Thread-safe Priority Queues

Priyankit, Gabriel, Taranpreet

Overview

- Priority Queues
- Blocking Queues
- Java's implementation of thread safe PriorityBlockingQueue
- Limitations of the PriorityBlockingQueue
- Alternative implementations (Pipelined PQ & Lock-free PQ)
- Project goals
- Testing and Evaluation of alternate implementation.

Priority Queues

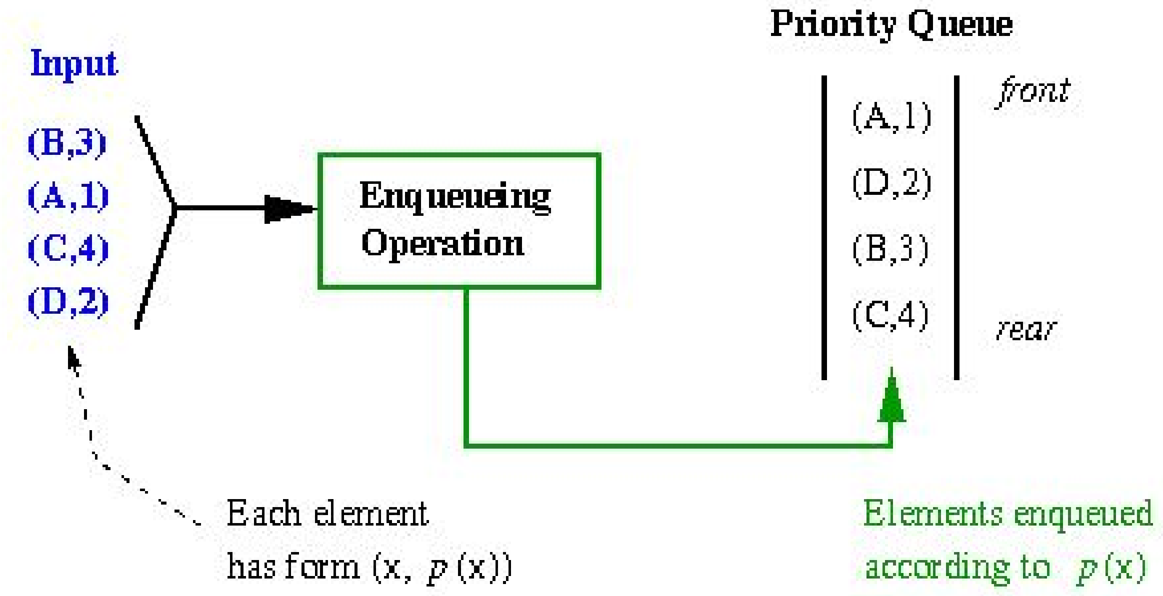
Standard queue - FIFO.

Priority queue - element with higher priority served before element with a lower priority.

- Sorted order imposed on items it contains (in queue terms).
- Highest priority = head of queue.
- Efficient insertion and removal (sequential), by using a binary heap structure.

Use cases - Job scheduling algorithms in OS, algorithms that require a heap.

