Queues

CPE 212 -- Lecture 10

UAHuntsville

Outline

- Queue ADT
  - Concepts
  - Implementations
- Summary

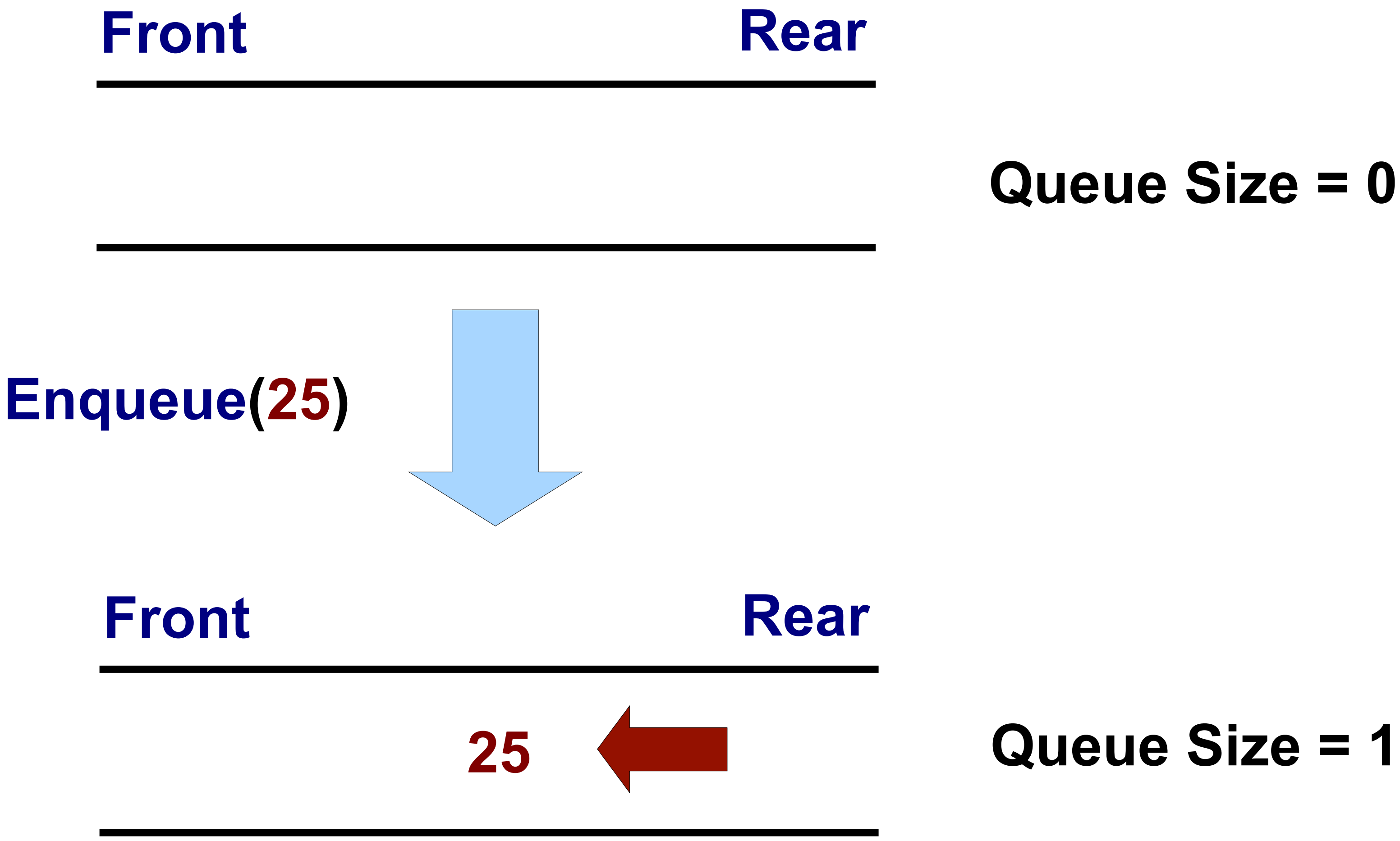
Queue ADT – Basic Concepts

- An ordered homogeneous data structure in which ***elements are added to the rear and removed from the front***
- **FIFO** - **F**irst **I**n, **F**irst **O**ut
- Example:
  - Check out line at the grocery store

Queue ADT - Basic Operations

- **Enqueue**
  - Adds one element to the rear of the queue
- **Dequeue**
  - Removes and returns item from the front of the queue
- **IsEmpty**
  - Determines whether the queue is empty
- **IsFull**
  - Determines whether the queue is full
- **MakeEmpty**
  - Initializes the queue to the empty state

Enqueue Operation - 1



Front

Rear

Queue Size = 0

Enqueue(25)

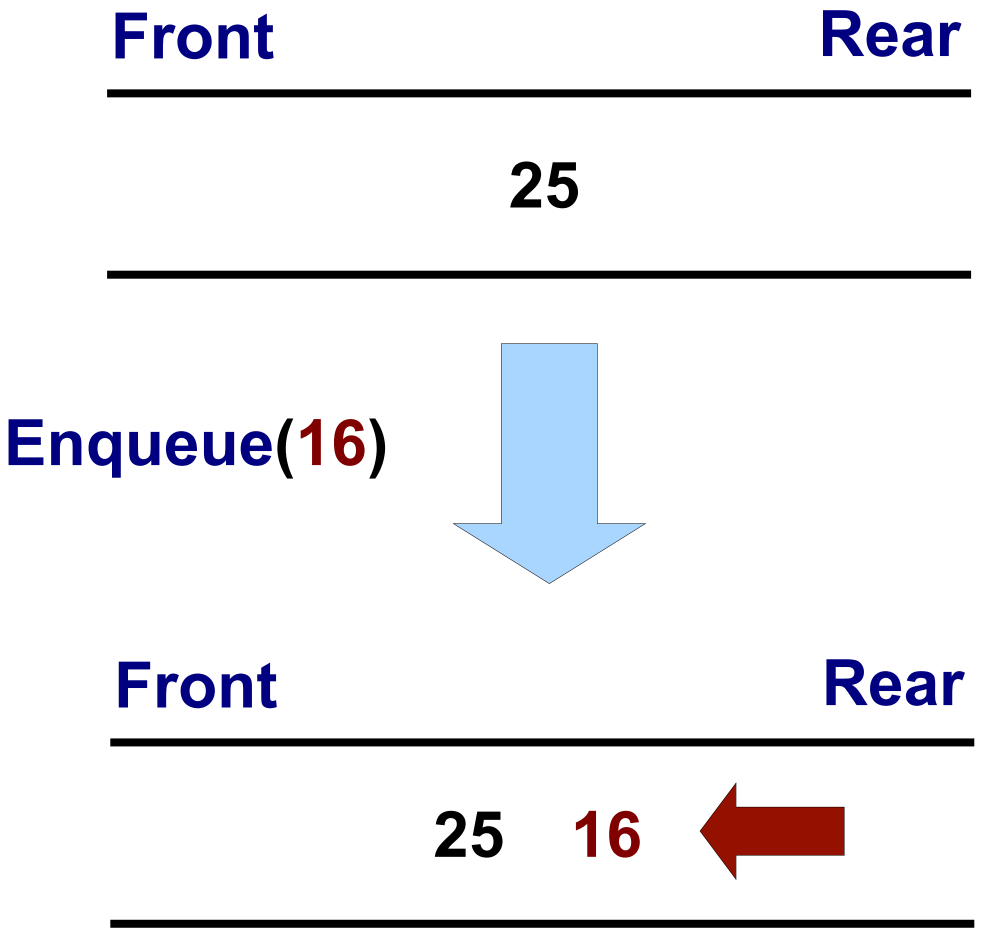
Front

Rear

25

Queue Size = 1

Enqueue Operation - 2



Front Rear

25

Queue Size = 1

Enqueue(16)

