

Data Structures and Algorithms in Java

Sorting: Priority Queues

Outline

① Priority Queue

② Elementary Implementation

③ Heap-based Implementation

④ Heap Sort

Priority Queue

Priority Queue

A priority queue (PQ for short) is a data structure that allows us to process keys in order, without storing them in full sorted order all at once

Priority Queue

A priority queue (PQ for short) is a data structure that allows us to process keys in order, without storing them in full sorted order all at once

Fundamental operations: insert and remove the minimum (or maximum)

Priority Queue

Priority Queue

```
dsa.MinPQ<Key> implements java.lang.Iterable<Key>
```

MinPQ()	constructs an empty minPQ
MinPQ(Comparator<Key> c)	constructs an empty minPQ with the given comparator
MinPQ(int capacity)	constructs an empty minPQ with the given capacity
MinPQ(int capacity, Comparator<Key> c)	constructs an empty minPQ with the given capacity and comparator
boolean isEmpty()	returns <code>true</code> if this minPQ is empty, and <code>false</code> otherwise
int size()	returns the number of keys in this minPQ
void insert(Key key)	adds <code>key</code> to this minPQ
Key min()	returns the smallest key in this minPQ
Key delMin()	removes and returns the smallest key in this minPQ
Iterator<Key> iterator()	returns an iterator to iterate over the keys in this minPQ in ascending order
String toString()	returns a string representation of this minPQ

Priority Queue

Priority Queue

```
dsa.MaxPQ<Key> implements java.lang.Iterable<Key>
```

MaxPQ()	constructs an empty maxPQ
MaxPQ(Comparator<Key> c)	constructs an empty maxPQ with the given comparator
MaxPQ(int capacity)	constructs an empty maxPQ with the given capacity
MaxPQ(int capacity, Comparator<Key> c)	constructs an empty maxPQ with the given capacity and comparator
boolean isEmpty()	returns <code>true</code> if this maxPQ is empty, and <code>false</code> otherwise
int size()	returns the number of keys in this maxPQ
void insert(Key key)	adds <code>key</code> to this maxPQ
Key max()	returns the largest key in this maxPQ
Key delMax()	removes and returns the largest key in this maxPQ
Iterator<Key> iterator()	returns an iterator to iterate over the keys in this maxPQ in descending order
String toString()	returns a string representation of this maxPQ

Priority Queue

Priority Queue

Program: TopM.java

Priority Queue

Program: `TopM.java`

- Command-line input: m (int)

Priority Queue

Program: `TopM.java`

- Command-line input: m (int)
- Standard input: sequence of transactions

Priority Queue

Program: `TopM.java`

- Command-line input: m (int)
- Standard input: sequence of transactions
- Standard output: top m transactions in decreasing order of amount

Priority Queue

Program: TopM.java

- Command-line input: m (int)
- Standard input: sequence of transactions
- Standard output: top m transactions in decreasing order of amount

```
>_ ~/workspace/dsaj/programs
```

```
$ -
```

Priority Queue

Program: TopM.java

- Command-line input: m (int)
- Standard input: sequence of transactions
- Standard output: top m transactions in decreasing order of amount

```
>_ ~/workspace/dsaj/programs  
$ cat ../data/tinyBatch.txt
```

Priority Queue

Program: TopM.java

- Command-line input: m (int)
- Standard input: sequence of transactions
- Standard output: top m transactions in decreasing order of amount

```
>_ ~/workspace/dsaj/programs  
  
$ cat ../data/tinyBatch.txt  
Turing      6/17/1990    644.08  
vonNeumann  3/26/2002    4121.85  
Dijkstra     8/22/2007    2678.40  
vonNeumann  1/11/1999    4409.74  
Dijkstra     11/18/1995    837.42  
Hoare        5/10/1993    3229.27  
vonNeumann  2/12/1994    4732.35  
Hoare        8/18/1992    4381.21  
Turing        1/11/2002    66.10  
Thompson     2/27/2000    4747.08  
Turing        2/11/1991    2156.86  
Hoare        8/12/2003    1025.70  
vonNeumann  10/13/1993   2520.97  
Dijkstra     9/10/2000    708.95  
Turing        10/12/1993   3532.36  
Hoare        2/10/2005    4050.20  
$ _
```

