CSE 332 Data Abstractions:

Priority Queues, Heaps, and a Small Town Barber

Kate Deibel
Summer 2012

Announcements

David's Super Awesome Office Hours
- Mondays 2:30-3:30 CSE 220
- Wednesdays 2:30-3:30 CSE 220
- Sundays 1:30-3:30 Allen Library Research Commons
- Or by appointment

Kate's Fairly Generic But Good Quality Office Hours
- Tuesdays, 2:30-4:30 CSE 210
- Whenever my office door is open
- Or by appointment

Announcements

- Remember to use cse332-staff@cs
  - Or at least e-mail both me and David
  - Better chance of a speedy reply
- Kate is not available on Thursdays
  - I've decided to make Thursdays my focus on everything but Teaching days
  - I will not answer e-mails received on Thursdays until Friday

Today

- Amortized Analysis Redux
- Review of Big-Oh times for Array, Linked-List and Tree Operations
- Priority Queue ADT
- Heap Data Structure

Thinking beyond one isolated operation

AMORTIZED ANALYSIS

Amortized Analysis

- What happens when we have a costly operation that only occurs some of the time?
- Example:
  My array is too small. Let's enlarge it.

  Option 1: Increase array size by 5
  Copy old array into new one

  Option 2: Double the array size
  Copy old array into new one

We will now explore amortized analysis!

Stretchy Array (version 1)

```
StretchyArray:
  maxSize: positive integer (starts at 0)
  array: an array of size maxSize
  count: number of elements in array

  put(x): add x to the end of the array
    if maxSize == count
      make new array of size (maxSize + 5)
      copy old array contents to new array
      maxSize = maxSize + 5
    array[count] = x
    count = count + 1
```


Stretchy Array (version 2)

```
StretchyArray:
  maxSize: positive integer (starts at 0)
  array: an array of size maxSize
  count: number of elements in array

  put(x): add x to the end of the array
    if maxSize == count
      make new array of size (maxSize * 2)
      copy old array contents to new array
      maxSize = maxSize * 2
    array[count] = x
    count = count + 1
```


Performance Cost of put(x)

In both stretchy array implementations, put(x)is defined as essentially:

```
if maxSize == count
  make new array of bigger size
  copy old array contents to new array
  update maxSize
array[count] = x
count = count + 1
```

What f(n) is put(x) in O(f(n))?

Performance Cost of put(x)

In both stretchy array implementations, put(x)is defined as essentially:

```
if maxSize == count                      O(1)
  make new array of bigger size          O(1)
  copy old array contents to new array   O(n)
  update maxSize                         O(1)
array[count] = x                         O(1)
count = count + 1                        O(1)
```

In the worst-case, put(x) is O(n) where n is the current size of the array!!

But…

- We do not have to enlarge the array each time we call put(x)
- What will be the average performance if we put n items into the array?

$$\frac{\sum_{i=1}^{n} \text{cost of calling put for the ith time}}{n} = O(?)$$

- Calculating the average cost for multiple calls is known as *amortized analysis*

Amortized Analysis of StretchyArray Version 1

i	maxSize	count	cost	*comments*
	0	0		Initial state
1	5	1	0 + 1	Copy array of size 0
2	5	2	1	
3	5	3	1	
4	5	4	1	
5	5	5	1	
6	10	6	5 + 1	Copy array of size 5
7	10	7	1	
8	10	8	1	
9	10	9	1	
10	10	10	1	
11	15	11	10 + 1	Copy array of size 10
⁞	⁞	⁞	⁞	⁞

Amortized Analysis of StretchyArray Version 1

i	maxSize	count	cost	*comments*
	0	0		Initial state
1	5	1	0 + 1	Copy array of size 0
2	5	2	1	
			1	
			1	
			1	
			5 + 1	Copy array of size 5
			1	
8	10	8	1	
9	10	9	1	
10	10	10	1	
11	15	11	10 + 1	Copy array of size 10
⋮	⋮	⋮	⋮	⋮

