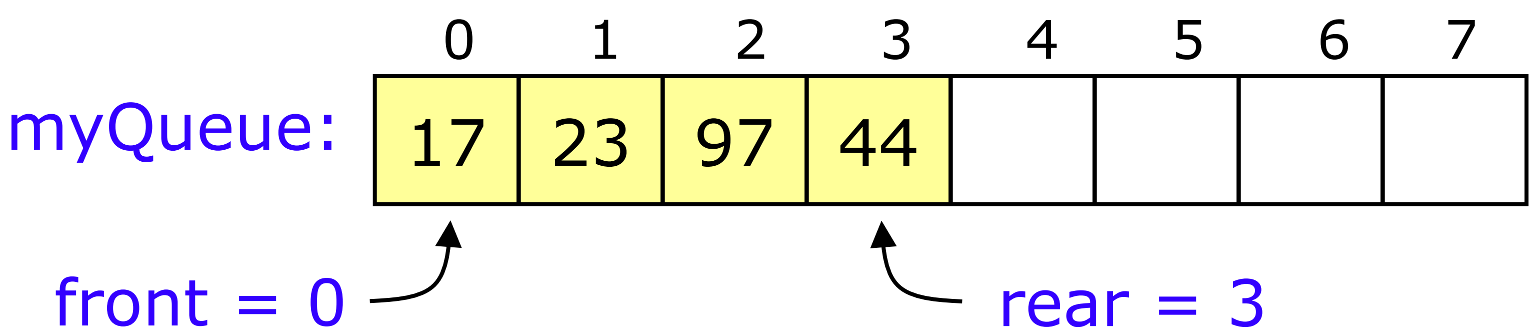
Queues

Stacks, Queues, and Deques

- A stack is a last in, first out (LIFO) data structure
  - Items are removed from a stack in the reverse order from the way they were inserted
- A queue is a first in, first out (FIFO) data structure
  - Items are removed from a queue in the same order as they were inserted
- A deque is a double-ended queue—items can be inserted and removed at either end