Priority Queue

Program: TopM.java

- Command-line input: m (int)
- Standard input: sequence of transactions
- Standard output: top m transactions in decreasing order of amount

```
>_ ~/workspace/dsaj/programs

$ cat ../data/tinyBatch.txt
Turing      6/17/1990   644.08
vonNeumann 3/26/2002   4121.85
Dijkstra    8/22/2007   2678.40
vonNeumann 1/11/1999   4409.74
Dijkstra    11/18/1995   837.42
Hoare       5/10/1993   3229.27
vonNeumann 2/12/1994   4732.35
Hoare       8/18/1992   4381.21
Turing      1/11/2002   66.10
Thompson   2/27/2000   4747.08
Turing      2/11/1991   2156.86
Hoare       8/12/2003   1025.70
vonNeumann 10/13/1993  2520.97
Dijkstra    9/10/2000   708.95
Turing      10/12/1993  3532.36
Hoare       2/10/2005   4050.20
$ java TopM 5 < ../data/tinyBatch.txt
```

Priority Queue

Program: TopM.java

- Command-line input: m (int)
- Standard input: sequence of transactions
- Standard output: top m transactions in decreasing order of amount

```
>_ ~/workspace/dsaj/programs

$ cat ../data/tinyBatch.txt
Turing      6/17/1990   644.08
vonNeumann 3/26/2002   4121.85
Dijkstra    8/22/2007   2678.40
vonNeumann 1/11/1999   4409.74
Dijkstra    11/18/1995   837.42
Hoare       5/10/1993   3229.27
vonNeumann 2/12/1994   4732.35
Hoare       8/18/1992   4381.21
Turing      1/11/2002   66.10
Thompson   2/27/2000   4747.08
Turing      2/11/1991   2156.86
Hoare       8/12/2003   1025.70
vonNeumann 10/13/1993  2520.97
Dijkstra    9/10/2000   708.95
Turing      10/12/1993  3532.36
Hoare       2/10/2005   4050.20
$ java TopM 5 < ../data/tinyBatch.txt
Thompson   2/27/2000   4747.08
vonNeumann 2/12/1994   4732.35
vonNeumann 1/11/1999   4409.74
Hoare       8/18/1992   4381.21
vonNeumann 3/26/2002   4121.85
$ _
```

Priority Queue

Priority Queue

</> TopM.java

```
import dsa.LinkedStack;
import dsa.MinPQ;
import dsa.Transaction;
import stdlib.StdIn;
import stdlib.StdOut;

public class TopM {
    public static void main(String[] args) {
        int m = Integer.parseInt(args[0]);
        MinPQ<Transaction> pq = new MinPQ<Transaction>(m + 1);
        while (StdIn.hasNextLine()) {
            String line = StdIn.readLine();
            Transaction transaction = new Transaction(line);
            pq.insert(transaction);
            if (pq.size() > m) {
                pq.delMin();
            }
        }
        LinkedStack<Transaction> stack = new LinkedStack<Transaction>();
        for (Transaction transaction : pq) {
            stack.push(transaction);
        }
        for (Transaction transaction : stack) {
            StdOut.println(transaction);
        }
    }
}
```

Elementary Implementation

Elementary Implementation

Data Structure	insert()	delMin() / delMax()
Ordered array	n	1
Unordered array	1	n
Ordered linked list	n	1
Unordered linked list	1	n

Heap-based Implementation

Heap-based Implementation

A binary tree is a tree data structure in which each node has at most two children

Heap-based Implementation

A binary tree is a tree data structure in which each node has at most two children

A max-heap is a binary tree in which the key in each node is larger than or equal to the keys in that node's children

Heap-based Implementation

A binary tree is a tree data structure in which each node has at most two children

A max-heap is a binary tree in which the key in each node is larger than or equal to the keys in that node's children

The largest key in a max-heap is found at the root

Heap-based Implementation

A binary tree is a tree data structure in which each node has at most two children

A max-heap is a binary tree in which the key in each node is larger than or equal to the keys in that node's children