Every five steps, we have to do a multiple of five more work

Amortized Analysis of StretchyArray Version 1

Assume the number of puts is n=5k

- We will make n calls to array[count]=x
- We will stretch the array k times and will cost:

$$0 + 5 + 10 + \cdots + 5(k-1)$$

Total cost is then:

n + (0 + 5 + 10 + ⋯ + 5(k-1))

= n + 5(1 + 2 + ⋯ +(k-1))

= n + 5(k-1)(k-1+1)/2

= n + 5k(k-1)/2

≈ n + n^2/10

Amortized cost for put(x) is

$$\frac{n + \frac{n^2}{10}}{n} = 1 + \frac{n}{10} = O(n)$$

Amortized Analysis of StretchyArray Version 2

i	maxSize	count	cost	*comments*
	1	0		Initial state
1	1	1	1	
2	2	2	1 + 1	Copy array of size 1
3	4	3	2 + 1	Copy array of size 2
4	4	4	1	
5	8	5	4 + 1	Copy array of size 4
6	8	6	1	
7	8	7	1	
8	8	8	1	
9	16	9	8 + 1	Copy array of size 8
10	16	10	1	
11	16	11	1	
⋮	⋮	⋮	⋮	⋮

Amortized Analysis of StretchyArray Version 2

i	maxSize	count	cost	*comments*
	1	0		Initial state
1	2	1	1	
2	2	2	1 + 1	Copy array of size 1
3	4	3	2 + 1	
4	4	4	1	
5	8	5	4 + 1	
6	8	6	1	
7	8	7	1	
8	8	8	1	
9	16	9	8 + 1	Copy array of size 8
10	16	10	1	
11	16	11	1	
⋮	⋮	⋮	⋮	⋮

Enlarge steps happen basically when *i* is a power of 2

Amortized Analysis of StretchyArray Version 2

Assume the number of puts is n=2^k

- We will make n calls to array[count]=x
- We will stretch the array k times and will cost:

$$\approx 1 + 2 + 4 + \cdots + 2^{k-1}$$

Total cost is then:

≈ n + (1 + 2 + 4 + ⋯ + 2^{k-1})

≈ n + 2^k – 1

≈ 2n - 1

Amortized cost for put(x) is

$$\frac{2n-1}{n} = 2 - \frac{1}{n} = O(1)$$

The Lesson

With amortized analysis, we know that over the long run (on average):

- If we stretch an array by a constant amount, each put(x) call is O(n) time
- If we double the size of the array each time, each put(x) call is O(1) time

In general, paying a high-cost infrequently can pay off over the long run.

What about wasted space?

Two options:

- We can adjust our growth factor
  - As long as we multiply the size of the array by a factor >1, amortized analysis holds
- We can also shrink the array:
  - A good rule of thumb is to halve the array when it is only 25% full
  - Same amortized cost

Memorize these. They appear all the time.

ARRAY, LIST, AND TREE PERFORMANCE

Very Common Interactions

When we are working with data, there are three very common operations:

- Insert(x): insert x into structure
- Find(x): determine if x is in structure
- Remove(i): remove item as position i
- Delete(x): find and delete x from structure

Note that when we usually delete, we

- First find the element to remove
- Then we remove it

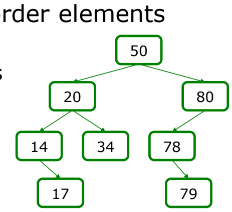
Overall time is O(Find + Remove)

Arrays and Linked Lists

- Most common data structures
- Several variants
  - Unsorted Array
  - Unsorted Circular Array
  - Unsorted Linked List
  - Sorted Array
  - Sorted Circular Array
  - Sorted Linked List
- We will ignore whether the list is singly or doubly-linked
  - Usually only leads to a constant factor change in overall performance

Binary Search Tree (BST)

- Another common data structure
- Each node has at most two children
  - Left child's value is less than its parent
  - Right child's value is greater than parent
- Structure depends on order elements were inserted into tree
  - Best performance occurs if the tree is balanced
- General properties
  - Min is leftmost node
  - Max is rightmost node