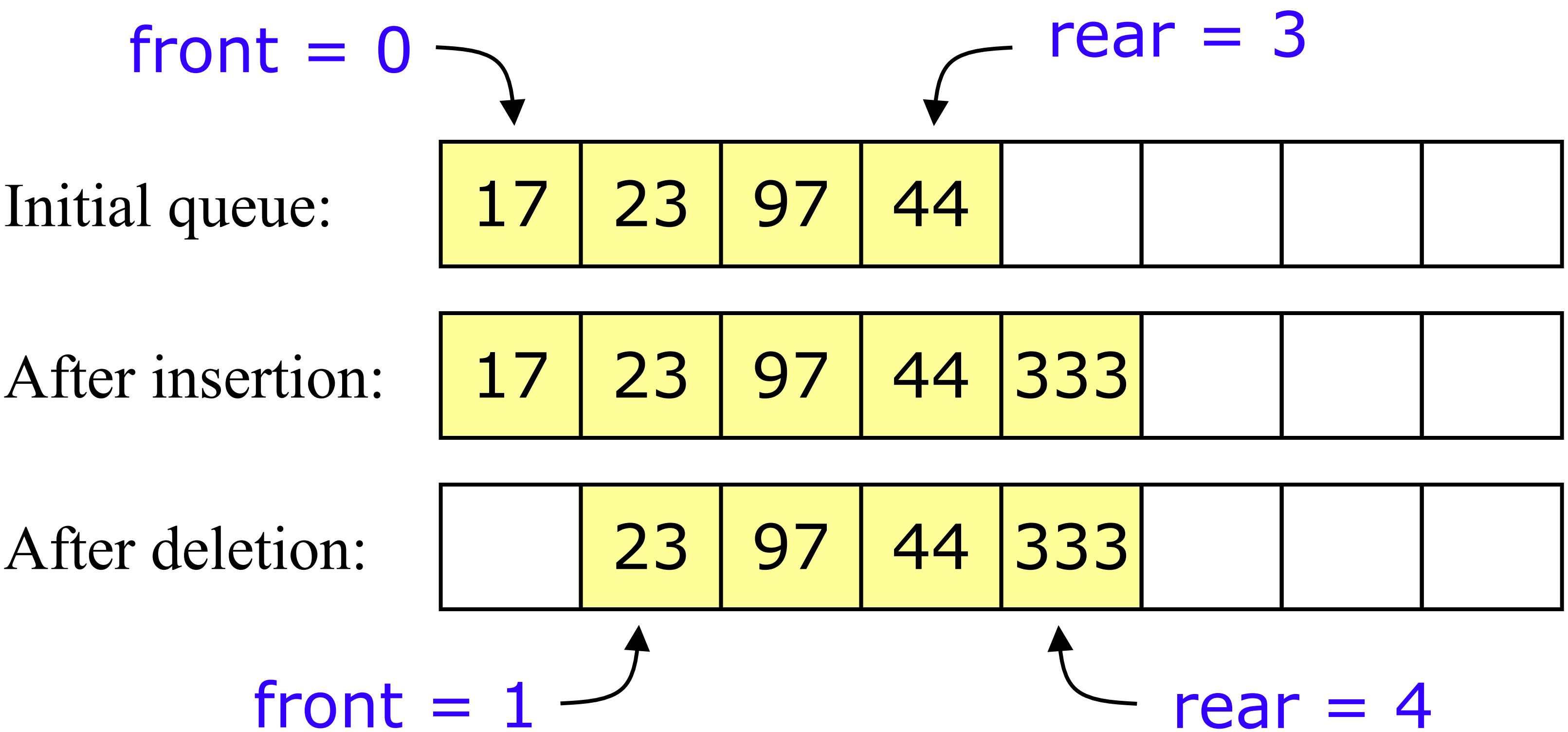
Array implementation of queues

- A queue is a first in, first out (FIFO) data structure
- This is accomplished by inserting at one end (the rear) and deleting from the other (the front)



- **To insert:** put new element in location 4, and set rear to 4
- **To delete:** take element from location 0, and set front to 1

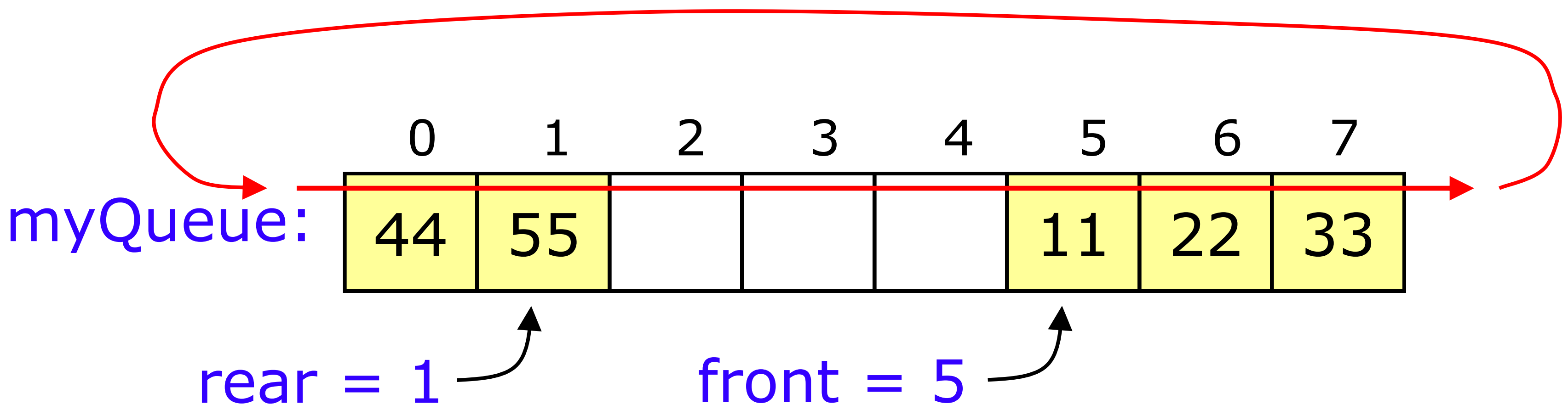
Array implementation of queues



- Notice how the array contents "crawl" to the right as elements are inserted and deleted
- This will be a problem after a while!

Circular arrays

- We can treat the array holding the queue elements as circular (joined at the ends)



- Elements were added to this queue in the order `11`, `22`, `33`, `44`, `55`, and will be removed in the same order
- Use: `front = (front + 1) % myQueue.length;` and: `rear = (rear + 1) % myQueue.length;`

Full and empty queues

- If the queue were to become completely full, it would look like this:

myQueue:

0	1	2	3	4	5	6	7
44	55	66	77	88	11	22	33

rear = 4 front = 5

- If we were then to remove all eight elements, making the queue completely empty, it would look like this:

myQueue:

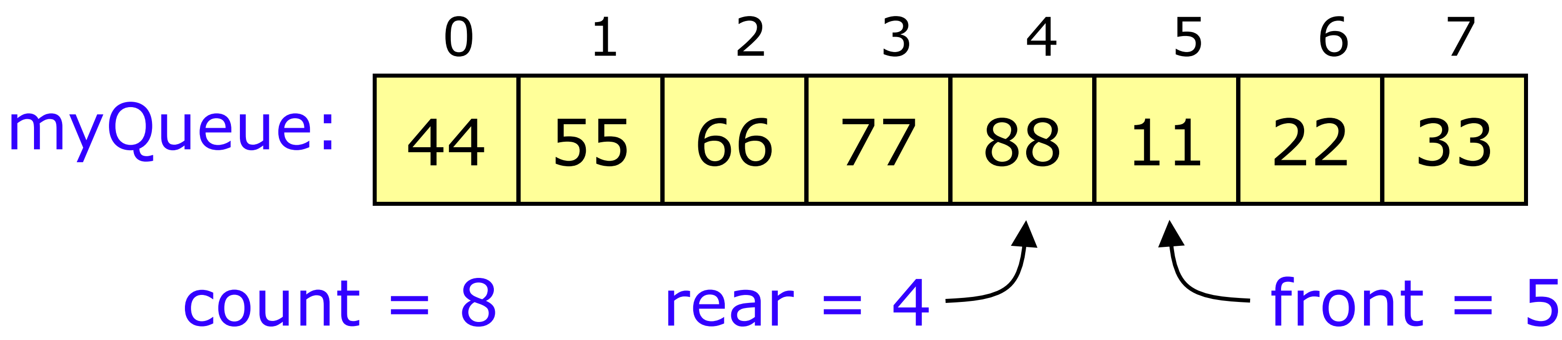
0	1	2	3	4	5	6	7

rear = 4 front = 5

This is a problem!

Full and empty queues: solutions

- **Solution:** Keep an additional variable



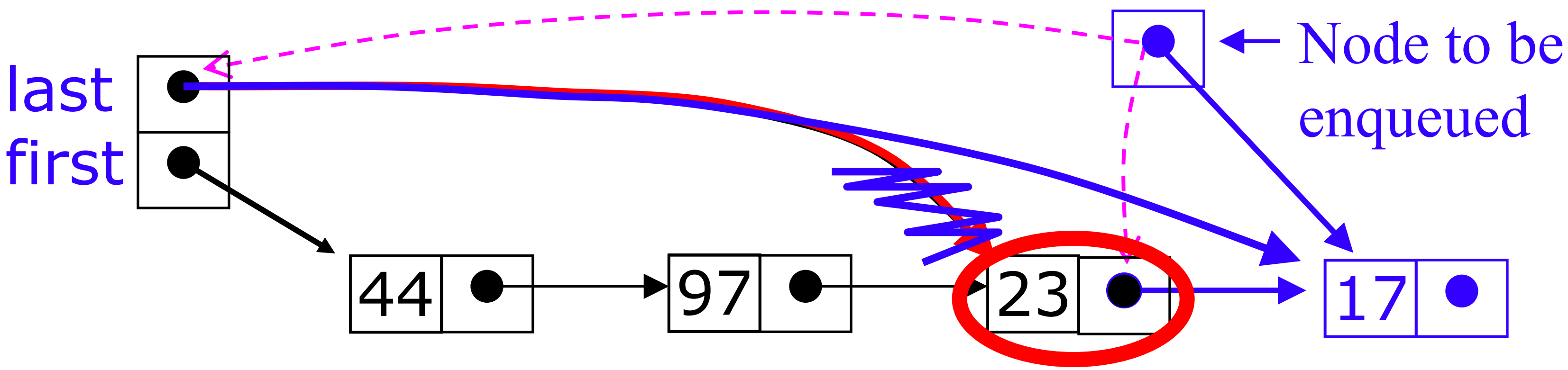
Linked-list implementation of queues

- In a queue, insertions occur at one end, deletions at the other end
- Operations at the front of a singly-linked list (SLL) are O(1), but at the other end they are O(n)
  - Because you have to find the last element each time
- BUT: there is a simple way to use a singly-linked list to implement both insertions and deletions in O(1) time
  - You always need a pointer to the first thing in the list
  - You can keep an additional pointer to the *last* thing in the list

SLL implementation of queues

- In an SLL you can easily find the successor of a node, but not its predecessor
  - Remember, pointers (references) are one-way
- If you know where the *last* node in a list is, it's hard to remove that node, but it's easy to add a node after it
- Hence,
  - Use the *first* element in an SLL as the *front* of the queue
  - Use the *last* element in an SLL as the *rear* of the queue
  - Keep pointers to *both* the front and the rear of the SLL

