Objectives

- Data structure: Heaps
- Implementing a Priority Queue

Review: Priority Queues for Sorting

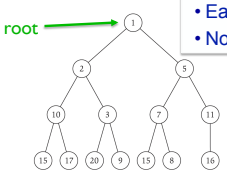
1. Add elements into PQ with the number's value as its priority
2. Then extract the smallest number until done
   - Come out in sorted order

Sorting *n* numbers takes O(n logn) time, which is our goal running time. However, "known" data structures won't give us that running time.

Already know our "loops" will be O(n)

Heap Defined

- Combines benefits of sorted array and list
- Balanced binary tree

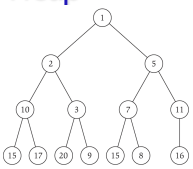


- Each node has *at most* 2 children
- Node value is its *key*

Heap order: each node's key is at least as large as its parent's

Note: **not** a binary search tree

Review: Implementing a Heap



- Option 1: Use pointers
  - Each node keeps
    - Element it stores, key
    - 3 pointers: 2 children, parent
- Option 2: No pointers
  - Requires knowing upper bound on *n*
  - For node at position *i*
    - left child is at *2i*
    - right child is at *2i+1*

Implementing a Heap: Operations

- Finding the minimal element?

Implementing a Heap: Operations

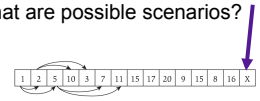
- Finding the minimal element
  - First element
  - O(1)

Implementing a Heap: Operations

- Adding an element?
  - Assume heap has less than N elements

Implementing a Heap: Operations

- Adding an element?
  - Could add element to last position
    - What are possible scenarios?



Implementing a Heap: Operations

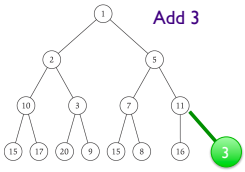
- Adding an element?
  - Could add element to last position
    - What are possible scenarios?
      - Heap is no longer balanced
      - Something that is almost a heap but a little off
      - Need `Heapify-up` procedure to fix our heap

Heapify-Up

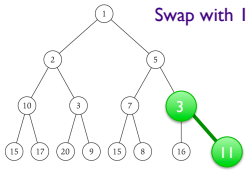
Heap — Position where node added

```
Heapify-up(H, i):
  if i > 1 then
      j=parent(i)=floor(i/2)
      if key[H[i]] < key[H[j]] then
        swap array entries H[i] and H[j]
        Heapify-up(H, j)
```

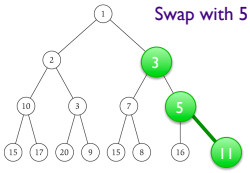

Practice: Heapify-Up



Practice: Heapify-Up



Practice: Heapify-Up



Heapify-Up

- Claim. Assuming array H is almost a heap with key of H[i] too small, Heapify-Up fixes the heap property in O(log i) time
  - Can insert a new element in a heap of *n* elements in O(log n) time

Heapify-Up

- Claim. Assuming array H is almost a heap with key of H[i] too small, Heapify-Up fixes the heap property in O(log i) time
  - Can insert a new element in a heap of *n* elements in O(log n) time
- Proof. By induction
  - If i=1 …

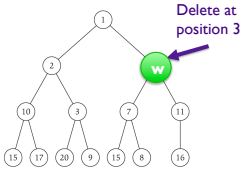
Heapify-Up

- Claim. Assuming array H is almost a heap with key of H[i] too small, Heapify-Up fixes the heap property in O(log i) time
  - Can insert a new element in a heap of *n* elements in O(log n) time
- Proof. By induction
  - If i=1, is already a heap → O(1)
  - If i>1, …

Heapify-Up

- Claim. Assuming array H is almost a heap with key of H[i] too small, Heapify-Up fixes the heap property in O(log i) time
  - Can insert a new element in a heap of *n* elements in O(log n) time
- Proof. By induction
  - If i=1, is already a heap → O(1)
  - If i>1,
    - Swaps are O(1)
    - Swaps continue up to root (max) → log i

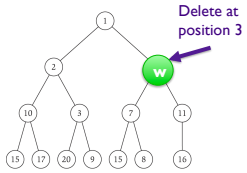
Deleting an Element



Deleting an Element

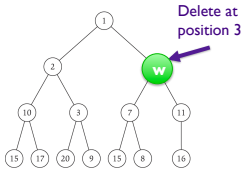
- Delete at position *i*
- Removing an element:
  - Messes up heap order
  - Leaves a "hole" in the heap
- Not as straightforward as `Heapify-Up`
- Algorithm
  1. Fill in element where hole was
     - Patch hole: move n^{th} element into i^{th} spot
  2. Adjust heap to be in order
     - At position *i* because moved n^{th} item up to *i*

Deleting an Element



- What are the possibilities when we move n^{th} element (*w*) into spot where element was removed?

