(Binary) Heaps

- So far, the binary trees we've seen are linked structures
  - (Just Like a linked list, but each node has two child references)

- Binary trees can also be implemented using arrays
  - An array-based binary tree is called a *binary heap (or just "heap")*
    - Nodes are accessed using indices!!
    - Advantage: Heaps are always balanced!
    - Disadvantage: Same with ArrayList
      - Array-based, so need to re-allocate sometimes as heap grows
      - Can waste space with un-used array slots
Binary Heap organization:

- Root node is at index $i=0$
- For any node at index $i$
  - Its left child is at index: ???
  - Its right child is at index: ???
  - Its parent node is at index: ???

What is stored:

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>25</th>
<th>50</th>
<th>40</th>
<th>56</th>
<th>59</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

What it represents:
Binary Heap organization:

- Root node is at index $i=0$
- For any node at index $i$
  - Its left child is at index: $i*2+1$
  - Its right child is at index: $i*2+2$
  - Its parent node is at index: $(i-1)/2$
    - Except for root

What is stored:

What it represents:

$$\begin{array}{ccccccc}
0 & 1 & 2 & 3 & 4 & 5 \\
10 & 25 & 50 & 40 & 56 & 59 \\
\end{array}$$
Outline

- (Binary) Heaps
- Min-Heap
  - Properties
  - Add
    - Percolate up
  - Remove
    - Percolate down
- Performance and Applications
- Conclusion
One popular heap organization is called a *Min-Heap*

**Important: Min-Heap Properties (Recursive)**

- The value at the root is smaller than its children
- Every sub-tree is a min-heap

(Think "bottom heavy")

*Note that Min-Heap's properties are more relaxed than BST's!*
Min-Heap

- One popular heap organization is called a **Min-Heap**
- Important: Min-Heap Properties (Recursive)
  - The value at the root is smaller than its children
  - Every sub-tree is a min-heap

- Note that Min-Heap's properties are more relaxed than BST's!
One popular heap organization is called a *Min-Heap*

Important: Min-Heap Properties (Recursive)

- The value at the root is smaller than its children
- Every sub-tree is a min-heap

Note that Min-Heap's properties are more relaxed than BST's!
One popular heap organization is called a **Min-Heap**

**Important: Min-Heap Properties (Recursive)**

- The value at the root is smaller than its children
- Every sub-tree is a min-heap

Note that Min-Heap's properties are more relaxed than BST's!
Min-Heap?
Min-Heap?
(Binary) Heaps

Min-Heap
- Properties
  - Add
    - Percolate up
  - Remove
    - Percolate down

Performance and Applications

Conclusion
Adding Items

- Add the following items: 43, 18, 2

```
size: 10
capacity: 40
```

```
0  1  2  3  4  5  6  7  8  9  10  ...
5 14 23 32 41 87 90 50 64 53 ...
```
Adding Items

- Add the following items: 43, 18, 2

Size: 11
Capacity: 40
Adding Items

- Add the following items: 43, 18, 2

Check min-heap property: 43 > 41, so we're done!

size: 11
capacity: 40
Adding Items

- Add the following items: 43, 18, 2

```
size: 11
capacity: 40
```
Adding Items

- Add the following items: 43, 18, 2

![Heap Diagram]

- size: 12
- capacity: 40

```plaintext
5  14  23  32  41  87  90  50  64  53  43  18 ...
0  1  2  3  4  5  6  7  8  9  10  11
```
Adding Items

- Add the following items: 43, 18, 2

Check min-heap property: 18 < 87, so it violates!!

size: 12
capacity: 40
Adding Items

- Add the following items: 43, 18, 2

Percolate 18 up!

Swap!

size: 12
capacity: 40
Adding Items

- Add the following items: 43, 18, 2

Percolate 18 up!

Size: 12
Capacity: 40
Adding Items

- Add the following items: 43, 18, 2

Check min-heap property: 18 < 23, so it violates!!

size: 12
capacity: 40
Adding Items

- Add the following items: 43, 18, 2

Percolate 18 up!

