Lists (Continued)

Ordered Lists

Definition

- Ordered lists are ordinary lists, where the point of insertion of elements is controlled by the list implementation, rather than the user, and such that some order of the elements is ensured

Types of Ordered Lists

1. Chronologically Ordered Lists
   - Elements are ordered as their arrival time
   - Insertion is always made at the last element of the list
   - The time complexity of the insert operation is O(1).

Types of Ordered Lists (cont.)

2. Sorted Lists
   - Elements are placed in the list according to the values of their key fields
   - The condition: key_i <= key_{i+1} is true for all i = 1 to n
   - Associated time complexities:
     - Building a chronologically ordered list of n elements is O(n)
     - Searching an unsorted list of n elements is O(n)
     - Building a sorted list of n elements is, in the worst case: 1 + 2 + 3 + … + n-1= $O(n^2)$
     - Searching a sorted list if arrays are used is O(log n)
     - Sorting an unsorted list of n elements using a good sorting algorithm is O(n log n)

Types of Ordered Lists (cont.)

3. Frequency-Ordered Lists
   - Elements are placed in the list according to an associated probability of being the target of the search operation
   - The element of highest probability is first in the list, and so on.
   - The time complexity of the search operation depends on the probability distribution. It may range:
     - From: $(n+1)/2$, for equal distribution
     - to: $2-2^{(1-n)}$, for the $(1/2, 1/4, 1/8, \ldots, 1/(2^{n-1}))$ distribution
   - A more common distribution follows the 80-20 rule, the search operation in this case takes: $0.122\ n$

Types of Ordered Lists (cont.)

- The ordering can be done without knowing the probabilities by using the self-organizing lists. They use one of the following three methods:
  - If search is successful, increase a frequency counter field by one and move the element forward until it passes all elements of lower frequency counter values.
  - If search is successful, move the element one position forward.
  - If search is successful, move the element to the first position in the list.

Linear Structures

Priority Queues

Abstract Definition

A Priority Queue is a collection of homogeneous elements (i.e. a list), where each element in the list is associated with a priority value. Elements in the queue are served according to their priority, such that a higher priority element is served first.

Specifications

- A priority queue is a HPIFO "highest priority in, first out" structure, which contains elements of some data type, such that each element in the queue is associated with a priority value supplied at insertion time.

- The priorities must be of an ordered type.

Entries

- An entry into the priority queue is an object-oriented composition pattern which defines a single object composed of other objects.
- The simplest entry is a pair, consisting of:
  - Priority, p and
  - An element.
- A Java interface for a simple entry class can be defined as:

```
public interface Entry<P, E> {
    public P getPriority();
    public E getElement();
}
```

- The implementation constructor would set the values of the instance fields

Comparators

- How priorities or keys are compared?
  1. Implement a different priority queue for each priority type to be used.
     - Not very general;
     - Requires a lot of repeated code.
  2. Require that priority types or keys be able to compare themselves to one another.
     - Priority types may not know how they ought to be compared
     - e.g. How to compare two points?
  3. Use special comparator objects that are external to the keys to supply the comparison rules.
     - A priority queue is given a comparator when it is constructed.

Comparators

- A comparator is an object that compares two keys
- A comparator ADT is defined in the standard Java interface: **`java.util.Comprator`** as follows:

  compare(a,b): Returns an integer i such that:
  - i < 0 if a < b,
  - i = 0 if a = b,
  - i > 0 if a > b.

Operations

Remember that a priority queue is a specialized list.

Constructor()	Constructs an empty priority queue.
clear()	Set the priority queue to an empty state.
size()	Return the number of elements in the priority queue
isEmpty()	Check if the priority queue is empty.
isFull()	Check if the priority queue is full.
min()	Return, but do not remove the element of smallest p
insert(p, element)	Add element with priority, p, to queue, return entry.
removeMin()	Remove and return the element of highest priority (i.e. smallest p) from the priority queue.