The largest key in a max-heap is found at the root

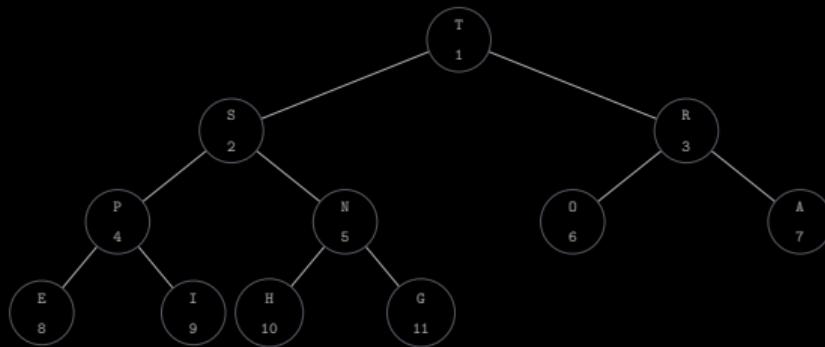
The height of a complete binary tree of size n is $\lfloor \lg n \rfloor$

Heap-based Implementation

Heap-based Implementation

Representation

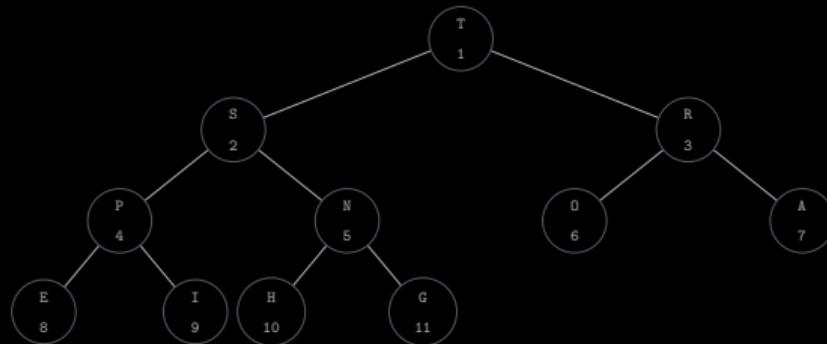
a[]											
0	1	2	3	4	5	6	7	8	9	10	11
-	T	S	R	P	N	O	A	E	I	H	G



Heap-based Implementation

Representation

a[]											
0	1	2	3	4	5	6	7	8	9	10	11
-	T	S	R	P	N	O	A	E	I	H	G

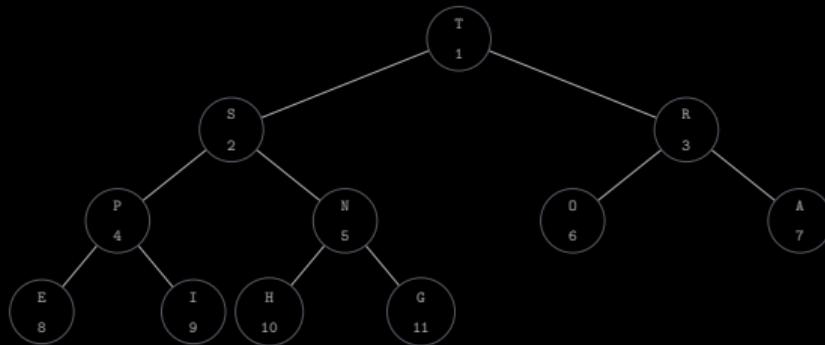


The left and right children of node i are nodes $2i$ and $2i + 1$ respectively

Heap-based Implementation

Representation

a[]											
0	1	2	3	4	5	6	7	8	9	10	11
-	T	S	R	P	N	O	A	E	I	H	G



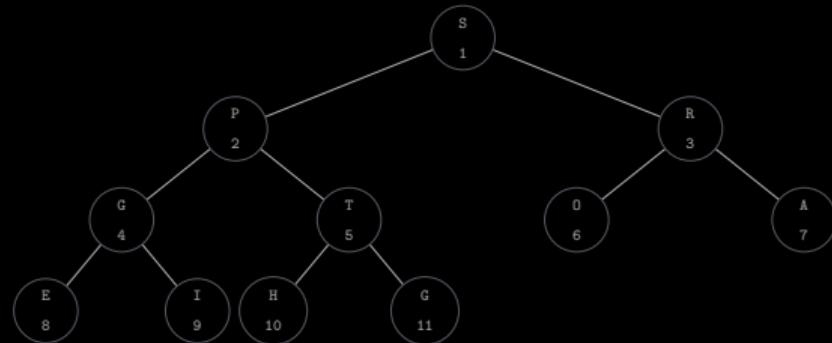
The left and right children of node i are nodes $2i$ and $2i + 1$ respectively