Swap!

size: 12
capacity: 40
Add the following items: 43, 18, 2

Percolate 18 up!
Adding Items

- Add the following items: 43, 18, 2

Check min-heap property: 18 > 5, so we're done!!

size: 12
capacity: 40
Adding Items

- Add the following items: 43, 18, 2

```
size: 12
capacity: 40
```
Your turn! Add: 2

- Add the following items: 43, 18, 2

Size: 12
Capacity: 40

Diagram of a binary heap with the added items highlighted.
Explore the Heap code (given)

- Need to compare heap's stored data for percolate up/down...

```java
/**
 * Heaps: A lot like ArrayList under the hood
 */
public class Heap<E extends Comparable<E>> {
    private final int INITIAL_CAPACITY = 10;
    private E[] the_data;
    private int capacity;
    private int size;

    /**
     * Creates an empty heap of default capacity of 10
     */
    public Heap() {
        this.capacity = INITIAL_CAPACITY;
        this.size = 0;
        this.the_data = (E[]) new Comparable[capacity];
    }
```
Implementing `add(E item)`

- **Sketch**
  - Increment size to gain a new item in the heap
    - (What if we're out of space?)
  - Percolate the item up recursively:
    - Base case 1: Root: do nothing
    - Base case 2: Item is already less than its parent: do nothing
    - Recursive case: Item > parent
      - Swap them
      - Recursively percolate parent up
/**
 * Adds an item to the heap
 * @param item The element to add to the heap
 */
public void add(E item) {
    if (this.size == this.capacity) { // out of space; double capacity
        this.reallocate();
    }
    this.the_data[this.size] = item; // add element to the end of the array
    this.percolateUp(size); // possibly need to percolate item up the heap
    this.size++;
}

private void percolateUp(int i) {
    // only attempt to percolate up if current node is not root
    if (i > 0) {
        // Compare with parent
        int parent = (i-1)/2;
        if (this.the_data[i].compareTo(this.the_data[parent]) < 0) {
            E tmp = this.the_data[i];
            this.the_data[i] = this.the_data[parent];
            this.the_data[parent] = tmp;
            // recursively percolate parent node up
            this.percolateUp(parent);
        }
    }
}
Outline

- (Binary) Heaps
- Min-Heap
  - Properties
  - Add
    - Percolate up
  - Remove
    - Percolate down
- Performance and Applications
- Conclusion
Removing Items: remove(int index)

- Remove the following items: 53 (index 9), 14 (index 1), 5 (index 0)

![Heap Diagram]

- size: 12
- capacity: 40
Removing Items

- Remove the following items: 53 (index 9), 14 (index 1), 5 (index 0)

```
size: 12
capacity: 40
```
Removing Items

- Remove the following items: 53 (index 9), 14 (index 1), 5 (index 0)

size: 12
capacity: 40
Removing Items

- Remove the following items: 53 (index 9), 14 (index 1), 5 (index 0)

![Binary Heap Diagram]

- Size: 12
- Capacity: 40
Removing Items

- Remove the following items: 53 (index 9), 14 (index 1), 5 (index 0)

Remove last node by decrementing size

size: 11
capacity: 40
Removing Items

- Remove the following items: **53 (index 9)**, **14 (index 1)**, **5 (index 0)**

Min-heap property satisfied!
(Done)

Size: **11**
Capacity: **40**
Removing Items

- Remove the following items: 53 (index 9), 14 (index 1), 5 (index 0)

```
size: 11
capacity: 40
```
Removing Items

- Remove the following items: 53 (index 9), 14 (index 1), 5 (index 0)

Min-heap property violated!
(Percolate 43 down!)

size: 10
capacity: 40
Removing Items

- Remove the following items: 53 (index 9), 14 (index 1), 5 (index 0)

Min-heap property violated!
(Percolate 43 down!)

size: 10
capacity: 40
Removing Items

- Remove the following items: 53 (index 9), 14 (index 1), 5 (index 0)

Min-heap property violated!
(Percolate 43 down!)