Worst-Case Run-Times

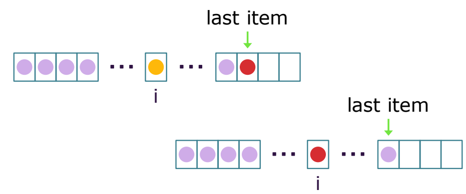
	Insert	Find	Remove	Delete
Unsorted Array				
Unsorted Circular Array				
Unsorted Linked List				
Sorted Array				
Sorted Circular Array				
Sorted Linked List				
Binary Search Tree				
Balanced BST				

Worst-Case Run-Times

	Insert	Find	Remove	Delete
Unsorted Array	O(1)	O(n)	O(1)	O(n)
Unsorted Circular Array	O(1)	O(n)	O(1)	O(n)
Unsorted Linked List	O(1)	O(n)	O(1)	O(n)
Sorted Array	O(n)	O(log n)	O(n)	O(n)
Sorted Circular Array	O(n/2)	O(log n)	O(n/2)	O(n/2)
Sorted Linked List	O(n)	O(n)	O(1)	O(n)
Binary Search Tree	O(n)	O(n)	O(n)	O(n)
Balanced BST	O(log n)	O(log n)	O(log n)	O(log n)

Remove in an Unsorted Array

- Let's say we want to remove the item at position *i* in the array
- All that we do is move the last item in the array to position i



Remove in a Binary Search Tree

Replace node based on following logic

- If no children, just remove it
- If only one child, replace node with child
- If two children, replace node with the smallest data for the right subtree
- See Weiss 4.3.4 for implementation details

Balancing a Tree

How do you guarantee that a BST will always be balanced?

- Non-trivial task
- We will discuss several implementations next Monday

The immediate priority is to discuss heaps.

PRIORITY QUEUES

Scenario

What is the difference between waiting for service at a pharmacy versus an ER?

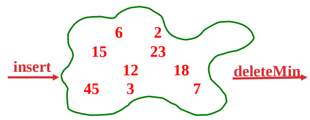
Pharmacies usually follow the rule First Come, First Served — Queue

Emergency Rooms assign priorities based on each individual's need — Priority Queue

New ADT: Priority Queue

Each item has a "priority"
- The next/best item has the lowest priority
- So "priority 1" should come before "priority 4"
- Could also do maximum priority if so desired

Operations:
- insert
- deleteMin



deleteMin returns/deletes item with lowest priority
- Any ties are resolved arbitrarily
- Fancier PQueues may use a FIFO approach for ties

Priority Queue Example

insert *a* with priority *5*
insert *b* with priority *3*
insert *c* with priority *4*
w = **deleteMin**
x = **deleteMin**
insert *d* with priority *2*
insert *e* with priority *6*
y = **deleteMin**
z = **deleteMin**

after execution:

w = b
x = c
y = d
z = a

To simplify our examples, we will just use the priority values from now on

Applications of Priority Queues

PQueues are a major and common ADT
- Forward network packets by urgency
- Execute work tasks in order of priority
  - "critical" before "interactive" before "compute-intensive" tasks
  - allocating idle tasks in cloud environments
- A fairly efficient sorting algorithm
  - Insert all items into the PQueue
  - Keep calling deleteMin until empty

Advanced PQueue Applications

- "Greedy" algorithms
  - Efficiently track what is "best" to try next
- Discrete event simulation (e.g., virtual worlds, system simulation)
  - Every event e happens at some time t and generates new events e_1, …, e_n at times $t+t_1$, …, $t+t_n$
  - Naïve approach:
    - Advance "clock" by 1, check for events at that time
  - Better approach:
    - Place events in a priority queue (priority = time)
    - Repeatedly: deleteMin and then insert new events
    - Effectively "set clock ahead to next event"

From ADT to Data Structure

- How will we implement our PQueue?

