Data Structures and Algorithms
(資料結構與演算法)

Lecture 7: Queue

Hsuan-Tien Lin (林軒田)

htlin@csie.ntu.edu.tw

Department of Computer Science
& Information Engineering

National Taiwan University
(國立台灣大學資訊工程系)

intuition

Visual Intuition of Queue

figure by Kuro-Historian, licensed under CC0 1.0 via Wikimedia Commons

first-in-first-out (FIFO)

- waiting queue for tickets
- job queue in printer

queue: a restricted data structure,
but also important for computer science

Task Queue in Multithread System

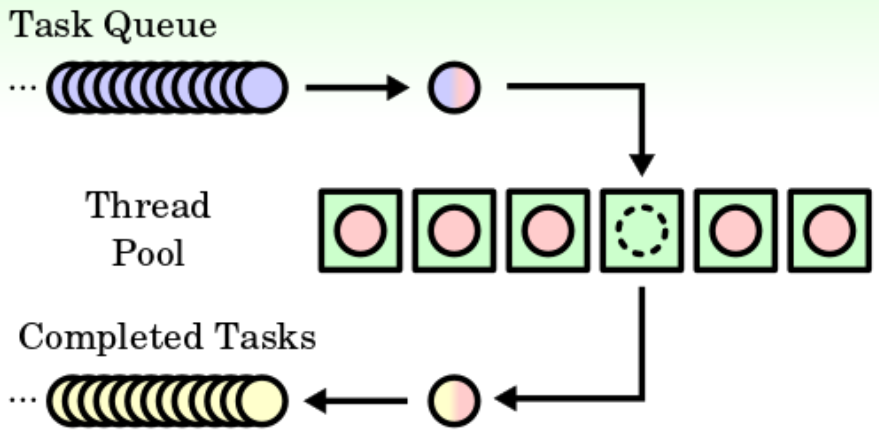
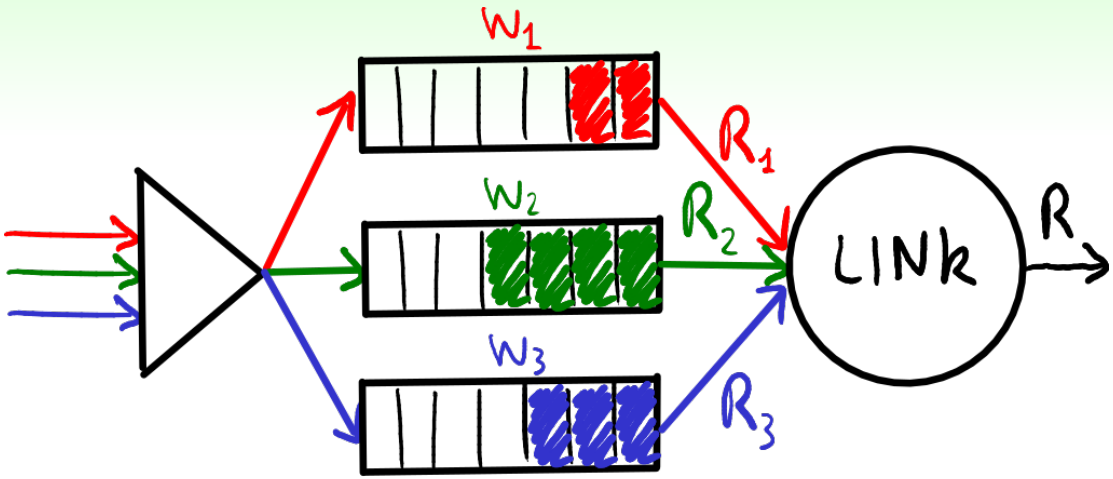


figure by Cburnett, licensed under CC BY-SA 3.0 via Wikimedia Commons

first ready-job, first serve (by available thread)
—but long tasks can occupy the resources (unless round-robin)

