Outline

1. Chapter 13: Heaps, Balanced Trees, and Hash Tables
   - Priority Queues and Heaps
   - Binary 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 Queue in Python

This would be the interface to a Python class implementing the Priority Queue ADT:

```
class PQueue:
   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 (by priority) 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 linked list is sorted by priority, it takes $\Theta(n)$ to find the position at which to insert the item, and $\Theta(1)$ to insert it.
  Otherwise (if we will append items at the end of the list and search by highest priority), `dequeue` takes $\Theta(n)$ to go through all items in an unsorted list to find the highest priority item.

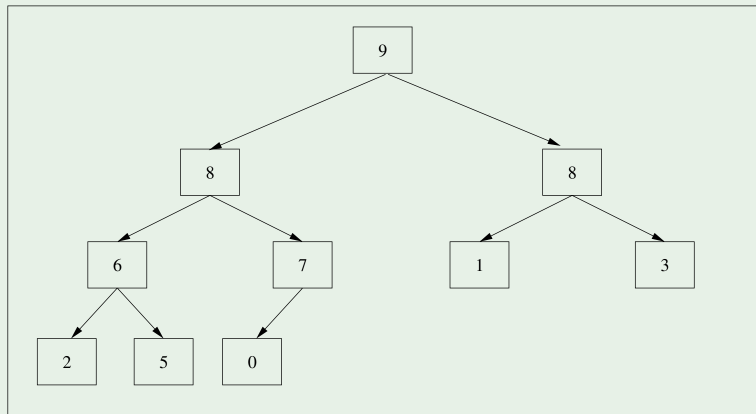
Implementing a Priority Queue As A Heap

For better performance, we will 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.
- 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

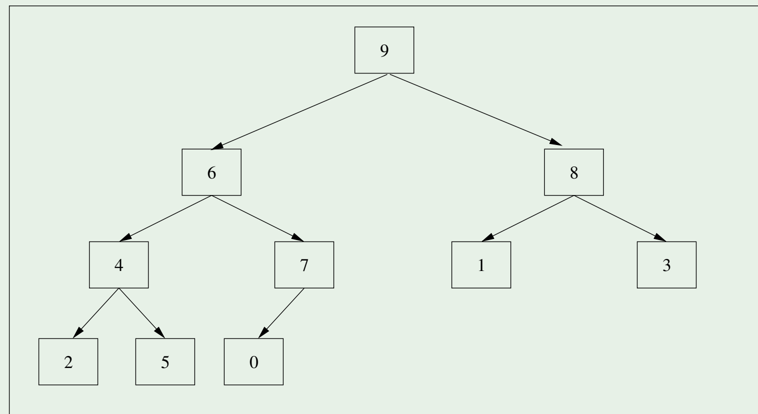
A tree with the heap property



Notice how fast it is to find the node with the highest priority (it's at the top of the heap).

Implementing a Priority Queue As A Heap

A tree without the heap property



Binary Heaps

Binary Heap

- is a complete binary tree, whose nodes are labeled with integer values (**priorities**), that
- has the Heap property:
  For any node, no node below it has a higher priority.
- Therefore, the highest priority node is at the top of the heap!
- `insert` method for the `Heap` class to insert elements, and
- `delete_max` method to delete the top of the heap.

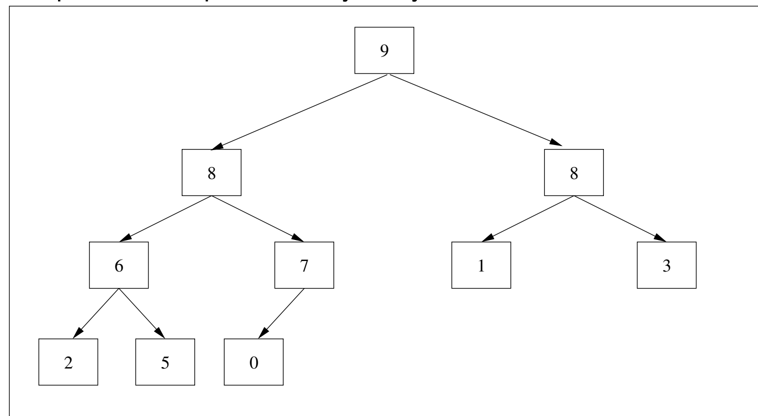
Binary Heaps

Implementation:

- The `insert` and `delete_max` 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++: dynamic 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	...

Heap: constructor in Python: __init__

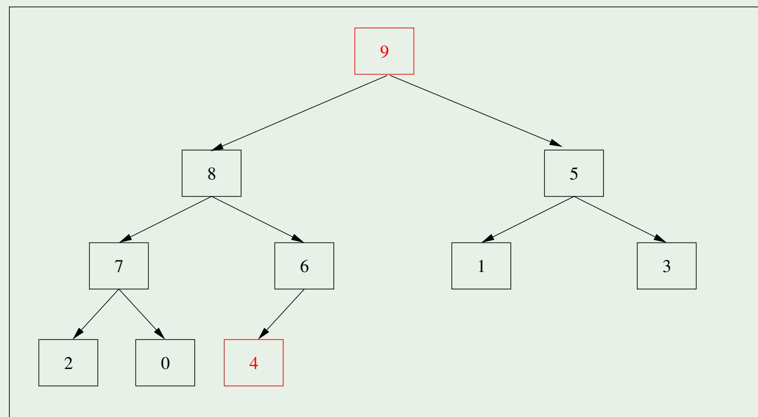
```
def __init__(self, items=None):
  '''post:  a heap is created with specified
items'''
  self.heap = [None]
  if items is None:
    self.heap_size = 0
  else:
    self.heap += items
    self.heap_size = len(items)
    self._build_heap()
```

Heap: class declaration in C++:

```
  template <typename Item>
class Heap{
public:
   Heap(const Item seq[]=NULL, const int &size=0);
   ~ Heap();
   unsigned int size() const { return heap_size; }
   void asList(std::string &s);
private:
   Heap(const Heap<Item> &h);
   Heap<Item>& operator=(const Heap<Item> &h);
   void _build_heap();
   void resize(unsigned int newSize);
   Item *items_;
   unsigned int heap_size, capacity_; };
#include "Heap.template"
```

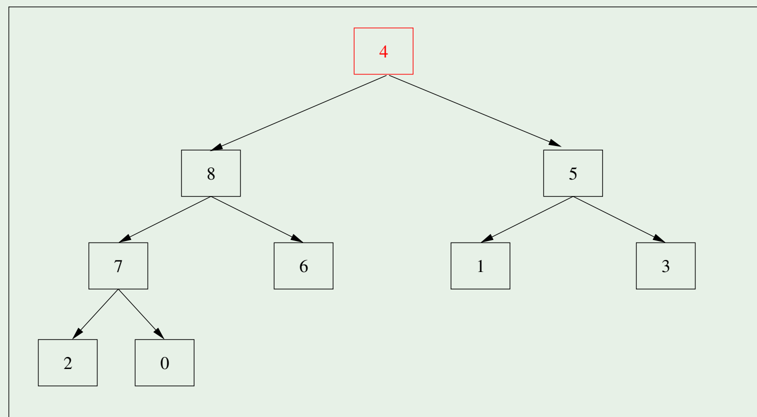
Heaps: operations delete_max and _heapify

Removing the highest priority item



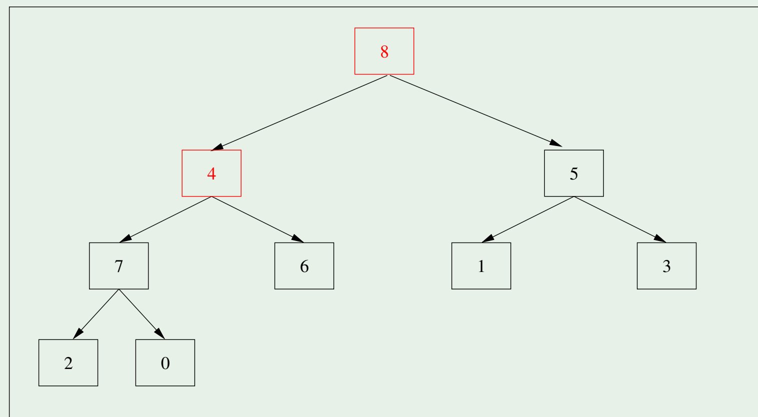
Heaps: operations delete_max and _heapify

Save top item and replace with last



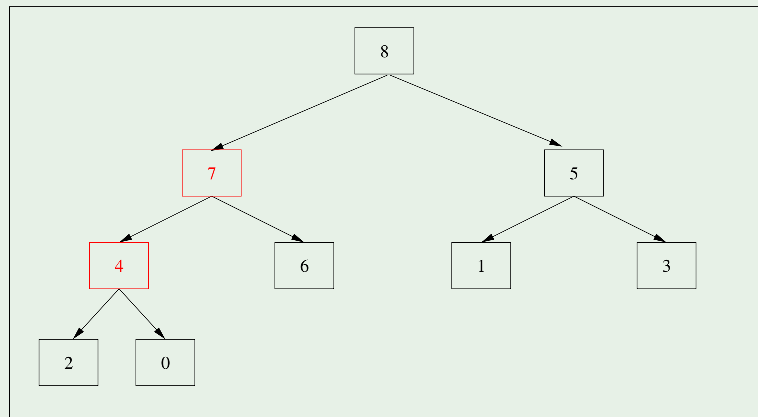
Heaps: operations delete_max and _heapify

Percolate down until...



Heaps: operations delete_max and _heapify

The heap property is restored



Heaps: operation delete_max in Python

