This is repeated until a free slot is found.
- When the new key is used to access the data, the original address is inspected. If the data is not found, the linear function is used to calculate the next likely location for the data. This is repeated until the data is found.
- This policy can lead to a high collision rate as items cluster around a few locations.

Collision Resolution

Open addressing–Quadratic Probe

- If a hash function produces a slot address that is already in use, a quadratic function is used to calculate an alternate location for the new key's data. This is repeated until a free slot is found.
- When the new key is used to access the data, the original address is inspected. If the data is not found, the quadratic function is used to calculate the next likely location for the data. This is repeated until the data is found.
- This policy can lead to fewer collisions than a linear probe policy.

Collision Resolution

Double Hashing

- If a hash function produces a slot address that is already in use, an alternate hashing function is used to calculate an alternate location for the new key's data. This is repeated until a free slot is found.
- When the new key is used to access the data, the original address is inspected. If the data is not found, the alternate hashing function is used to calculate the next likely location for the data. This is repeated until the data is found.
- This policy can lead to fewer collisions than a quadratic probe policy.

Collision Resolution

Separate Chaining

- If a hash function produces a slot address that is already in use, a linked list is begun at that slot which contains all the data for colliding keys at that slot. Subsequent keys that hash to that same slot are appended to the linked list.
- When the new key is used to access the data, the original address is inspected. If there is a linked list at that slot, the list is searched for that key's data.
- This policy still leads to $\Theta(1)$ performance as long as the load factor for the table is not too high.

In-class Work / suggested homework

In-class Work / suggested homework

Implement heapsort in C++ version of Heap class.

Do the problem 4, from Part II from the **Final Exam Sample**