Lecture 11

Queue ADT

Lecture 11

Today:

- Queue ADT
- Algorithms that use a Queue
- Implementing a Queue (with a Linked List)
- Implementing a Queue (with a Static Array)

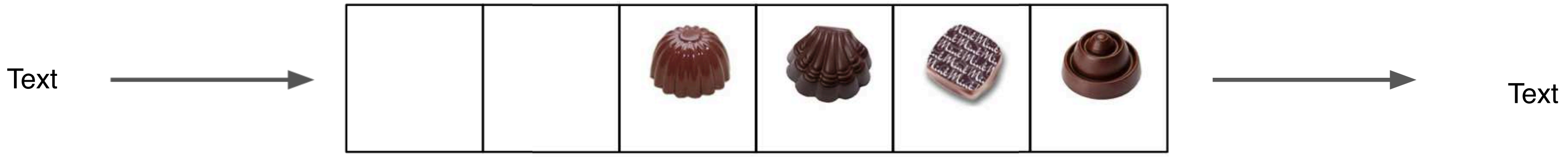
Queue ADT

Queue ADT: A *queue* is a sequence of data, but the insert and remove operations work on

- order is first-in-first-out (FIFO)
- like a line-up

Used in simulations and modeling

- to model sequences of work and their processors, e.g, assembly lines
- Operations Research (OR)



Queue of items

E.g., Fleshing out Queue ADT

Q. What data / properties and operations / methods define a queue?

Data / Properties:

- a sequence of data
- first in first out order

Operations / Methods:

- insert to back(enqueue)
- remove from front (dequeue)
- isEmpty
- top
- size (length)

Search Algorithms

Problem: Given a map and a starting location *x*, find the shortest number of steps to all other locations.

- Or … find the closest location that obeys some property (*goal state*).

Strategy:

- Start by finding all locations adjacent to x (distance = 1)
- Then find all locations adjacent to these (distance = 2)
- Repeat until everything that's reachable has been visited

use a queue to keep track of where to go next

Queue-Based Searching

Breadth-First Search

Problem: Find all locations that are reachable from the start, and compute their distance.

Algorithm:

```
create an empty queue Q;
initialize all distances ← −1 (unreachable), except distance(start) <— 0
while Q not empty {
        dequeue from Q —> current
    if next  is neighbor of current and distance(next)==-1
        distance(next) = distance(current) + 1
        enqueue next → Q
    }
}
```

Sample Map:

50	51	54	57	65	69
48	52	51	58	64	64
47	53	52	54	60	63
45	48	49	56	64	61
44	45	51	57	58	60
42	46	50	52	58	59

Distance:

0	−1	−1	−1	−1	−1
−1	−1	−1	−1	−1	−1
−1	−1	−1	−1	−1	−1
−1	−1	−1	−1	−1	−1
−1	−1	−1	−1	−1	−1
−1	−1	−1	−1	−1	−1

Q: (0,0)(0,1)(1,0)(1,1)(2,0)

Dist: 0. 1. 1. 2. 2

Implementation of Queue ADT

First implementation of a Queue will use a linked list

Q. What's the running time of:

- `.create()?` Text
- `.isEmpty()?` Text
- `.enqueue(x)?` Text
- `.dequeue()?` Text

For `.enqueue(x)` and `.dequeue()`, only issue is to decide which end of the list

`.enqueue(x):`

```
list.append(x);
```

`.dequeue():`

```
return list.removeHead();
```

Queue Implementation: Static Array

Array implementation presents an interesting algorithmic problem

Strategy: Make the back of the queue the size of the array and initially make the front of the queue at index 0.

- `.enqueue(x)` is easy
- but dequeue() is not

Options:

- shift all remaining items down   - O(N)
- increment the front index
  - *O*(1), but   wastes space

```
class Queue {
   private:
      int arr[Q_CAP];
      unsigned size;
       unsigned front;

   public:
      Queue();

      bool isEmpty();
      void enqueue(int x);
      int dequeue();

      ...
};
```

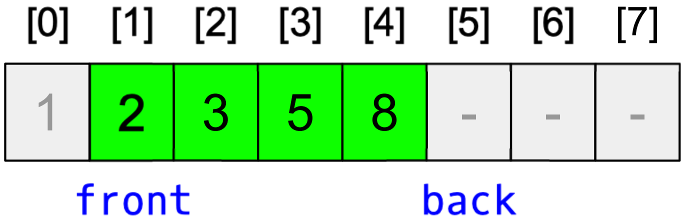
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
arr:	1	2	3	5	8	-	-	-

Queue Implementation: Circular Array

Strategy: Maintain indices of the `front` and `back` elements

- on `.dequeue()` increment `front`:
  - `front =` (front + 1) % capacity;
- on `.enqueue(x)` increment `back`:
  - `back =` (back + 1) % capacity;
- use modulus operator (%) to cycle through indices
  - makes last index adjacent to 0

alternate:
front = (front + 1 == capacity) ? 0 : (front + 1);
back = (back + 1 == capacity) ? 0 : (back + 1);



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
1	2	3	5	8	-	-	-
	`front`				`back`