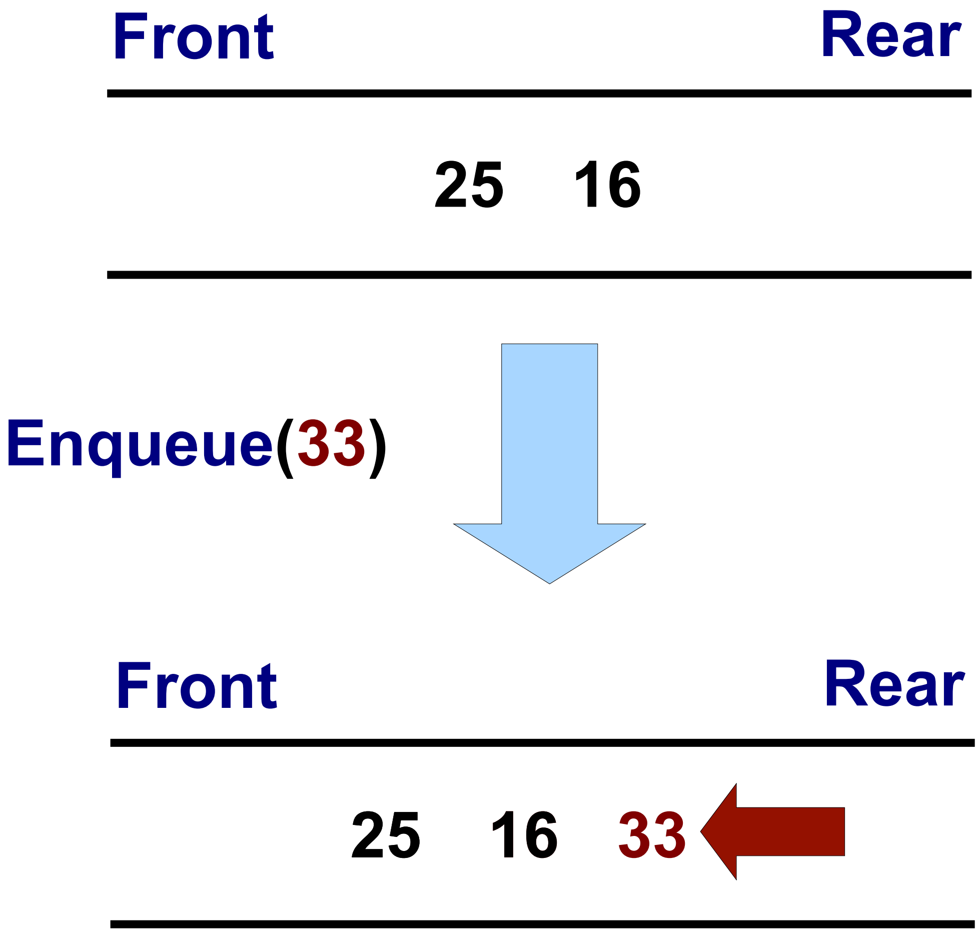
Front Rear

25 16

Queue Size = 2

UAHuntsville

Enqueue Operation - 3



Front

Rear

25 16

Queue Size = 2

Enqueue(33)

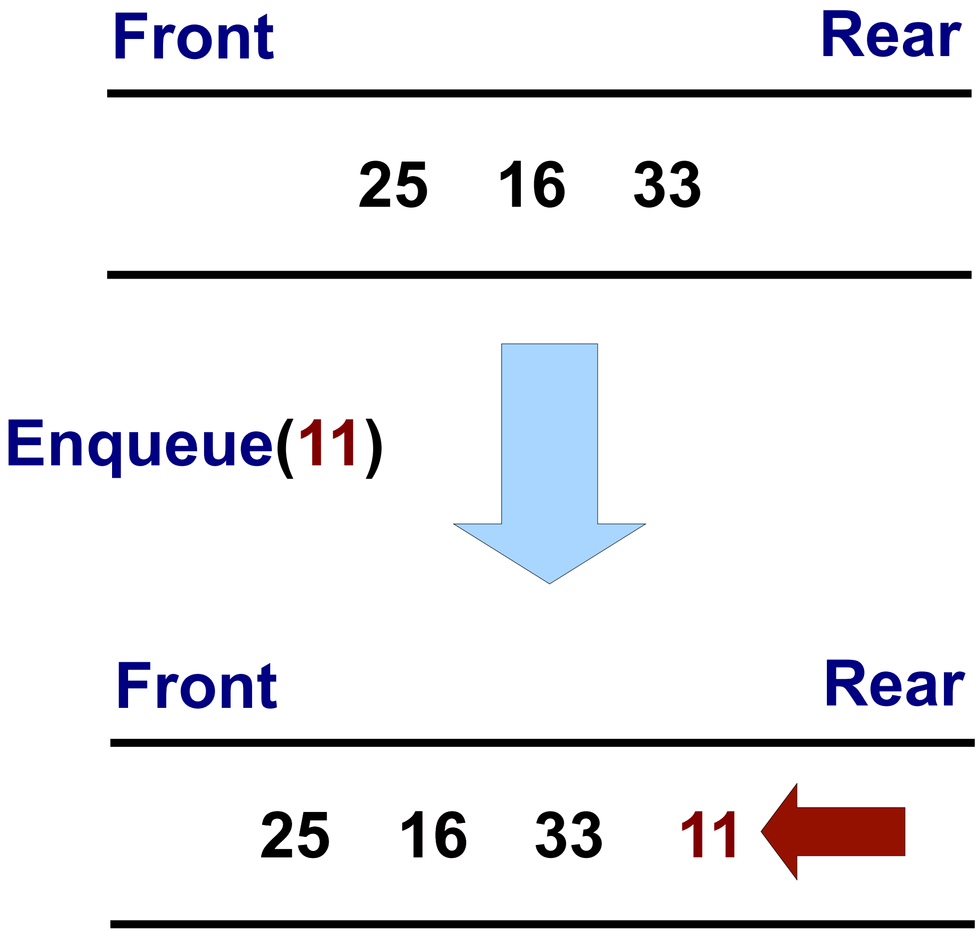
Front

Rear

25 16 33

Queue Size = 3

Enqueue Operation - 4



Front Rear

25 16 33

Queue Size = 3

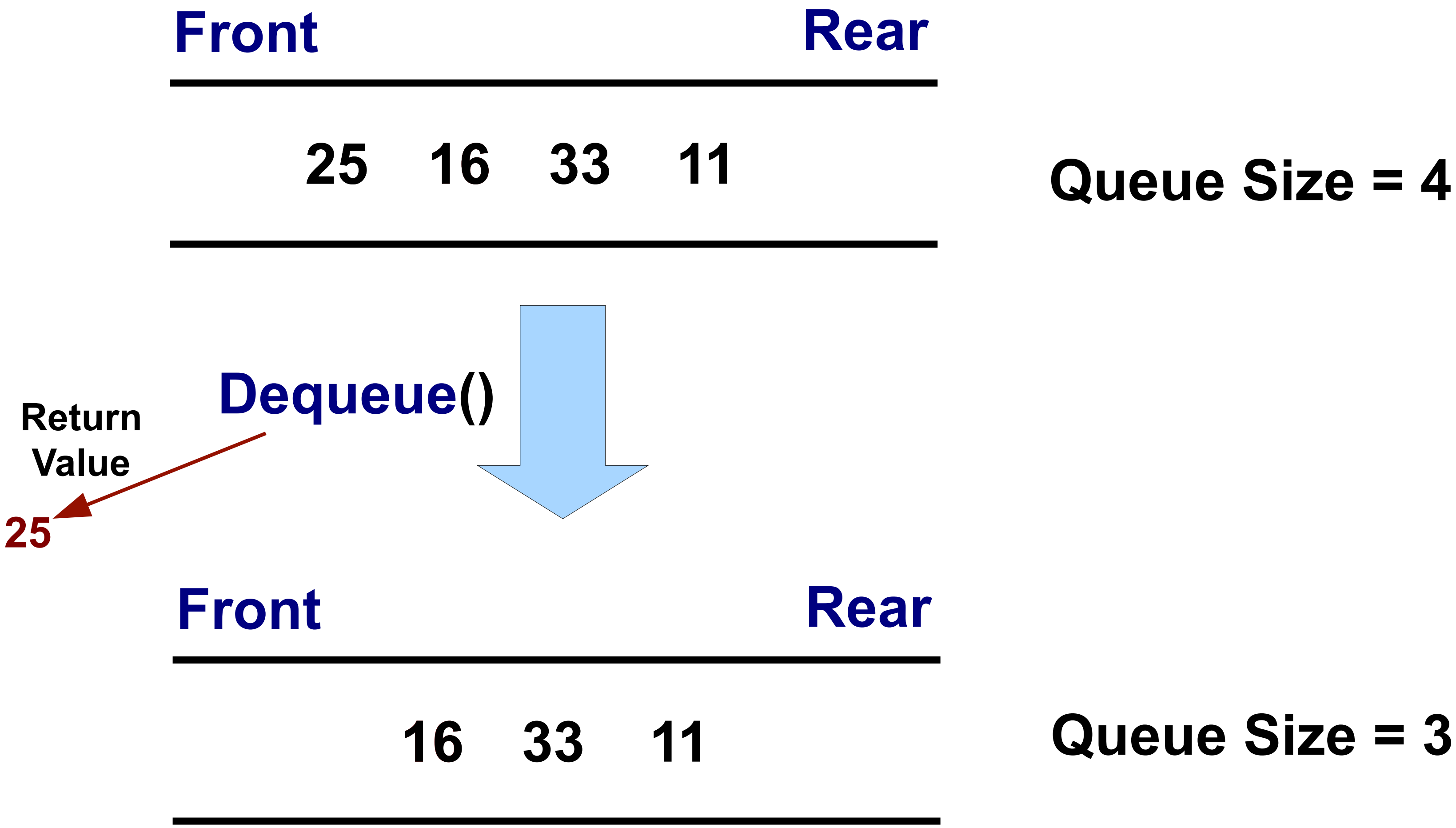
Enqueue(11)

