Priority Queues

Contents

Abstract Priority Queue

Source: `AbstractPriorityQueue.java`

```
public abstract class AbstractPriorityQueue<T extends Comparable<T>> {
    protected Comparator<T> comparator;

    public AbstractPriorityQueue(Comparator<T> comparator) {
        this.comparator = comparator;
    }
```

- Class types: `class` vs. `abstract class` vs. `interface`.
- Generics: `<T extends Comparable<T>>`.
- Member types: `private` vs. `package` vs. `protected` vs. `public`.
- Constructor: `this`.

```java
/**
 * Adds a comparable key to this queue.
 * @param key the comparable key.
 */
abstract public void add(T key);

/**
 * Removes the key with the highest priority if exists.
 * @return the key with the highest priority if exists; otherwise, {@code null}.
 */
abstract protected T remove();

/** @return the size of this queue. */
abstract public int size();

/** @return {@code true} if the queue is empty; otherwise, {@code false}. */
public boolean isEmpty() {
    return size() == 0;
}
```

- Abstract methods: `add()`, `remove()`, `size()`.
- Regular method: `isEmpty()`.
- Javadoc.

Lazy Priority Queue

Source: `LazyPriorityQueue.java`

```
public class LazyPriorityQueue<T extends Comparable<T>> extends AbstractPriorityQueue<T> {
    private List<T> keys;

    public LazyPriorityQueue(Comparator<T> comparator) {
        super(comparator);
        keys = new ArrayList<>();
    }

    public LazyPriorityQueue() {
        this(Comparator.naturalOrder());
    }
```

- Inheritance: `extends AbstractPriorityQueue<T>`.
- Constructors: default vs. parameters, `this` vs. `super`.

```java
    /**
     * Adds a key to the back of the list.
     * @param key the comparable key.
     */
    @Override
    public void add(T key) { keys.add(key); }

    /**
     * Finds the key with the highest priority, and removes it from the list.
     * @return the key with the highest priority if exists; otherwise, {@code null}.
     */
    @Override
    protected T remove() {
        if (isEmpty()) return null;
        T max = Collections.max(keys, comparator);
        keys.remove(max);
        return max;
    }

    @Override
    public int size() { return keys.size(); }
```

- Annotation: `@Override`.
- Edge case handling: `remove()`.
- Standard API: `Collections.max()`.
- Complexity: `add()`, `remove()`.

Eager Priority Queue

Source: `EagerPriorityQueue.java`

```java
/**
 * Adds a key to the list according to the priority.
 * @param key the comparable key.
 */
@Override
public void add(T key) {
    int index = Collections.binarySearch(keys, key, comparator);
    if (index < 0) index = -(index + 1);
    keys.add(index, key);
}

/**
 * Remove the last key in the list.
 * @return the key with the highest priority if exists; otherwise, {@code null}.
 */
@Override
protected T remove() {
    return isEmpty() ? null : keys.remove(keys.size() - 1);
}
```

- Standard API: Collections.binarySearch().
- Ternary conditional operator: `condition ? :` .
- Complexity: `add()` , `remove()` .

Binary Heap

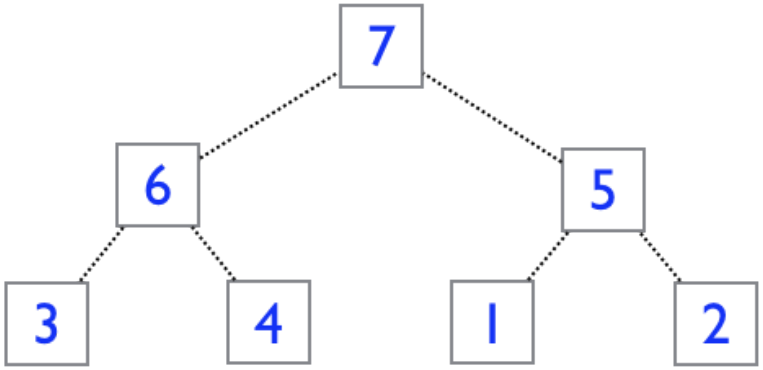
What is a heap?