	Insert	Find	Remove	Delete
Unsorted Array	O(1)	O(n)	O(1)	O(n)
Unsorted Circular Array	O(1)	O(n)	O(1)	O(n)
Unsorted Linked List	O(1)	O(n)	O(1)	O(n)
Sorted Array	O(n)	O(log n)	O(n/2)	O(n/2)
Sorted Circular Array	O(n/2)	O(log n)	O(n/2)	O(n/2)
Sorted Linked List	O(n)	O(n)	O(1)	O(n)
Binary Search Tree	O(n)	O(n)	O(n)	O(n)
Balanced BST	O(log n)	O(log n)	O(log n)	O(log n)

- We need to add one more analysis to the above: finding the min value

Finding the Minimum Value

	FindMin
Unsorted Array	
Unsorted Circular Array	
Unsorted Linked List	
Sorted Array	
Sorted Circular Array	
Sorted Linked List	
Binary Search Tree	
Balanced BST	

Finding the Minimum Value

	FindMin
Unsorted Array	O(n)
Unsorted Circular Array	O(n)
Unsorted Linked List	O(n)
Sorted Array	O(1)
Sorted Circular Array	O(1)
Sorted Linked List	O(1)
Binary Search Tree	O(n)
Balanced BST	O(log n)

Best Choice for the PQueue?

	Insert	FindMin+Remove
Unsorted Array	O(1)	O(n)+O(1)
Unsorted Circular Array	O(1)	O(n)+O(1)
Unsorted Linked List	O(1)	O(n)+O(1)
Sorted Array	O(n)	O(1)+O(n)
Sorted Circular Array	O(n/2)	O(1)+O(n/2)
Sorted Linked List	O(n)	O(1)+O(1)
Binary Search Tree	O(n)	O(n)+O(n)
Balanced BST	O(log n)	O(log n)+O(log n)

None are that great

We generally have to pay linear time for either insert or deleteMin

- Made worse by the fact that:
  # inserts ≈ # deleteMins

Balanced trees seem to be the best solution with O(log n) time for both

- But balanced trees are complex structures
- Do we really need all of that complexity?

Our Data Structure: The Heap

Key idea: Only pay for functionality needed

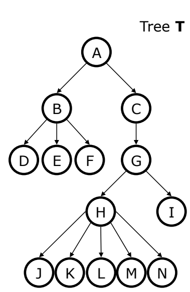
- Do better than scanning unsorted items
- But do not need to maintain a full sort

The Heap:

- O(log n) insert and O(log n) deleteMin
- If items arrive in random order, then the average-case of insert is O(1)
- Very good constant factors

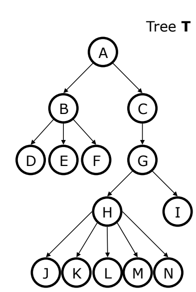
Reviewing Some Tree Terminology

root(**T**): A
leaves(**T**): D-F, I, J-N
children(B): D, E, F
parent(H): G
siblings(E): D, F
ancestors(F): B, A
descendents(G): H, I, J-N
subtree(G): G and its children



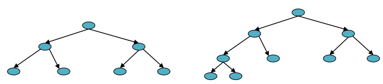
Some More Tree Terminology

depth(B): *1*
height(G): *2*
height(**T**): *4*
degree(B): *3*
branching factor(**T**): *0-5*



Types of Trees

- Binary tree: Every node has ≤2 children
- n-ary tree: Every node as ≤n children
- Perfect tree: Every row is completely full
- Complete tree:
  All rows except the bottom are completely full, and it is filled from left to right



Perfect Tree Complete Tree

Some Basic Tree Properties

Nodes in a perfect binary tree of height h?

$2^{h+1}-1$

Leaves in a perfect binary tree of height h?

2^h

Height of a perfect binary tree with n nodes?

$\lfloor \log_2 n \rfloor$

Height of a complete binary tree with n nodes?

$\lfloor \log_2 n \rfloor$

Properties of a Binary Min-Heap

More commonly known as a binary heap or simply a heap

- Structure Property:
  A complete [binary] tree
- Heap Property:
  The priority of every non-root node is greater than the priority of its parent

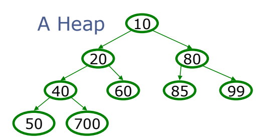
How is this different from a binary search tree?