Front Rear

25 16 33 11

Queue Size = 4

Dequeue Operation - 1



Front Rear

25 16 33 11

Queue Size = 4

Dequeue()

Return Value

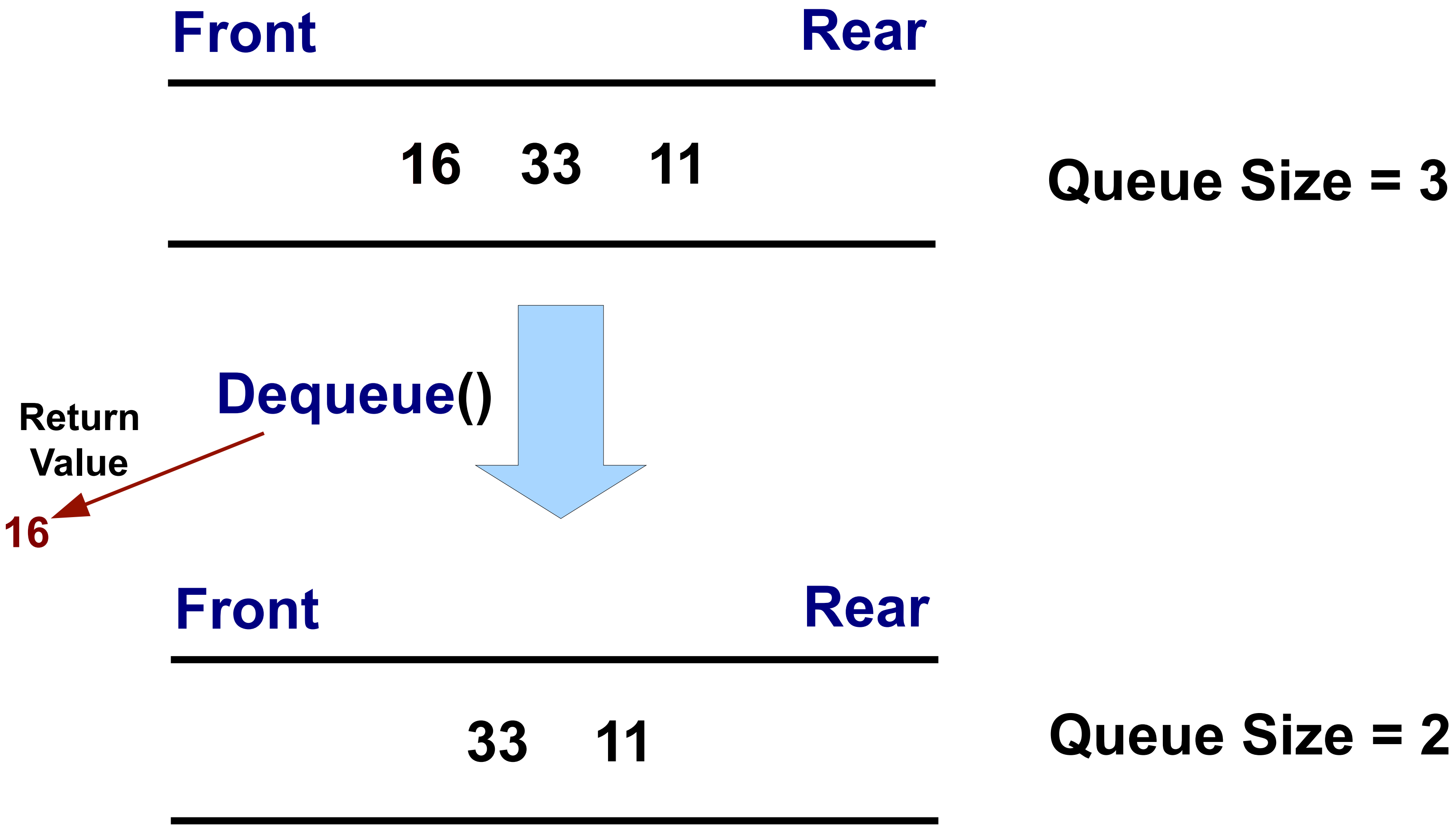
25

Front Rear

16 33 11

Queue Size = 3

Dequeue Operation - 2



Front Rear

16 33 11

Queue Size = 3

Dequeue()

Return Value

16

Front Rear

33 11

Queue Size = 2

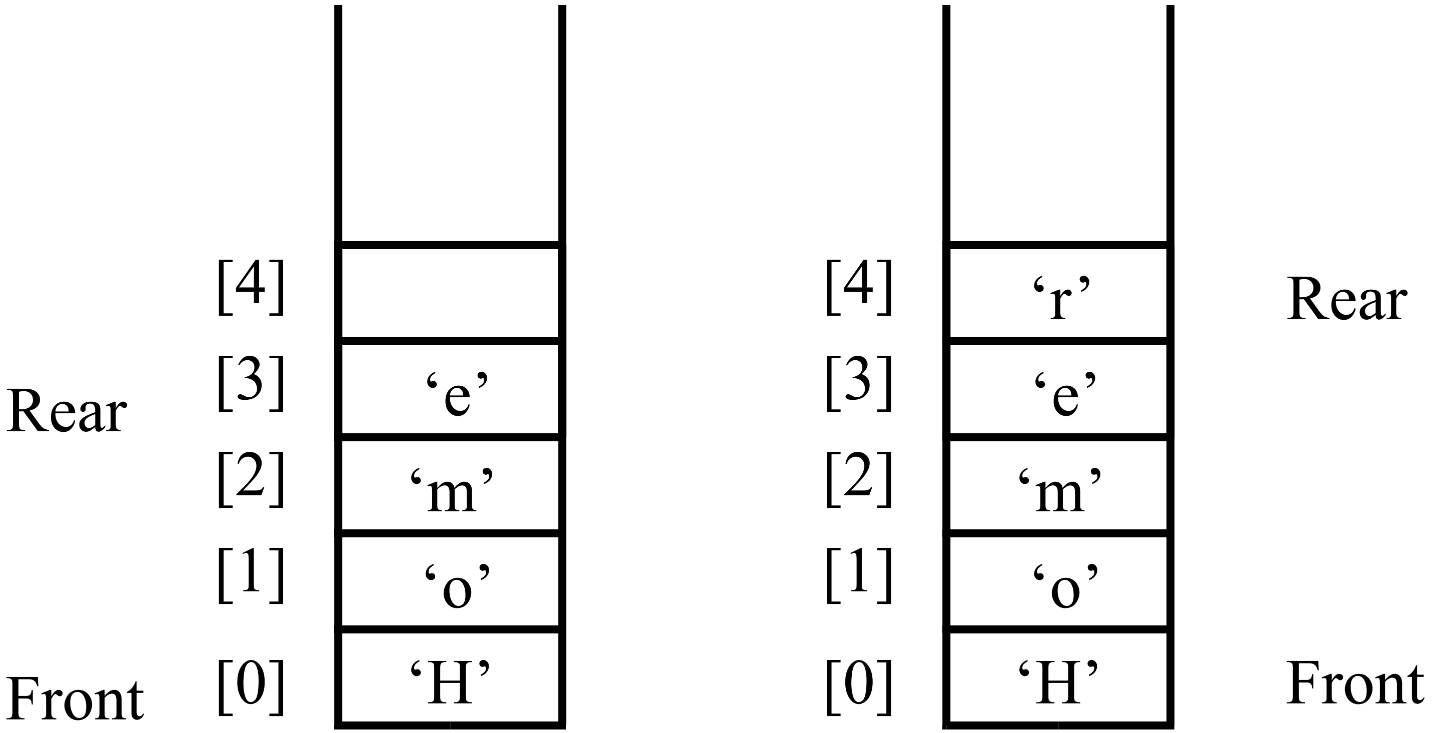
Limits of Enqueue and Dequeue

- What happens when **Enqueue** is invoked when the queue is **full**?
- What happens when **Dequeue** is invoked when the queue is **empty**?
- Same options as with **Stack ADT**
  - **Option #1 – Client is responsible**
  - **Option #2 – Container is responsible**