Priority Queues

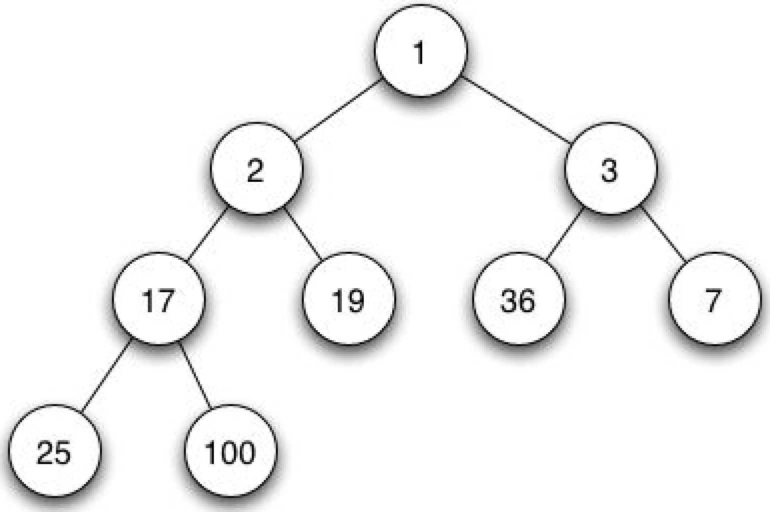


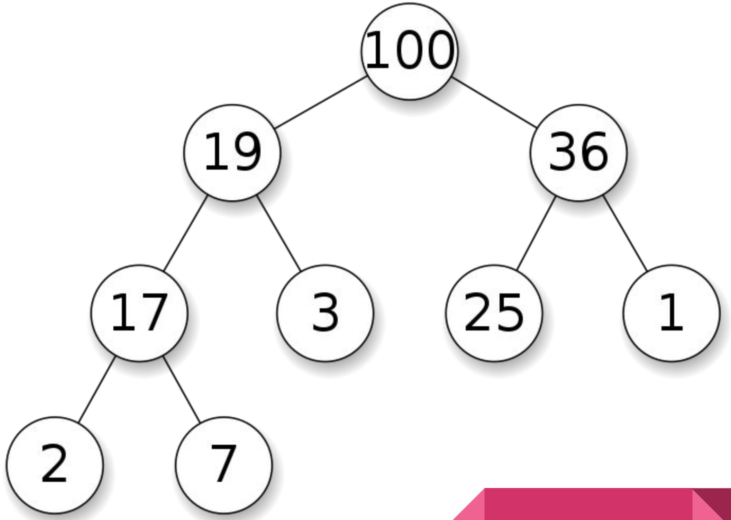
Binary Heaps

Binary heaps are binary trees with the two additional properties:

- Complete binary trees
- Heap property - the key stored in each node is either greater than (≥) or less than (≤) the keys in the node's children (one or the other)

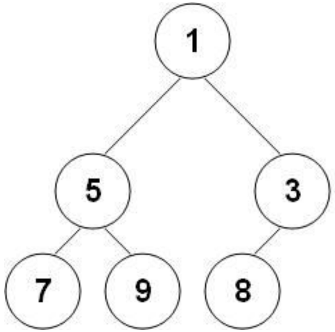
Binary Heaps





Binary Heaps - Implementation

- Arrays
  - Root at index 0
  - Each element at index i has children at indices $2i + 1$ and $2i + 2$ and parent at $\lfloor (i - 1) / 2 \rfloor$



Node	1	5	3	7	9	8
Index	0	1	2	3	4	5

Binary Heaps - Implementation II

Node struct

```
{
    Object value;
    Node left, right;
}
```

O(log n) lookup

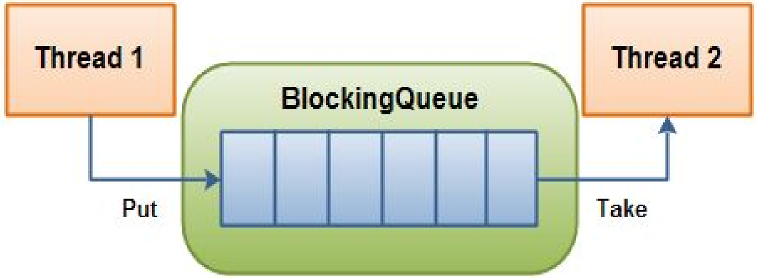
Blocking Queue

Interface that is primarily used for producer/consumer problem.

A queue that is thread-safe. All queuing methods are atomic.

- `put(E e)` : Insert elements. If the queue is full, it waits for the space to be available.
- `E take()` : Retrieves and removes the element from the head. If queue is empty it waits for the element to be available.

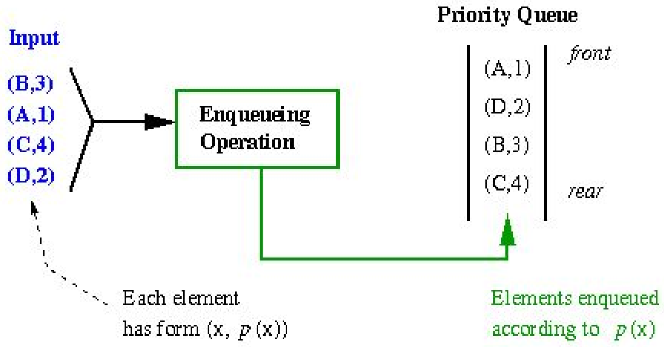
Queue can grow to a fixed size (bounded) or have no set size (unbounded)