Properties of a Binary Min-Heap

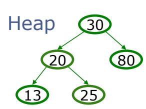
More commonly known as a binary heap or simply a heap

- Structure Property:
  A complete [binary] tree
- Heap Property:
  The priority of every non-root node is greater than the priority of its parent

A Heap



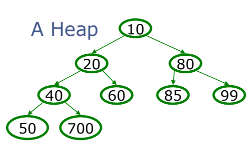
Not a Heap



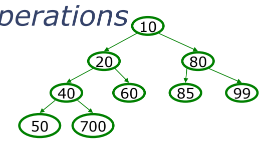
Properties of a Binary Min-Heap

- Where is the minimum priority item?
  At the root
- What is the height of a heap with n items?
  $\lfloor \log_2 n \rfloor$

A Heap



Basics of Heap Operations



findMin:

- return root.data

deleteMin:

- Move last node up to root
- Violates heap property, so *Percolate Down* to restore

insert:

- Add node after last position
- Violate heap property, so *Percolate Up* to restore

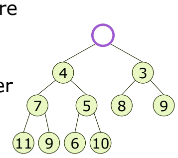
The general idea:

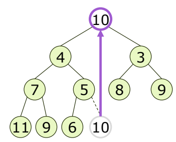
- Preserve the complete tree structure property
- This likely breaks the heap property
- So percolate to restore the heap property

DeleteMin Implementation

1. Delete value at root node (and store it for later return)
2. There is now a "hole" at the root. We must "fill" the hole with another value, must have a tree with one less node, and it must still be a complete tree
3. The "last" node is the is obvious choice, but now the heap property is violated
4. We percolate down to fix the heap

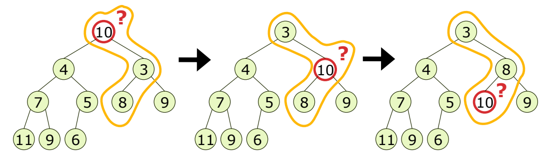
```
While greater than either child
   Swap with smaller child
```





Percolate Down

```
While greater than either child
   Swap with smaller child
```



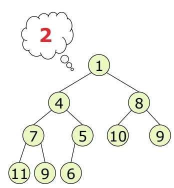
What is the runtime?
O(log n)

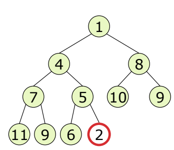
Why does this work?
Both children are heaps

Insert Implementation

1. To maintain structure property, there is only one node we can insert into that keeps the tree complete.
   We put our new data there.
2. We then percolate up to fix the heap property:

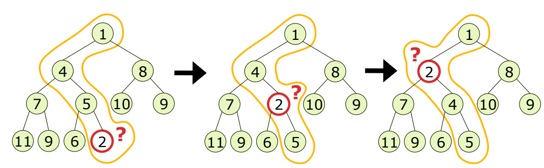
```
While less than parent
   Swap with parent
```





Percolate Up

```
While less than parent
   Swap with parent
```



What is the runtime?
O(log n)

Why does this work?
Both children are heaps

Achieving Average-Case O(1) insert

Clearly, insert and deleteMin are worst-case O(log n)
- But we promised average-case O(1) insert

Insert only requires finding that one special spot
- Walking the tree requires O(log n) steps

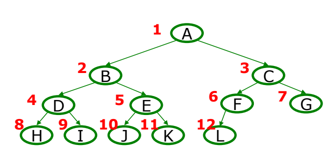
We should only pay for the functionality we need
- Why have we insisted the tree be complete?

All complete trees of size n contain the same edges
- So why are we even representing the edges?

Here comes the really clever bit about implementing heaps!!!

