CSI33 Data Structures

Department of Mathematics and Computer Science
Bronx Community College

November 27, 2017

Outline

1. Chapter 13: Heaps, Balanced Trees and Hash Tables
   - Priority Queues and Heaps

Outline

1. Chapter 13: Heaps, Balanced Trees and Hash Tables
   - Priority Queues and Heaps

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

Applications:

- A hospital emergency room.
- An event handler in a computer's operating system. Different processes running at the same time share access to the CPU. Essential services have higher priority than user applications.
- Pattern-matching algorithms (voice or handwriting recognition) where input is compared with stored patterns. The best matches will get the highest scores and saved in a priority queue for further processing.

Priority Queues

This would be the interface to a Python class implementing the 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'''
```

Implementing a Priority Queue As A Heap

Worst-case running times for structures we have seen:

- Sorted list: `enqueue` is $\Theta(n)$. An array would allow $\Theta(\log n)$ to find the position (Binary search), but $\Theta(n)$ is needed to insert by moving the higher items out of the way.
- Linked list: `enqueue` or `dequeue` is $\Theta(n)$. If the list is sorted, `enqueue` takes $\Theta(n)$ to find the position at which to insert the item. Otherwise, `dequeue` takes $\Theta(n)$ to go through all items in an unsorted list to find the highest priority item.

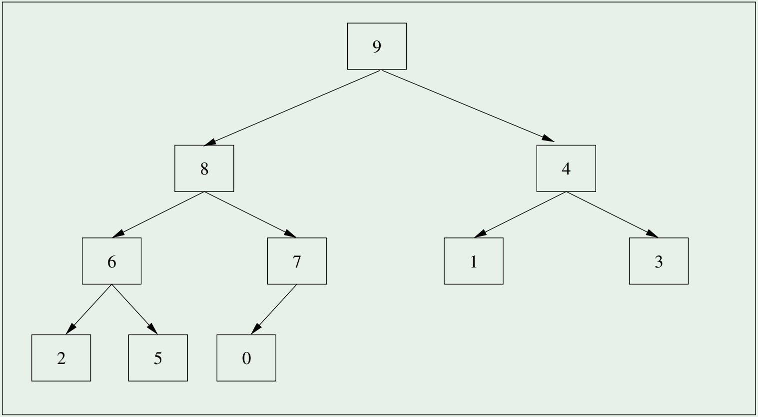