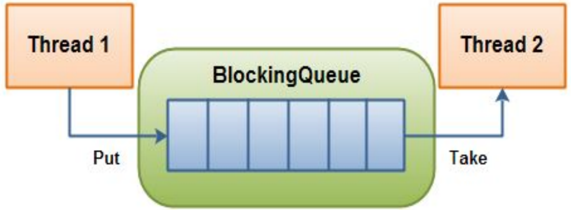


PriorityBlockingQueue

Java’s implementation for a thread-safe priority queue.

Implements an unbounded BlockingQueue and adds features of a PriorityQueue.





pri

Limitations of Java's PBQ implementation

```java
public E peek() {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        return q.peek();
    } finally {
        lock.unlock();
    }
}

public E poll() {
    final ReentrantLock lock = this.lock;
    lock.lock();
    try {
        return q.poll();
    } finally {
        lock.unlock();
    }
}
```

Mutual exclusion (locking)

- All methods wrapped with ReentrantLock locking/unlocking => one thread restricted to access the Queue at any given time

Alternative Implementations

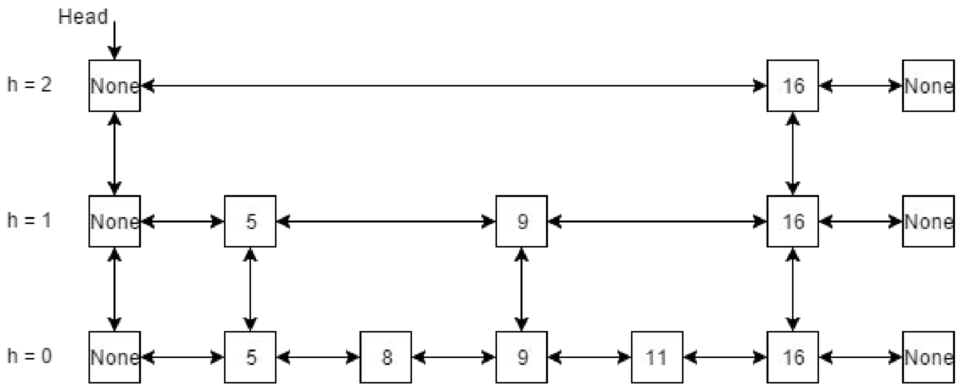
1. Lock-free: Randomised Skiplist implementation
2. Pipelined: - Heap based implementation

Lock free PQ

- Lock-free implementation (no mutual exclusion)
- Guarantee: Always at least one operation progresses
- Implementation based on a randomized skip list
- O(logN) for insertions and deletions

Skiplist

- The heights of the nodes are determined by probability.
- Insertion and Deletion
- Ordered list, first element is highest priority.
- For lock-free, synchronisation primitives are used:
    - CAS - compare and swap
    - FAA - fetch and add
- Pros:
    - Balancing is probabilistic
    - No need to pre-allocate all memory



Pipeline PQ

- insert() delete() replace() peek()
- Heap based (called P-heap)
- B = binary heap array
- T = token array