A Java Priority Queue Interface

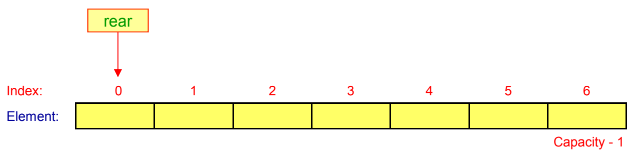
```
public interface PriorityQueue<P, E> {
    public void clear();
    public int size();
    public boolean isEmpty();
    public boolean isFull();
    public Entry<P,E> insert(P priority, E element);
    public Entry<P,E> removeMin();
    public Entry<P,E> min();
}
```

Priority Queue Implementation Using Arrays – Unsorted List

- Use a partially filled array of fixed capacity
- Elements in the list are not ordered according to their priority values
- The priority queue is considered as an unsorted list
- Use one integer variable called rear, that points to the last element of the priority queue
- An empty priority queue is initialized by setting rear = 0.

Priority Queue Implementation Using Arrays – Unsorted List (cont.)

- The queue is empty if the condition: rear == 0 is true.
- The queue is full if the condition: rear == CAPACITY is true.
- How does the insert and removeMin operations work?



Mapping Operations for Unsorted List: insert(p, element) ... O(1)

- If not full, then:
  - Store entry in the array at rear.
  - Increment rear.
- Example:
  - Suppose we have the shown priority queue.
  - insert(2,G)

rear ↓ (index 4)

Index:	0	1	2	3	4	5	6
Element:	(3 , A)	(10 , B)	(1 , C)	(5 , D)			

Capacity - 1

Mapping Operations for Unsorted List: insert(p, element) ... O(1)

- If not full, then:
  - Store entry in the array at rear.
  - Increment rear.
- Example:
  - Suppose we have the shown priority queue.
  - insert(2,G)

rear ↓ (index 4)

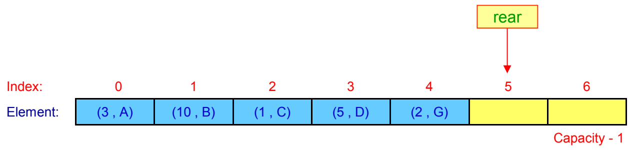
Index:	0	1	2	3	4	5	6
Element:	(3 , A)	(10 , B)	(1 , C)	(5 , D)	(2 , G)		

Capacity - 1

Mapping Operations for Unsorted List: insert(p, element) ... O(1)

- If not full, then:
  - Store entry in the array at rear.
  - Increment rear.

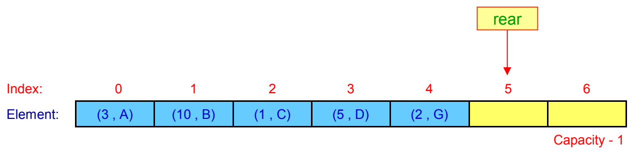
- Example:
  - Suppose we have the shown priority queue.
  - insert(2,G)



Mapping Operations for Unsorted List: removeMin() ... O(3n/2)

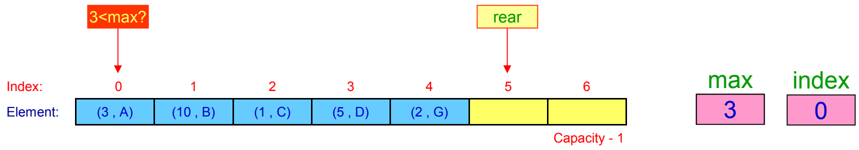
- If not empty, then:
  - Search for the entry having maximum priority in the unsorted list – O(n)
  - Remove the entry of highest priority from the list – O(1)
  - Shift all entries behind the one removed, one position forward – O(n/2)
  - Decrement rear – O(1)

- Example:
  - removeMin() ➔ will return C.