task queue: the simplest scheduling mechanism

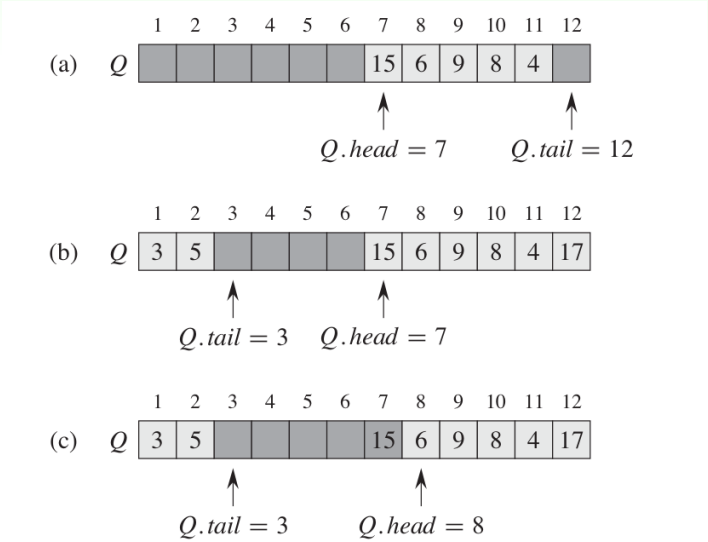
Packet Queue in Networking



multiple queues with different priority for weighted fair queuing

real-world use: from simple/single queue to complicated/multiple queues

Queue Implemented on Circular Array



(Textbook Figure 10.2)

(a) queue with 5 elements between array [7..11]

(b) after ENQUEUE(Q, 17), ENQUEUE(Q, 3), ENQUEUE(Q, 5)

(c) after DEQUEUE(Q) which returns 15

ENQUEUE(Q, *data*)

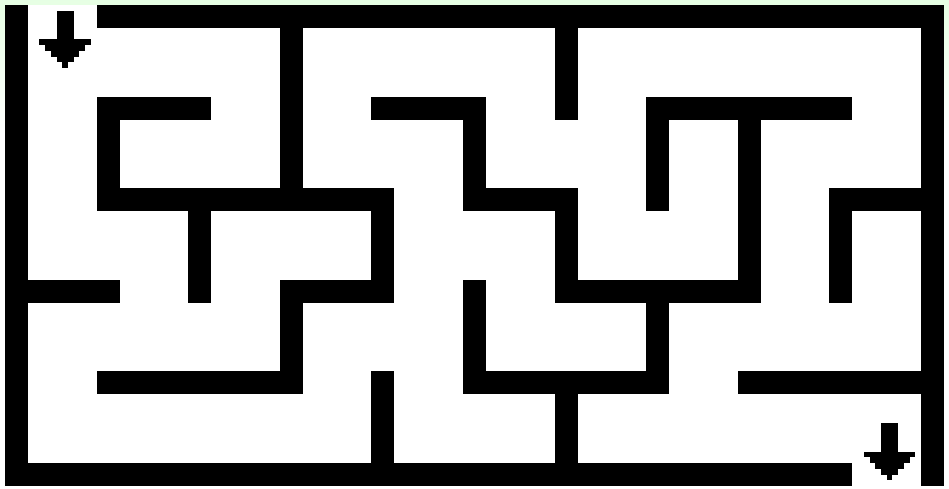
```
1  Q.arr[Q.tail] = data
2  if Q.tail == Q.length
3        Q.tail = 1
4  else
5        Q.tail = Q.tail + 1
6
```

DEQUEUE(Q)

```
1  x = Q.arr[Q.head]
2  if Q.head == Q.length
3        Q.head = 1
4  else
5        Q.head = Q.head + 1
6  return x
```

application: maze solving

The Maze Problem



http://commons.wikimedia.org/wiki/File:Maze01-01.png

Given a (2D) maze, is there a way out?