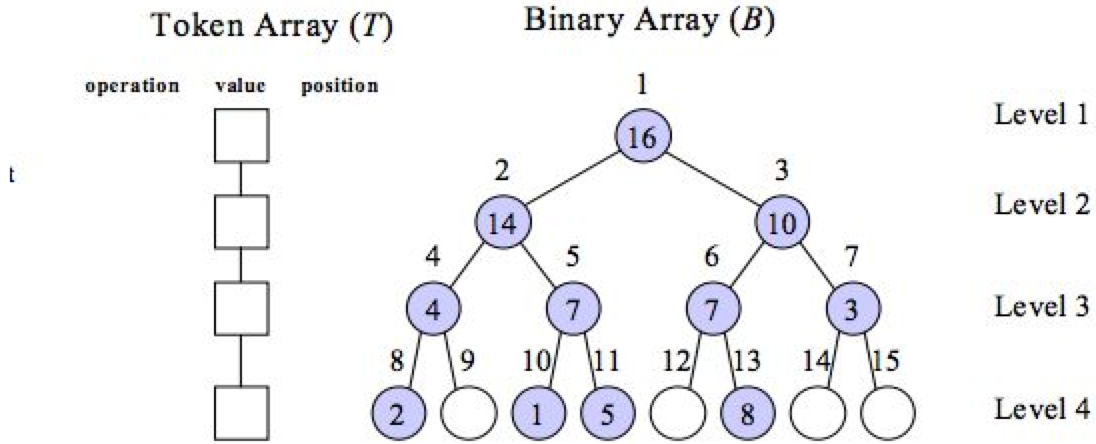
Binary Heap Array

- Similar to binary heap implemented using an array but with three fields:
  - `B[i].active` = boolean set to true if the node is active, false otherwise
  - `B[i].value` = if active, holds the value of the node
  - `B[i].capacity` = if active, contains number of inactive nodes rooted at node B[i]
- P-heap property: If x is a node in B, and y is its immediate child, then,
  - If they are both active, then x.value >= y.value
  - If y is active, then x must be active

Token array

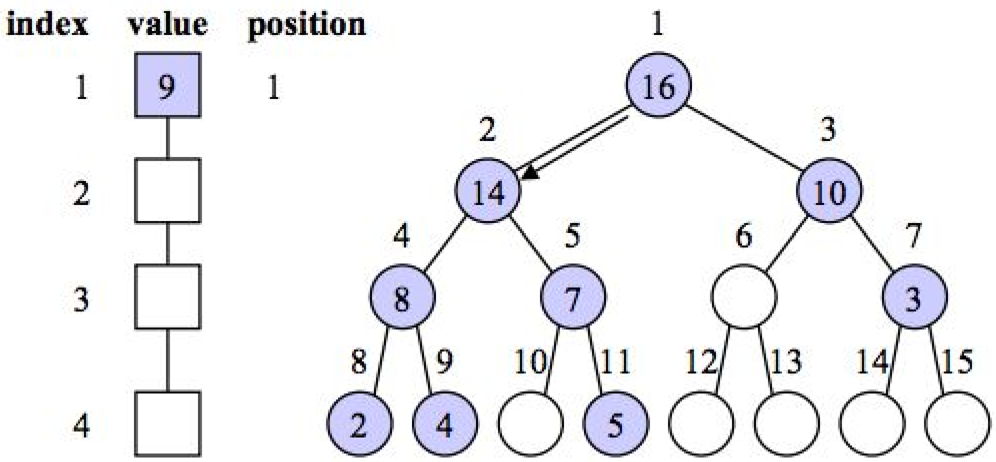
- An array with size equal to the number of levels in the binary tree
- Each element has three fields:
  - `T[i].operation` = either of insert(), remove(), peek(), replace()
  - `T[i].value` = may hold a value that needs to be inserted into B
  - `T[i].position` = may hold an index of a node at level L(i) of B

Binary Heap Array and Token Array

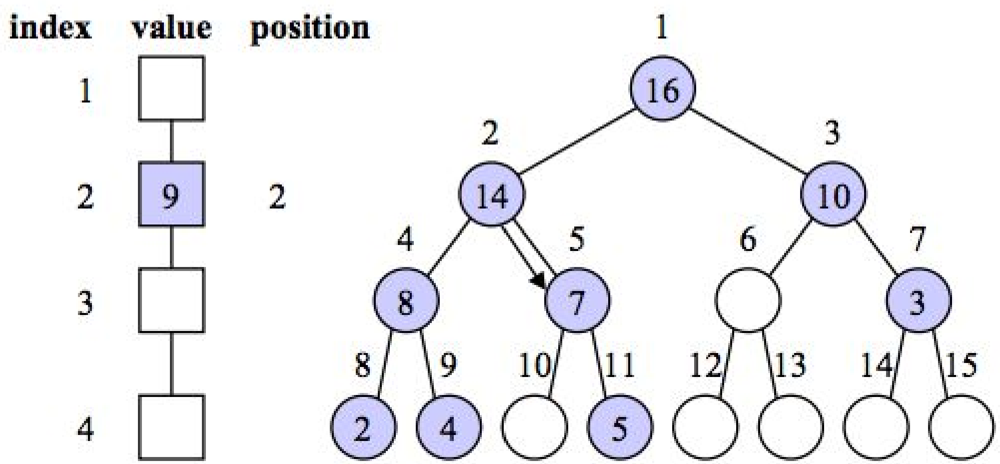


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
value	16	14	10	4	7	7	3	2		1	5		8		
capacity	4	1	3	1	0	1	2	0	1	0	0	1	0	1	1