- A tree where each node has a higher (or equal) priority than its children.
- The tree must be balanced at all time.
- What is a binary heap?

Operations

- Add: swim.
- Remove: sink.
- Both operations can be done in $O(\log n)$.

Input = 7 3 2 4 6 1 5



list = ∅ 7 6 5 3 4 1 2

- Binary heap can be represented by a list.
- Index of the parent: $k/2$.
- Index of the children: $k * 2$ and $(k * 2) + 1$.

Source: `BinaryHeap.java`

```java
public class BinaryHeap<T extends Comparable<T>> extends AbstractPriorityQueue<T> {
    private List<T> keys;

    public BinaryHeap(Comparator<T> comparator) {
        super(comparator);
        keys = new ArrayList<>();
        keys.add(null);    // initialize the first item as null
    }

    public BinaryHeap() {
        this(Comparator.naturalOrder());
    }

    @Override
    public int size() {
        return keys.size() - 1;
    }
```

- Handle the `null` key at the front.

```java
@Override
public void add(T key) {
    keys.add(key);
    swim(size());
}

private void swim(int k) {
    while (1 < k && comparator.compare(keys.get(k / 2), keys.get(k)) < 0) {
        Collections.swap(keys, k / 2, k);
        k /= 2;
    }
}
```

- Add each key to the end of the list and swim until it becomes a heap.
- `comparator.compare()` : compare itself to its parent.

```java
@Override
protected T remove() {
    if (isEmpty()) return null;
    Collections.swap(keys, 1, size());
    T max = keys.remove(size());
    sink(1);
    return max;
}

private void sink(int k) {
    for (int i = k * 2; i <= size(); k = i, i *= 2) {
        if (i < size() && comparator.compare(keys.get(i), keys.get(i + 1)) < 0) i++;
        if (comparator.compare(keys.get(k), keys.get(i)) >= 0) break;
        Collections.swap(keys, k, i);
    }
}
```

- Replace the root with the last key in the list and sink until it becomes a heap.
- Compare two children.
- Compare itself to the greater child.