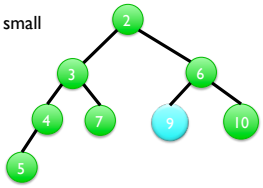
Deleting an Element



- Two possibilities: element *w* is
  - Too small: violation is between it and parent → `Heapify-Up`
  - Too big: with one or both children → `Heapify-Down` (example: w = 12)

Deleting an Element

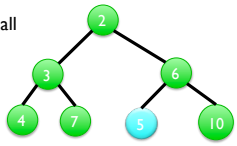
Example where new key is too small



- Delete 9
- Replace with 5

Deleting an Element

Example where new key is too small



- Delete 9
- Replace with 5
- But 5 < 6, so need to `Heapify-Up`

Heapify-Down

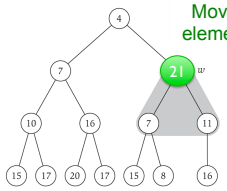
```
Heapify-down(H, i):
   n = length(H)
   if 2i > n then          Why can we stop?
      Terminate with H unchanged
   else if 2i < n then
      left=2i and right=2i+1
      j be index that minimizes
            key[H[left]] and key[[H[right]]
   else if 2i = n then
      j=2i

   if key[H[j]] < key[H[i]] then
      swap array entries H[i] and H[j]
      Heapify-down(H, j)
```

Heapify-Down

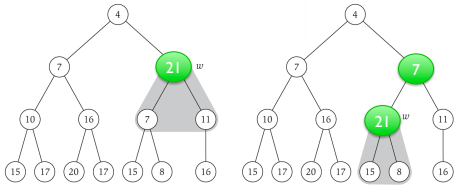
```
Heapify-down(H, i):
   n = length(H)
   if 2i > n then          i is a leaf – nowhere to go
      Terminate with H unchanged
   else if 2i < n then
      left=2i and right=2i+1
      j be index that minimizes
            key[H[left]] and key[[H[right]]
   else if 2i = n then
      j=2i

   if key[H[j]] < key[H[i]] then
      swap array entries H[i] and H[j]
      Heapify-down(H, j)
```


Practice: Heapify-Down

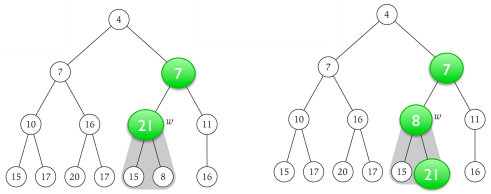
Moved 21 to where element was removed



Practice: Heapify-Down



Practice: Heapify-Down



Runtime of Heapify-Down?

```
Heapify-down(H, i):
   n = length(H)
   if 2i > n then
      Terminate with H unchanged
   else if 2i < n then
      left=2i and right=2i+1
      j be index that minimizes    O(1)
            key[H[left]] and key[[H[right]]
   else if 2i = n then
      j=2i

   if key[H[j]] < key[H[i]] then
      swap array entries H[i] and H[j] O(1)
      Heapify-down(H, j)
```

Num swaps: O(log n)

Implementing Priority Queues with Heaps

Operation	Description	Run Time
StartHeap(N)	Creates an empty heap that can hold N elements	
Insert(v)	Inserts item v into heap	
FindMin()	Identifies minimum element in heap but does not remove it	
Delete(i)	Deletes element in heap at position i	
ExtractMin()	Identifies and deletes an element with minimum key from heap	

Implementing Priority Queues with Heaps

Operation	Description	Run Time
StartHeap(N)	Creates an empty heap that can hold N elements	O(N)
Insert(v)	Inserts item v into heap	O(log n)
FindMin()	Identifies minimum element in heap but does not remove it	O(1)
Delete(i)	Deletes element in heap at position i	O(log n)
ExtractMin()	Identifies and deletes an element with minimum key from heap	O(log n)

Comparing Data Structures

Operation	Heap	Unsorted List	Sorted List
StartHeap(N)			
Insert(v)			
FindMin()			
Delete(i)			
ExtractMin()			

Comparing Data Structures

Operation	Heap	Unsorted List	Sorted List
StartHeap(N)	O(N)		
Insert(v)	O(log n)		
FindMin()	O(1)		
Delete(i)	O(log n)		
ExtractMin()	O(log n)		

Comparing Data Structures

Operation	Heap	Unsorted List	Sorted List
StartHeap(N)	O(N)	O(1)	O(1)
Insert(v)	O(log n)	O(1)	O(n)
FindMin()	O(1)	O(1)	O(1)
Delete(i)	O(log n)	O(n)	O(1)
ExtractMin()	O(log n)	O(n)	O(1)

Additional Heap Operations

- Access elements in PQ by name
  - Maintain additional array `Position` that stores current position of each element in heap

- Operations:
  - Delete(Position[v])
    - Does not increase overall running time
  - ChangeKey(v, α)
    - Changes key of element v to key(v) = α
    - Identify position of element v in array (`Position` array)
    - Change key, heapify

Assignments

- Journals: Finish Chapter 2 for Wednesday
- Problem Set 2 due Friday