优先队列实现原理分析

ziwenxie.site/2017/03/30/algorithm-priority-queue/

引言

优先队列是在实际工程中被广泛应用的一种数据结构，不管是在操作系统的进程调度中，还是在相关的图算法比如Prim算法和Dijkstra算法中，我们都可以看到优先队列的身影，本文我们就来分析一下优先队列的实现原理。

优先队列

以操作系统的进程调度为例，比如我们在使用手机的过程中，手机分配给来电的优先级都会比其它程序高，在这个业务场景中，我们不要求所有元素全部有序，因为我们需要处理的只是当前键值最大的元素(优先级最高的进程)。在这种情况下，我们需要实现的只是删除最大的元素(获取优先级最高的进程)和插入新的元素(插入新的进程)，这种数据结构就叫做优先队列。

我们先来定义一个优先队列，下面我们将使用`pq[]`来保存相关的元素，在构造函数中可以指定堆的初始化大小，如果不指定初始化大小值，默认初始化值为1。p.s: 在下面我们会实现相关的`resize()`方法用来动态调整数组的大小。

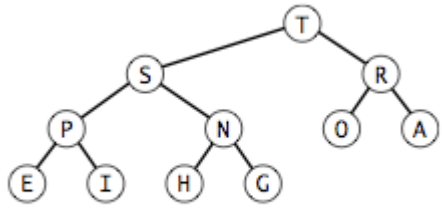
```
public class MaxPQ<Key> implements Iterable<Key> {
    private Key[] pq;
    private int n;
    private Comparator<Key> comparator;

     * Initializes an empty priority queue with the given initial capacity.
     *
     * @param  initCapacity the initial capacity of this priority queue
     */
    public MaxPQ(int initCapacity) {
        pq = (Key[]) new Object[initCapacity + 1];
        n = 0;
    }

     * Initializes an empty priority queue.
     */
    public MaxPQ() {
        this(1);
    }
}
```

堆的基本概念

在正式进入优先队列分析之前，我们有必要先了解一下对于堆的相关操作。我们定义当一棵二叉树的每个结点都要大于等于它的两个子结点的时候，称这棵二叉树堆有序。如下图就是一棵典型的堆有序的完全二叉树。

堆有序的完全二叉树

堆上浮和下沉操作



A heap-ordered complete binary tree

对了保证堆有序，对于堆我们要对它进行上浮和下沉操作，我们先来实现两个常用的工具方法，其中`less()`用于比较两个元素的大小，`exch()`用于交换数组的两个元素：

```
private boolean less(int i, int j) {
    if (comparator == null) {
        return ((Comparable<Key>)
pq[i]).compareTo(pq[j]) < 0;
    }
    else {
        return comparator.compare(pq[i], pq[j]) <
0;
    }
}
private void exch(int i, int j) {
    Key swap = pq[i];
    pq[i] = pq[j];
    pq[j] = swap;
}
```

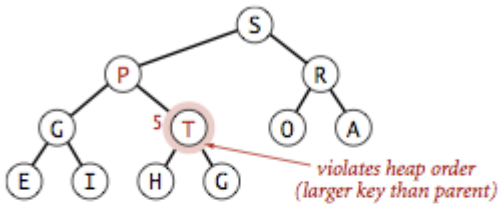
上浮操作

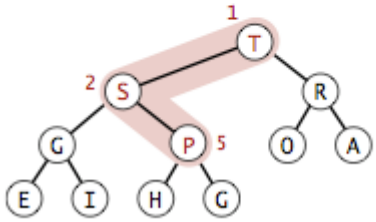
根据下图我们首先来分析一下上浮操作，以`swim(5)`为例子，我们来看一下上浮的过程。对于堆我们进行上浮的目的是保持堆有序性，即一个结点的值大于它的子结点的值，所以我们将`a[5]`和它的父结点`a[2]`相比较，如果它大于父结点的值，我们就交换两者，然后继续`swim(2)`。

上浮操作

具体的实现代码如下：

```
private void swim(int k) {
    while (k > 1 && less(k/2, k)) {
        exch(k, k/2);
        k = k/2;
    }
}
```