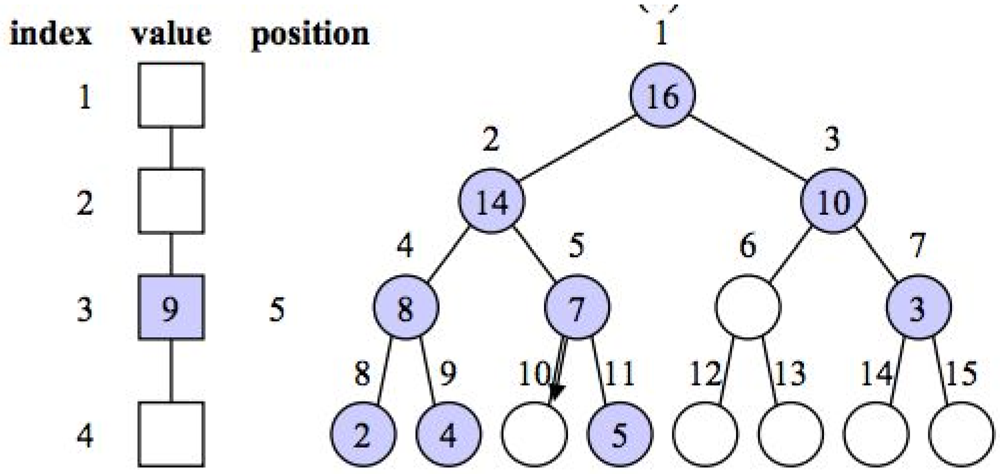
Pipelined PQ - Insertion



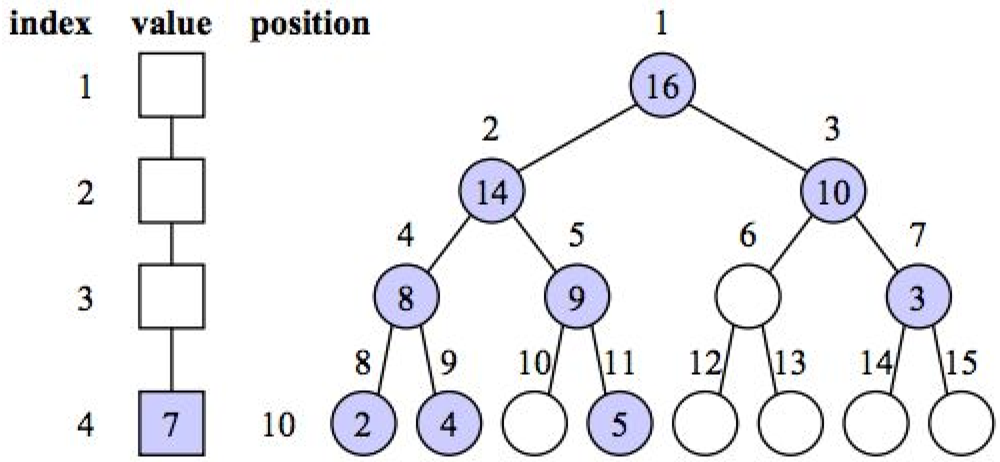
Pipelined PQ - Insertion II



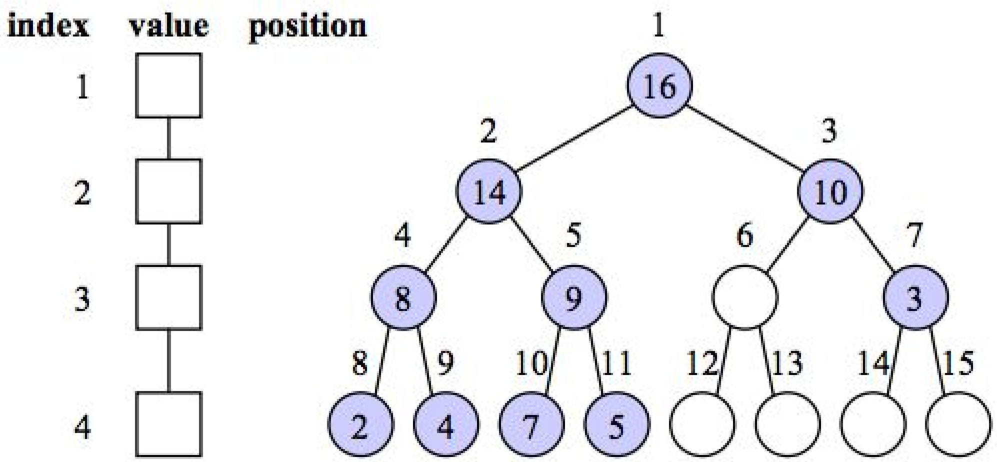
Pipelined PQ - Insertion III



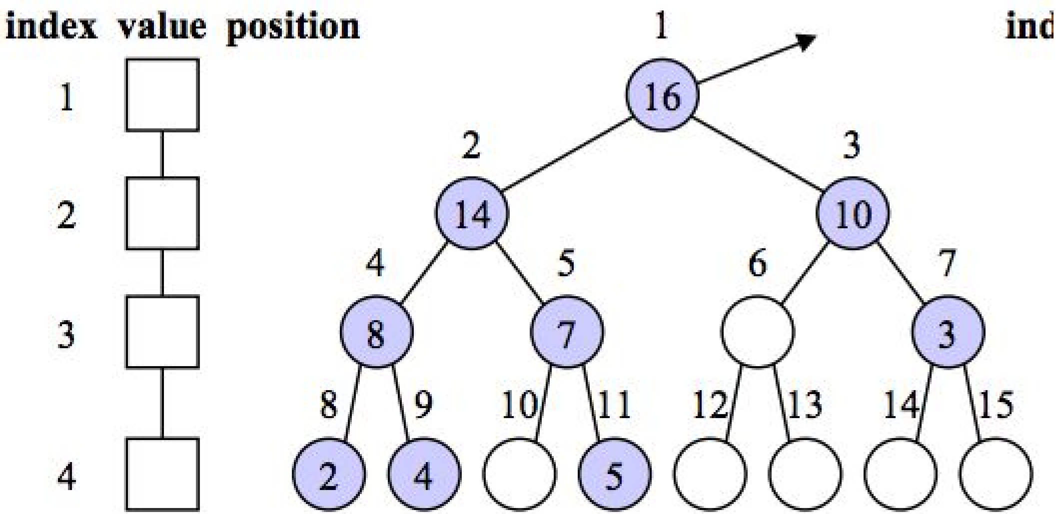
Pipelined PQ - Insertion IV



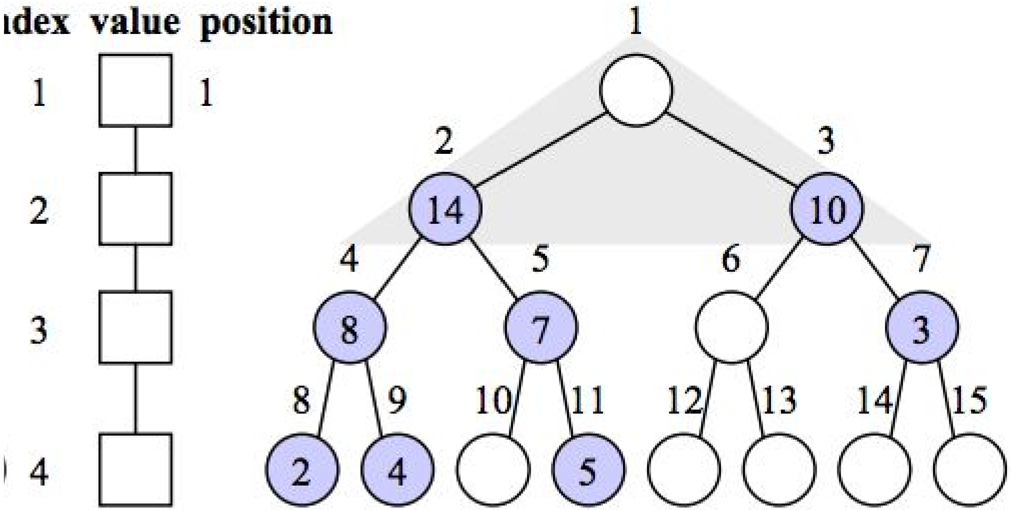
Pipelined PQ - Insertion V



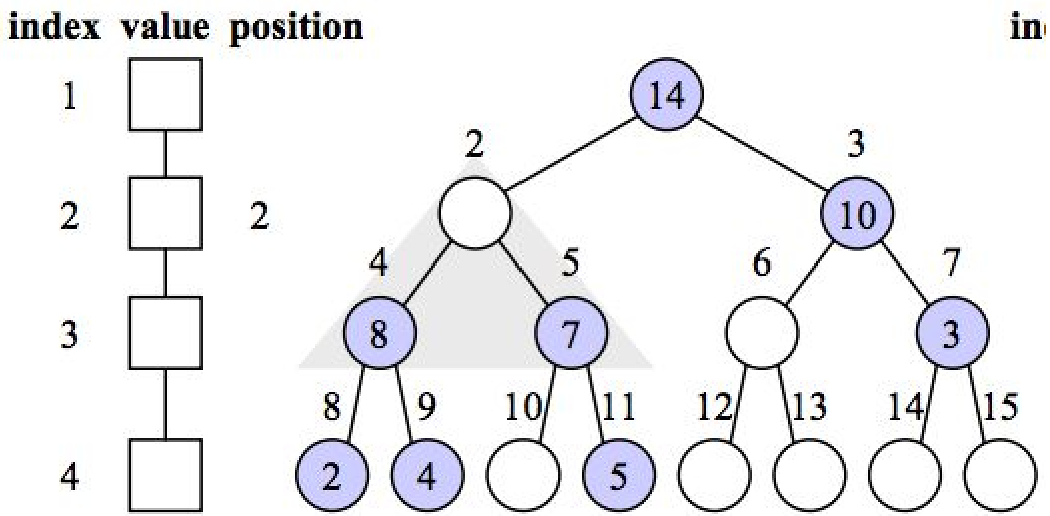
Pipelined PQ - Deletion



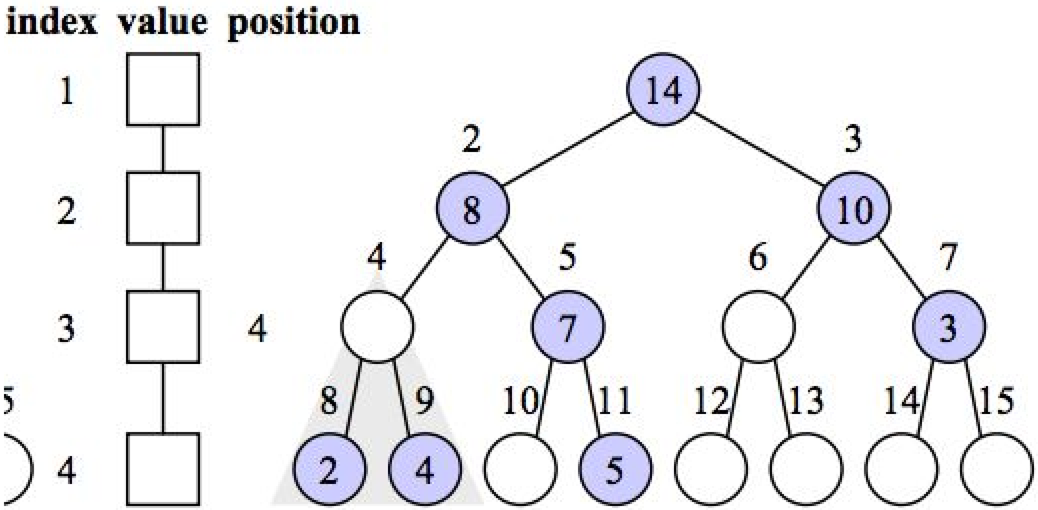
Pipelined PQ - Deletion II



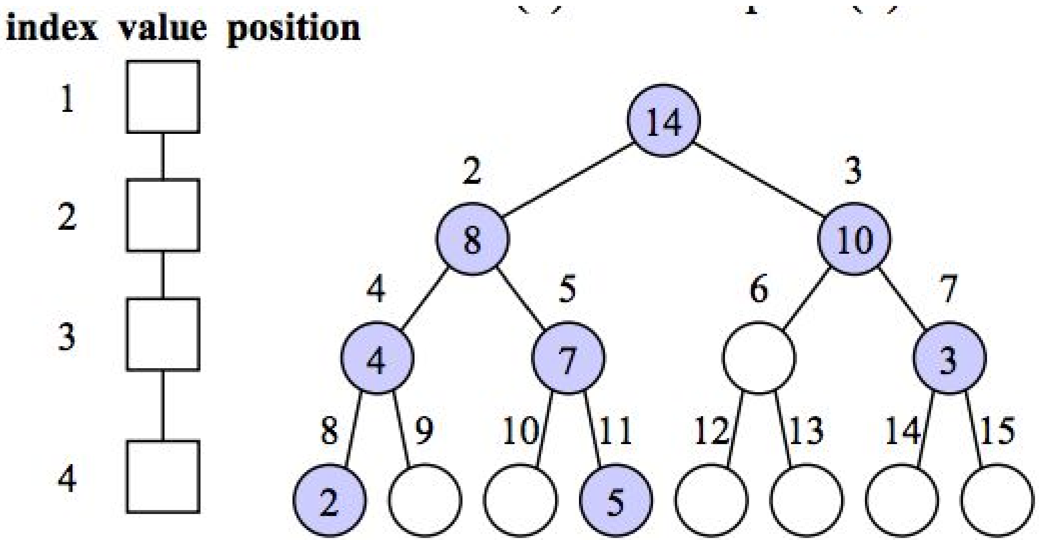
Pipelined PQ - Deletion III



Pipelined PQ - Deletion IV



Pipelined PQ - Deletion V



Parallelisation in Pipelined PQs

- Each operation belongs “within” two consecutive levels of the heap
