Outline

1. Chapter 13: Heaps, Balances Trees and Hash Tables
   - Priority Queues and Heaps (continues)
   - Hash Tables
   - In-class Work / Suggested homework

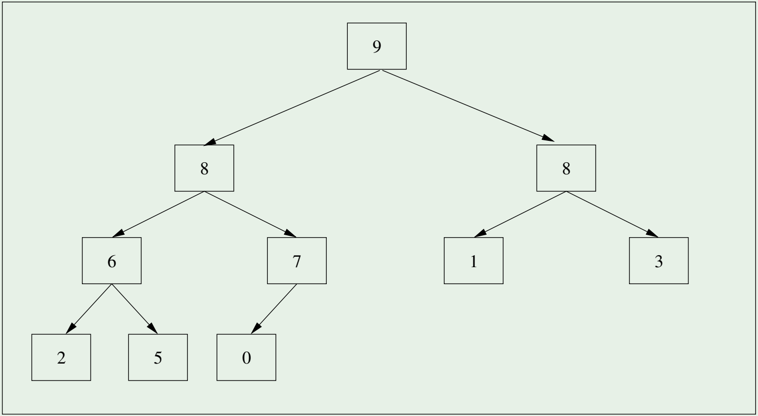
Binary Heaps

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.

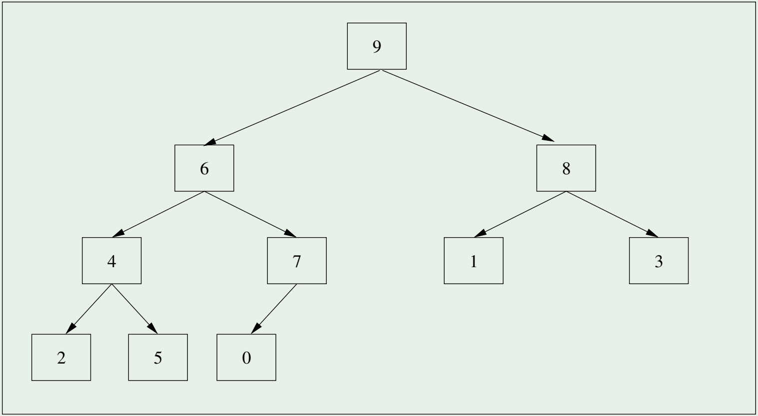
Binary Heaps

A tree with the heap property



Binary Heaps

A tree without the heap property



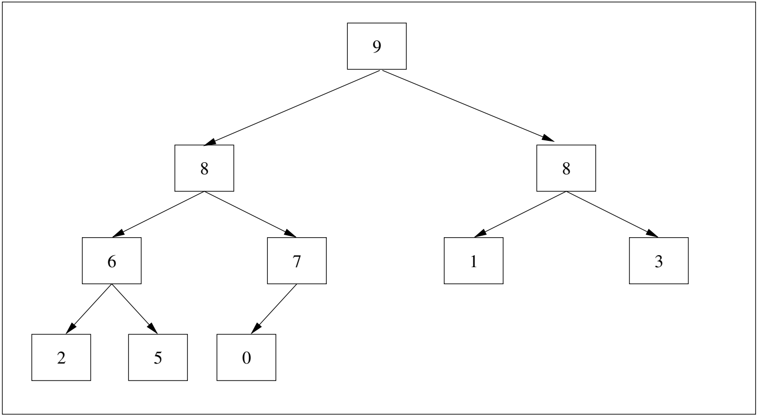
Binary Heaps

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 are 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__

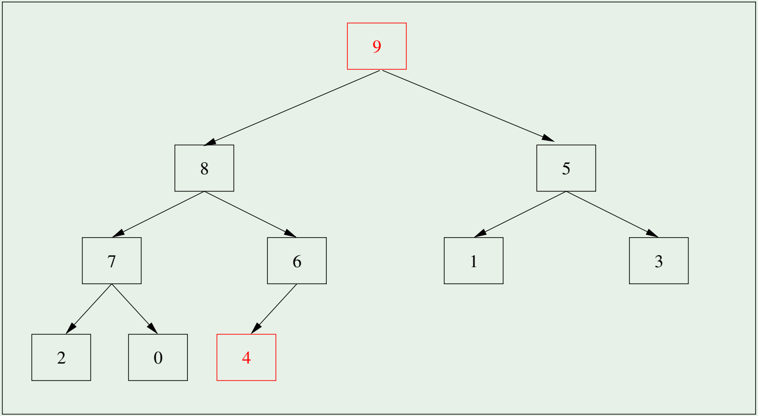
```
  def __init__(self, items=None):
   '''post:  a heap is created with specified
items'''
   self.heap = [None]
   if items is None:
      self.heap_size = 0
   else:
      self.heap += items
      self.heap_size = len(items)
      self._build_heap()
```

