Chapter 3 - QUEUE

- Definition of Queue
- Specifications for Queue
- Implementations of Queue
  - Linked Queue
  - Contiguous Queue
- Applications of Queue
Linear List Concepts

- Linear lists
  - General
    - Unordered
    - Ordered
  - Restricted
    - FIFO (Queue)
    - LIFO (stack)
Queue - FIFO data structure

- Queues are one of the most common of all data-processing structures.

- Queues are used where someone must wait one's turn before having access to something.

- Queues are used in every operating system and network: processing system services and resource supply: printer, disk storage, use of the CPU, ...

- Queues are used in business online applications: processing customer requests, jobs, and orders.
Queue ADT

DEFINITION: A **Queue** of elements of type T is a finite sequence of elements of T, in which data can be inserted only at one end, called the **rear**, and deleted from the other end, called the **front**.

Queue is a First In - First Out (FIFO) data structure.

**Basic operations:**

- **Construct** a Queue, leaving it empty.
- **Enqueue** an element.
- **Dequeue** an element.
- **QueueFront**.
- **QueueRear**.
Basic operation of Queue (EnQueue)

Before

<table>
<thead>
<tr>
<th>rear</th>
<th>front</th>
</tr>
</thead>
</table>

After

<table>
<thead>
<tr>
<th>rear</th>
<th>front</th>
</tr>
</thead>
</table>

EnQueue

a) **Successful operation**: function returns *success*

b) **Unsuccessful operation**: function returns *overflow*
Basic operation of Queue (DeQueue)

Before

<table>
<thead>
<tr>
<th>rear</th>
<th>front</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

After

<table>
<thead>
<tr>
<th>rear</th>
<th>front</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DeQueue

a) **Successful operation**: function returns *success*

b) **Unsuccessful operation**: function returns *underflow*
Basic operation of Queue (QueueFront)

a) Successful operation: function returns *success*

b) Unsuccessful operation: function returns *underflow*
Basic operation of Queue (QueueRear)

Before

<table>
<thead>
<tr>
<th>rear</th>
<th>front</th>
</tr>
</thead>
</table>

After

<table>
<thead>
<tr>
<th>rear</th>
<th>front</th>
</tr>
</thead>
</table>

Received data: X

Queue remains unchanged

a) **Successful operation**: function returns *success*

b) **Unsuccessful operation**: function returns *underflow*
Queue ADT (cont.)

Extended operations:

• Determine whether the queue is *empty* or not.
• Determine whether the queue is *full* or not.
• Find the *size* of the queue.
• *Clear* the queue to make it empty.
• Determine the total number of elements that have ever been placed in the queue.
• Determine the average number of elements processed through the queue in a given period.
• …
Specifications for Queue ADT

<void> Create()
<ErrorCode> EnQueue (val DataIn <DataType>)
<ErrorCode> DeQueue ()
<ErrorCode> QueueFront (ref DataOut <DataType>)
<ErrorCode> QueueRear (ref DataOut <DataType>)
<boolean> isEmpty ()
<boolean> isFull ()
<void> Clear ()
<integer> Size () // the current number of elements in the queue.

Variants:
ErrorCode DeQueue (ref DataOut <DataType>)
...

CuuDuongThanCong.com
https://fb.com/tailieudientucntt
Built Queue ADT

Queue may be fully inherited from a List, inside its operations calling List’s operations.

<ErrorCode> **EnQueue** (val DataIn <DataType>)
Call List::InsertTail(DataIn)
or
Call List::Insert(DataIn, Size()) // insert after last element
end EnQueue

<ErrorCode> **DeQueue** (val DataOut <DataType>)
Call List::RemoveHead(DataOut)
or
Call List::Remove(DataOut, 0) // remove element from the 1st position
end EnQueue

Similar for other operations of Queue…
Implementations of Queue

✓ Contiguous Implementation.

✓ Linked Implementation.
Linked Queue

a) Conceptual

Node
Data <DataType>
link <pointer>
end Node

Queue
front <pointer>
rear <pointer>
count <integer>
end Queue

b) Physical

front
rear
count
Create an Empty Linked Queue

Before

front = ?
rear = ?
count = ?

After

front = NULL
rear = NULL
count = 0
Create Linked Queue

<void> Create()

Creates an empty linked queue

Pre none

Post An empty linked queue has been created.

1. front = NULL
2. rear = NULL
3. count = 0
4. Return
end Create
**EnQueue**

**Before:**
- count: 4
- front
- rear

**After:**
- count: 5
- front
- rear

- **pNew->data = DataIn**
- **pNew->link = NULL**
- **rear->link = pNew**
- **rear = pNew**
- **count = count + 1**

CuDuongThanCong.com  https://fb.com/tailieudientucntt
Before:

After:

EnQueue (cont.)