Unit Tests

Source: `PriorityQueueTest.java`

Accuracy

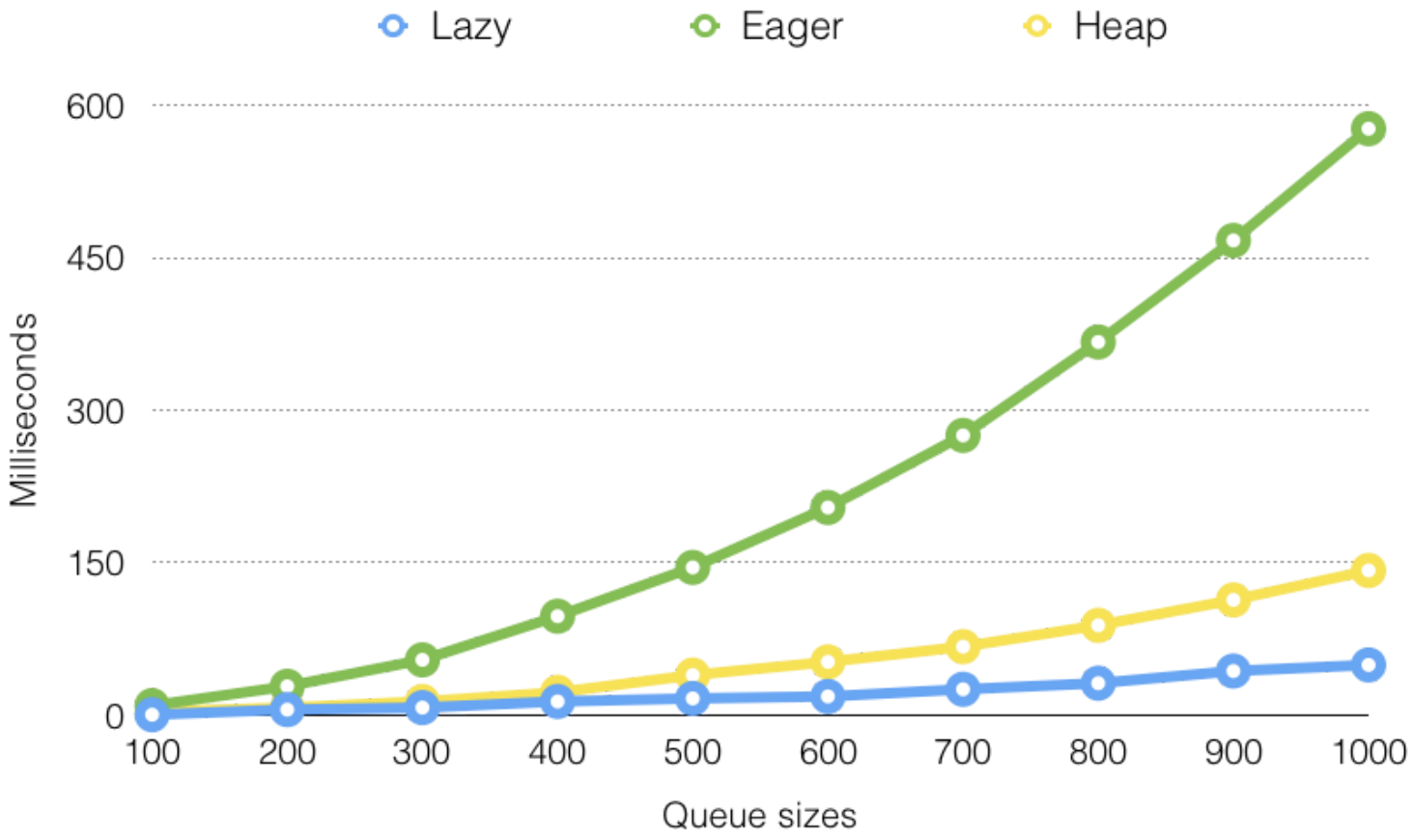
```java
@Test
public void testAccuracy() {
    testAccuracy(new LazyPriorityQueue<>(), Comparator.reverseOrder());
    testAccuracy(new EagerPriorityQueue<>(), Comparator.reverseOrder());
    testAccuracy(new BinaryHeap<>(), Comparator.reverseOrder());
}

private void testAccuracy(AbstractPriorityQueue<Integer> q, Comparator<Integer> sort) {
    List<Integer> keys = new ArrayList<>(Arrays.asList(4, 1, 3, 2, 5, 6, 8, 3, 4, 7, 5, 9, 7));
    keys.forEach(q::add);
    keys.sort(sort);
    keys.forEach(key -> assertEquals(key, q.remove()));
}
```

- Annotation: `@Test`.