Array Representation of a Binary Heap

From node i:
- left child: 2i
- right child: 2i+1
- parent: i / 2



	A	B	C	D	E	F	G	H	I	J	K	L	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

- We skip index 0 to make the math simpler
- Actually, it can be a good place to store the current size of the heap

Pseudocode: insert

This pseudocode uses ints. In real use, you will have data nodes with priorities.

```
void insert(int val) {
  if(size == arr.length-1)
    resize();
  size = size + 1;
  i = percolateUp(size, val);
  arr[i] = val;
}
```

```
int percolateUp(int hole, int val) {
  while(hole > 1 && val < arr[hole/2])
    arr[hole] = arr[hole/2];
    hole = hole / 2;
  }
  return hole;
}
```


Pseudocode: deleteMin

This pseudocode uses ints. In real use, you will have data nodes with priorities.

```
int deleteMin() {
  if(isEmpty()) throw…
  ans = arr[1];
  hole = percolateDown(1,arr[size]);
  arr[hole] = arr[size];
  size--;
  return ans;
}
```


Pseudocode: deleteMin

```
int percolateDown(int hole, int val) {
  while(2 * hole <= size) {
    left  = 2 * hole;
    right = left + 1;
    if(arr[left] < arr[right] || right > size)
      target = left;
    else
      target = right;

    if(arr[target] < val) {
      arr[hole] = arr[target];
      hole = target;
    }
    else
      break;
  }
  return hole;
}
```

Note that *percolateDown* is more complicated than *percolateUp* because a node might not have two children

Advantages of Array Implementation

Minimal amount of wasted space:

- Only index 0 and any unused space on right
- No "holes" due to complete tree property
- No wasted space representing tree edges

Fast lookups:

- Benefit of array lookup speed
- Multiplying and dividing by 2 is extremely fast (can be done through bit shifting)
- Last used position is easily found by using the PQueue's size for the index

Disadvantages of Array Implementation

May be too clever:

- Will you understand it at 3am three months from now?

What if the array gets too full or too empty?

- Array will have to be resized
- Stretchy arrays give us O(1) amortized performance

Advantages outweigh disadvantages: This is how heaps are implemented. Period.

So why O(1) average-case insert?

- Yes, insert's worst case is O(log n)
- The trick is that it all depends on the order the items are inserted
- Experimental studies of randomly ordered inputs shows the following:
  - Average 2.607 comparisons per insert (# of percolation passes)
  - An element usually moves up 1.607 levels
- deleteMin is average O(log n)
  - Moving a leaf to the root usually requires re-percolating that value back to the bottom

I promised you a small-town barber…

MORE ON HEAPS

Other Common Heap Operations

decreaseKey(i, p): O(log n)

- given pointer to object in priority queue (e.g., its array index), lower its priority to p
- Change priority and percolate up

increaseKey(i, p): O(log n)

- given pointer to object in priority queue (e.g., its array index), raise its priority to p
- Change priority and percolate down

remove(i): O(log n)

- given pointer to object in priority queue (e.g., its array index), remove it from the queue
- decreaseKey to p = -∞, then deleteMin

Building a Heap

Suppose you have n items to put in a new priority queue… what is the complexity?

- Sequence of n inserts → O(n log n)
  Worst-case is Θ(n log n)

Can we do better?

- If we only have access to the insert and deleteMin operations, then NO
- There is a faster way, but that requires the ADT to have buildHeap operation

When designing an ADT, adding more and more specialized operations leads to tradeoffs with Convenience, Efficiency, and Simplicity

Proof that n inserts can be *Θ(n log n)*

Worst performance is if insert has to percolate up to the root (Occurs when inserted in reverse order)

If n = 7, worst case is

- insert(7) takes 0 percolations
- insert(6) takes 1 percolation
- insert(5) takes 1 percolation
- insert(4) takes 2 percolations
- insert(3) takes 2 percolations
- insert(2) takes 2 percolations
- insert(1) takes 2 percolations

More generally…

If $n = 2^k - 1$, then the worst-case number of percolations will be:

$$0 \cdot 1 + 1 \cdot 2 + 2 \cdot 4 + 3 \cdot 8 + \cdots + (\mathrm{k}-1) \cdot 2^{k-1}$$
$$= 0 \cdot 2^0 + 1 \cdot 2^1 + 2 \cdot 2^2 + \cdots + (\mathrm{k}-1) \cdot 2^{k-1}$$
$$= \sum_0^{k-1} i \cdot 2^i$$