- Odd numbered levels can be accessed simultaneously
- Even numbered levels can be accessed simultaneously
- In this way multiple operations can take place in the heap concurrently

Comparison of Lock free and Pipelined PQ

Lock-free:

- Probabilistic time complexity of $O(\log N)$ for insertion, deletion
- $\Theta(1)$, $O(N)$ for peek
- Relies on atomic primitives
- Risk: can cause starvation
- Keys/priorities have to be unique

Pipelined PQ:

$O(\log n)$ for insertion, deletion

$O(1)$ for peek

Each level can complete an operation (and pass it down as needed) and move to the next

Relies on locks

Goals for this project

To produce a faster thread-safe priority queue than Java’s PriorityBlockingQueue.

Plan to analyze pseudocode for both further. Implement at least one of the strategies discussed (both if time permits).

- Implement our version of a PBQ.
- Evaluate this implementation by comparing performance with PBQ.

Evaluation/Testing methods

Use Java's PriorityBlockingQueue implementation as a benchmark, using the three main PriorityBlockingQueue operations - `E peek(), E take(), put(E)`

Test our implementation for correctness on small inputs.

Tests for performance on large input, one thread per core, different cases:

- Multiple consumer, single producer
- Single consumer, multiple producer
- Multiple consumer, multiple producer (random inserts/deletes).

Evaluation/Testing methods

The priorities of inserted items will be chosen uniformly at random, modeling a realistic input.

Threads accessing the data structure randomly choose whether to insert a random priority item or apply a Delete-min operation

Experiment by varying the proportion of inserts/deletes (increasing or decreasing the size of the structure gradually).

References

- A. Ioannou and M. G. H. Katevenis, "Pipelined Heap (Priority Queue) Management for Advanced Scheduling in High-Speed Networks”
- Ranjita Bhagwan, Bill Lin, “Fast and Scalable Priority Queue Architecture for High-Speed Network Switches”
- https://en.wikipedia.org/wiki/Priority_queue
- http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/java/util/concurrent/PriorityBlockingQueue.java
- http://www.non-blocking.com/download/SunT03_PQueue_TR.pdf