Queue ADT - Implementations

- **Arrays**
  - Fixed front
  - Floating front
  - Circular
- **Linked Lists**
  - Singly linked
  - Double links

Array - Fixed Front: Enqueue



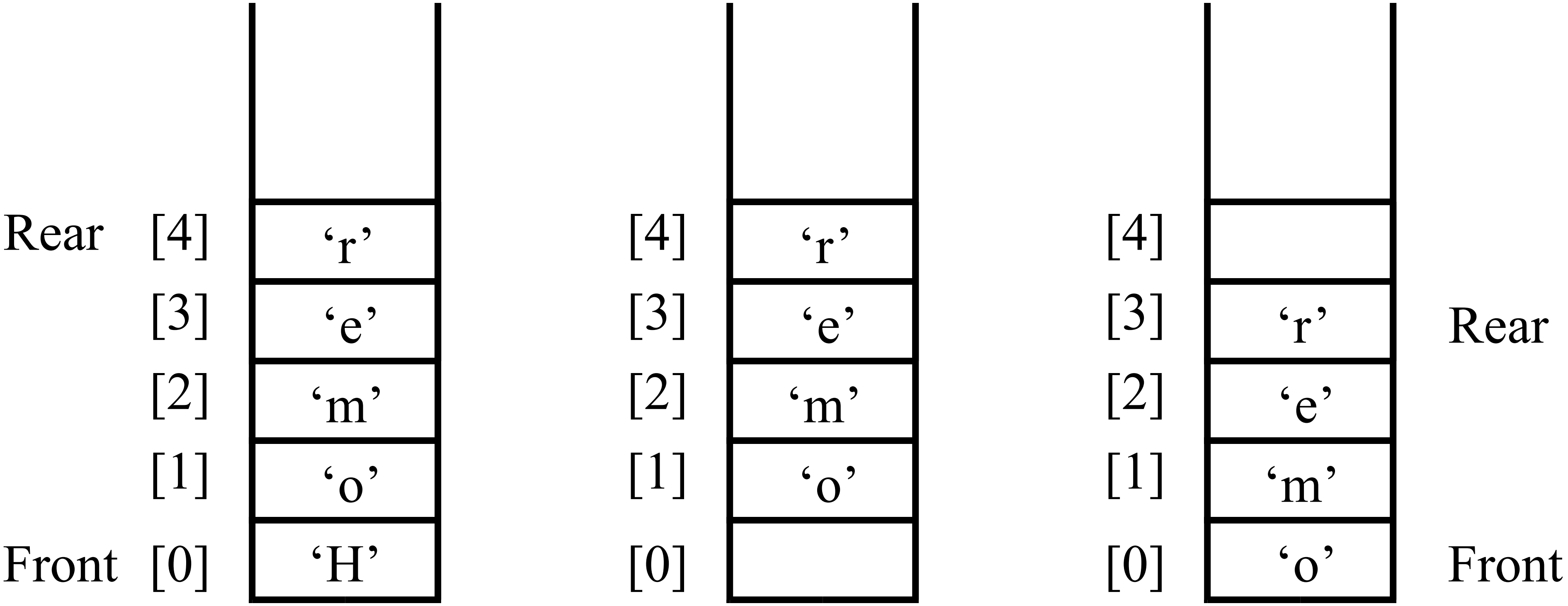
[4]
[3] ‘e’
[2] ‘m’
[1] ‘o’
[0] ‘H’

Rear
Front

[4] ‘r’
[3] ‘e’
[2] ‘m’
[1] ‘o’
[0] ‘H’

Rear
Front

Array - Fixed Front: Dequeue

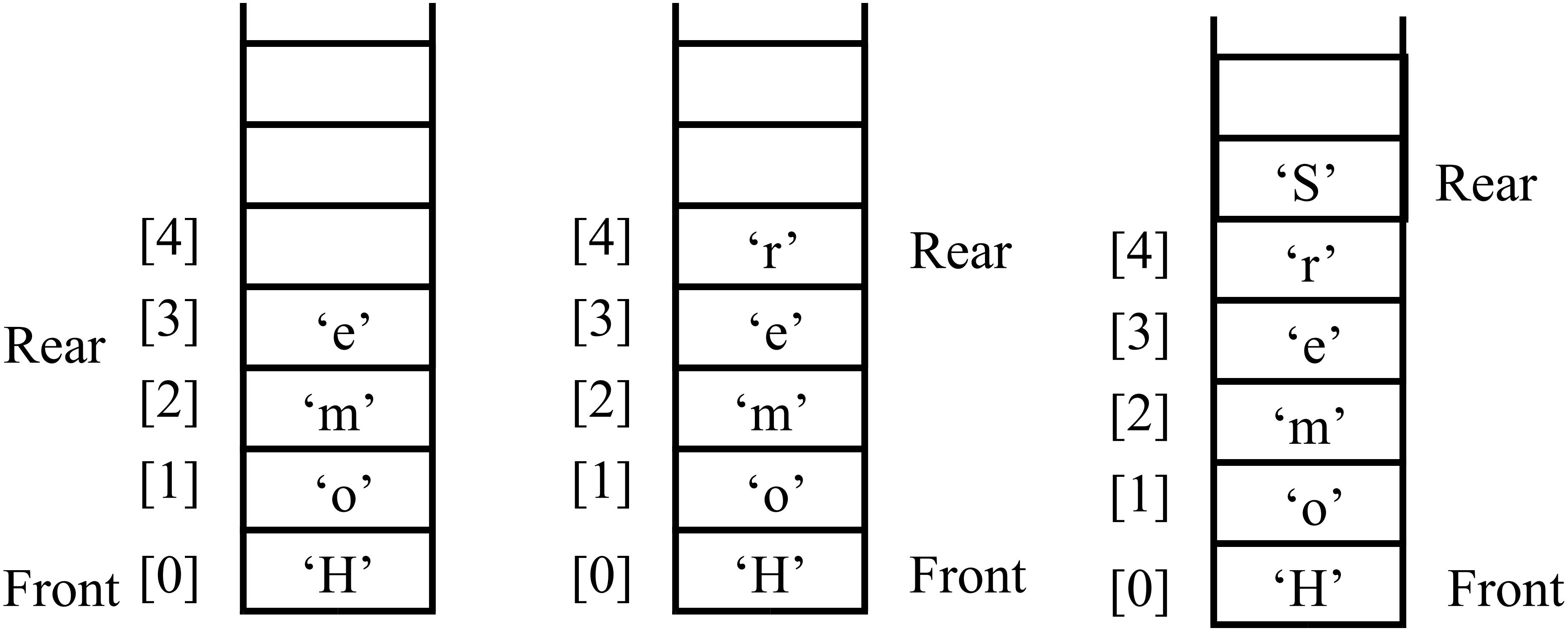


Rear	[4]	‘r’	[4]	‘r’	[4]		
	[3]	‘e’	[3]	‘e’	[3]	‘r’	Rear
	[2]	‘m’	[2]	‘m’	[2]	‘e’	
	[1]	‘o’	[1]	‘o’	[1]	‘m’	
Front	[0]	‘H’	[0]		[0]	‘o’	Front

Array - Fixed Front

- **Advantages**
  - Simple to code
  - Easy to determine status Full/Empty
  - Easy to identify place to add new item
- **Disadvantages**
  - Sliding elements downward takes time
  - Resizing

Array - Floating Front: Enqueue



Rear

Front

[4]

[3] ‘e’

[2] ‘m’

[1] ‘o’

[0] ‘H’

