Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Priority Queues

Learning Objectives

- To be able to understand and describe the priority queue ADT and the heap;
- To be able to analyze the complexity of the priority queue ADT methods;
- To be able to implement a priority queue ADT with a heap;
- To be able to apply the priority queue ADT and the heap.

Reading

M. T. Goodrich, R. Tamassia and M. H. Goldwasser, *Data Structures and Algorithms in Java*, 6th Edition, 2014.

- **Chapter 9. Heaps and Priority Queues**
- **Sections 9.1-9.4**
- **pp. 328-357**

Priority Queue

A priority queue is an abstract data type for storing a collection of prioritized elements that supports

- arbitrary element insertion,
- removal of elements in order of priority (the element with first priority can be removed at any time).

任意元素插入（arbitrary element insertion）可以将任意带优先级的元素加入队列中。
按优先级移除元素（removal of elements in order of priority）优先级高的元素会**先被移除**（例如最小值或最大值）。
与普通队列不同，元素不一定按插入顺序被处理，而是按其优先级

在 **最小优先队列** 中，min() 或 removeMin() 会移除最小的元素（优先级最高）。
在 **最大优先队列** 中，max() 或 removeMax() 移除最大的元素。

insert(k, v): 插入键为 k 的条目。
removeMin(): 移除并返回最小键条目。
min(): 返回最小键条目（不移除）。
size(), isEmpty(): 查询大小/是否为空。

Priority Queue ADT

- A priority queue stores a collection of entries
- Each entry is a pair (key, value)
- Main methods of the Priority Queue ADT
  - insert(k, v) inserts an entry with key k and value v
  - removeMin() removes and returns the entry with smallest key, or null if the the priority queue is empty
- Additional methods
  - min() returns, but does not remove, an entry with smallest key, or null if the the priority queue is empty
  - size(), isEmpty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market
  - Operating Systems
  - Graph algorithms
  - Heap sort
  - … 操作系统调度、图算法（如 Dijkstra）、股票市场、堆排序等。

步骤	操作	返回值	优先队列内容
1	insert(5, A)	–	[(5, A)]
2	insert(9, C)	–	[(5, A), (9, C)]
3	insert(3, B)	–	[(3, B), (9, C), (5, A)]
4	min()	(3, B)	[(3, B), (9, C), (5, A)]
5	removeMin()	(3, B)	[(5, A), (9, C)]
6	insert(7, D)	–	[(5, A), (9, C), (7, D)]
7	removeMin()	(5, A)	[(7, D), (9, C)]
8	removeMin()	(7, D)	[(9, C)]
9	removeMin()	(9, C)	[]
10	removeMin()	None	[]
11	isEmpty()	True	[]

Example

- Consider a sequence of priority queue methods:
  - insert(5,A), insert(9,C), insert(3,B),
  - min(), removeMin(), insert(7,D),
  - removeMin(), removeMin(), removeMin(), removeMin(), isEmpty()
- What are the returned value and priority queue content in each step?

Example

- A sequence of priority queue methods:

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min()	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin()	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin()	(5,A)	{ (7,D), (9,C) }
removeMin()	(7,D)	{ (9,C) }
removeMin()	(9,C)	{ }
removeMin()	null	{ }
isEmpty()	true	{ }

- How to implement a priority queue?

Implementing a Priority Queue

- One challenge in implementing a priority queue is that we must keep track of both an element and its key, even as entries are relocated within a data structure.

在实现优先队列时，**我们必须同时跟踪每个元素和它的键**，即使它们在数据结构中被移动。这是一个挑战。

为什么这是个挑战？

在 **堆结构** 中，插入、上浮（upheap）、下沉（downheap）等操作会不断移动元素在数组中的位置。如果我们只记录 (key, value)，一旦位置变化，就很难直接找到或更新特定元素的位置。

- How to deal with this challenge?

1. **封装为 Entry 对象**

将每个元素封装成一个 Entry 对象（包含 key, value, 以及在堆中的位置）：

```python
class Entry:
    def __init__(self, key, value):
        self.key = key
        self.value = value
```

2. **使用数组或列表来维护堆**，每次交换节点时也同时交换其 Entry 对象的位置。

3. **维护位置映射（可选）**

如果需要快速查找某个元素，可以用字典（如 element → index）来同步追踪每个条目的位置。

Entry ADT

- An entry in a priority queue is simply a key-value pair
- Priority queues store entries to allow for efficient insertion and removal based on keys
- Methods:
  - getKey: returns the key for this entry
  - getValue: returns the value associated with this entry

- As a Java interface:

```
/**
 * Interface for a key-value
 * pair entry
**/
public interface Entry<K,V>
  {
     K getKey();
     V getValue();
  }
```

概念	解释
Entry	一个 key-value 对
getKey()	获取关键字，用于排序和优先级判断
getValue()	获取值，表示实际存储的数据内容

1. **可比性（Comparability）**：

- 对于任意两个元素 x 和 y，满足：

x ≤ y 或 y ≤ x

2. **反对称性（Antisymmetric）**：

- 如果 x ≤ y 且 y ≤ x，那么：

x = y

3. **传递性（Transitivity）**：

- 如果 x ≤ y 且 y ≤ z，则有：

x ≤ z

Total Order Relations

键（key）可以是任意对象，只要**可以定义一个顺序。**
可以有**多个不同的 entry 具有相同的** key

- Keys in a priority queue can be arbitrary objects on which an order is defined.
- Two distinct entries in a priority queue can have the same key.