```
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: operation delete_max in C++

```
template <typename Item>
Item Heap<Item>::delete_max(){
   Item max_item;
   if (heap_size > 0) {
      max_item = items_[1];
      items_[1] = items_[heap_size];
      heap_size -= 1;
      if (heap_size > 0){
         _heapify_(1);
      }
   }
   return max_item;
}
```

Heaps: operation Heapify in Python

Heapify

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

Heaps: operation Heapify in Python

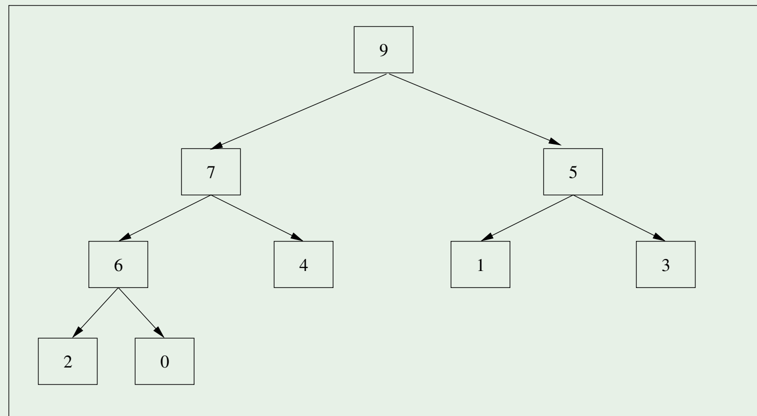
```
def _heapify(self, position):
 '''pre:  heap property is satisfied below position
 post:  heap property is satisfied at and below position'''
 item = self.heap[position]
 while position * 2 <= self.heap_size:
    child = position * 2
    # if right child exists, determine maximum of two children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

Heaps: operation Heapify in Python