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"
```

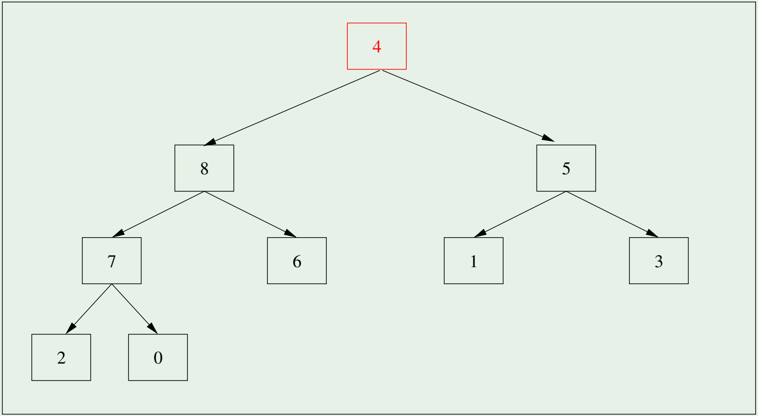
Heaps: operations `delete_max` and `_heapify`

Removing the highest priority item



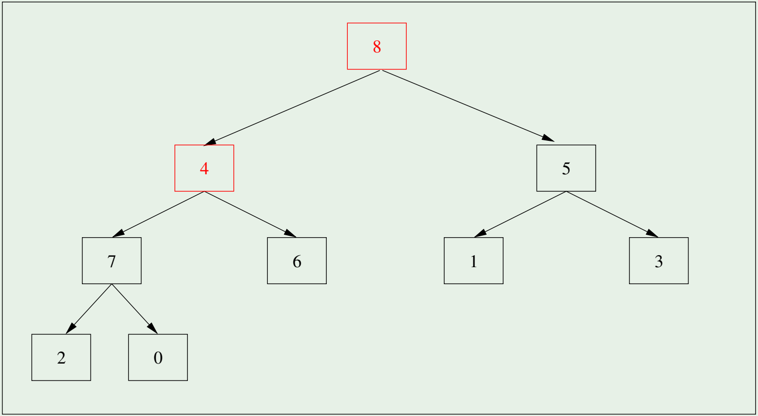
Heaps: operations `delete_max` and `_heapify`

Save top item and replace with last



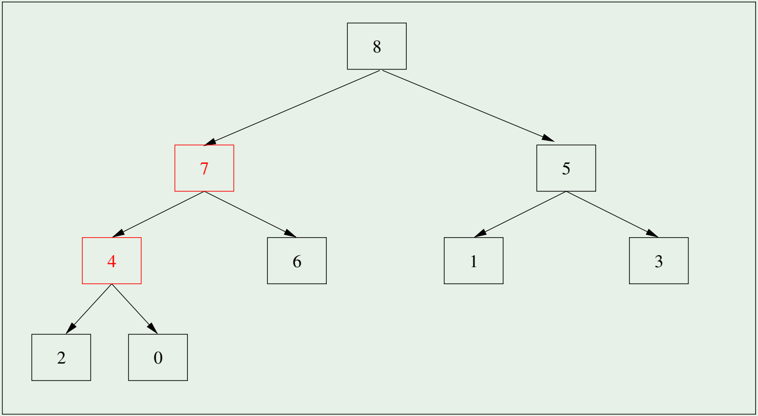
HEAPS: OPERATIONS `DELETE_MAX` AND `_HEAPIFY`

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

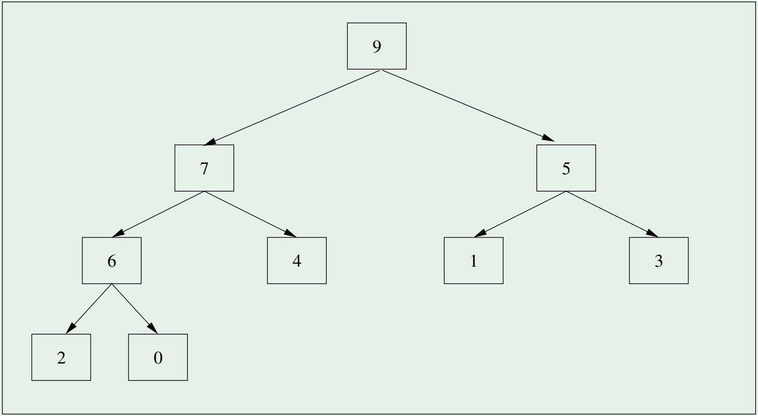
```
def _heapify(self, position):
 '''pre:  heap property is satisfied below position
 post:  heap property is satisfied at and below position'''
 item = self.heap[position]
 while position * 2 <= self.heap_size:
    child = position * 2
    # if right child exists, determine maximum of two children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