If we focus on just the last item, then

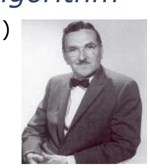
$$(k-1) \cdot 2^{k-1} = k \cdot 2^{k-1} - 2^{k-1} = \frac{k}{2}(2^k - 1) + \frac{k}{2} - \frac{1}{2} 2^k$$
$$= \frac{1}{2}(\mathrm{n} \cdot \log_2 n + \log_2 n - \log_2(n+1))$$
$$= \Theta(n \cdot \log n)$$

BuildHeap using Floyd's Algorithm

We can actually build a heap in O(n)

The trick is to use our general strategy for working with the heap:

- Preserve structure property
- Break and restore heap property



Thank you, Floyd the barber, for your cool O(n) algorithm!!

Floyd's Algorithm:

- Create a complete tree by putting the n items in array indices 1,…,n
- Fix the heap-order property

Floyd's Algorithm

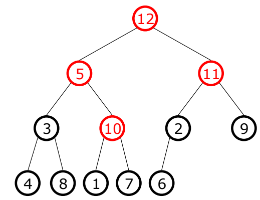
Bottom-up fixing of the heap

- Leaves are already in heap order
- Work up toward the root one level at a time

```
void buildHeap() {
  for(i = size/2; i>0; i--) {
    val  = arr[i];
    hole = percolateDown(i,val);
    arr[hole] = val;
  }
}
```

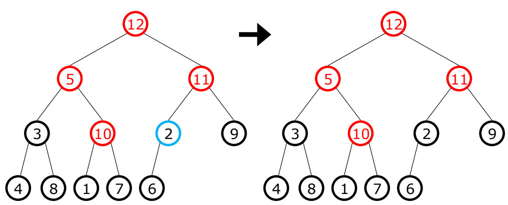

Example

- We use a tree for readability purposes
- Red nodes are not less than children
- No leaves are red
- We start at i=size/2



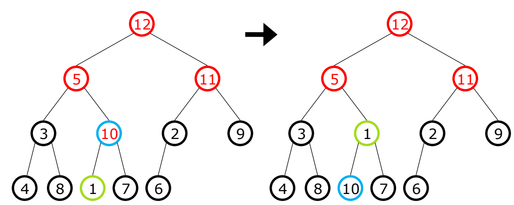
Example

i = 6, node is 2
no change is needed



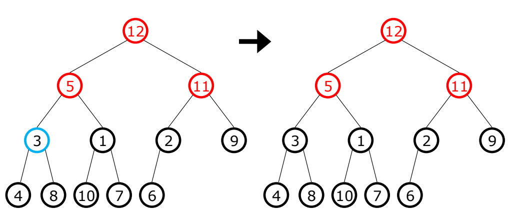
Example

i = 5, node is 10
10 percolates down; 1 moves up



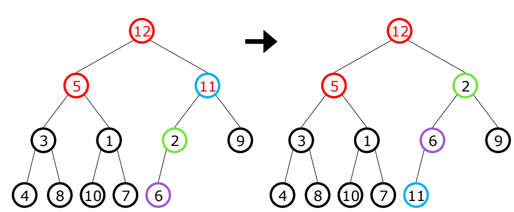
Example

i = 4, node is 3
no change is needed



Example

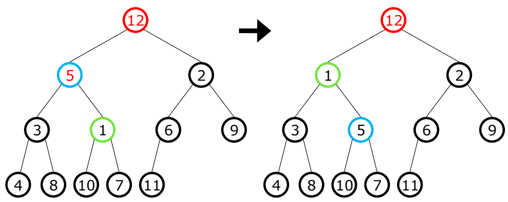
i = 3, node is 11
11 percolates down twice; 2 and 6 move up



Example

i = 2, node is 5

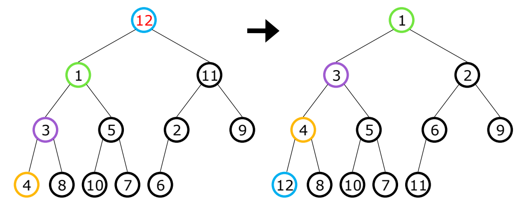
5 percolates down; 1 moves up (again)



Example

i = 1, node is 12

12 percolates down; 1, 3, and 4 move up



But is it right?

