CSE 12 – Basic Data Structures

Prof. Christine Alvarado
[Slides borrowed/adapted from slides by Cynthia Lee, Rakesh Varna, & Roshni Chandrashekhar]
Announcements

1. Don’t copy code. Not even starter code.
2. Watch Piazza for announcements about rescheduled midterm
3. Midterm review options:
   1. Review sheet coming next week
   2. Review sessions:
      1. Thursday 5-6:20pm (Shari Haynes, in B00 lecture Pepper Canyon 109)
      2. Friday 10-10:50am (me, in this room)
Queue Design

Which of the following are behaviors inherent in queues, which are decisions we'll make in specifying the behavior of our queue, which are decisions we'll make when implementing our queue.

1. When you dequeue, you get out the oldest element in the queue
2. When you insert, the element is inserted at the beginning of the linked list.
3. When you dequeue an empty queue, it throws an exception.

A. 1 2 3
B. 2 1 3
C. 1 3 2
D. 2 3 1
E. None of these is correct
ArrayQueue: Using a *circular* array underlying data structure

**Solution:** Be more creative!

View the array as *circular* and allow both *front* and *rear* to advance through (around) the array.

This will require *no* data movement for enqueues or dequeues!
public E dequeue()
    { // potential issue if empty,
      // for now, assume not empty
      size--;
      E e = array[front];
      <YOUR CODE HERE>
      return e;
    }

Select the correct code to insert from below:

A
  array[front] = null;
  front++; // Do I need to say
  if(front == array.length)
    front = 0;

B
  rear = rear - 1;
  if(rear < 0)
    rear = array.length - 1;

C
  for(int i = 0; i < rear; i++) {
    array[i] = array[i+1];
  }
  rear = rear - 1;
  if(rear < 0)
    rear = array.length - 1;

D
  None of these are correct
public void enqueue(E e) {
    // potential issue if full,
    // for now, assume not full
    <YOUR CODE HERE>
    size++;
}

Select the correct code to insert from below:

A
    rear++;  // rear++
    if (rear == array.length)  // if(rear == array.length)
        rear = 0;
        array[rear] = e;  // array[rear] = e;

B
    rear++
    array[rear] = e;

C
    for (int i = front; i < rear; i++) {
        array[i] = array[i + 1];
    }
    array[rear] = e;
    front--;

D
    None of these are correct
Design decisions: Where do front and rear point?

Which of these choices will work?

A  
1  12  8  
front  rear  

B  
1  12  8  
front  rear  

C  
1  12  8  
front  rear  

D  
Any of these could work  

It’s your choice, but make sure you know what you’re doing!
Queues using circular Array

Initially empty:

<table>
<thead>
<tr>
<th>Front, rear</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

enqueue(10)

<table>
<thead>
<tr>
<th>10 (front)</th>
<th>rear</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

enqueue(11)

<table>
<thead>
<tr>
<th>10 (front)</th>
<th>11</th>
<th>rear</th>
<th></th>
<th></th>
</tr>
</thead>
</table>
Queues using circular Array

Initially empty:

| Front, rear |   |   |   |   |

enqueue(10)

| 10 (front) | rear |   |   |   |

enqueue(11)

| 10 (front) | 11   | rear |   |   |

What should be the value of front after the next dequeue?
A. 0  B. 1  C. 2  D. 5
Initially empty:

<table>
<thead>
<tr>
<th>Front, rear</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

enqueue(10)

<table>
<thead>
<tr>
<th>10 (front)</th>
<th>rear</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

enqueue(11)

<table>
<thead>
<tr>
<th>10 (front)</th>
<th>11</th>
<th>rear</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What should be the value of arr[0] after the next dequeue?

