Heaps, continued.
Review from last lecture

• A heap is a complete binary tree whose last level of nodes is filled left-to-right and which satisfies the heap condition.

• Heap condition:
  • The root of every sub-tree is no smaller than any node in the sub-tree. (For max-heap).

• The heap condition ensures that the largest element is always stored at the root:
  • $O(1)$ time-cost for findLargest
  • $O(\log n)$ time-cost for removeLargest
Adding to a heap

• To add a new object \( o \) to the heap:
  
  • Create a new node \( n \) containing \( o \), and add \( n \) to the last level of the tree (at the left-most position).
  
  • This may violate the heap condition.

  • Repeatedly “bubble up” \( n \) towards the root whenever \( n > \text{parent}(n) \).
Adding to a heap

• To add a new object \( o \) to the heap:
  • Create a new node \( n \) containing \( o \), and add \( n \) to the last level of the tree (at the left-most position).
  • This may violate the heap condition.
  • Repeatedly “bubble up” \( n \) towards the root whenever \( n > \text{parent}(n) \).
Adding to a heap

- To add a new object $o$ to the heap:
  - Create a new node $n$ containing $o$, and add $n$ to the last level of the tree (at the left-most position).
  - This may violate the heap condition.
  - Repeatedly “bubble up” $n$ towards the root whenever $n > \text{parent}(n)$.
Adding to a heap

• To add a new object $o$ to the heap:
  • Create a new node $n$ containing $o$, and add $n$ to the last level of the tree (at the left-most position).
  • This may violate the heap condition.
  • Repeatedly “bubble up” $n$ towards the root whenever $n > \text{parent}(n)$. 

![Tree diagram with nodes 5, 4, 3, 1, 2, 8]
Adding to a heap

- To add a new object o to the heap:
  - Create a new node n containing o, and add n to the last level of the tree (at the left-most position).
  - This may violate the heap condition.
  - Repeatedly “bubble up” n towards the root whenever n > parent(n).
Adding to a heap

• To add a new object $o$ to the heap:
  • Create a new node $n$ containing $o$, and add $n$ to the last level of the tree (at the left-most position).

  • This may violate the heap condition.

  • Repeatedly “bubble up” $n$ towards the root whenever $n > \text{parent}(n)$.

The tree is now a valid heap again.
Removing the largest element from a heap

• The largest element is always stored at the top of the heap.
  • Hence, just remove the root.

• We must then replace it with something.
  • Remove the last node $n$ in the heap (right-most child of last level) and make it the new root of the tree.
    • This may violate the heap condition.
  • We will then have to recursively swap $n$ with one of its children (i.e., back down the tree) until the heap condition is restored. This is called “trickling down”.
Removing the *largest* element from a heap

```c
void removeLargest () {
    _nodeArray[0] = _nodeArray[_numNodes - 1];
    _numNodes--;
    trickleDown(0);
}

void trickleDown (int index) {
    If node at index is less than one of its children:
    Swap node with the largest child node. Recursive implementation
    trickleDown(largestChild(index));
}

Or

void trickleDown (int index) {
    While node at index is less than one of its children:
    Swap node with the largest child node. Iterative implementation
    index = largestChild(index);
}
```
Removing the largest element from a heap

```java
void removeLargest () {
    _nodeArray[0] = _nodeArray[_numNodes - 1];
    _numNodes--;
    trickleDown(0);
}

void trickleDown (int index) {
    While node at index is less than one of its children:
        Swap node with the largest child node.
    index = largestChild(index);
}
```

![Heap Diagram]

Removing the largest element from a heap

```c
void removeLargest () {
    _nodeArray[0] = _nodeArray[_numNodes - 1];
    _numNodes--;
    trickleDown(0);
}

void trickleDown (int index) {
    While node at index is less than one of its children:
        Swap node with the largest child node.
    index = largestChild(index);
}
```
Removing the largest element from a heap

```c
void removeLargest () {
    _nodeArray[0] = _nodeArray[_numNodes - 1];
    _numNodes--;
    trickleDown(0);
}

void trickleDown (int index) {
    While node at index is less than one of its children:
        Swap node with the largest child node.
    index = largestChild(index);
}
```

```
2
/ \
5   3
| |
3 4
```
Removing the *largest* element from a heap

```java
void removeLargest () {
    _nodeArray[0] = _nodeArray[_numNodes - 1];
    _numNodes--;
    trickleDown(0);
}

void trickleDown (int index) {
    While node at index is less than one of its children:
        Swap node with the largest child node.
    index = largestChild(index);
}
```