Implementing a Priority Queue As A Heap

For better performance, we use a new structure; 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.
- Notice how fast it is to find the node with the highest priority (it's 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.

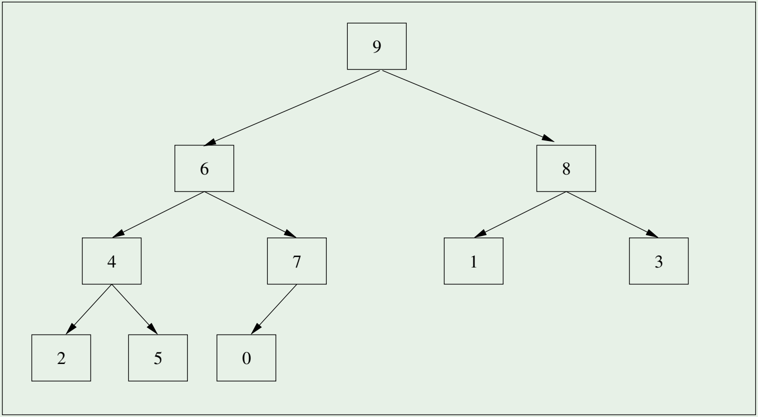
Implementing a Priority Queue As A Heap

A tree with the heap property



Implementing a Priority Queue As A Heap

A tree without the heap property



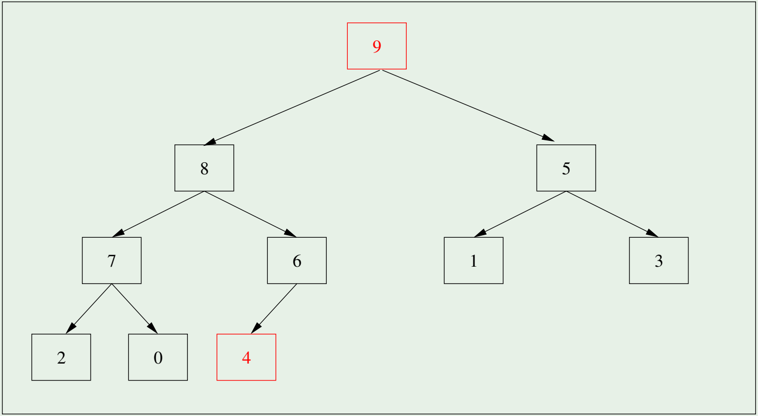
Implementing a Priority Queue As A Heap

Implementation issues:

- 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`.)
- We will use Python and its list class to implement binary heaps, so resizing will not be a problem when items are enqueued.

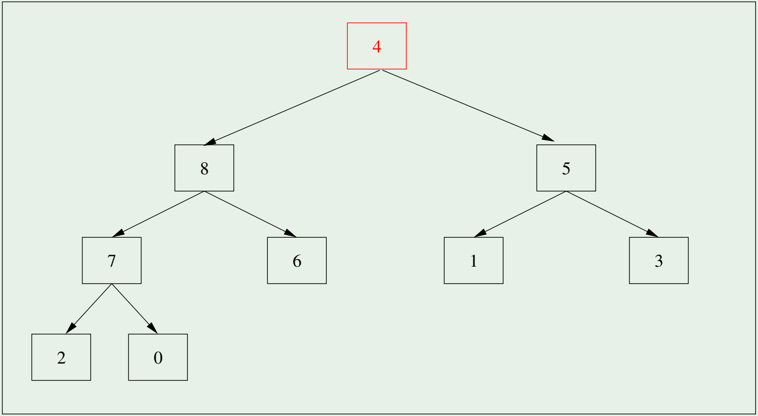
DELETE_MAX

Want to remove the highest priority item



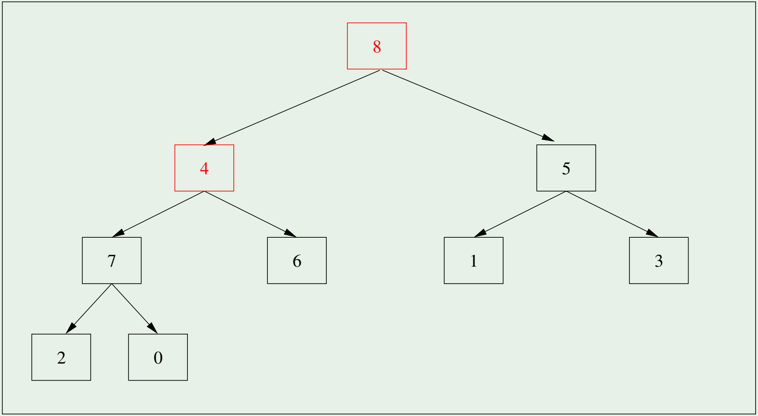
DELETE_MAX

Save top item and replace with last



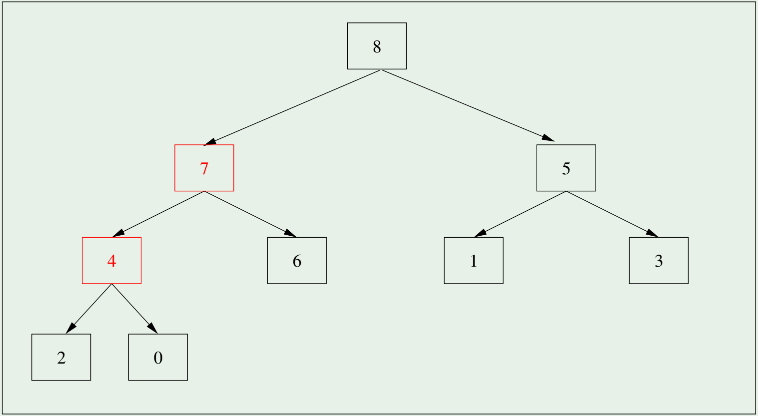
DELETE_MAX

Percolate down until...



DELETE_MAX

The heap property is restored



DELETE_MAX

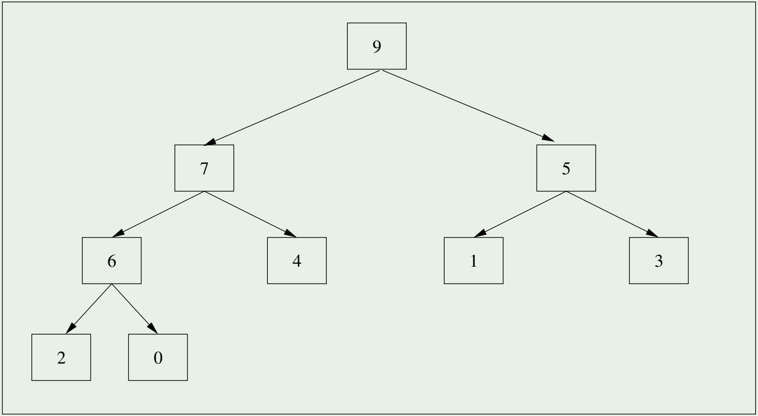
```
def delete_max(self):
    '''pre:  heap property is satisfied
    post:  maximum element in heap is removed and returned'''
    if self.heap_size > 0:
        max_item = self.heap[1]
        self.heap[1] = self.heap[self.heap_size]
        self.heap_size -= 1
        self.heap.pop()
        if self.heap_size > 0:
            self._heapify(1)
        return max_item
```

DELETE_MAX

