Queues

Professor Frank Kreimendahl

School of Computing and Data Science

Wentworth Institute of Technology

October 3, 2022

Queue ADT

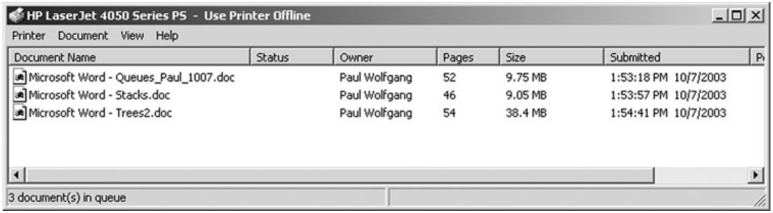
Queue ADT

- A queue is a fundamental data structure in computer science
- A queue works like a queue:
  - Only the front item can be removed
  - Items can only be inserted at the back
  - Only one item can be inserted or extracted at a time
- The front of the queue is the least recently added item in the queue
- The queue is a *First-in, First-out* (FIFO) data structure

Queue Usage

Operating Systems use queues for a variety of reasons:

- Keeping track of program requests for a limited resource
- Ensuring programs get served in a fair order
- CPU, Network, Keyboard input, etc.

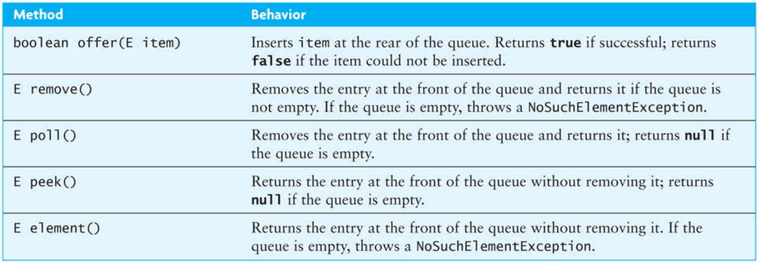

HP LaserJet 4050 Series PS - Use Printer Offline
Printer Document View Help

Document Name	Status	Owner	Pages	Size	Submitted	P
Microsoft Word - Queues_Paul_1007.doc		Paul Wolfgang	52	9.75 MB	1:53:18 PM 10/7/2003	
Microsoft Word - Stacks.doc		Paul Wolfgang	46	9.05 MB	1:53:57 PM 10/7/2003	
Microsoft Word - Trees2.doc		Paul Wolfgang	54	38.4 MB	1:54:41 PM 10/7/2003	

3 document(s) in queue

Windows Print Queue

Queue Interface



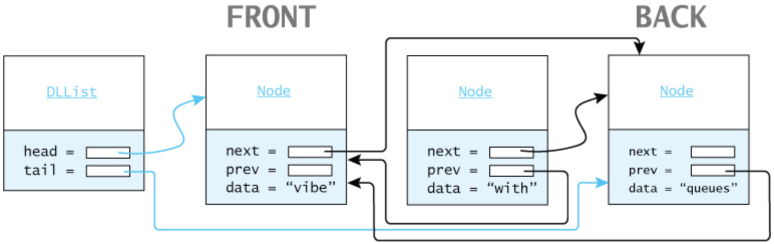
Method	Behavior
`boolean offer(E item)`	Inserts `item` at the rear of the queue. Returns **`true`** if successful; returns **`false`** if the item could not be inserted.
`E remove()`	Removes the entry at the front of the queue and returns it if the queue is not empty. If the queue is empty, throws a `NoSuchElementException`.
`E poll()`	Removes the entry at the front of the queue and returns it; returns **`null`** if the queue is empty.
`E peek()`	Returns the entry at the front of the queue without removing it; returns **`null`** if the queue is empty.
`E element()`	Returns the entry at the front of the queue without removing it. If the queue is empty, throws a `NoSuchElementException`.

The `Queue` interface implements the `Collection` interface in Java, which means any Java implementation must have an iterator as well.

Double-Linked List Implementation

- Use a double-linked list to implement a Queue interface
- List's head corresponds to Queue's front
- List's tail corresponds to Queue's back
- `offer` and `remove` are both efficient
  - Both can be completed without traversing through an entire list
  - Efficient operations on a data structure are important for fast-executing programs

Queue Implementation



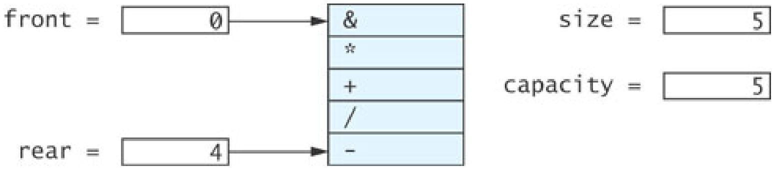
List with queue ends marked

- `offer(E item)` calls a list's `add(E item)`
- `remove()` and `poll()` call a list's `remove(0)`
- `peek()` and `element` call a list's `get(0)`

Circular Array Implementation

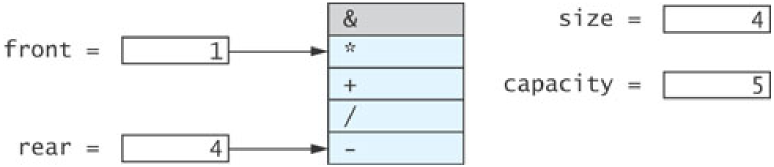
- popping from the front of a regular array is inefficient – we need to shift all of the values forward, one at a time
- solve this issue with a "circular array"
- A circular array holds several pieces of important information:
  - The array with all the items
  - A back index where the most recent value was pushed
  - A front index where the next value can be popped (not necessarily 0)
  - Current size and capacity

Circular Array Example



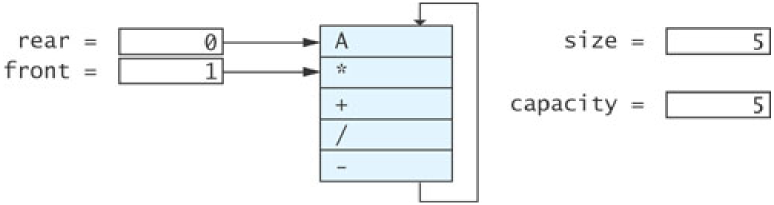
A full queue, with separate variables to maintain the front/back indices

Circular Array Example



Removing from the queue, with an index update

Circular Array Example



Adding “A” to the queue, with an index update

Constructor

```
public ArrayQueue(int initCapacity) {
  capacity = initCapacity;
  theData = (E[]) new Object[capacity];
  front = 0;
  rear = capacity - 1;
  size = 0;
}
```

offer

```
public boolean offer(E item) {
  if (size == capacity)
    reallocate();
  size++;
  rear = (rear + 1) % capacity;
  theData[rear] = item;
  return true;
}
```

poll

```
public E poll() {
  if (size == 0)
    return null;
  E result = theData[front];
  front = (front + 1) % capacity;
  size--;
  return result;
}
```

Iter

```
private class Iter implements Iterator<E> {
  private int index;
  private int count = 0;

  public Iter() {
    index = front;
  }

  @Override
  public boolean hasNext() {
    return count < size;
  }

 ....
```

Iter

```
@Override
 public E next() {
   if (!hasNext())
     throw new NoSuchElementException();
   E returnValue = theData[index];
   index = (index + 1) % capacity;
   count++;
   return returnValue;
 }

 @Override
 public void remove() {
   throw new UnsupportedOperationException();
 }
}
```