True

```
  2
 / \
5   3
 /   /
3   4
```
Removing the largest element from a heap

```java
void removeLargest () {
    _nodeArray[0] = _nodeArray[_numNodes - 1];
    _numNodes--;
    trickleDown(0);
}

void trickleDown (int index) {
    While node at index is less than one of its children:
        Swap node with the largest child node.
    index = largestChild(index);
}
```

![Heap Diagram]
Removing the largest element from a heap

```java
void removeLargest () {
    _nodeArray[0] = _nodeArray[_numNodes - 1];
    _numNodes--;
    trickleDown(0);
}

void trickleDown (int index) {
    While node at index is less than one of its children:
        Swap node with the largest child node.
    index = largestChild(index);
}
```

![Heap Diagram]
Removing the **largest** element from a heap

```java
void removeLargest () {
    _nodeArray[0] = _nodeArray[_numNodes - 1];
    _numNodes--;
    trickleDown(0);
}

void trickleDown (int index) {
    While node at index is less than one of its children:
        Swap node with the largest child node.
    index = largestChild(index);
}
```

True
Removing the \textit{largest} element from a heap

```java
void removeLargest () {
    _nodeArray[0] = _nodeArray[_numNodes - 1];
    _numNodes--;
    trickleDown(0);
}

void trickleDown (int index) {
    While node at index is less than one of its children:
        Swap node with the \textit{larger} child node.
    index = largestChild(index);
}
```

It’s crucial we swap with the \textit{larger} child to maintain the heap condition.
Removing the largest element from a heap

```java
void removeLargest () {
    _nodeArray[0] = _nodeArray[_numNodes - 1];
    _numNodes--;
    trickleDown(0);
}

void trickleDown (int index) {
    While node at index is less than one of its children:
    Swap node with the largest child node.
    index = largestChild(index);
}
```

![Binary heap diagram]

- 5
  - 4
    - 3
      - 2
Removing the largest element from a heap

```java
void removeLargest () {
    _nodeArray[0] = _nodeArray[_numNodes - 1];
    _numNodes--;
    trickleDown(0);
}

void trickleDown (int index) {
    While node at index is less than one of its children:
    Swap node with the largest child node.
    index = largestChild(index);
}
```
Removing the largest element from a heap

```java
void removeLargest () {
    _nodeArray[0] = _nodeArray[_numNodes - 1];
    _numNodes--;
    trickleDown(0);
}

void trickleDown (int index) {
    While node at index is less than one of its children:
        Swap node with the largest child node.
    index = largestChild(index);
}
```

Done.
Finding an arbitrary node

• Heaps offer fast access to the largest node in the heap.

• However, despite their binary tree representation, they offer no advantage over simple lists in terms of finding an arbitrary element.

• If the element o that the user wishes to find is not the largest, then o could be anywhere in the heap.

• This contrasts with binary search trees (more later).

• Hence, to find an object o within a heap, we must search through the entire heap.
Finding an arbitrary node

public T find (T o) {
    final int index = findNode(0, o);
    if (index < 0) {
        throw new NoSuchElementException();
    }
    return _nodeArray[index];
}

private int findNode (int rootIdx, T o) {
    if (_nodeArray[rootIdx].equals(o)) {
        return rootIdx;
    }
    int idx;
    if (leftChild(rootIdx) < _numNodes &&
        (idx = find(leftChild(rootIdx), o)) >= 0) {
        return idx;
    } else if (rightChild(rootIdx) < _numNodes &&
               (idx = find(rightChild(rootIdx), o)) >= 0) {
        return idx;
    } else {
        return -1;
    }
}
Finding an arbitrary node

But this is much easier (and slightly faster too).

```java
int findNode (T o) {
    for (int i = 0; i < _numNodes; i++) {
        if (_nodeArray[i].equals(o)) {
            return i;
        }
    }
    return -1;  // Or any other appropriate value.
}
```

- This is one of the conveniences of representing the tree as an array.
- Only possible for complete trees in which there are no “holes” in the array (i.e., missing child nodes).
Removing an arbitrary node

- Removing an arbitrary node requires that we first find the node \( n \) to be removed.
- We can use the findNode(o) method we just constructed.
- Once found, we can swap the last node in the heap (right-most child of last level) with \( n \).
- Then we just trickleDown that node and we’re done, right?
Removing an arbitrary node

• Removing an arbitrary node requires that we first find the node $n$ to be removed.

• We can use the `findNode(o)` method we just constructed.

• Once found, we can swap the last node in the heap (right-most child of last level) with $n$.

• Then we just trickleDown that node and we’re done, right?  Wrong.
Removing an arbitrary node

- The above procedure worked for \texttt{removeLargest()} because we always started from the top (root) of the heap.

- By trickling down from the top, we guarantee that every sub-tree (starting from the very top) is a valid heap.