The parent of node i is $\left\lfloor \frac{i}{2} \right\rfloor$

Heap-based Implementation

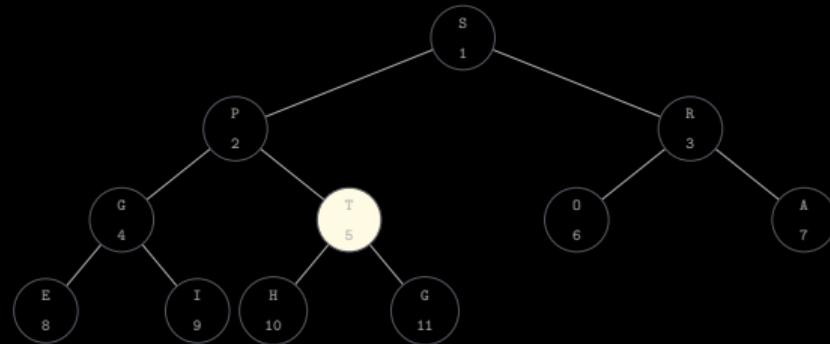
Heap-based Implementation

Bottom-up reheapify (swim)



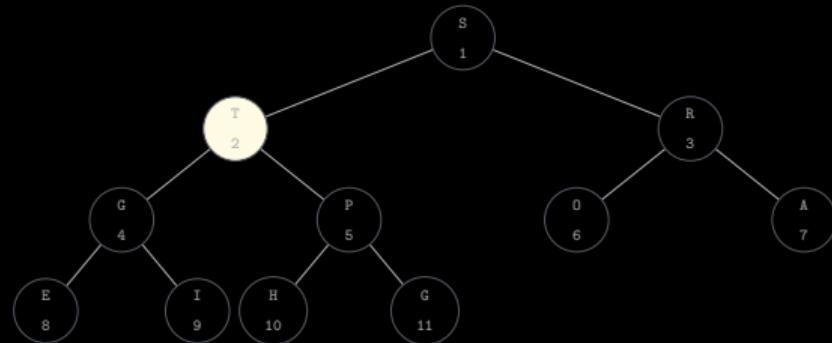
Heap-based Implementation

Bottom-up reheapify (swim)



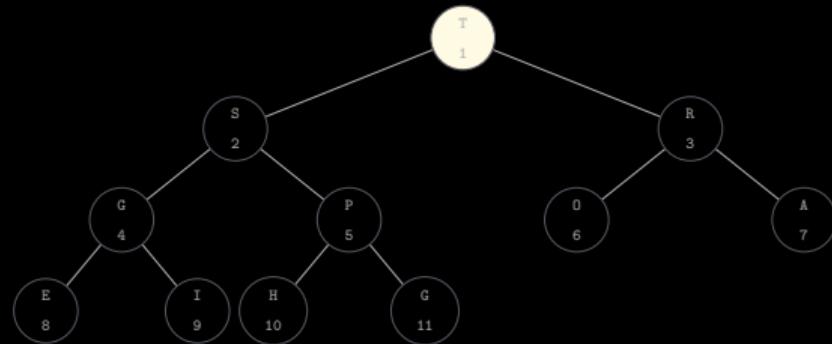
Heap-based Implementation

Bottom-up reheapify (swim)



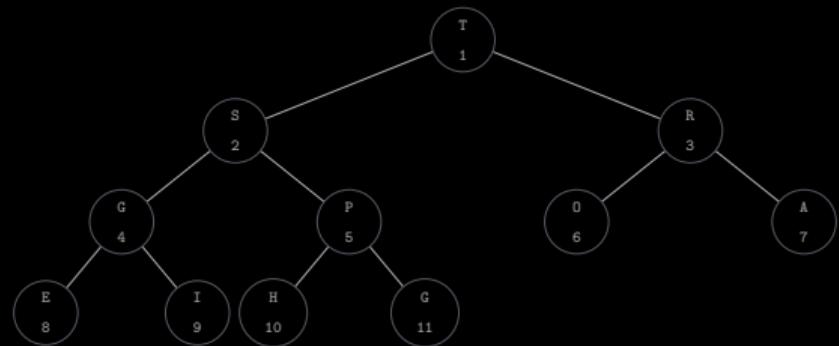
Heap-based Implementation

Bottom-up reheapify (swim)