- Mathematical concept of total order relation $\leq$
  - Comparability property: either $x \leq y$ or $y \leq x$
  - Antisymmetric property: $x \leq y$ and $y \leq x \Rightarrow x = y$
  - Transitive property: $x \leq y$ and $y \leq z \Rightarrow x \leq z$

Comparator ADT

- A comparator encapsulates the action of comparing two objects according to a given total order relation.
- A generic priority queue uses an auxiliary comparator.
- The comparator is external to the keys being compared.
- When the priority queue needs to compare two keys, it uses its comparator.

- Primary method of the Comparator ADT
- compare(a, b): returns an integer i such that
  - i < 0 if a < b,
  - i = 0 if a = b,
  - i > 0 if a > b.
  - An error occurs if a and b cannot be compared.

Example Comparator

- Lexicographic comparison of 2D points:

```
/** Comparator for 2D points under the
    standard lexicographic order. */
public class Lexicographic implements
    Comparator {
  int xa, ya, xb, yb;
  public int compare(Object a, Object b)
    throws ClassCastException {
     xa = ((Point2D) a).getX();
     ya = ((Point2D) a).getY();
     xb = ((Point2D) b).getX();
     yb = ((Point2D) b).getY();
     if (xa != xb)
           return (xb - xa);
     else
           return (yb - ya);
  }
}
```

- Point objects:

```
/** Class representing a point in the
    plane with integer coordinates */
public class Point2D {
  protected int xc, yc; // coordinates
  public Point2D(int x, int y) {
    xc = x;
    yc = y;
  }
  public int getX() {
        return xc;
  }
  public int getY() {
        return yc;
  }
}
```

Sequence-based Priority Queue

- Implementation with an unsorted list (Sec.9.2.4)

- Performance:
  - insert takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - removeMin and min take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

- Implementation with a sorted list (Sec.9.2.5)

- Performance:
  - insert takes $O(n)$ time since we have to find the place where to insert the item
  - removeMin and min take $O(1)$ time, since the smallest key is at the beginning

插入阶段：将所有元素一个个用 insert() 插入优先队列
移除阶段：用 removeMin() 一次次移除最小元素，将其插入结果序列，从而得到升序排列

Priority Queue Sorting (Sec.9.4)

- We can use a priority queue to sort a list of comparable elements
  1. Insert the elements one by one with a series of insert operations
  2. Remove the elements in sorted order with a series of removeMin operations
- The running time of this sorting method depends on the priority queue implementation.

Algorithm *PQ-Sort*(*S*, *C*)
  Input list *S*, comparator *C* for the elements of *S*
  Output list *S* sorted in increasing order according to *C*
  P ← priority queue with comparator *C*
  while ¬*S.isEmpty* ()
    e ← *S.remove*(*S.first* ())
    P.insert (*e*, **null**)
  while ¬*P.isEmpty*()
    e ← *P.removeMin*().*getKey*()
    S.addLast(*e*)

优先队列实现方式	插入时间	删除最小值时间	总时间复杂度
堆 (Heap)	O(log n)	O(log n)	**O(n log n)** ✅ 最快
无序列表	O(1)	O(n)	O(n²) ❌ 慢
有序列表	O(n)	O(1)	O(n²) ❌ 慢

Selection-Sort

- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence
- Running time of Selection-sort:
  1. Inserting the elements into the priority queue with n insert operations takes $O(n)$ time
  2. Removing the elements in sorted order from the priority queue with n removeMin operations takes time proportional to

$$1 + 2 + \ldots + n$$

- Selection-sort runs in $O(n^2)$ time

Selection-Sort Example

	Sequence S	Priority Queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(7,4)
..	..	..
(g)	()	(7,4,8,2,5,3,9)
Phase 2		
(a)	(2)	(7,4,8,5,3,9)
(b)	(2,3)	(7,4,8,5,9)
(c)	(2,3,4)	(7,8,5,9)
(d)	(2,3,4,5)	(7,8,9)
(e)	(2,3,4,5,7)	(8,9)
(f)	(2,3,4,5,7,8)	(9)
(g)	(2,3,4,5,7,8,9)	()

Insertion-Sort

- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence.
- Running time of Insertion-sort:
  1. Inserting the elements into the priority queue with n insert operations takes time proportional to

     $$1 + 2 + \ldots + n$$

  2. Removing the elements in sorted order from the priority queue with a series of n removeMin operations takes $O(n)$ time.
- Insertion-sort runs in $O(n^2)$ time.

Insertion-Sort Example

	Sequence S	Priority queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(4,7)
(c)	(2,5,3,9)	(4,7,8)
(d)	(5,3,9)	(2,4,7,8)
(e)	(3,9)	(2,4,5,7,8)
(f)	(9)	(2,3,4,5,7,8)
(g)	()	(2,3,4,5,7,8,9)
Phase 2		
(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
..	..	..
(g)	(2,3,4,5,7,8,9)	()

In-place Insertion-Sort

- Instead of using an external data structure, we can implement selection-sort and insertion-sort in-place
- A portion of the input sequence itself serves as the priority queue
- For in-place insertion-sort
  - We keep sorted the initial portion of the sequence
  - We can use swaps instead of modifying the sequence