- When removing an \textit{arbitrary} node, the \texttt{trickleDown} process will “fix” the sub-tree rooted at \( n \), but \textit{not necessarily} the whole tree.

- What’s an example heap in which this problem would arise?
Removing an arbitrary node

• Suppose we wish to remove the node containing 4.
• If we just replace it with the “last” node (6)...
Removing an arbitrary node

- ...then the `trickleDown()` method will do nothing (6 is already bigger than its children).
- Moreover, 6 is now bigger than its parent -- a *violation of the heap condition*.
Removing an arbitrary node

- In a correct implementation of `remove(o)` for arbitrary `o`, we need to `sometimes bubbleUp` and `sometimes trickleDown`:

```java
void remove (T o) {
    Find the node `n` containing `o`.
    Replace `n` with the "last" node `l` in the heap.
    If `l < n`:
        trickleDown on `n`.
    Else:
        bubbleUp on `n`.
}
```
Removing an arbitrary node

- In a correct implementation of `remove(o)` for arbitrary `o`, we need to *sometimes* `bubbleUp` *and* *sometimes* `trickleDown`:

```java
void remove (T o) {
    // Find the node n containing o.
    // Replace n with the "last" node l in the heap.
    // If l < n:
    //     trickleDown on n.
    // Else:
    //     bubbleUp on n.
}
```

![Valid heap diagram]
Removing an arbitrary node

• In a correct implementation of `remove(o)` for arbitrary `o`, we need to *sometimes* `bubbleUp` *and* *sometimes* `trickleDown`:

```java
void remove (T o) {
    Find the node `n` containing `o`.
    Replace `n` with the "last" node `l` in the heap.
    If `l < n`:
        `trickleDown` on `n`.
    Else:
        `bubbleUp` on `n`.
}
```

![Binary heap example]

• In a correct implementation of `remove(o)` for arbitrary `o`, we need to *sometimes* `bubbleUp` *and* *sometimes* `trickleDown`:

```java
void remove (T o) {
    Find the node `n` containing `o`.
    Replace `n` with the "last" node `l` in the heap.
    If `l < n`:
        `trickleDown` on `n`.
    Else:
        `bubbleUp` on `n`.
}
```

![Binary heap example]
Removing an arbitrary node

• In a correct implementation of remove(o) for arbitrary o, we need to sometimes `bubbleUp` and sometimes `trickleDown`:

```java
void remove (T o) {
    Find the node n containing o.
    Replace n with the “last” node l in the heap.
    If l < n:  // n was 4, l is 6
        trickleDown on n.
    Else:
        bubbleUp on n.
}
```

```plaintext
      9
     / \
   5   8
  / \ /\
3 6 1 7 8
```

---

Wednesday, July 25, 12
Removing an arbitrary node

- In a correct implementation of `remove(o)` for arbitrary `o`, we need to sometimes `bubbleUp` and sometimes `trickleDown`:

```java
void remove (T o) {
    Find the node `n` containing `o`.
    Replace `n` with the "last" node `l` in the heap.
    If `l < n`:  // `n` was 4, `l` is 6
        `trickleDown` on `n`.
    Else:
        `bubbleUp` on `n`.
}
```

```plaintext
6 1 7 8
3 ...
```

```
9
5 8
3
```
Removing an arbitrary node

- In a correct implementation of `remove(o)` for arbitrary `o`, we need to sometimes `bubbleUp` and sometimes `trickleDown`:

```java
void remove (T o) {
    Find the node n containing o.
    Replace n with the "last" node l in the heap.
    If l < n:  // n was 4, l is 6
        trickleDown on n.
    Else:
        bubbleUp on n.
}
```

```plaintext
6 8 5 1 9 7 8 3 ...
```
Removing an arbitrary node

• In a correct implementation of \texttt{remove(o)} for arbitrary \texttt{o}, we need to sometimes \texttt{bubbleUp} and sometimes \texttt{trickleDown}:

\begin{verbatim}
void remove (T o) {
    Find the node \texttt{n} containing \texttt{o}.
    Replace \texttt{n} with the “last” node \texttt{l} in the heap.
    If \texttt{l} < \texttt{n}:  // \texttt{n} was 4, \texttt{l} is 6
        \texttt{trickleDown} on \texttt{n}.
    Else:
        \texttt{bubbleUp} on \texttt{n}.
}
\end{verbatim}

Valid heap again.
Heap operations: time costs

- The implementations for the add/find/removeLargest/remove methods depend on the methods bubbleUp and trickleDown.

```java
void bubbleUp (int idx) {
    While node at idx is “larger” than its parent:
        Swap data in the node and its parent;
        Set idx to be parentIdx(idx);
}
```

- At each loop iteration, `idx` moves one step closer from a leaf to the root of the heap.