```
  template <typename Item>
void Heap<Item>::_heapify_(unsigned int position) {
   Item it;
   unsigned int child;
   it = items_[position];
   while (position * 2 <= heap_size) // if left child exists
   {    child = position * 2; // left child
      if (child != heap_size && items_[child + 1] >
items_[child]) { child += 1;}
      if (items_[child] > it) {
         items_[position] = items_[child];
         position = child; } //advance the position
      else { break; }
      items_[position] = it;
   } }
```

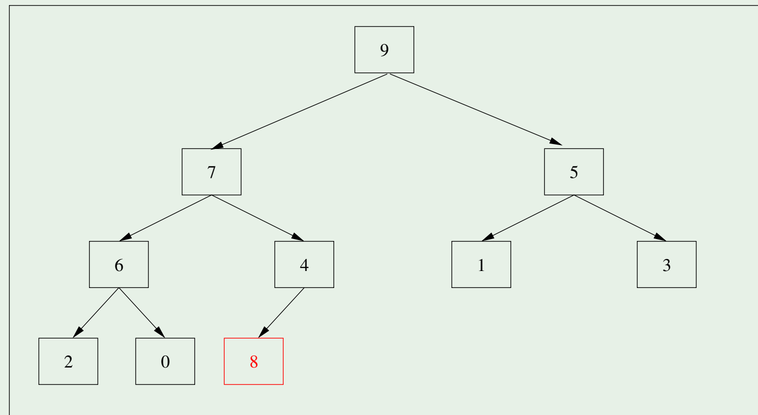
Heap: operation insert

Want to insert item with priority 8



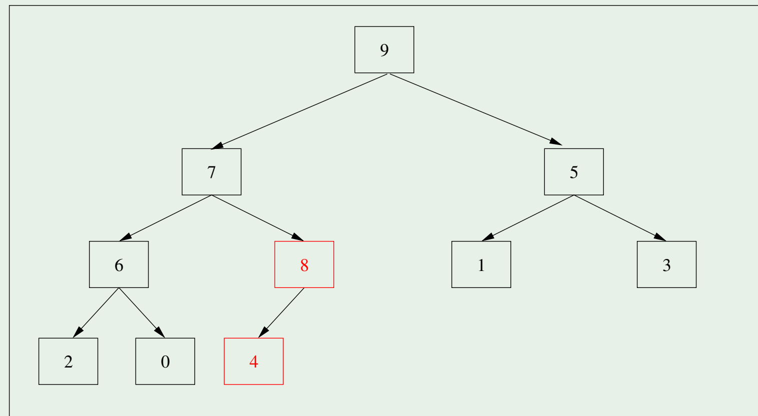
Heap: operation insert

Add the new item at the end



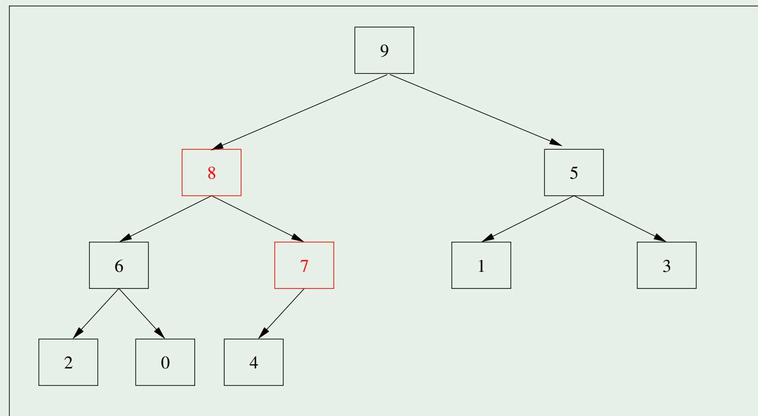
Heap: operation insert

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
```

Heap: operation insert

```
template <typename Item>
void Heap<Item>::insert(Item a) {
   if(heap_size + 1 > capacity_-1) {
      resize(2*heap_size+1); }
   heap_size += 1;
   int position = heap_size, parent = position / 2;
   while (parent > 0 && items_[parent] < a){
      items_[position] = items_[parent];
      position = parent;
      parent = position / 2; }
   items_[position] = a; }
```

_build_heap in Python and C++

```
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)
```

```
template <typename Item>
void Heap<Item>::_build_heap() {
for (unsigned int i = heap_size / 2; i > 0; i--) {
    _heapify_(i); }
```

TIME ANALYSIS

The tree is complete, hence 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.

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
```

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.
- Define `Node` class, with two attributes: `priority, item`; To compare two Node instances, `Node1 < Node2`, compare their priorities, i.e.
  `priority1 < priority2`

Notes on Heap and Priority Queue Implementations

Using C++

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