A. 10    B. 0    C. null    D. It doesn’t matter
Queues using circular Array

dequeue()

<table>
<thead>
<tr>
<th>10 or null</th>
<th>11 (front)</th>
<th>rear</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

dequeue()

<table>
<thead>
<tr>
<th>10 or null</th>
<th>11 or null</th>
<th>Front, rear</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

enqueue(12)

<table>
<thead>
<tr>
<th>10 or null</th>
<th>11 or null</th>
<th>12 (Front)</th>
<th>rear</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Queues using circular Array

<table>
<thead>
<tr>
<th>10 or null</th>
<th>11 or null</th>
<th>12 (Front)</th>
<th>8</th>
<th>3</th>
<th>rear</th>
</tr>
</thead>
</table>

enqueue(20)

What is the value of rear after this enqueue?
A. 5
B. 0
C. 1
D. 2
E. Other
The game of Minesweeper

<table>
<thead>
<tr>
<th>row</th>
<th>col 0</th>
<th>col 1</th>
<th>col 2</th>
<th>col 3</th>
<th>col 4</th>
<th>col 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>row 0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>row 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>row 2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>row 3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>row 4</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>row 5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
The game of Minesweeper

When the user clicks on the cell outlined in red, which other cells should open?

A. Yellow
B. Yellow and blue
C. Yellow, blue and green
D. Yellow, blue, green and purple
E. All non-white cells
Searching for exposed cells

The highlighted yellow region can all be considered “connected” by the cells with 0 neighbors. To find all of these cells, you just explore outward from the first cell, wandering around until you can’t explore any further.

You can imagine that the cells with 0 neighbors have “paths” to each of their neighbors in all directions.
Searching for exposed cells

In CS terms, the cells in the grid form what is called a **graph**. Graphs are data structures with:

- **Vertices** (sometimes called nodes) – in this case the cells in the grid
- **Edges**: In this case the imaginary “handles” from the 0-neighbor cells to the cells next to them.
The cells in yellow form one (strongly) **connected component** in the graph. That is, all the cells you can reach from any other cell in the yellow section.

The cells in orange are the neighbors of the strongly connected component cells and are also revealed in the minesweeper game.
Searching for exposed cells

In this graph, which colors combine to form a different (strongly) connected component?

A. Yellow only
B. Yellow and purple
C. Yellow, purple and green
D. Yellow, purple, green and blue
E. All of the cells form a connected component

The purple and green cells are the neighbors (reachable) of the strongly connected component, but not technically part of it.
### Finding connected components via Search

<table>
<thead>
<tr>
<th>row</th>
<th>col 0</th>
<th>col 1</th>
<th>col 2</th>
<th>col 3</th>
<th>col 4</th>
<th>col 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>row 0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>row 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>row 2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>row 3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>row 4</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>row 5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

#### SearchForConnectedComponent
1. A **queue** to hold all the nodes in the Connected Component (CC)
2. A **task list** to hold nodes as we search
   - Mark starting node as visited
   - Put starting node on task list
   - While **task list** is not empty
     - Remove node from task list
     - Add node to **CC queue**
     - For each of node’s unvisited neighbors (reachable via edge/”handle”):
       - Mark neighbor as visited
       - Add neighbor to **task list**
   - **Return CC queue**

The ADT used for the task list determines the order in which nodes are visited and placed in the queue!
## Finding connected components via Depth First Search

<table>
<thead>
<tr>
<th>row</th>
<th>column 0</th>
<th>column 1</th>
<th>column 2</th>
<th>column 3</th>
<th>column 4</th>
<th>column 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### SearchForConnectedComponentDFS

- Initialize:
  1. A queue to hold all the nodes in the CC
  2. A Stack to hold nodes as we search
- Mark starting node as visited
- Put starting node on task list
- While Stack is not empty
  - Pop node from Stack
  - Add node to CC queue
  - For each of node’s unvisited neighbors:
    - Mark neighbor as visited
    - Push neighbor on Stack
- Return CC queue

**RC Queue**

(2,5) (3,5) (4,5)
Finding connected components via Breadth First Search

SearchForConnectedComponentDFS
• Initialize:
  1. A queue to hold all the nodes in the CC
  2. A Queue to hold nodes as we search
• Mark starting node as visited
• Put starting node on task list
• While Queue is not empty
  • Dequeue node from Queue
  • Add node to CC queue
  • For each of node’s unvisited neighbors:
    • Mark neighbor as visited
    • Enqueue neighbor in Queue
• Return CC queue
Search and HW4

- You are given code for BFS. Make sure you can map it to the algorithm we saw today! (exam question possibility…)
- You will write code for DFS. Trivial if you understand what’s going on.
- Then you will explore the different behaviors of the two algorithms.