[4] ‘r’ Rear

[3] ‘e’

[2] ‘m’

[1] ‘o’

[0] ‘H’ Front

‘S’ Rear

[4] ‘r’

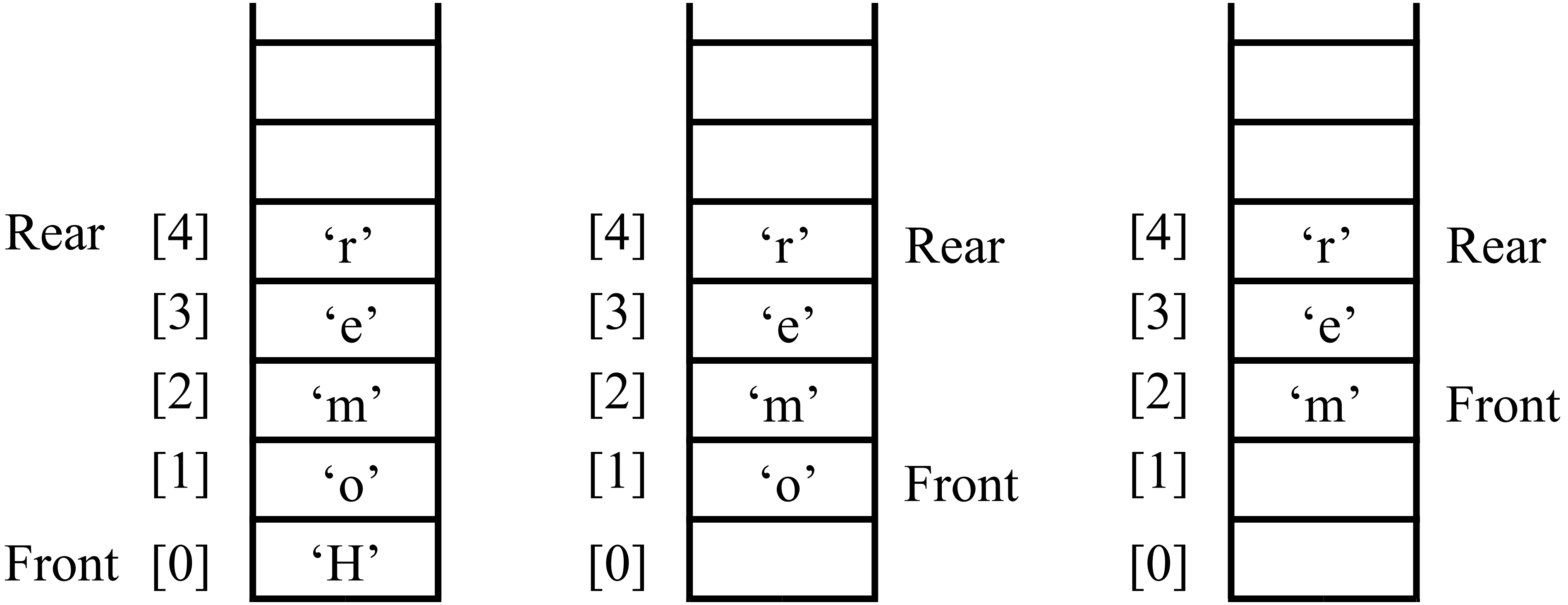
[3] ‘e’

[2] ‘m’

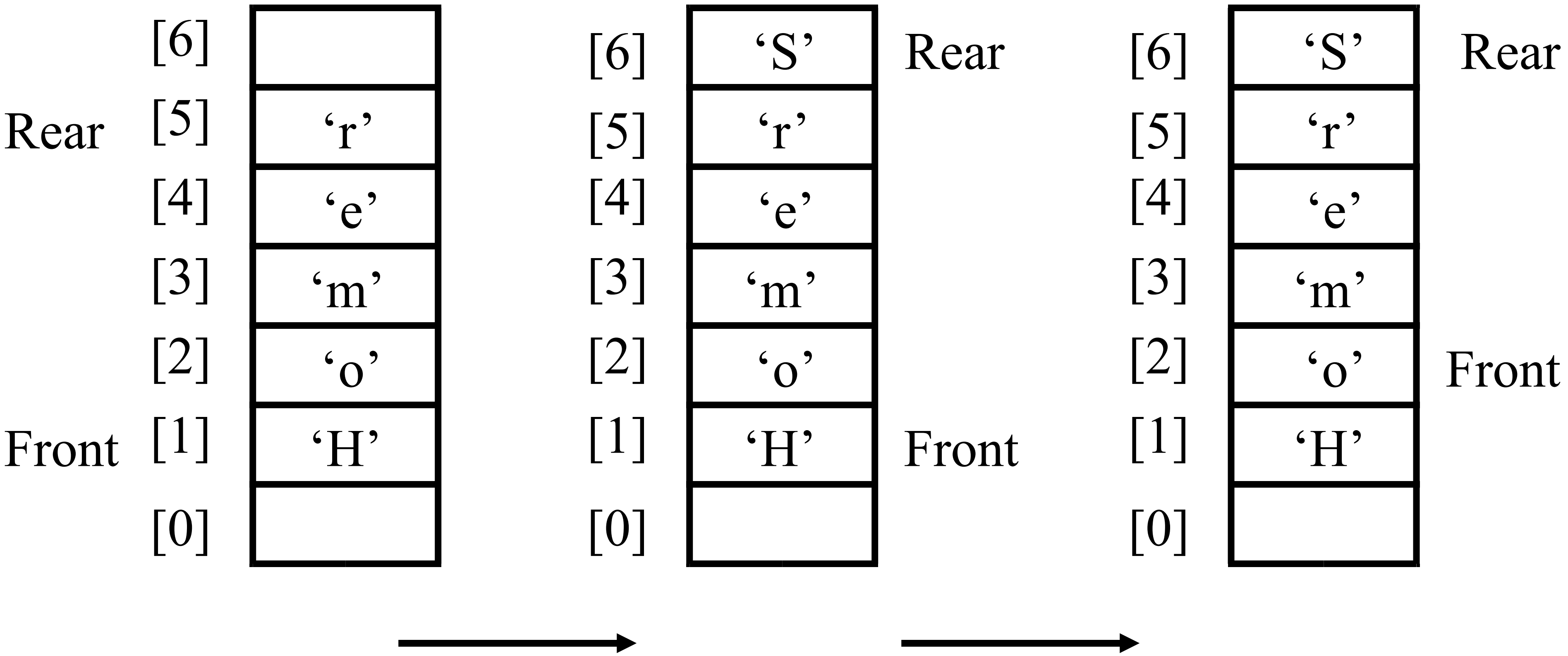
[1] ‘o’

[0] ‘H’ Front

Array - Floating Front: Dequeue



Array - Floating Front: Problem



[6]
Rear [5] ‘r’
[4] ‘e’
[3] ‘m’
[2] ‘o’
Front [1] ‘H’
[0]

[6] ‘S’ Rear
[5] ‘r’
[4] ‘e’
[3] ‘m’
[2] ‘o’
[1] ‘H’ Front
[0]

[6] ‘S’ Rear
[5] ‘r’
[4] ‘e’
[3] ‘m’
[2] ‘o’ Front
[1] ‘H’
[0]