Heap-based Implementation

Bottom-up reheapify (swim)



Heap-based Implementation

Heap-based Implementation

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

Heap-based Implementation

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

Insert: add the new key at the end of the array, increment the size of the heap, and swim up to restore the heap order

Heap-based Implementation

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

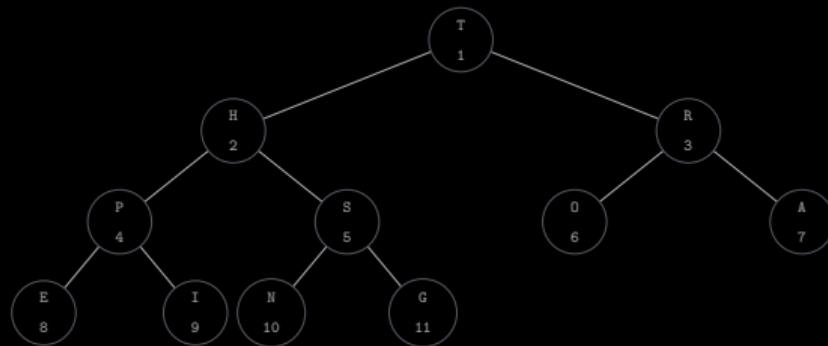
Insert: add the new key at the end of the array, increment the size of the heap, and swim up to restore the heap order

$$T(n) = \log n$$

Heap-based Implementation

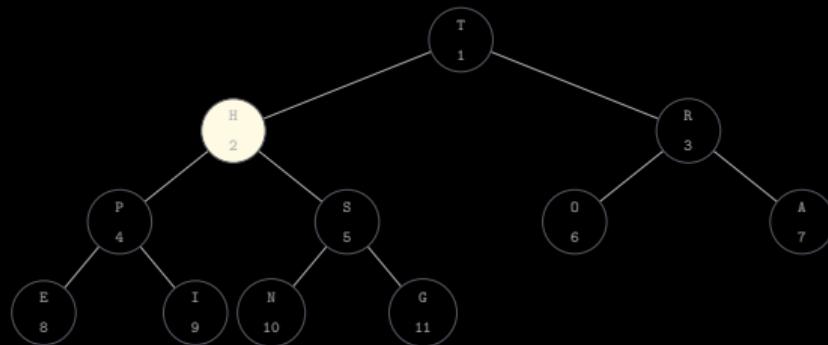
Heap-based Implementation

Top-down reheapify (sink)



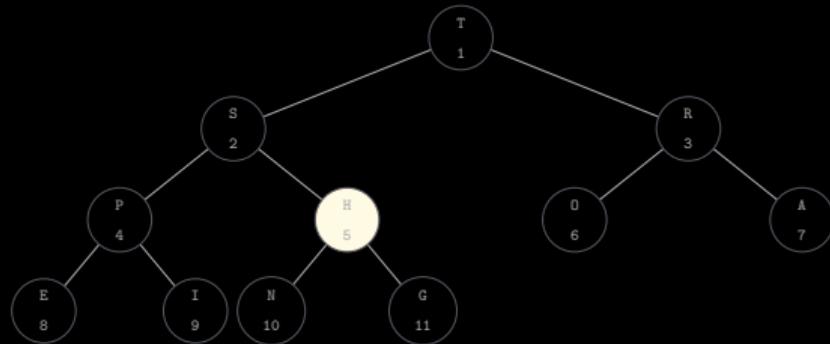
Heap-based Implementation

Top-down reheapify (sink)



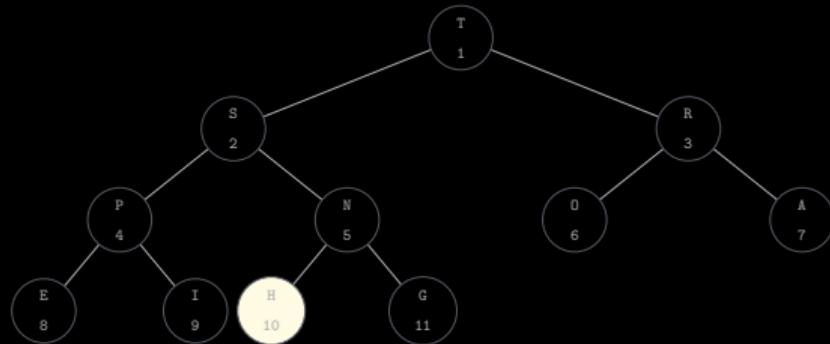
Heap-based Implementation

Top-down reheapify (sink)