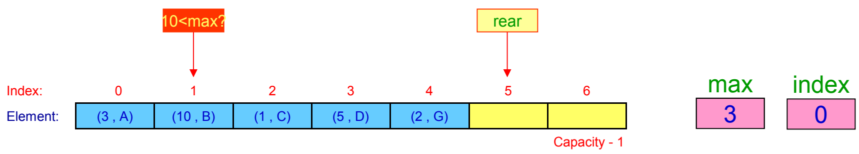
Mapping Operations for Unsorted List: removeMin() ... $O(3n/2)$

- If not empty, then:
  - Search for the entry having maximum priority in the unsorted list – $O(n)$
  - Remove the entry of highest priority from the list – $O(1)$
  - Shift all entries behind the one removed, one position forward – $O(n/2)$
  - Decrement rear – $O(1)$
- Example:
  - removeMin() ➔ will return C.



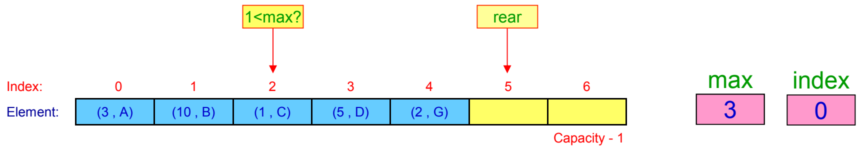
Mapping Operations for Unsorted List: removeMin() ... $O(3n/2)$

- If not empty, then:
  - Search for the entry having maximum priority in the unsorted list – $O(n)$
  - Remove the entry of highest priority from the list – $O(1)$
  - Shift all entries behind the one removed, one position forward – $O(n/2)$
  - Decrement rear – $O(1)$
- Example:
  - removeMin() ➔ will return C.



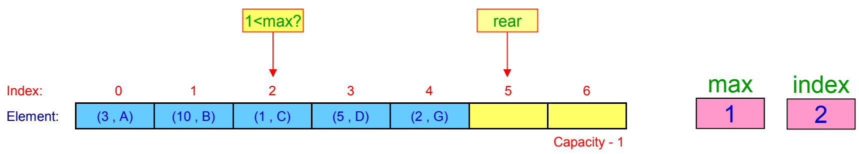
Mapping Operations for Unsorted List: removeMin() ... $O(3n/2)$

- If not empty, then:
  - Search for the entry having maximum priority in the unsorted list – $O(n)$
  - Remove the entry of highest priority from the list – $O(1)$
  - Shift all entries behind the one removed, one position forward – $O(n/2)$
  - Decrement rear – $O(1)$
- Example:
  - removeMin() ➔ will return C.



Mapping Operations for Unsorted List: removeMin() ... $O(3n/2)$

- If not empty, then:
  - Search for the entry having maximum priority in the unsorted list – $O(n)$
  - Remove the entry of highest priority from the list – $O(1)$
  - Shift all entries behind the one removed, one position forward – $O(n/2)$
  - Decrement rear – $O(1)$
- Example:
  - removeMin() ➔ will return C.



Mapping Operations for Unsorted List: removeMin() ... $O(3n/2)$

- If not empty, then:
  - Search for the entry having maximum priority in the unsorted list – $O(n)$
  - Remove the entry of highest priority from the list – $O(1)$
  - Shift all entries behind the one removed, one position forward – $O(n/2)$
  - Decrement rear – $O(1)$
- Example:
  - removeMin() ➔ will return C.

5<max? (index 3)

rear (index 5)

Index:	0	1	2	3	4	5	6
Element:	(3 , A)	(10 , B)	(1 , C)	(5 , D)	(2 , G)		

Capacity - 1

max	index
1	2

Mapping Operations for Unsorted List: removeMin() ... $O(3n/2)$

- If not empty, then:
  - Search for the entry having maximum priority in the unsorted list – $O(n)$
  - Remove the entry of highest priority from the list – $O(1)$
  - Shift all entries behind the one removed, one position forward – $O(n/2)$
  - Decrement rear – $O(1)$
- Example:
  - removeMin() ➔ will return C.