Enqueueing a node



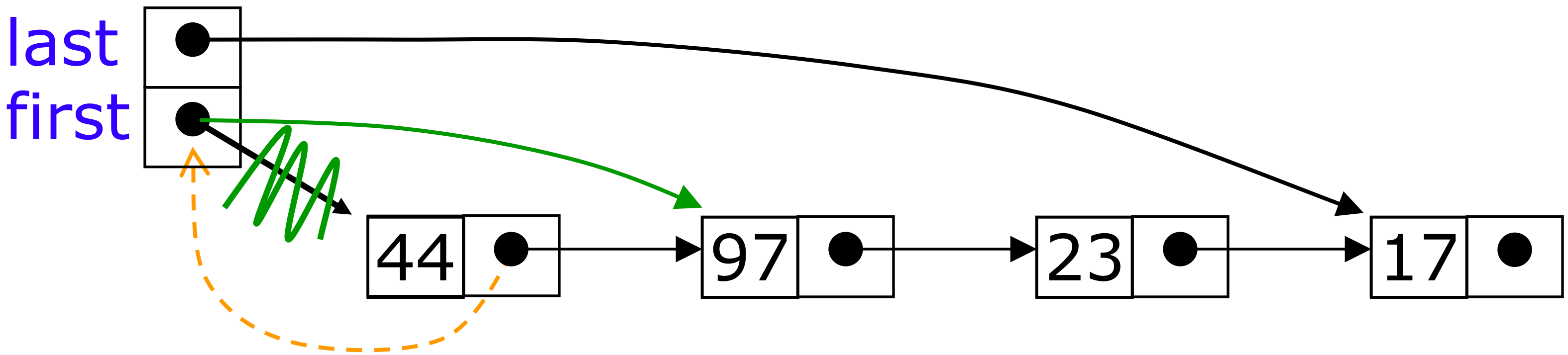
To enqueue (add) a node:

Find the current last node

Change it to point to the new last node

Change the last pointer in the list header

Dequeuing a node



- To dequeue (remove) a node:
  - Copy the pointer from the first node into the header

Queue implementation details

- With an array implementation:
  - you can have both overflow and underflow
  - you should set deleted elements to `null`

- With a linked-list implementation:
  - you can have underflow
  - overflow is a global out-of-memory condition
  - there is no reason to set deleted elements to `null`

A queue ADT

- Here are the usual operations on a queue:
  - Queue(): the constructor
  - boolean isEmpty()
  - Object enqueue(Object item): add an element at the rear
  - Object dequeue(): remove an element from the front
  - Object peek(): look at the front element
  - int search(Object o): Returns the 1-based position from the front of the queue

java.util Interface Queue<E>

- Java provides a queue *interface* and several implementations
- boolean add(E e)
  - Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an IllegalStateException if no space is currently available.
- E element()
  - Retrieves, but does not remove, the head of this queue.
- boolean offer(E e)
  - Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.
- E peek()
  - Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
- E poll()
  - Retrieves and removes the head of this queue, or returns null if this queue is empty.
- E remove()
  - Retrieves and removes the head of this queue.

Source: Java 6 API

A deque ADT

- Here are the operations expected of a deque:
  - Deque(): the constructor
  - boolean isEmpty()
  - Object addAtFront(Object item)
  - Object addAtRear(Object item)
  - Object getFromFront()
  - Object getFromRear()
  - Object peekAtFront()
  - Object peekAtRear()
  - int search(Object o): Returns the 1-based position from the front of the deque

Using ArrayLists

- You could implement a deque with java.util.ArrayList:
  - addAtFront(Object) → add(0, Object)
  - addAtRear(Object item) → add(Object)
  - getFromFront() → remove(0)
  - getFromRear() → remove(size() – 1)
- If you did this, should you ***extend*** ArrayList or use it as a field in your Deque class?
- Would this be a good implementation?
- Why or why not?

java.util Interface Deque<E>

- Java 6 has a Deque interface
- There are 12 methods:
  - Add, remove, or examine an element...
  - ...at the head or the tail of the queue...
  - ...and either throw an exception, or return a special value (null or false) if the operation fails

	First Element (Head)		**Last Element (Tail)**	
	Throws exception	*Special value*	*Throws exception*	*Special value*
Insert	`addFirst(e)`	`offerFirst(e)`	`addLast(e)`	`offerLast(e)`
Remove	`removeFirst()`	`pollFirst()`	`removeLast()`	`pollLast()`
Examine	`getFirst()`	`peekFirst()`	`getLast()`	`peekLast()`

