Priority Queues

Priority queue

- A stack is first in, last out
- A queue is first in, first out
- A priority queue is *least-first-out*
  - The "smallest" element is the first one removed
    - (You could also define a *largest-first-out* priority queue)
  - The definition of "smallest" is up to the programmer (for example, you might define it by implementing `Comparator` or `Comparable`)
  - If there are several "smallest" `elements`, the implementer must decide which to remove first
    - Remove any "smallest" element (don't care which)
    - Remove the first one added

A priority queue ADT

- Here is one possible ADT:
  - `PriorityQueue()`: a constructor
  - `void add(Comparable o)`: inserts `o` into the priority queue
  - `Comparable remove()`: removes and returns the least element
  - `Comparable peek()`: returns (but does not remove) the least element
  - `boolean isEmpty()`: returns true iff empty
  - `int size()`: returns the number of elements
  - `void clear()`: discards all elements

Evaluating implementations

- When we choose a data structure, it is important to look at usage patterns
  - If we load an array once and do thousands of searches on it, we want to make searching fast—so we would probably sort the array
  - If we load a huge array and expect to do only a few searches, we probably *don't* want to spend time sorting the array
- For almost all uses of a queue (including a priority queue), we eventually remove everything that we add
- Hence, when we analyze a priority queue, neither "add" nor "remove" is more important—we need to look at the timing for "add + remove"

Array implementations

- A priority queue could be implemented as an *unsorted* array (with a count of elements)
  - Adding an element would take O(?) time (why?)
  - Removing an element would take O(?) time (why?)
  - Hence, adding *and* removing an element takes O(?) time
  - This is an inefficient representation
- A priority queue could be implemented as a *sorted* array (again, with a count of elements)
  - Adding an element would take O(?) time (why?)
  - Removing an element would take O(?) time (why?)
  - Hence, adding *and* removing an element takes O(?) time
  - Again, this is inefficient

Array implementations

- A priority queue could be implemented as an *unsorted* array (with a count of elements)
  - Adding an element would take `O(1)` time (why?)
  - Removing an element would take `O(n)` time (why?)
  - Hence, adding *and* removing an element takes `O(n)` time
  - This is an inefficient representation
- A priority queue could be implemented as a *sorted* array (again, with a count of elements)
  - Adding an element would take `O(n)` time (why?)
  - Removing an element would take `O(1)` time (why?)
  - Hence, adding *and* removing an element takes `O(n)` time
  - Again, this is inefficient

Linked list implementations

- A priority queue could be implemented as an *unsorted* linked list
  - Adding an element would take O(?) time (why?)
  - Removing an element would take O(?) time (why?)
- A priority queue could be implemented as a *sorted* linked list
  - Adding an element would take O(?) time (why?)
  - Removing an element would take O(?) time (why?)
- As with array representations, adding *and* removing an element takes O(?) time
  - Again, these are inefficient implementations

Linked list implementations

- A priority queue could be implemented as an *unsorted* linked list
  - Adding an element would take `O(1)` time (why?)
  - Removing an element would take `O(n)` time (why?)
- A priority queue could be implemented as a *sorted* linked list
  - Adding an element would take `O(n)` time (why?)
  - Removing an element would take `O(1)` time (why?)
- As with array representations, adding *and* removing an element takes `O(n)` time
  - Again, these are inefficient implementations

Binary tree implementations

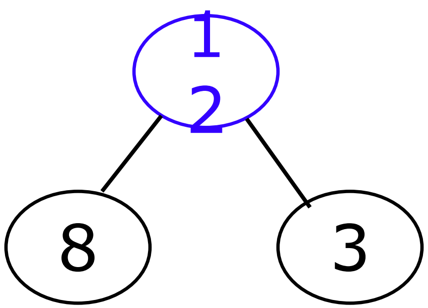
- A priority queue could be represented as a binary search tree
  - Insertion times would range from O(?) to O(?) (why?)
  - Removal times would range from O(?) to O(?) (why?)
- A priority queue could be represented as a *balanced* binary search tree
  - Insertion and removal could destroy the balance
  - We need an algorithm to *rebalance* the binary tree
  - Good rebalancing algorithms require only O(?) time, but are complicated

Binary tree implementations

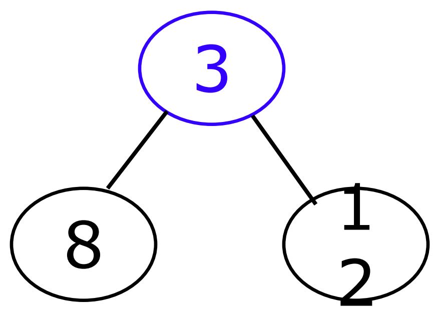
- A priority queue could be represented as a binary search tree
  - Insertion times would range from O(log n) to O(n) (why?)
  - Removal times would range from O(log n) to O(n) (why?)
- A priority queue could be represented as a *balanced* binary search tree
  - Insertion and removal could destroy the balance
  - We need an algorithm to *rebalance* the binary tree
  - Good rebalancing algorithms require only O(log n) time, but are complicated