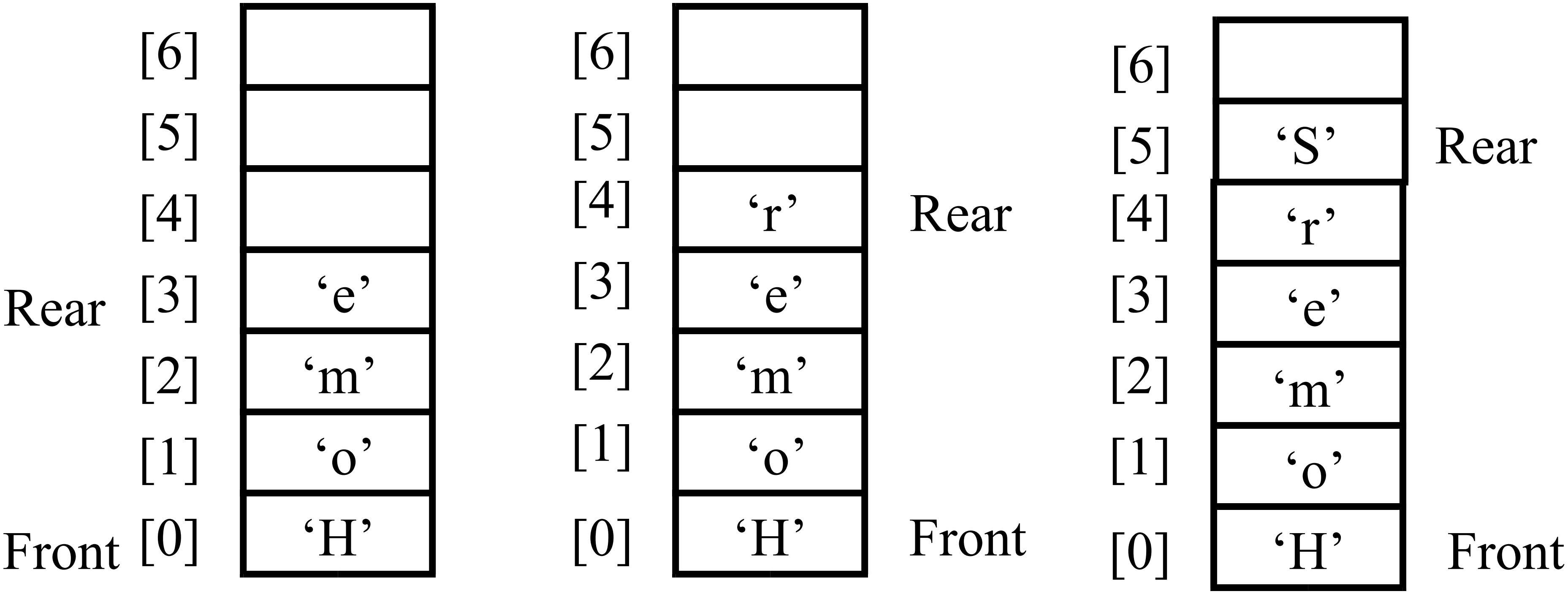
Enqueue ‘S’

Enqueue ‘i’ ???

Array - Floating Front

- **Advantages**
  - Simple to code
  - Easy to determine status Full/Empty
- **Disadvantages**
  - Running out of elements
  - Resizing

Array - Circular: Enqueue

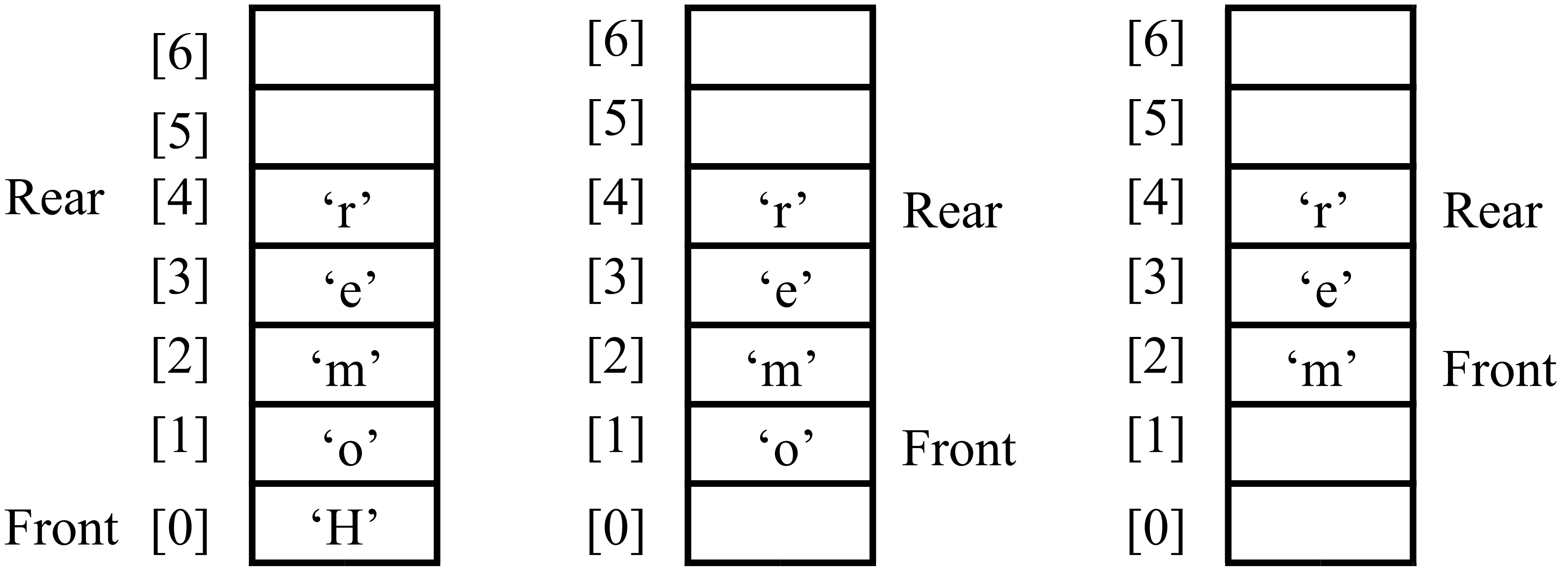


[6]
[5]
[4]
Rear [3] ‘e’
[2] ‘m’
[1] ‘o’
Front [0] ‘H’

[6]
[5]
[4] ‘r’ Rear
[3] ‘e’
[2] ‘m’
[1] ‘o’
[0] ‘H’ Front

[6]
[5] ‘S’ Rear
[4] ‘r’
[3] ‘e’
[2] ‘m’
[1] ‘o’
[0] ‘H’ Front

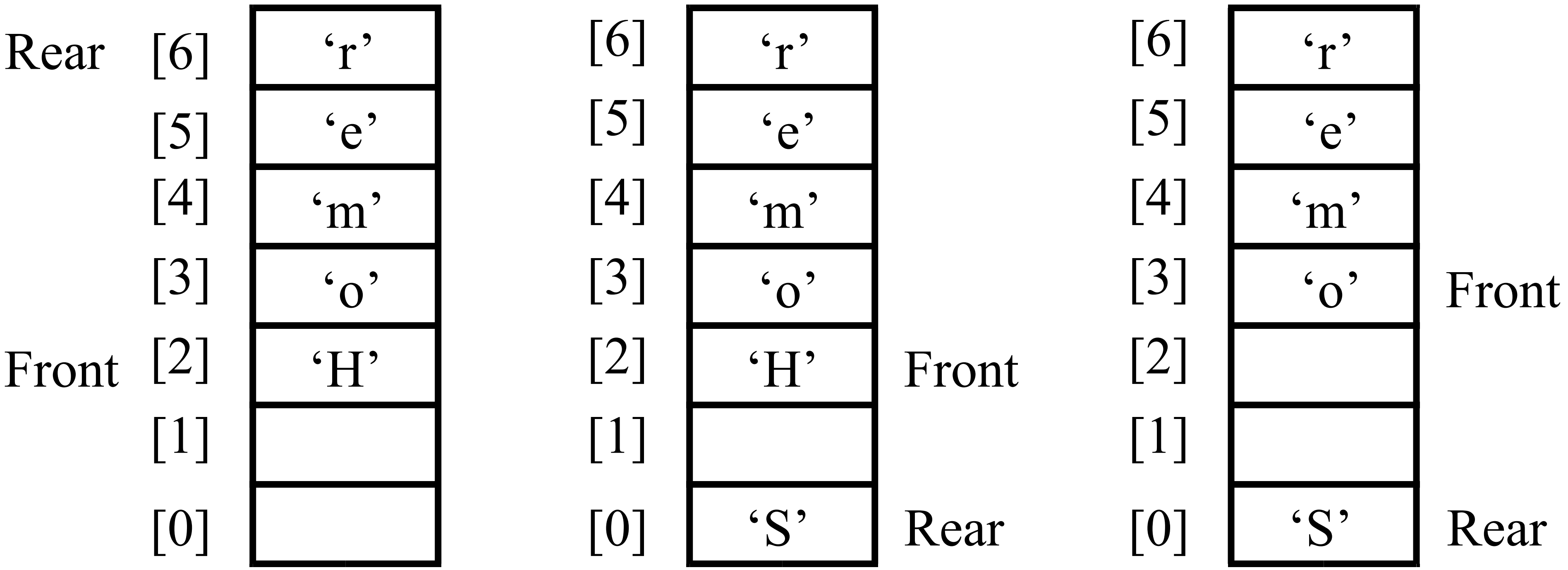
Array - Circular: Dequeue



	[6]		[6]			[6]		
	[5]		[5]			[5]		
Rear	[4]	‘r’	[4]	‘r’	Rear	[4]	‘r’	Rear
	[3]	‘e’	[3]	‘e’		[3]	‘e’	
	[2]	‘m’	[2]	‘m’		[2]	‘m’	Front
	[1]	‘o’	[1]	‘o’	Front	[1]		
Front	[0]	‘H’	[0]			[0]		