Before:

Count: 0
front: null
rear: null

After:

Count: 1
front: pNew
rear: pNew

pNew->data = DataIn
pNew->link = NULL
front = pNew
rear = pNew
count = count + 1
DeQueue

Before:

Count 4
front
rear

After:

Count 3
front
rear

\( pDel = \text{front} \)
\( \text{front} = \text{front} \rightarrow \text{link} \)
\( \text{recycle} \ pDel \)
\( \text{count} = \text{count} - 1 \)
DeQueue

Before:
- Count: 1
- front
- rear

After:
- Count: 0
- front
- rear

pDel = front
front = NULL
rear = NULL
recycle pDel
count = count - 1
EnQueue & DeQueue Algorithm

- EnQueue is successful when queue is not full.
- DeQueue successful when queue is not empty.

**Regular cases:**
- EnQueue: only rear must be updated (*points to new element*).
- DeQueue: only front must be updated (*points to next element if exists*).

**Irregular cases:**
- EnQueue an element to an empty queue: both rear and front must be updated (*point to new element*).
- DeQueue a queue having only one element: both rear and front must be updated (*receive NULL value*).

- In any successful case, count must be updated.
EnQueue Algorithm

<ErrorCode> **EnQueue** (val **DataIn** <DataType>)

Inserts one element at the rear of the queue.

**Pre**  **DataIn** contains data to be inserted.

**Post** If queue is not full, **DataIn** has been inserted at the rear of the queue; otherwise, queue remains unchanged.

**Return** *success* or *overflow*.
<ErrorCode> EnQueue (val DataIn <DataType>)

// For Linked Queue
1. Allocate pNew
2. If (allocation was successful)
   1. pNew->data = Data
   2. pNew->link = NULL
   3. if (count = 0)
      1. front = pNew
   4. else
      1. rear->link = pNew
5. rear = pNew
6. count = count + 1
7. return success
3. Else
   1. return overflow
end EnQueue

Empty queue:
- pNew->data = DataIn
- pNew->link = NULL
- front = pNew
- rear = pNew
- count = count + 1

Not empty queue:
- pNew->data = DataIn
- pNew->link = NULL
- rear->link = pNew
- rear = pNew
- count = count + 1
DeQueue Algorithm

<ErrorCode> DeQueue()

Deletes one element at the front of the queue.

Pre  none

Post  If the queue is not empty, the element at the front of the queue has been removed; otherwise, the queue remains unchanged.

Return success or underflow.
Deque Algorithm (cont.)

<ErrorCode> DeQueue()
// For Linked Queue
1. If (count > 0)
   1. pDel = front
   2. front = front->link
   3. if (count = 1)
      1. rear = NULL
   4. recycle pDel
   5. count = count - 1
   6. return success
2. else
   1. return underflow
3. end DeQueue

Queue has more than one element:
pDel = front
front = front->link
recycle pDel
count = count - 1

Queue has only one element:
pDel = front
front = NULL // = front->link
rear = NULL
recycle pDel
count = count - 1
QueueFront Algorithm

<ErrorCode> QueueFront (ref DataOut <DataType>)

Retrieves data at the front of the queue without changing the queue.

Pre none.

Post if the queue is not empty, DataOut receives data at its front. The queue remains unchanged.

Return success or underflow.

// For Linked Queue

1. If (count > 0)
   1. DataOut = front->data
   2. Return success
2. Else
   1. Return underflow
3. End QueueFront
Contiguous Implementation Of Queue

Circular queue

Linear implementation

front

rear

occupied

max – 1

max – 2

max – 1

max – 2

CuuDuongThanCong.com

https://fb.com/tailieudientucntt
Contiguous Implementation Of Queue (cont.)

Boundary conditions

- Queue containing one item:
  - rear
  - front
  - Remove the item.

- Empty queue:
  - rear
  - front

- Queue with one empty position:
  - rear
  - front
  - Insert an item.

- Full queue:
  - rear
  - front
Contiguous Implementation Of Queue (cont.)

- The physical model: a linear array with the front always in the first position and all elements moved up the array whenever the front is deleted.
- A linear array with two indices always increasing.
- A circular array with front and rear indices and one position left vacant.
- A circular array with front and rear indices and a Boolean flag to indicate fullness (or emptiness).
- A circular array with front and rear indices and an integer counter of elements
Contiguous Implementation Of Queue (cont.)

```plaintext
Queue
  front <integer>
  rear <integer>
  data <array of <DataType>>
  count <integer>
End Queue // Automatically Allocated Array

<void> Create()
Pre  none.
Post  An empty queue has been created.
       1. count = 0
       2. front = -1
       3. rear = -1
end Create
```
EnQueue & DeQueue Algorithm

- EnQueue is successful when queue is not full.
- DeQueue is successful when queue is not empty.

Regular cases:
- EnQueue: only rear must be updated (increases by 1)
- DeQueue: only front must be updated (increases by 1)

Irregular cases:
- EnQueue an element to an empty queue: both rear and front must be updated (receive 0 value – 1st position in array).
- DeQueue a queue having only one element: both rear and front must be updated (receive -1 value).