Maze by Trial-and-Error

iteratively

- start from some location
- record some neighboring locations as future candidates

issue

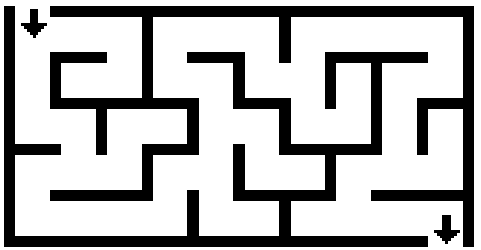
avoid visited location—so no infinite loop

need: storage of future candidates; record of visited

A General Maze Algorithm

start from some location; storage of future candidates; record of visited

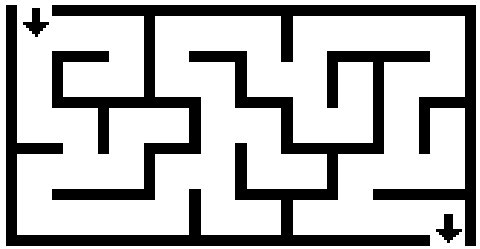
```
MAZE-OUT(M)
 1  Candidate = {}; all Visited[i, j] = FALSE
 2  Candidate.INSERT(M.begin_i, M.begin_j)
    // get out of M starting from (M.begin_i, M.begin_j)
 3  while Candidate not empty
 4        (i, j) = Candidate.REMOVE()
 5        if Visited[i, j] is FALSE
 6              Visited[i, j] = TRUE
 7              for each non-Visited (k, ℓ) neighbor of (i, j)
 8                    if (k, ℓ) is exit
 9                          return SUCCEED
10                    else
11                          Candidate.INSERT(k, ℓ)
12                          // possible duplicates in Candidate, why?
13  return NO-WAY-OUT
```



drunken walk to exit: by random remove from *Candidate*

Maze Algorithm by Stack

```
MAZE-OUT-STACK(M)
 1  Candidate = {}; all Visited[i, j] = FALSE
 2  Candidate.PUSH(M.begin_i, M.begin_j)
    // get out of M starting from (M.begin_i, M.begin_j)
 3  while Candidate not empty
 4        (i, j) = Candidate.POP()
 5        if Visited[i, j] is FALSE
 6              Visited[i, j] = TRUE
 7              for each non-Visited (k, ℓ) neighbor of (i, j)
 8                    if (k, ℓ) is exit
 9                          return SUCCEED
10                    else
11                          Candidate.PUSH(k, ℓ)
12                          // possible duplicates in Candidate, why?
13  return NO-WAY-OUT
```

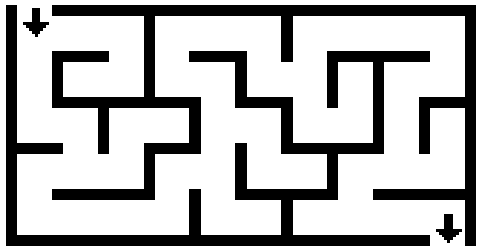


LIFO of Stack ⇒ later neighbor first ⇒ last path out

Stack: last path out by depth-first search

Maze Algorithm by Queue

```
MAZE-OUT-QUEUE(M)
 1  Candidate = {}; all Visited[i, j] = FALSE
 2  Candidate.ENQUEUE(M.begin_i, M.begin_j)
    // get out of M starting from (M.begin_i, M.begin_j)
 3  while Candidate not empty
 4        (i, j) = Candidate.DEQUEUE()
 5        if Visited[i, j] is FALSE
 6              Visited[i, j] = TRUE
 7              for each non-Visited (k, ℓ) neighbor of (i, j)
 8                    if (k, ℓ) is exit
 9                          return SUCCEED
10                    else
11                          Candidate.ENQUEUE(k, ℓ)
12                          // possible duplicates in Candidate, why?
13  return NO-WAY-OUT
```



FIFO of Queue ⇒ nearby neighborS first ⇒ shortest unit-step path out

Queue: shortest path out by breadth-first search

stack + queue = deque

Deques

Deque = Stack + Queue + push_front

- action: [constant-time] push_back (like push and enqueue), pop_back (like pop), pop_front (like dequeue), push_front

- application: job scheduling

can be implemented by circular array or doubly-linked list

Summary

Lecture 7: Queue

- intuition

waiting in line like a decent citizen

- application: maze solving

different data structure causes different algo. behavior

- stack + queue = deque

union of both worlds