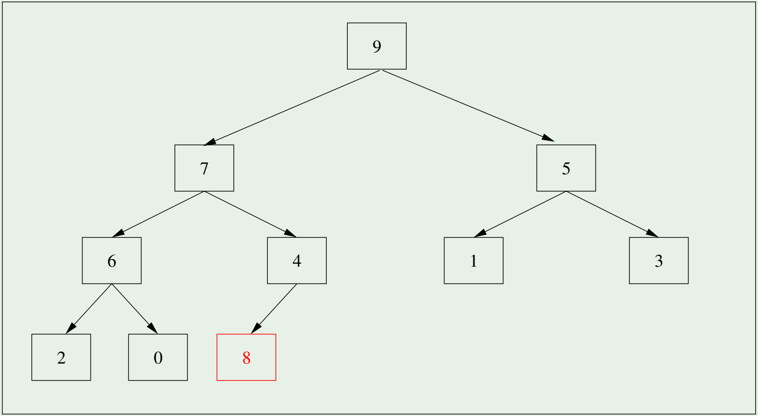
Heap: operation insert

Want to insert item with priority 8



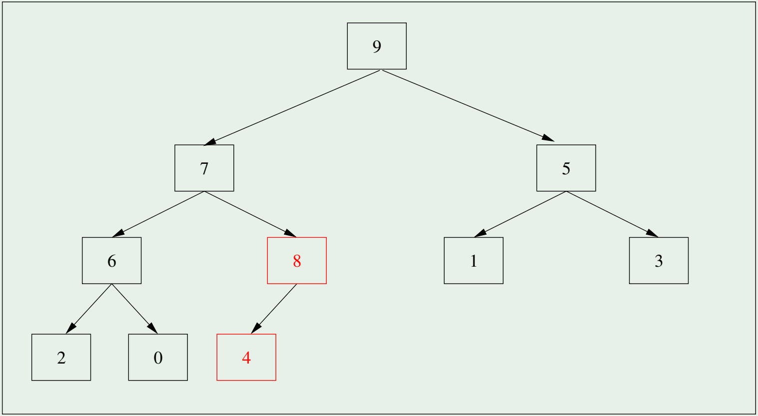
Heap: operation insert

Add the new item at the end



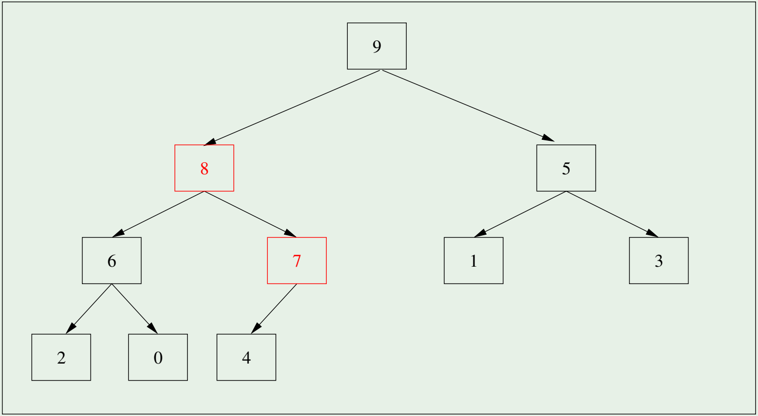
Heap: operation insert

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.

Python: dictionaries; C++: maps

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; C++: maps

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; C++: maps

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 so 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.

Python: dictionaries; C++: maps

In C++, maps are *associative containers* that store elements formed by a combination of a key value and a mapped value, following a specific order.

Internally, the elements in a map are always sorted by its key.

Maps are typically implemented as *binary search trees*, and as such, are slower than unordered_map containers.

Internally, the elements in the unordered_map are not sorted in any particular order with respect to either their key or mapped values, but organized into buckets depending on their *hash values* to allow for fast access to individual elements directly by their key values (with a constant average time complexity on average).

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

1. Implement *heapsort* in C++ version of Heap class.
2. Implement the Priority Queue using Binary Heaps in Python and/or in C++ using templates
   use the PQueue_ideas.py and usingPQueue.py files.
3. Do the problem 4, from Part II from the **Final Exam Sample**