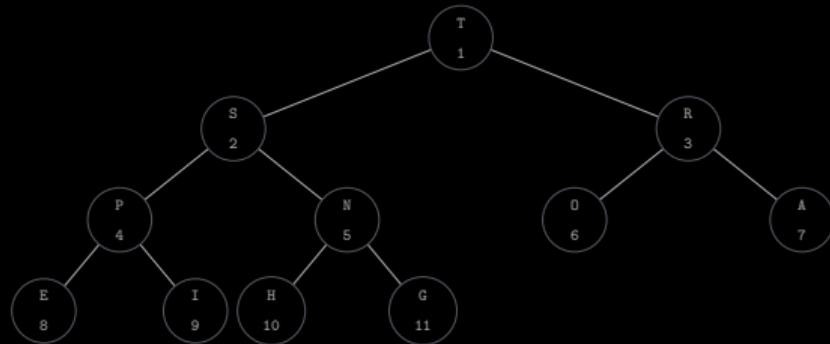
Heap-based Implementation

Top-down reheapify (sink)



Heap-based Implementation

Top-down reheapify (sink)



Heap-based Implementation

Heap-based Implementation

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

Heap-based Implementation

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

Remove the maximum: take the largest key off the top, put the key from the end of the heap at the top, decrement the size of the heap, and sink down to restore the heap order

Heap-based Implementation

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

Remove the maximum: take the largest key off the top, put the key from the end of the heap at the top, decrement the size of the heap, and sink down to restore the heap order

$$T(n) = \log n$$

Heap-based Implementation

Heap-based Implementation

```
</> MaxPQ.java

package dsa;

import java.util.Comparator;
import java.util.Iterator;
import java.util.NoSuchElementException;

import stdlib.StdIn;
import stdlib.StdOut;

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

    public MaxPQ() {
        this(1);
    }

    public MaxPQ(int capacity) {
        pq = (Key[]) new Object[capacity + 1];
        n = 0;
    }

    public MaxPQ(Comparator<Key> c) {
        this(1, c);
    }

    public MaxPQ(int capacity, Comparator<Key> c) {
        pq = (Key[]) new Object[capacity + 1];
        n = 0;
        this.c = c;
    }

    public boolean isEmpty() {
        return n == 0;
    }

    public void insert(Key key) {
        pq[n+1] = key;
        swim(n+1);
        n++;
    }

    public Key deleteMax() {
        if (n == 0)
            throw new NoSuchElementException("Priority queue underflow");
        Key max = pq[1];
        exch(1, n);
        pq[n] = null;
        sink(1);
        n--;
        return max;
    }

    public Key max() {
        if (n == 0)
            throw new NoSuchElementException("Priority queue underflow");
        return pq[1];
    }

    public Iterator<Key> iterator() {
        return new PQIterator();
    }

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

    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;
        }
    }

    private void exch(int i, int j) {
        Key temp = pq[i];
        pq[i] = pq[j];
        pq[j] = temp;
    }

    private boolean less(int i, int j) {
        return c.compare(pq[i], pq[j]) < 0;
    }

    private class PQIterator implements Iterator<Key> {
        private int k = 1;

        public boolean hasNext() {
            return k <= n;
        }

        public Key next() {
            if (!hasNext())
                throw new NoSuchElementException();
            return pq[k++];
        }
    }
}
```

Heap-based Implementation

```
</> MaxPQ.java

}

public int size() {
    return n;
}

public void insert(Key key) {
    if (n == pq.length - 1) {
        resize(2 * pq.length);
    }
    pq[++n] = key;
    swim(n);
}

public Key max() {
    if (isEmpty()) {
        throw new NoSuchElementException("Priority queue is empty");
    }
    return pq[1];
}

public Key delMax() {
    if (isEmpty()) {
        throw new NoSuchElementException("Priority queue is empty");
    }
    Key max = pq[1];
    exchange(1, n--);
    sink(1);
    pq[n + 1] = null;
    if (n > 0 && n == (pq.length - 1) / 4) {
        resize(pq.length / 2);
    }
    return max;
}
```