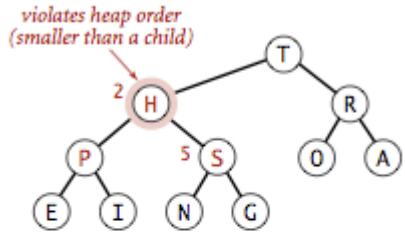
下沉操作

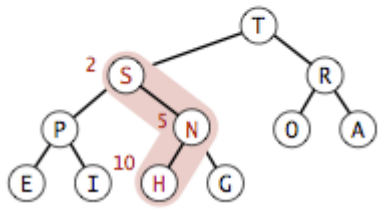
根据下图我们来分析一下下沉操作，以sink(2)为例子，我们先将结点a[2]和它两个子结点中较小的结点相比较，如果小于子结点，我们就交换两者，然后继续sink(5)。

下沉操作

具体的实现代码如下：

```
private void sink(int k) {
    while (2*k <= n) {
        int j = 2*k;
        if (j < n && less(j, j+1)) j++;
        if (!less(k, j)) break;
        exch(k, j);
        k = j;
    }
}
```

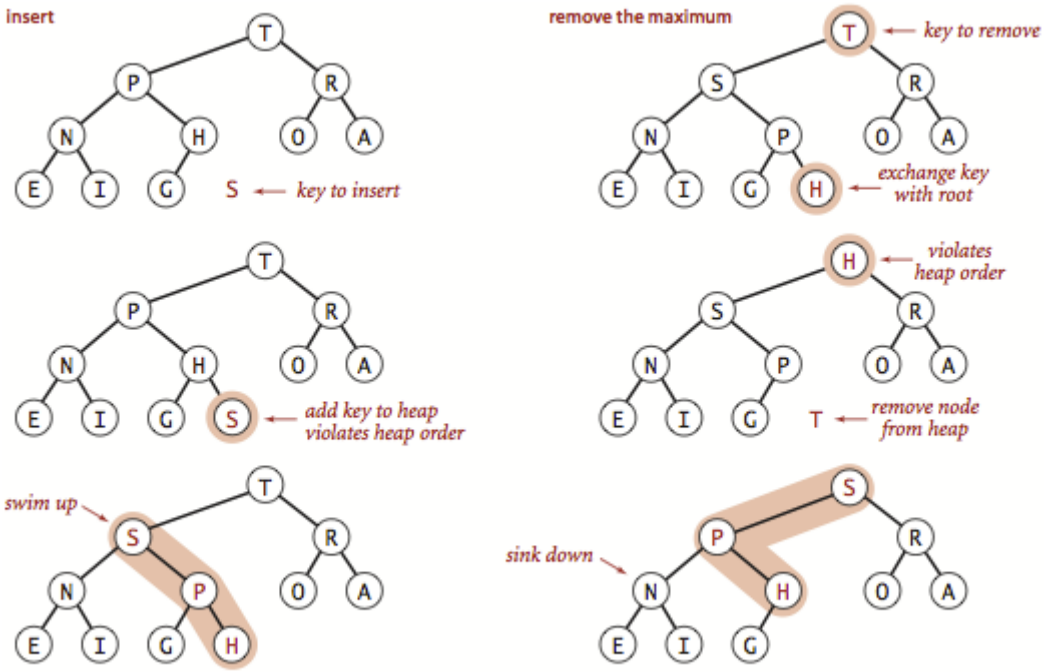




实现

我们来分析一下插入一个元素的过程，如果我们要在堆中新插入一个元素s的话，首先我们默认将这个元素插入到数组中pq[++n] 中(数组是从1开始计数的)。当我们插入s后，打破了堆的有序性，所以我们采用上浮操作来维持堆的有序性，当上浮操作结束之后，我们依然可以保证根结点的元素是数组中最大的元素。

接下来我们来看一下删除最大元素的过程，我们首先将最大的元素a[1]和a[n]交换，然后我们删除最大元素a[n]，这个时候堆的有序性已经被打破了，所以我们继续通过下沉操作来重新维持堆的有序性，保持根结点元素是所有元素中最大的元素。