Speed

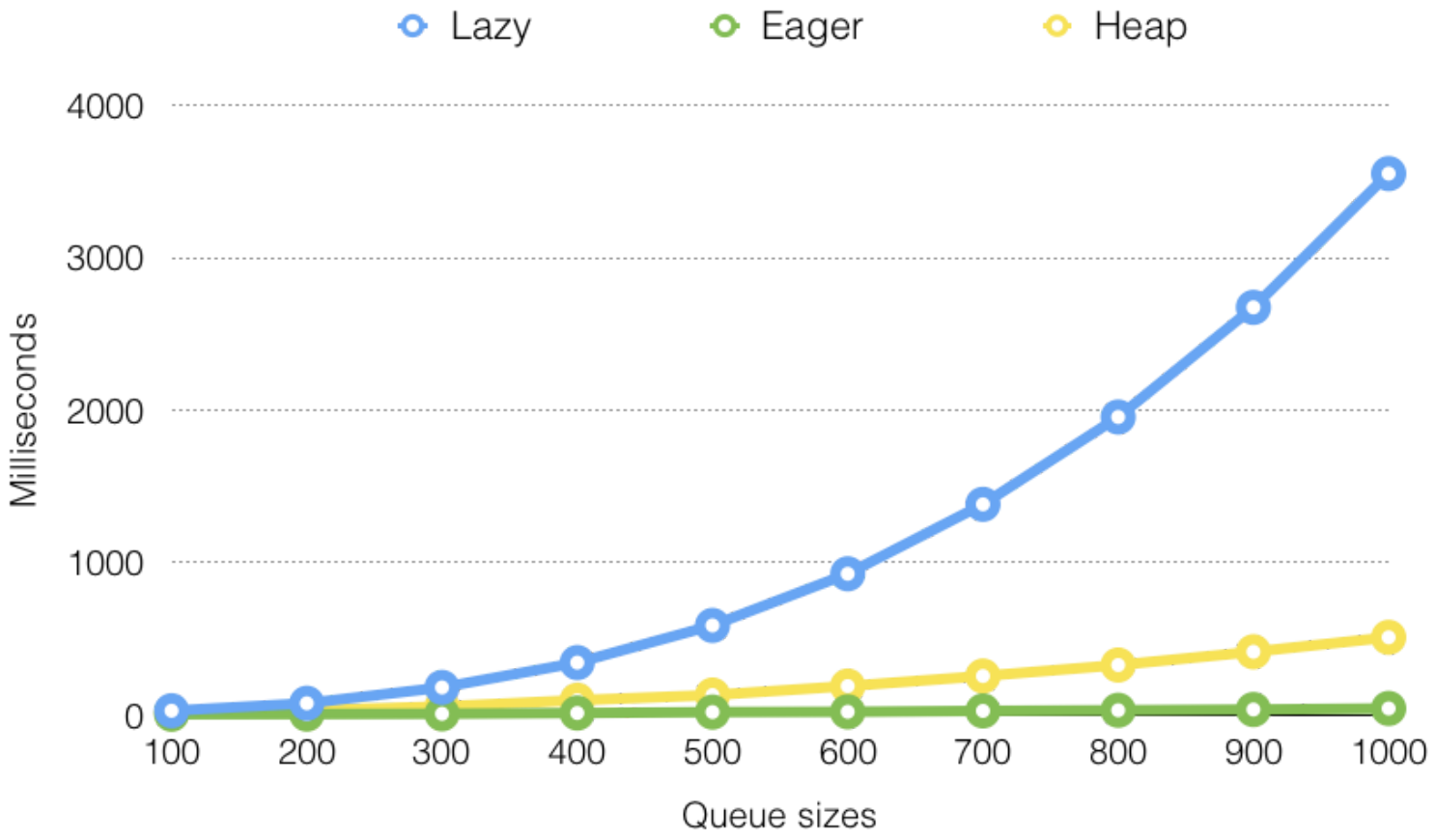
```
private void addRuntime(AbstractPriorityQueue<Integer> queue, long[] times, int[] keys) {
    long st, et;

    st = System.currentTimeMillis();

    for (int key : keys)
        queue.add(key);

    et = System.currentTimeMillis();
    times[0] += et - st;

    st = System.currentTimeMillis();

    while (!queue.isEmpty())
        queue.remove();

    et = System.currentTimeMillis();
    times[1] += et - st;
}
```

Speed Comparison - Add



- Lazy: $O(1)$ vs. Eager: $O(\log n)$? vs. Heap: $O(\log n)$.

Speed Comparison - Remove



- Lazy: $O(n)$, Eager: $O(1)$, Heap: $O(\log n)$.

References

- Priority queue.
- Binary heap.
- Generics in Java.