Swap with smaller child

size: 10
capacity: 40
Removing Items

- Remove the following items: 53 (index 9), 14 (index 1), 5 (index 0)

No one to swap with (done!)

size: 10
capacity: 40
Your Turn! Remove 5

- Remove the following items: 53 (index 9), 14 (index 1), 5 (index 0)
Your Turn! Remove 5

- Remove the following items: 53 (index 9), 14 (index 1), 5 (index 0)
Heap after remove(5)

- Remove the following items: 53 (index 9), 14 (index 1), 5 (index 0)

```
0
1
3
4
5
6
7
8
9
10
11
...
```

size: 9
capacity: 40
Outline

- (Binary) Heaps
- Min-Heap
  - Properties
  - Add
    - Percolate up
  - Remove
    - Percolate down
- Performance and Applications
- Conclusion
Observe: A Heap is a balanced binary tree!

- Percolating up and down are the time-dominant methods
  - Worst case: they just traverse all levels of the heap!

<table>
<thead>
<tr>
<th>Operations</th>
<th>ArrayList</th>
<th>LinkedList</th>
<th>BST</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>add an item</td>
<td>$O(n)$ avg/worst</td>
<td>$O(n)$ avg/worst</td>
<td>$O(H)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>contains an item</td>
<td>$O(n)$ avg/worst</td>
<td>$O(n)$ avg/worst</td>
<td>$O(H)$</td>
<td>$O(n)$ - why? **</td>
</tr>
<tr>
<td>remove an item</td>
<td>$O(n)$ avg/worst</td>
<td>$O(n)$ avg/worst</td>
<td>$O(H)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

** Heap doesn't help us search.

An item could be down either left or right subtree, so heaps don't help "prune" like a BST does.
What good is a Heap if it doesn't help you search quickly?

- Min-Heap guarantee: smallest element always at top (root) of the heap
  - Takes $O(1)$ time to identify... just return `the_data[0]`

These are the efficient (viable) operations...

- What can we do with them?

<table>
<thead>
<tr>
<th>Operations</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add an item</strong></td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td><strong>contains an item</strong></td>
<td>$O(n)$</td>
</tr>
<tr>
<td><strong>remove an item</strong></td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td><strong>find smallest item</strong></td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
One Application: HeapSort

- HeapSort is one interesting use of heaps...
  - Exploit the fact that it takes $O(1)$ to find min, $O(\log n)$ to remove it

- Algorithm:

```java
/**
 * HeapSort!
 * @param list a list of ints
 * @return a sorted list of ints
 */
public static int[] heapSort(int[] list) {
    Heap<Integer> heap = new Heap<>();

    // build min-heap -- add all items from the list
    for (int i = 0; i < list.length; i++) {
        heap.add(list[i]);
    }

    // re-build the list by continuously pulling the min from the heap
    for (int i = 0; i < list.length; i++) {
        list[i] = heap.remove(0);
    }

    return list;
}
```
Another App: Java's PriorityQueue\(<E>\) Class

- Most priority queue implementations are heap based?
- Don't care how items are ordered internally
  - Only care that the smallest item is at the head of the queue

<table>
<thead>
<tr>
<th>Signature</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>public boolean offer(E item)</td>
<td>(O(\log n))</td>
</tr>
<tr>
<td>public E remove()</td>
<td>(O(\log n))</td>
</tr>
<tr>
<td>public E poll()</td>
<td>(O(\log n))</td>
</tr>
<tr>
<td>public E peek()</td>
<td>(O(1))</td>
</tr>
<tr>
<td>public E element()</td>
<td>(O(1))</td>
</tr>
</tbody>
</table>
(Binary) Heaps

Min-Heap
- Properties
- Add
  - Percolate up
- Remove
  - Percolate down

Performance and Applications

Conclusion
Reminders:

- Hwk 6 (Huffman Encoding) due Monday!
  - Questions?
- Hwk 5 regrades due Wednesday

Last time…

- BST time analysis: it's all about tree height (and tree balance)
- Wrote a couple recursive BST methods (contains(), add(), smallest())

Today:

- An always-balanced, array-based binary tree