Heap implementation

- A priority queue can be implemented as a heap
- In order to do this, we have to define the *heap property*
  - In Heapsort, a node has the heap property if it is *at least as large* as its children
  - For a priority queue, we will define a node to have the heap property if it is *as least as small* as its children (since we are using smaller numbers to represent higher priorities)

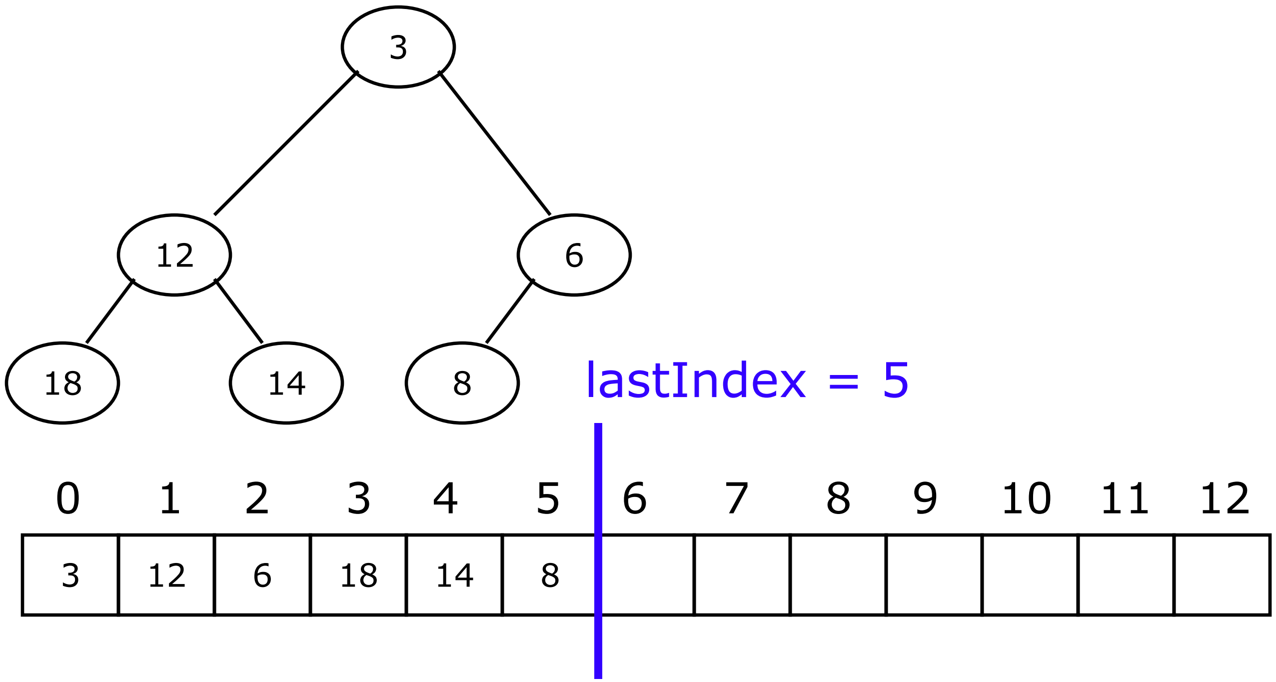


Heapsort: Blue node has the heap property



Priority queue: Blue node has the heap property

Array representation of a heap



- Left child of node i is 2*i + 1, right child is 2*i + 2
  - Unless the computation yields a value larger than lastIndex, in which case there is no such child
- Parent of node i is (i – 1)/2
  - Unless i == 0

Using the heap

- To add an element:
  - Increase `lastIndex` and put the new value there
  - Reheap the newly added node
    - This is called up-heap bubbling or percolating up
    - Up-heap bubbling requires `O(log n)` time
- To remove an element:
  - Remove the element at location `0`
  - Move the element at location `lastIndex` to location `0`, and decrement `lastIndex`
  - Reheap the new root node (the one now at location `0`)
    - This is called down-heap bubbling or percolating down
    - Down-heap bubbling requires `O(log n)` time
- Thus, it requires `O(log n)` time to add *and* remove an element

Comments

- A priority queue is a data structure that is designed to return elements in order of priority
- Efficiency is usually measured as the *sum* of the time it takes to add and to remove an element
  - Simple implementations take O(n) time
  - Heap implementations take O(log n) time
  - Balanced binary tree implementations take O(log n) time
  - Binary tree implementations, without regard to balance, can take O(n) (linear) time
- Thus, for any sort of heavy-duty use, heap or balanced binary tree implementations are better

Java 5 java.util.PriorityQueue

- Java 5 finally has a PriorityQueue class, based on heaps
  - Has redundant methods because it implements two similar interfaces
  - PriorityQueue<E> queue = new PriorityQueue<E>();
    - Uses the *natural ordering* of elements (that is, Comparable)
    - There is another constructor that takes a Comparator argument
  - boolean add(E o) – from the Collection interface
  - boolean offer(E o) – from the Queue interface
  - E peek() – from the Queue interface
  - boolean remove(Object o) – from the Collection interface
  - E poll() – from the Queue interface (returns null if queue is empty)
  - void clear()
  - int size()

The End