In any successful case, count must be updated.
Queuing Theory

• A field of applied mathematics that is used to predict the performance of queues.

• Two types of queues:
  – Single-server queue: provides service to only one customer at a time.
  – Multi-server queue: provides service to many customers at a time.
Queuing Theory

- The two factors that most dramatically affect the queue:
  - **Arrival rate**: the rate at which customers arrive in the queue for service.
  - **Service time**: the average time required to complete the processing of a customer request.
Queuing Theory

• Performance of a queue is measured by:
  – **Queue time**: the average length of time customers wait in the queue.
  – **Response time** = queue time + service time
  – The average size of the queue.
  – The maximal queue size
Queuing Theory

• For a banking queue:
  – If the average service time is reduced by 15%, how many fewer tellers would we need?
  – Given a growing system that is currently under capacity, how long will it be before we need to add another service.
Queue Applications

- Polynomial Arithmetic
- Categorizing Data
- Evaluate a Prefix Expression
- Radix Sort
- Queue Simulation
void PolynomialSum(val p1<Queue>, val p2<Queue>, ref q<Queue>)

Calculates  q = p1 + p2

Pre  p1 and p2 are two polynomials, each element in them consists of a coefficient and an exponent. Elements in a polynomial appear with descending exponents.

Post  q is the sum of p1 and p2

Uses  Queue ADT
void PolynomialSum (val p1 <Queue>, val p2 <Queue>, ref q <Queue>)
1. q.Clear()
2. loop ( NOT p.isEmpty() OR NOT q.isEmpty() )
   1. p1.QueueFront(p1Data)
   2. p2.QueueFront(p2Data)
3. if (p1Data.degree > p2Data.degree)
   1. p1.DeQueue()
   2. q.EnQueue(p1Data)
4. else if (p2Data.degree > p1Data.degree)
   1. p2.DeQueue()
   2. q.EnQueue(p2Data)
5. else
   1. p1.DeQueue()
   2. p2.DeQueue()
   3. if (p1Data.coefficient + p2Data.coefficient <> 0)
      1. qData.coefficient = p1Data.coefficient + p2Data.coefficient
      2. qData.degree = p1Data.degree
      3. q.EnQueue(qData)
End PolynomialSum
Categorizing Data

✓ Sometimes data need to rearrange without destroying their basic sequence.

✓ Samples:

  • Ticket selling: several lines of people waiting to purchase tickets and each window sell tickets of a particular flight.
  
  • Delivery center: packages are arranged into queues base on their volumes, weights, destinations,...
Categorizing Data (cont.)

Rearrange data without destroying their basic sequence.

\[
\begin{array}{cccccccccccccccc}
3 & 22 & 12 & 6 & 10 & 34 & 65 & 29 & 9 & 30 & 81 & 4 & 5 & 19 & 20 & 57 & 44 & 99 \\
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
3 & 6 & 9 & 4 & 5 & 12 & 10 & 19 & 22 & 29 & 20 & 34 & 65 & 30 & 81 & 57 & 44 & 99 \\
\end{array}
\]

\[
\begin{array}{cccccccccccccccc}
< 10 & 10 \rightarrow 19 & 20 \rightarrow 29 & \geq 30 \\
\end{array}
\]
Categorizing Data (cont.)

Multiple Queue

3  6

12  10

22

34
Algorithm **Categorize**

Groups a list of numbers into four groups using four queues.

1. queue1, queue2, queue3, queue4 <Queue>

2. loop (not EOF)
   1. read (number)
   2. if (number < 10)
      1. queue1.EnQueue(number)
   3. else if (number < 20)
      1. queue2.EnQueue(number)
   4. else if (number < 30)
      1. queue3.EnQueue(number)
   5. else
      1. queue4.EnQueue(number)

3. // Takes data from each queue.

4. End Categorize
Evaluate a Prefix Expression

Use two queues in turns to evaluate a prefix expression:

front
- + * 9 + 2 8 * + 4 8 6 3 (q1)

- + * 9 10 * 12 6 3 (q2)

- + 90 72 3 (q1)

- 162 3 (q2)

159 (q1)
Radix Sort

✓ Algorithm applied to data that use character string as key.

✓ Very efficient sorting method that use linked queues.

✓ Consider the key one character at a time

✓ Devide the elements into as many sublists as there are possibilities for given character from the key.

✓ To eleminate multiplicity of sublists, consider characters in the key from right to left.
Radix Sort

Sorted by letter 3

```
Radix: front
  mop -> map -> top -> rap

Radix: qr
  car -> tar

Radix: qt
  rat -> cat -> cot
```

rat, mop, cat, map, car, top, cot, tar, rap
Radix Sort (cont.)
Radix Sort (cont.)

Sorted by letter 2
Radix Sort (cont.)
Radix Sort (cont.)

Sorted by letter 1
Radix Sort (cont.)

Sorted list
Queue Simulation

• A modelling activity to generate statistics about the performance of queues.

• Gilberg-Forouzan’s textbook:
  – Queue simulation program