- Hence, loop can execute maximum of `h` times (`h` is tree height). For heap of `n` nodes, `h` is $\log_2(n)$.

- Inside loop, the time cost is about 2 operations.

- Hence, time cost is $O(\log n)$. 
Heap operations: time costs

- void trickleDown (int index) {
    While node at index is less than one of its children:
    Swap node with the larger child node.
    index = largerChild(index);
}

- At each loop iteration, idx moves one step closer from the root of the heap to a leaf.
  - Hence, number of iterations is bounded by \( h = \log_2(n) \).
- Inside loop, the time cost is about 2 operations.
- Hence, time cost is \( O(\log n) \).
Heap operations: time costs

- Given the time costs of `bubbleUp` and `trickleDown`, we can compute the worst-case time costs of the fundamental heap operations:
  - `add(o)`: $O(1) + O(\log n) = O(\log n)$
    - Append a new node to the heap. $O(1)$
    - Bubble it up. $O(\log n)$
  - `removeLargest()`: $O(1) + O(\log n) = O(\log n)$
    - Swap last node with root. $O(1)$
    - Trickle root down. $O(\log n)$
Heap operations: time costs

- **find(o):** $O(n)$
  - Search through all nodes. $O(n)$

- **remove():** $O(n) + O(1) + O(\log n) = O(n)$
  - Find the node. $O(n)$
  - Swap node-to-remove with root. $O(1)$
  - *Either* trickle node down *or* bubble it up. $O(\log n)$
General heaps

• We have just described the minimal implementation of a binary heap.

• Binary heaps are the most common.

• In theory, however, any tree can be a heap as long as it satisfies the heap condition that the root of every sub-tree is no smaller than any node in the sub-tree.

• In particular, we can define a $d$-ary tree in which each node has $d$ child nodes (instead of always 2).
$d$-ary heaps

3-ary (ternary) heap

4-ary (quaternary) heap

$d$-ary heap

...
d-ary heaps: Why?

- d-ary heaps can offer a time cost savings compared to binary heaps.

- Consider:
  - The height $h$ of a binary heap is at most $\log_2(n)$.
  - The height $h$ of a ternary heap is at most $\log_3(n)$.
  - The height $h$ of a d-ary heap is at most $\log_d(n)$.

- As the base of the logarithm ($d$) gets larger, the value of the logarithm itself grows smaller.

- Hence, for larger $d$, operations that depend on the height of the tree will become faster.
d-ary heaps: Why?

- On the other hand, as \( d \) increases, so does the number of children per node.

- The time cost of \texttt{trickleDown} (but not \texttt{bubbleUp}) is affected by the number of children:
  
  ```java
  void trickleDown (int index) {
    While node at index is less than one of its children:
      ...
  }
  ```

  - Each loop iteration implicitly requires a comparison to all \( d \) children.

- The loop runs for at most \( h \) iterations (\( h = \log_d n \)), and each iteration takes at least \( d \) operations.

- Hence, time cost for \texttt{trickleDown} is \( O(hd) = O(d \log_d n) \).
bubbleUp: $O(\log_d n)$

bubbleUp is faster when $d$ is large.
trickleDown: $O(d \log_d n)$

trickleDown is faster when $d$ is small.
trickleDown versus bubbleUp

• In scenarios where \textit{bubbleUp} is called more frequently than \textit{trickleDown}, better time costs can be achieved using a larger value of $d$.

• Such scenarios can happen with \textit{priority queues} when the user changes the \textit{priority} of the data while they are still in the heap.
Increasing/decreasing priority

• Example:
  
  ```java
  heap.add(o1);  // Priority 7
  heap.add(o2);  // Priority 6
  ...
  heap.add(o7);  // Priority 6
  ```
Increasing/decreasing priority

• Example:
  heap.add(o1);  // Priority 7
  heap.add(o2);  // Priority 6
  ...
  heap.add(o7);  // Priority 6

• Later on:
  heap.increasePriority(o7);

Now we need to bubbleUp o7.
Increasing/decreasing priority

• Example:
  heap.add(o1);  // Priority 7
  heap.add(o2);  // Priority 6
  ...
  heap.add(o7);  // Priority 6

• Later on:
  heap.increasePriority(o7);

Done.
trickleDown versus bubbleUp

• *Increasing* the priority of an item requires `bubbleUp` to be called to maintain the heap condition.

• *Decreasing* the priority of an item requires `trickleDown` to be called to maintain the heap condition.

• In some applications, the user may want to *increase* the priority of items more frequently than they will *decrease* their priority.

• In this case, `bubbleUp` will be called more frequently than `trickleDown`.

• By using a $d$-ary heap and setting $d>2$, the time cost of the priority queue may be reduced compared to a binary heap.