```
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, 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
```

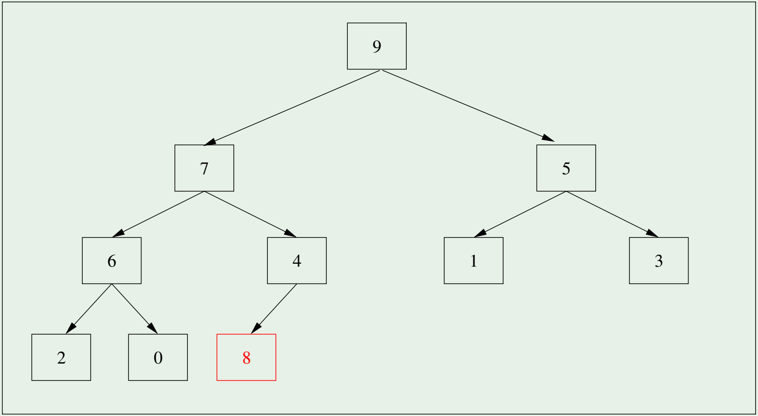
INSERT

Want to insert item with priority 8



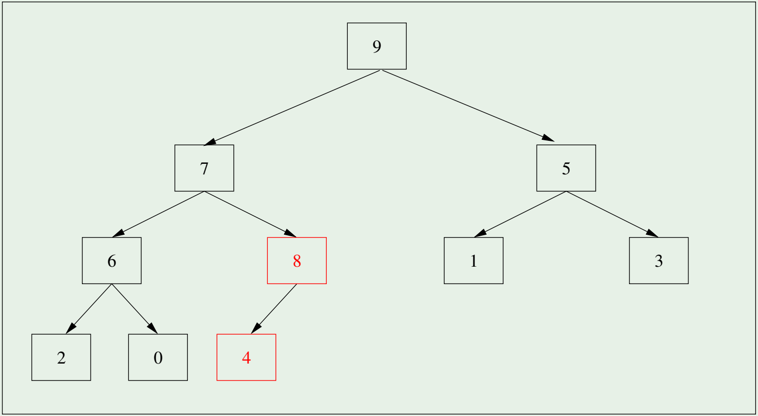
INSERT

Add the new item at the end



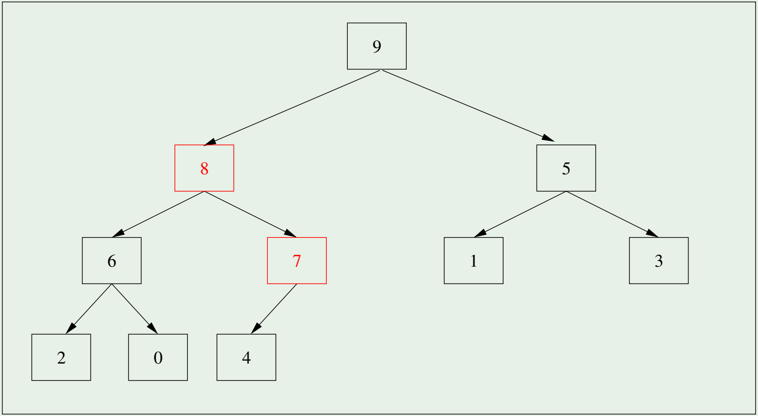
INSERT

Percolate up until...



INSERT

THE HEAP PROPERTY IS RESTORED



INSERT

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

__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)
```

Heapsort

```
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
```

Heapsort

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)`; Python will interpret `(priority1, item1) < (priority2, item2)` 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.