2<max? (index 4)

rear (index 5)

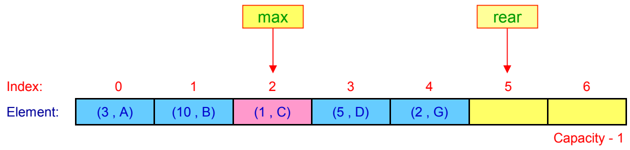
Index:	0	1	2	3	4	5	6
Element:	(3 , A)	(10 , B)	(1 , C)	(5 , D)	(2 , G)		

Capacity - 1

max	index
1	2

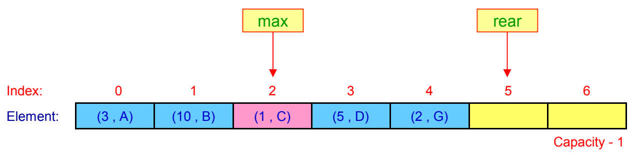
Mapping Operations for Unsorted List: removeMin() … $O(3n/2)$

- If not empty, then:
  - Search for the entry having maximum priority in the unsorted list – $O(n)$
  - Remove the entry of highest priority from the list – $O(1)$
  - Shift all entries behind the one removed, one position forward – $O(n/2)$
  - Decrement rear – $O(1)$
- Example:
  - removeMin() ➔ will return C.



Mapping Operations for Unsorted List: removeMin() … $O(3n/2)$

- If not empty, then:
  - Search for the entry having maximum priority in the unsorted list – $O(n)$
  - Remove the entry of highest priority from the list – $O(1)$
  - Shift all entries behind the one removed, one position forward – $O(n/2)$
  - Decrement rear – $O(1)$
- Example:
  - removeMin() ➔ will return C.



Mapping Operations for Unsorted List: removeMin() ... O(3n/2)

- If not empty, then:
  - Search for the entry having maximum priority in the unsorted list – $O(n)$
  - Remove the entry of highest priority from the list – $O(1)$
  - Shift all entries behind the one removed, one position forward – $O(n/2)$
  - Decrement rear – $O(1)$
- Example:
  - removeMin() ➔ will return C.

rear → 5

Index:	0	1	2	3	4	5	6
Element:	(3 , A)	(10 , B)	(1 , C)	(5 , D)	(2 , G)		

Capacity - 1

Mapping Operations for Unsorted List: removeMin() ... O(3n/2)

- If not empty, then:
  - Search for the entry having maximum priority in the unsorted list – $O(n)$
  - Remove the entry of highest priority from the list – $O(1)$
  - Shift all entries behind the one removed, one position forward – $O(n/2)$
  - Decrement rear – $O(1)$
- Example:
  - removeMin() ➔ will return C.

rear → 5

Index:	0	1	2	3	4	5	6
Element:	(3 , A)	(10 , B)	(5 , D)	(5 , D)	(2 , G)		

Capacity - 1

Mapping Operations for Unsorted List: removeMin() ... O(3n/2)

- If not empty, then:
  - Search for the entry having maximum priority in the unsorted list – $O(n)$
  - Remove the entry of highest priority from the list – $O(1)$
  - Shift all entries behind the one removed, one position forward – $O(n/2)$
  - Decrement rear – $O(1)$
- Example:
  - removeMin() ➔ will return C.

rear ↓ (index 5)

Index:	0	1	2	3	4	5	6
Element:	(3 , A)	(10 , B)	(5 , D)	(2 , G)	(2 , G)		

Capacity - 1

Mapping Operations for Unsorted List: removeMin() ... O(3n/2)

- If not empty, then:
  - Search for the entry having maximum priority in the unsorted list – $O(n)$
  - Remove the entry of highest priority from the list – $O(1)$
  - Shift all entries behind the one removed, one position forward – $O(n/2)$
  - Decrement rear – $O(1)$
- Example:
  - removeMin() ➔ will return C.

rear ↓ (index 4)

Index:	0	1	2	3	4	5	6
Element:	(3 , A)	(10 , B)	(5 , D)	(2 , G)	(2 , G)		