Floyd's algorithm "seems to work"

We will prove that it does work

- First we will prove it restores the heap property (correctness)
- Then we will prove its running time (efficiency)

Correctness

We claim the following is a loop invariant:

For all j>i, arr[j] is less than its children

True initially: If j > size/2, then j is a leaf

- Otherwise its left child would be at position > size

```
void buildHeap() {
  for(i = size/2; i>0; i--) {
    val  = arr[i];
    hole = percolateDown(i,val);
    arr[hole] = val;
  }
}
```


Correctness

We claim the following is a loop invariant:

For all j>i, arr[j] is less than its children

After an iteration: Still true

- We know that for j > i + 1, the heap property is maintained (from previous iteration)
- percolateDown maintains heap property
- arr[i] is fixed by percolate down
- Ergo, loop body maintains the invariant

```
void buildHeap() {
  for(i = size/2; i>0; i--) {
    val  = arr[i];
    hole = percolateDown(i,val);
    arr[hole] = val;
  }
}
```


Correctness

We claim the following is a loop invariant:

For all j>i, arr[j] is less than its children

Loop invariant implies that heap property is present

- Each node is less than its children
- We also know it is a complete tree

∴ It's a heap!

What type of proof was this?

```
void buildHeap() {
  for(i = size/2; i>0; i--) {
    val  = arr[i];
    hole = percolateDown(i,val);
    arr[hole] = val;
  }
}
```

Efficiency

Easy argument: buildHeap is O(n log n)

- We perform n/2 loop iterations
- Each iteration does one percolateDown, and costs O(log n)

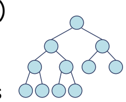
This is correct, but can make a tighter analysis.

The heights of each percolate are different!

```
void buildHeap() {
  for(i = size/2; i>0; i--) {
    val  = arr[i];
    hole = percolateDown(i,val);
    arr[hole] = val;
  }
}
```


Efficiency

Better argument: buildHeap is O(n)



- We perform n/2 loop iterations
- 1/2 iterations percolate at most 1 step
- 1/4 iterations percolate at most 2 steps
- 1/8 iterations percolate at most 3 steps
- etc.

# of percolations $< \frac{n}{2} \cdot \left(\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \cdots\right) = \frac{n}{2} \cdot \sum_i \frac{i}{2^i} = \frac{n}{2} \cdot 2 = n$

Ergo, buildHeap is O(n)

Proof of Summation

Let $S = \sum_i \frac{i}{2^i} = \frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + \cdots$

Then $2S = 1 + \frac{2}{2^1} + \frac{3}{2^2} + \frac{4}{2^3} + \frac{5}{2^4} + \cdots$

Then $2S - S = 1 + \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots = 2$

Lessons from buildHeap

- Without buildHeap, the PQueue ADT allows clients to implement their own buildHeap with worst-case Θ(n log n)
- By providing a specialized operation internal to the data structure (with access to the internal data), we can do a much better O(n) worst case

Our Analysis of buildHeap

Correctness:

- Example of a non-trivial inductive proof using loop invariants

Efficiency:

- First analysis easily proved it was at least O(n log n)
- A "tighter" analysis looked at individual steps to show algorithm is O(n)

Unrelated but consider reading up on the Fallacy of the Heap, also known as Loki's wager

PARTING THOUGHTS ON HEAPS

What to take away

- Priority Queues are a simple to understand ADT
- Making a useful data structure for them is tricky
  - Requires creative thinking for implementation
  - Resulting array allows for amazing efficiency

What we are skipping (see textbook)

d-heaps: have d children instead of 2
- Makes heaps shallower which is useful for heaps that are too big for memory
- The same issue arises for balanced binary search trees (we will study "B-Trees")

Merging heaps
- Given two PQueues, make one PQueue
- O(log n) merge impossible for binary heaps
- Can be done with specialized pointer structures

Binomial queues
- Collections of binary heap-like structures
- Allow for O(log n) insert, delete and merge