Heap operations

插入元素和删除最大元素

插入的实现代码如下：

```
 * Adds a new key to this priority queue.
 *
 * @param  x the new key to add to this priority queue
 */
public void insert(Key x) {

   if (n >= pq.length - 1) resize(2 * pq.length);

   pq[++n] = x;
   swim(n);
   assert isMaxHeap();
}
```

删除的实现代码如下：

```
     * Removes a maximum key and returns its associated index.
     *
     * @return an index associated with a maximum key
     * @throws NoSuchElementException if this priority queue is empty
     */
    public Key delMax() {
        if (isEmpty()) throw new NoSuchElementException("Priority queue
    underflow");
        Key max = pq[1];
        exch(1, n);
        n--;
        sink(1);
        pq[n+1] = null;
        if ((n > 0) && (n == (pq.length - 1) / 4)) resize(pq.length / 2);
        assert isMaxHeap();
        return max;
    }
```

上面我们在`insert()`过程中用到了`resize()`函数，它用于动态数组的大小，具体的实现代码如下：

```
    private void resize(int capacity) {
        assert capacity > n;
        Key[] temp = (Key[]) new Object[capacity];
        for (int i = 1; i <= n; i++) {
            temp[i] = pq[i];
        }
        pq = temp;
    }
    public boolean isEmpty() {
        return n == 0;
    }
```

而`isMaxHeap()`则用于判断当前数组是否满足堆有序原则，这在debug的时候非常的有用，具体的实现代码如下：

```
private boolean isMaxHeap() {
    return isMaxHeap(1);
}
private boolean isMaxHeap(int k) {
    if (k > n) return true;
    int left = 2*k;
    int right = 2*k + 1;
    if (left  <= n && less(k, left))  return false;
    if (right <= n && less(k, right)) return false;
    return isMaxHeap(left) && isMaxHeap(right);
}
```

到此我们的优先队列已经差不多完成了，注意我们上面实现了`Iterable<Key>`接口，所以我们来实现`iterator()`方法：

```
     * Returns an iterator that iterates over the keys on this priority queue
     * in descending order.
     * The iterator doesn't implement remove() since it's optional.
     *
     * @return an iterator that iterates over the keys in descending order
     */
    public Iterator<Key> iterator() {
        return new HeapIterator();
    }
    private class HeapIterator implements Iterator<Key> {

        private MaxPQ<Key> copy;


        public HeapIterator() {
            if (comparator == null) copy = new MaxPQ<Key>(size());
            else                    copy = new MaxPQ<Key>(size(), comparator);
            for (int i = 1; i <= n; i++)
                copy.insert(pq[i]);
        }
        public boolean hasNext()  { return !copy.isEmpty();                     }
        public void remove()      { throw new UnsupportedOperationException();  }
        public Key next() {
            if (!hasNext()) throw new NoSuchElementException();
            return copy.delMax();
        }
    }
```