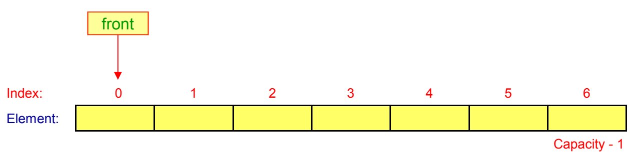
Capacity - 1

Priority Queue Implementation Using Arrays – Sorted List

- Use a partially filled array of fixed capacity
- Elements are assumed to be ordered according to their priority values
- The priority queue is considered as a sorted list
- Use one integer variable called front, that points to the last element of the list. This is the element of highest priority

Priority Queue Implementation Using Arrays – Sorted List (Cont.)

- An empty priority queue is initialized by setting front = 0.
- The queue is empty if the condition: front == 0 is true.
- The queue is full if the condition: front == CAPACITY is true.
- How does the insert and removeMin operations work?



Mapping Operations for Sorted List: insert(p, element) … $O(n)$

- If not full, then:
  - Search for the proper place of the new entry, keeping the list sorted according to priority – $O(n/2)$ on average
  - Shift all entries in front of found place, one position backward – $O(n/2)$
  - Insert the new entry at found place in the list – $O(1)$
  - Increment front – $O(1)$
- Example:
  - insert(2,G)

front

Index:	0	1	2	3	4	5	6
Element:	(10 , B)	(5 , D)	(3 , A)	(1 , C)			

Capacity - 1

Mapping Operations for Sorted List: insert(p, element) … $O(n)$

- If not full, then:
  - Search for the proper place of the new entry, keeping the list sorted according to priority – $O(n/2)$ on average
  - Shift all entries in front of found place, one position backward – $O(n/2)$
  - Insert the new entry at found place in the list – $O(1)$
  - Increment front – $O(1)$
- Example:
  - insert(2,G)

10<=2?

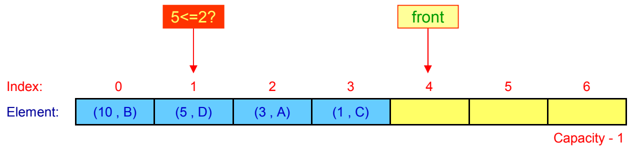
front

Index:	0	1	2	3	4	5	6
Element:	(10 , B)	(5 , D)	(3 , A)	(1 , C)			

Capacity - 1

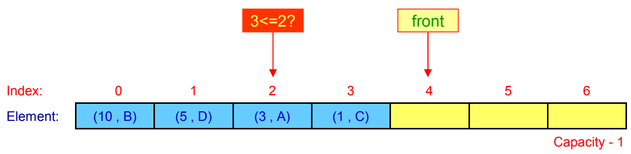
Mapping Operations for Sorted List: insert(p, element) ... $O(n)$

- If not full, then:
  - Search for the proper place of the new entry, keeping the list sorted according to priority – $O(n/2)$ on average
  - Shift all entries in front of found place, one position backward – $O(n/2)$
  - Insert the new entry at found place in the list – $O(1)$
  - Increment front – $O(1)$
- Example:
  - insert(2,G)



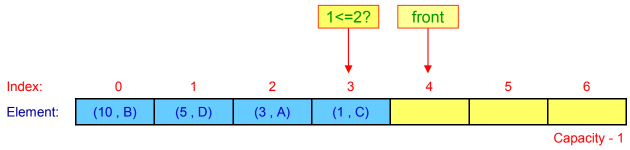
Mapping Operations for Sorted List: insert(p, element) ... $O(n)$

- If not full, then:
  - Search for the proper place of the new entry, keeping the list sorted according to priority – $O(n/2)$ on average
  - Shift all entries in front of found place, one position backward – $O(n/2)$
  - Insert the new entry at found place in the list – $O(1)$
  - Increment front – $O(1)$
- Example:
  - insert(2,G)