UAHuntsville

Array - Circular: Wrap Around



Rear	[6]	‘r’	[6]	‘r’		[6]	‘r’	
	[5]	‘e’	[5]	‘e’		[5]	‘e’	
	[4]	‘m’	[4]	‘m’		[4]	‘m’	
	[3]	‘o’	[3]	‘o’		[3]	‘o’	Front
Front	[2]	‘H’	[2]	‘H’	Front	[2]		
	[1]		[1]			[1]		
	[0]		[0]	‘S’	Rear	[0]	‘S’	Rear

Array - Circular

- **Advantages**
  - Can utilize the entire array
  - Speed
- **Disadvantages**
  - More complex implementation
  - Resizing

```
//*******  queue.h header **********

typedef char ItemType;

class Queue
{
  private:
    ItemType* items;                    // Queue items
    int maxQue;                         // Max number of queue items + 1
    int front;                          // Index of element before the front of queue
    int rear;                           // Index of rear element of queue

  public:
    Queue();                            // Constructor
    Queue(int max);                     // Parameterized constructor

    ~Queue();                           // Destructor

    void MakeEmpty();                   // Initialize queue to empty
    bool IsEmpty() const;               // Determine if queue is empty
    bool IsFull() const;                // Determine if queue is full
    void Enqueue(ItemType newItem);     // Add item to rear of queue
    void Dequeue(ItemType& item);       // Remove item from front of queue
};
```

```
//*******  queue.cpp header Circular Queue Implementation **********
#include “queue.h”

Queue::Queue()                          // Constructor
{
  maxQue = 501;                         // Want a queue which stores 500 elements
  front = maxQue - 1;                   // front == rear implies Empty
  rear = maxQue -1;
  items = new ItemType[maxQue];         // Allocate array for queue
} // End Queue::Queue()


Queue::Queue(int max)                   // Parameterized constructor
{
  maxQue = max + 1;                     // Want a queue which stores max elements
  front = maxQue - 1;                   // front == rear implies Empty
  rear = maxQue -1;
  items = new ItemType[maxQue];         // Allocate array for queue
} // End Queue::Queue(…)


Queue::~Queue()                         // Destructor
{
  delete [] items;                      // Deallocate the queue array
} // End Queue::~Queue()


void Queue::MakeEmpty()                 // Initialize queue to empty
{
  front = maxQue - 1;
  rear = maxQue -1;
} // Queue::MakeEmpty()
```

```
//*******  queue.cpp continued above **********

bool Queue::IsEmpty() const             // Determine if queue is empty
{
  return (front == rear);
} // End Queue::IsEmpty()


bool Queue::IsFull() const              // Determine if queue is full
{
  return ((rear + 1) % maxQue == front);
} // End Queue::IsFull()


void Queue::Enqueue(ItemType newItem)   // Add item to rear of queue
{                                       // Precondition:  queue NOT full
  rear = (rear+1) % maxQue;
  items[rear] = newItem;
} // End Queue::Enqueue(…)


void Queue::Dequeue(ItemType& item)     // Remove item from front of queue
{                                       // Precondition:  queue NOT empty
  front = (front + 1) % maxQue;
  item = items[front];
} // End Queue::Dequque(…)
```

Queue ADT

Array Implementation with Exception Handling

```
// Header File QueType.h
class FullQueue
{ /* No code here */ };
class EmptyQueue
{ /* No code here */ };

typedef char ItemType;

class QueType
{
  private:
    int front;
    int rear;
    ItemType* items;
    int maxQue;
  public:
    QueType();
    // Class constructor.
    // Because there is a default constructor, the precondition
    // that the queue has been initialized is omitted.
    QueType(int max);
    // Parameterized class constructor.
    ~QueType();
    // Class destructor.
    void MakeEmpty();
    // Function: Initializes the queue to an empty state.
    // Post: Queue is empty.
    bool IsEmpty() const;
    // Function: Determines whether the queue is empty.
    // Post: Function value = (queue is empty)
    bool IsFull() const;
    // Function: Determines whether the queue is full.
    // Post: Function value = (queue is full)
    void Enqueue(ItemType newItem);
    // Function: Adds newItem to the rear of the queue.
    // Post: If (queue is full) FullQueue exception is thrown
    //       else newItem is at rear of queue.
    void Dequeue(ItemType& item);
    // Function: Removes front item from the queue and returns it in item.
    // Post: If (queue is empty) EmptyQueue exception is thrown
    //       and item is undefined
    //       else front element has been removed from queue and
    //       item is a copy of removed element.
};
```

```
// Header QueType.cpp

#include "QueType.h"

QueType::QueType(int max)
// Parameterized class constructor
// Post: maxQue, front, and rear have been initialized.
//       The array to hold the queue elements has been dynamically
//       allocated.
{
  maxQue = max + 1;
  front = maxQue - 1;
  rear = maxQue - 1;
  items = new ItemType[maxQue];
}
QueType::QueType()          // Default class constructor
// Post: maxQue, front, and rear have been initialized.
//       The array to hold the queue elements has been dynamically
//       allocated.
{
  maxQue = 501;
  front = maxQue - 1;
  rear = maxQue - 1;
  items = new ItemType[maxQue];
}
QueType::~QueType()         // Class destructor
{
  delete [] items;
}

void QueType::MakeEmpty()
// Post: front and rear have been reset to the empty state.
{
  front = maxQue - 1;
  rear = maxQue - 1;
}

bool QueType::IsEmpty() const
// Returns true if the queue is empty; false otherwise.
{
  return (rear == front);
}
```

```
// QueType.cpp continued

bool QueType::IsFull() const
// Returns true if the queue is full; false otherwise.
{
  return ((rear + 1) % maxQue == front);
}

void QueType::Enqueue(ItemType newItem)
// Post: If (queue is not full) newItem is at the rear of the queue;
//       otherwise a FullQueue exception is thrown.
{
  if (IsFull())
    throw FullQueue();
  else
  {
    rear = (rear + 1) % maxQue;
    items[rear] = newItem;
  }
}

void QueType::Dequeue(ItemType& item)
// Post: If (queue is not empty) the front of the queue has been
//       removed and a copy returned in item;
//       othersiwe a EmptyQueue exception has been thrown.
{
  if (IsEmpty())
    throw EmptyQueue();
  else
  {
    front = (front + 1) % maxQue;
    item = items[front];
  }
}
```