Heap-based Implementation

</> MaxPQ.java

```
public Iterator<Key> iterator() {
    return new HeapIterator();
}

public String toString() {
    StringBuilder sb = new StringBuilder();
    for (Key key : this) {
        sb.append(key);
        sb.append(", ");
    }
    return n > 0 ? "[" + sb.substring(0, sb.length() - 2) + "]" : "[]";
}

private void resize(int capacity) {
    Key[] temp = (Key[]) new Object[capacity];
    for (int i = 1; i <= n; i++) {
        temp[i] = pq[i];
    }
    pq = temp;
}

private void swim(int i) {
    while (i > 1 && less(i / 2, i)) {
        exchange(i, i / 2);
        i /= 2;
    }
}

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

```

Heap-based Implementation

```
</> MaxPQ.java

        break;
    }
    exchange(i, j);
    i = j;
}
}

private boolean less(int i, int j) {
    if (c == null) {
        return ((Comparable) pq[i]).compareTo(pq[j]) < 0;
    }
    return c.compare(pq[i], pq[j]) < 0;
}

private void exchange(int i, int j) {
    Key swap = pq[i];
    pq[i] = pq[j];
    pq[j] = swap;
}

private class HeapIterator implements Iterator<Key> {
    private MaxPQ<Key> copy;

    public HeapIterator() {
        copy = (c == null) ? new MaxPQ<Key>(n) : new MaxPQ<Key>(n, c);
        for (int i = 1; i <= n; i++) {
            copy.insert(pq[i]);
        }
    }

    public boolean hasNext() {
        return !copy.isEmpty();
    }

    public Key next() {
```

Heap-based Implementation

```
</> MaxPQ.java

    if (!hasNext()) {
        throw new NoSuchElementException("Iterator is empty");
    }
    return copy.delMax();
}

public static void main(String[] args) {
    MaxPQ<String> pq = new MaxPQ<String>();
    while (!StdIn.isEmpty()) {
        String item = StdIn.readString();
        if (!item.equals("-")) {
            pq.insert(item);
        } else if (!pq.isEmpty()) {
            StdOut.print(pq.delMax() + " ");
        }
    }
    StdOut.println();
    StdOut.println(pq.size() + " keys left in the pq");
    StdOut.println(pq);
}
```

Heap Sort

Heap Sort

Heapify $a[0:n]$ containing n items to a max-heap, and repeatedly exchange the maximum key with the key at index n and reheapify the first $n - 1$ keys

Heap Sort

Heapify $a[]$ containing n items to a max-heap, and repeatedly exchange the maximum key with the key at index n and reheapify the first $n - 1$ keys

$$T(n) = n \log n$$

Heap Sort

Heap Sort

```
</> Heap.java

public class Heap {
    public static void sort(Comparable[] a) {
        int n = a.length;
        for (int i = n / 2; i >= 1; i--) {
            sink(a, i, n);
        }
        while (n > 1) {
            exchange(a, 1, n--);
            sink(a, 1, n);
        }
    }

    public static void sort(Object[] a, Comparator c) {
        int n = a.length;
        for (int i = n / 2; i >= 1; i--) {
            sink(a, i, n, c);
        }
        while (n > 1) {
            exchange(a, 1, n--);
            sink(a, 1, n, c);
        }
    }

    private static void sink(Comparable[] a, int i, int n) {
        while (2 * i <= n) {
            int j = 2 * i;
            if (j < n && less(a, j, j + 1)) {
                j++;
            }
            if (!less(a, i, j)) {
                break;
            }
            exchange(a, i, j);
            i = j;
        }
    }
}
```

Heap Sort

```
</> Heap.java

}

private static void sink(Object[] a, int i, int n, Comparator c) {
    while (2 * i <= n) {
        int j = 2 * i;
        if (j < n && less(a, j, j + 1, c)) {
            j++;
        }
        if (!less(a, i, j, c)) {
            break;
        }
        exchange(a, i, j);
        i = j;
    }
}

private static boolean less(Comparable[] a, int i, int j) {
    return a[i - 1].compareTo(a[j - 1]) < 0;
}

private static boolean less(Object[] a, int i, int j, Comparator c) {
    return c.compare(a[i - 1], a[j - 1]) < 0;
}

private static void exchange(Object[] a, int i, int j) {
    Object swap = a[i - 1];
    a[i - 1] = a[j - 1];
    a[j - 1] = swap;
}
}
```