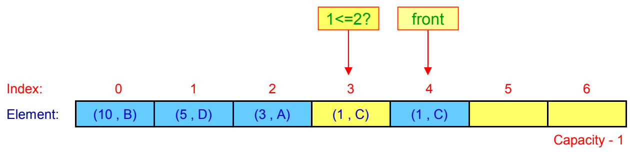
Mapping Operations for Sorted List: insert(p, element) ... $O(n)$

- If not full, then:
  - Search for the proper place of the new entry, keeping the list sorted according to priority – $O(n/2)$ on average
  - Shift all entries in front of found place, one position backward – $O(n/2)$
  - Insert the new entry at found place in the list – $O(1)$
  - Increment front – $O(1)$
- Example:
  - insert(2,G)



Mapping Operations for Sorted List: insert(p, element) ... $O(n)$

- If not full, then:
  - Search for the proper place of the new entry, keeping the list sorted according to priority – $O(n/2)$ on average
  - Shift all entries in front of found place, one position backward – $O(n/2)$
  - Insert the new entry at found place in the list – $O(1)$
  - Increment front – $O(1)$
- Example:
  - insert(2,G)



Mapping Operations for Sorted List: insert(p, element) ... $O(n)$

- If not full, then:
  - Search for the proper place of the new entry, keeping the list sorted according to priority – $O(n/2)$ on average
  - Shift all entries in front of found place, one position backward – $O(n/2)$
  - Insert the new entry at found place in the list – $O(1)$
  - Increment front – $O(1)$
- Example:
  - insert(2,G)

front

Index:	0	1	2	3	4	5	6
Element:	(10 , B)	(5 , D)	(3 , A)	(2 , G)	(1 , C)		

Capacity - 1

Mapping Operations for Sorted List: insert(p, element) ... $O(n)$

- If not full, then:
  - Search for the proper place of the new entry, keeping the list sorted according to priority – $O(n/2)$ on average
  - Shift all entries in front of found place, one position backward – $O(n/2)$
  - Insert the new entry at found place in the list – $O(1)$
  - Increment front – $O(1)$
- Example:
  - insert(2,G)

front

Index:	0	1	2	3	4	5	6
Element:	(10 , B)	(5 , D)	(3 , A)	(2 , G)	(1 , C)		

Capacity - 1

Mapping Operations for Sorted List: removeMin() … *O*(1)

- If not empty, then:
  - Remove the entry from the array at front – *O*(1).
  - Decrement front – *O*(1).
- Example:
  - removeMin() ➔ will return C.

front

Index:	0	1	2	3	4	5	6
Element:	(10 , B)	(5 , D)	(3 , A)	(1 , C)			

Capacity - 1

Mapping Operations for Sorted List: removeMin() … *O*(1)

- If not empty, then:
  - Remove the entry from the array at front – *O*(1).
  - Decrement front – *O*(1).
- Example:
  - removeMin() ➔ will return C.

front

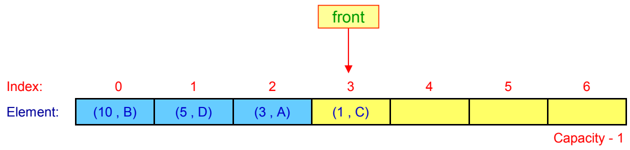
Index:	0	1	2	3	4	5	6
Element:	(10 , B)	(5 , D)	(3 , A)	(1 , C)			

Capacity - 1

Mapping Operations for Sorted List: removeMin() … *O*(1)

- If not empty, then:
  - Remove the entry from the array at front – *O*(1).
  - Decrement front – *O*(1).

- Example:
  - removeMin() ➔ will return C.



Priority Queue Applications

- Managing prioritized processes in a time-sharing environment

- Time-dependent simulations of real systems

- Numerical linear algebra

Advantages and Disadvantages

- Time complexity of one queue operation is O(n).
  - Not so efficient data structure.
- Fixed Storage space must be reserved in advance
  - Queue length is limited to the array size that was reserved.
  - May overflow or may waste unused space.