堆排序

将上面的优先队列稍微做一下改进，我们便可以实现堆排序，即对`pq[]`中的元素进行排序。对于堆排序的具体实现，下面我们分为两个步骤：

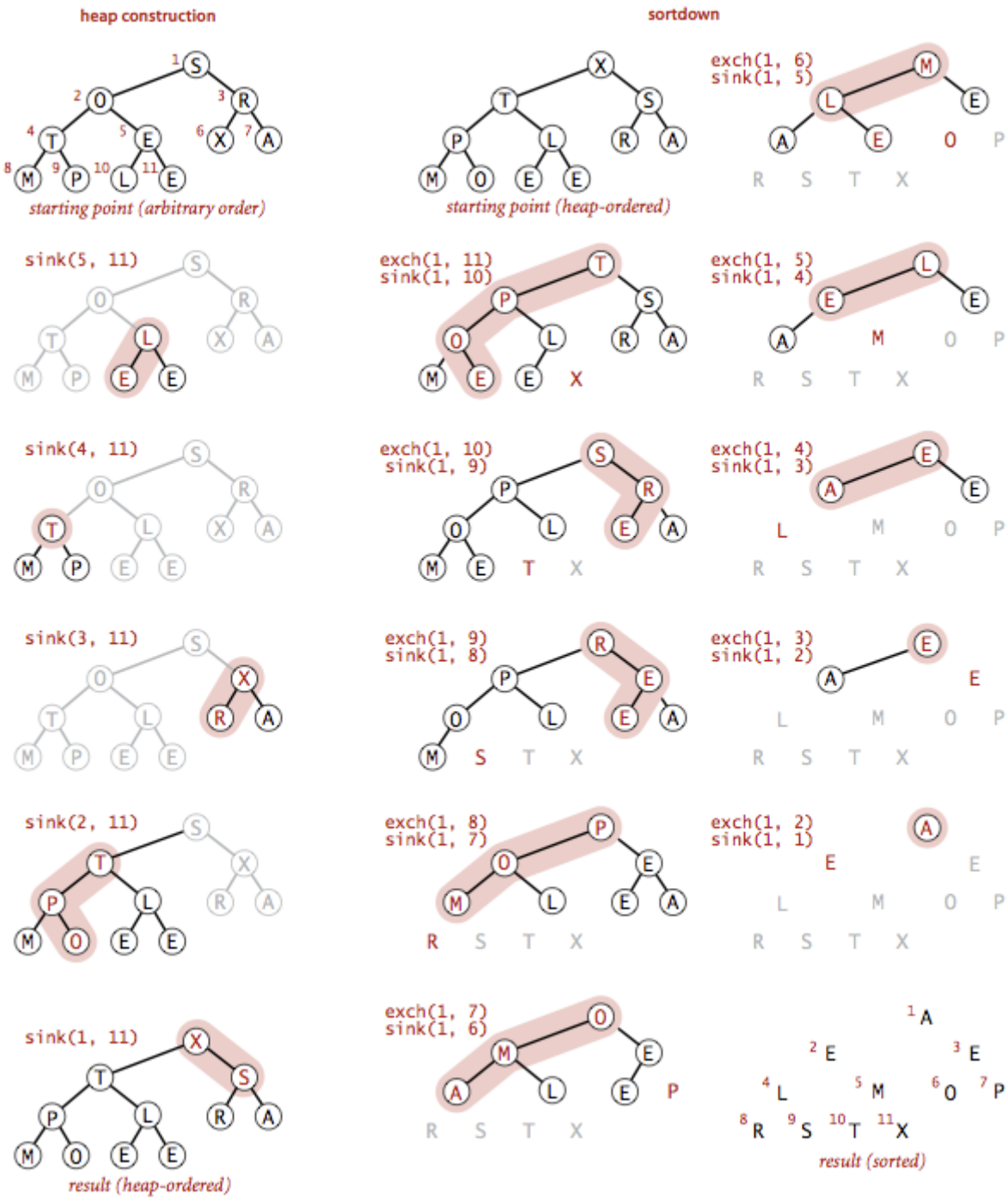
1. 首先我们先来构造一个堆。
2. 然后通过下沉的方式进行排序。

堆排序的实现代码非常的简短，我们首先来看一下具体的代码实现，然后我们再具体分析它的实现原理：

```
     * Rearranges the array in ascending order, using the natural order.
     * @param pq the array to be sorted
     */
    public static void sort(Comparable[] pq) {
        int n = pq.length;
        for (int k = n/2; k >= 1; k--)
            sink(pq, k, n);
        while (n > 1) {
            exch(pq, 1, n--);
            sink(pq, 1, n);
        }
    }
```

首先我们来看一下堆的构造过程(下图中的左图)。我们采用的方法是从右至左用`sink()`方法构造子堆。我们只需要扫描数组中的一半元素，即5, 4, 3, 2, 1。这样通过这几个步骤，我们可以得到一个堆有序的数组，即每个结点的大小都大于它的两个结点，并使最大元素位于数组的开头。

接下来我们来分析一下下沉排序的实现(下图中的右图)，这里我们采取的方法是每次都将最大的元素删除，然后重新通过`sink()`来维持堆有序，这样每一次`sink()`操作我们都可以的到数组中最大的元素。



Heapsort: constructing (left) and sorting down (right) a heap

堆排序过程

Referencs

ALGORITHM-4TH