Source: Java 6 API

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 removeLeast()`: removes and returns the least element
  - `Comparable getLeast()`: 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(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

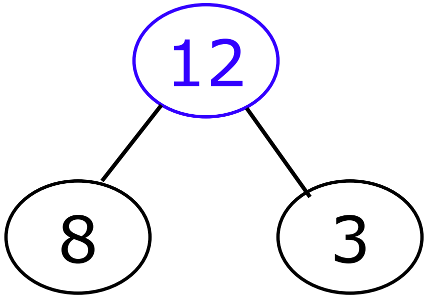
- A priority queue could be implemented as an *unsorted* linked list
  - Adding an element would take `O(1)` time (why?)
  - Removing an element would take `O(n)` time (why?)
- A priority queue could be implemented as a *sorted* linked list
  - Adding an element would take `O(n)` time (why?)
  - Removing 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

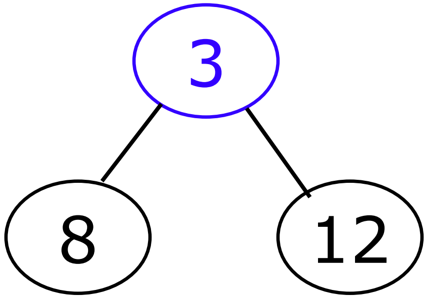
- A priority queue could be represented as a (not necessarily balanced) binary search tree
  - Insertion times would range from O(log n) to O(n) (why?)
  - Removal times would range from O(log n) to O(n) (why?)
- A priority queue could be represented as a *balanced* binary search tree
  - Insertion and 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)

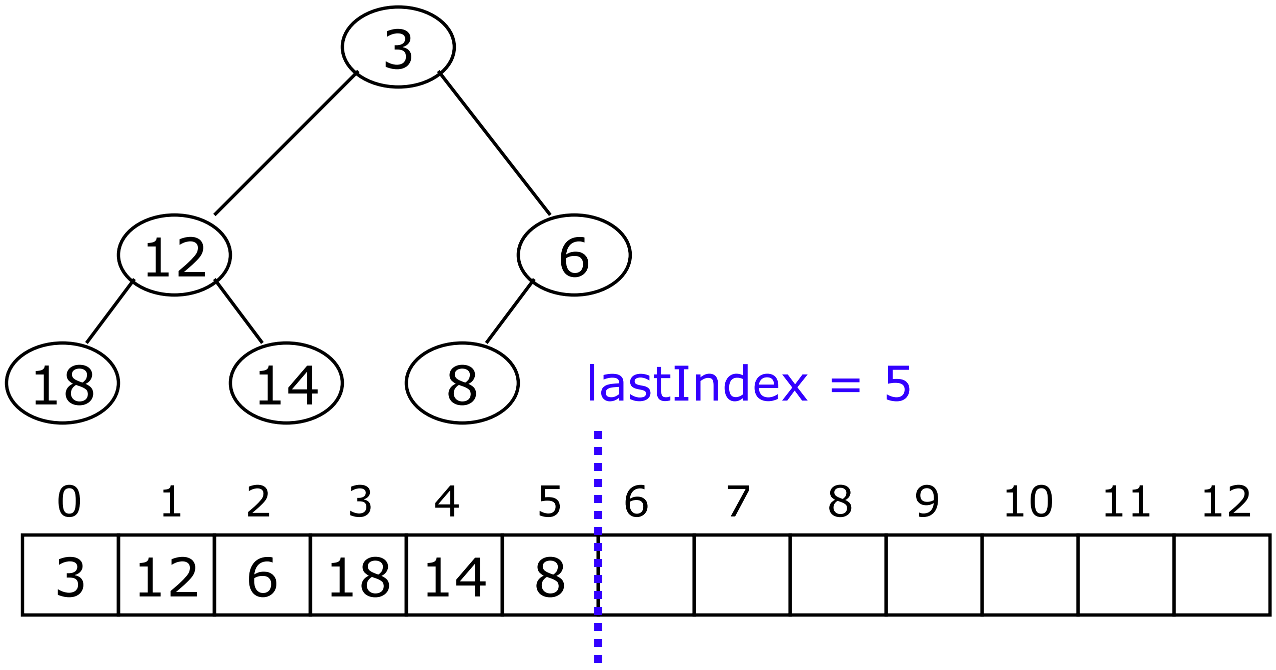


Heapsort: Blue node has the heap property



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
    - 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
    - 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()