CountedQueue ADT

Derived Class of QueType

```
// Header: CountedQueType.h

typedef char ItemType;

#include "QueType.h”

class CountedQueType : public QueType
{
  private:
    int length;

  public:
    CountedQueType(int max);
    void Enqueue(ItemType newItem);
    void Dequeue(ItemType& item);
    int LengthIs() const;
    // Returns the number of items on the queue.
};
```

```
// Header: CountedQueType.cpp

#include "CountedQueType.h"

void CountedQueType::Enqueue(ItemType newItem)
{

  try
  {
    QueType::Enqueue(newItem);
    length++;
  }
  catch(FullQueue)
  {
    throw FullQueue();
  }
}

void CountedQueType::Dequeue(ItemType& item)
{
  try
  {
    QueType::Dequeue(item);
    length--;
  }
  catch(EmptyQueue)
  {
    throw EmptyQueue();
  }
}

int CountedQueType::LengthIs() const
{
  return length;
}

CountedQueType::CountedQueType(int max) : QueType(max)
{
  length = 0;
}
```

```
// Test driver CQueDr.cpp
#include <iostream>
#include <fstream>


#include "CountedQueType.h"

using namespace std;

int main()
{

  ifstream inFile;       // file containing operations
  ofstream outFile;      // file containing output
  string inFileName;     // input file external name
  string outFileName;    // output file external name
  string outputLabel;
  string command;        // operation to be executed

  ItemType item;
  CountedQueType queue(5);
  int numCommands;


  // Prompt for file names, read file names, and prepare files
  cout << "Enter name of input command file; press return." << endl;
  cin  >> inFileName;
  inFile.open(inFileName.c_str());

  cout << "Enter name of output file; press return." << endl;
  cin  >> outFileName;
  outFile.open(outFileName.c_str());

  cout << "Enter name of test run; press return." << endl;
  cin  >> outputLabel;
  outFile << outputLabel << endl;

  inFile >> command;
// Test driver CQueDr.cpp continued
  numCommands = 0;
  while (command != "Quit")
  {
    try
    {
      if (command == "Enqueue")
      {
        inFile >> item;
        queue.Enqueue(item);
        outFile << item << " is enqueued." << endl;
      }
      else if (command == "Dequeue")
      {
        queue.Dequeue(item);
        outFile<< item  << " is dequeued. " << endl;
      }
      else if (command == "IsEmpty")
        if (queue.IsEmpty())
          outFile << "Queue is empty." << endl;
        else
          outFile << "Queue is not empty." << endl;

      else if (command == "IsFull")
        if (queue.IsFull())
          outFile << "Queue is full." << endl;
        else outFile << "Queue is not full."  << endl;
      else if (command == "LengthIs")
        outFile << "Length is " << queue.LengthIs() << endl;
    }
    catch (FullQueue)
    {
      outFile << "FullQueue exception thrown." << endl;
    }

    catch (EmptyQueue)
    {
      outFile << "EmtpyQueue exception thrown." << endl;
    }
    numCommands++;
    cout <<  " Command number " << numCommands << " completed."
         << endl;
    inFile >> command;
  }
  cout << "Testing completed."  << endl;
  inFile.close();
  outFile.close();
  return 0;
} // End main()
```

Sample Input

```
IsEmpty
LengthIs
Enqueue E
Enqueue G
Enqueue F
Enqueue I
LengthIs
Dequeue
Dequeue
Dequeue
Dequeue
Enqueue B
Enqueue C
Enqueue C
Enqueue D
Dequeue
Dequeue
Dequeue
Dequeue
Enqueue E
Dequeue
Enqueue C
Enqueue G
Dequeue
Dequeue
LengthIs
IsEmpty
Enqueue B
IsEmpty
Dequeue
IsEmpty
Enqueue B
Enqueue C
Enqueue C
Enqueue D
LengthIs
IsFull
Enqueue A
IsFull
LengthIs
Enqueue B
LengthIs
Dequeue
Enqueue B
Dequeue
Dequeue
Dequeue
Dequeue
Dequeue
LengthIs
Dequeue
LengthIs
Quit
```

Sample Output

```
CountedQueueTestRun
Queue is empty.
Length is 0
E is enqueued.
G is enqueued.
F is enqueued.
I is enqueued.
Length is 4
E is dequeued.
G is dequeued.
F is dequeued.
I is dequeued.
B is enqueued.
C is enqueued.
C is enqueued.
D is enqueued.
B is dequeued.
C is dequeued.
C is dequeued.
D is dequeued.
E is enqueued.
E is dequeued.
C is enqueued.
G is enqueued.
C is dequeued.
G is dequeued.
Length is 0
Queue is empty.
B is enqueued.
Queue is not empty.
B is dequeued.
Queue is empty.
B is enqueued.
C is enqueued.
C is enqueued.
D is enqueued.
Length is 4
Queue is not full.
A is enqueued.
Queue is full.
Length is 5
FullQueue exception thrown.
Length is 5
B is dequeued.
B is enqueued.
C is dequeued.
C is dequeued.
D is dequeued.
A is dequeued.
B is dequeued.
Length is 0
EmtpyQueue exception thrown.
Length is 0
```

Queue ADT

Linked List Implementation with Exception Handling

```
// Header file for Queue ADT:  QueType.h
class FullQueue
{};

class EmptyQueue
{};

typedef int ItemType;

struct NodeType
{
  ItemType info;
  NodeType* next;
};


class QueType
{
  private:
    NodeType* front;
    NodeType* rear;

  public:
    QueType();                      // Class constructor.

    ~QueType();                     // Class destructor.

    void MakeEmpty();               // Function: Initializes the queue to an empty state.

    bool IsEmpty() const;           // Function: Determines whether the queue is empty.

    bool IsFull() const;            // Function: Determines whether the queue is full.

    void Enqueue(ItemType newItem); // Function: Adds newItem to the rear of the queue.

    void Dequeue(ItemType& item);    // Function: Removes front item from the queue and returns it in item.
};
```

```
// QueType.h continued


QueType::QueType()          // Class constructor.
// Post:  front and rear are set to NULL.
{
  front = NULL;
  rear = NULL;
}


void QueType::MakeEmpty()
// Post: Queue is empty; all elements have been deallocated.
{
  NodeType* tempPtr;

  while (front != NULL)
  {
    tempPtr = front;
    front = front->next;
    delete tempPtr;
  }
  rear = NULL;
}



QueType::~QueType()     // Class destructor.
{
  MakeEmpty();
}


bool QueType::IsEmpty() const
// Returns true if there are no elements on the queue; false otherwise.
{
  return (front == NULL);
}
```

```
// QueType.h continued

bool QueType::IsFull() const
// Returns true if there is no room for another ItemType
//  on the free store; false otherwise.
{
  NodeType* location;
  try
  {
    location = new NodeType;
    delete location;
    return false;
  }
  catch(std::bad_alloc)
  {
    return true;
  }
}

void QueType::Enqueue(ItemType newItem)
// Adds newItem to the rear of the queue.
// Pre:  Queue has been initialized.
// Post: If (queue is not full) newItem is at the rear of the queue;
//       otherwise a FullQueue exception is thrown.

{
  if (IsFull())
    throw FullQueue();
  else
  {
    NodeType* newNode;

    newNode = new NodeType;
    newNode->info = newItem;
    newNode->next = NULL;
    if (rear == NULL)
      front = newNode;
    else
      rear->next = newNode;
    rear = newNode;
  }
}
```

```
// QueType.h continued


void QueType::Dequeue(ItemType& item)
// Removes front item from the queue and returns it in item.
// Pre:  Queue has been initialized and is not empty.
// Post: If (queue is not empty) the front of the queue has been
//       removed and a copy returned in item;
//       otherwise an EmptyQueue exception has been thrown.
{
  if (IsEmpty())
    throw EmptyQueue();
  else
  {
    NodeType* tempPtr;

    tempPtr = front;
    item = front->info;
    front = front->next;
    if (front == NULL)
      rear = NULL;
    delete tempPtr;
  }
}
```

```
// Test driver QueDr.cpp
#include <iostream>
#include <fstream>

#include "QueType.cpp"

using namespace std;

int main()
{

  ifstream inFile;       // file containing operations
  ofstream outFile;      // file containing output
  string inFileName;     // input file external name
  string outFileName;    // output file external name
  string outputLabel;
  string command;        // operation to be executed

  int item;
  QueType queue;
  int numCommands;


  // Prompt for file names, read file names, and prepare files
  cout << "Enter name of input command file; press return." << endl;
  cin  >> inFileName;
  inFile.open(inFileName.c_str());

  cout << "Enter name of output file; press return." << endl;
  cin  >> outFileName;
  outFile.open(outFileName.c_str());

  cout << "Enter name of test run; press return." << endl;
  cin  >> outputLabel;
  outFile << outputLabel << endl;

  inFile >> command;     // Priming read
```

```
// Test driver continued
  numCommands = 0;
  while (command != "Quit")
  {
    try
    {
      if (command == "Enqueue")
      {
        inFile >> item;
        queue.Enqueue(item);
        outFile << item << " is enqueued." << endl;
      }
      else if (command == "Dequeue")
      {
        queue.Dequeue(item);
        outFile<< item  << " is dequeued. " << endl;
      }
      else if (command == "IsEmpty")
        if (queue.IsEmpty())
          outFile << "Queue is empty." << endl;
        else
          outFile << "Queue is not empty." << endl;

      else if (command == "IsFull")
        if (queue.IsFull())
          outFile << "Queue is full." << endl;
        else outFile << "Queue is not full."  << endl;
    }
    catch (FullQueue)   // FullQueue exception handler
    {
      outFile << "FullQueue exception thrown." << endl;
    }

    catch (EmptyQueue)  // EmptyQueue exception handler
    {
      outFile << "EmtpyQueue exception thrown." << endl;
    }
    numCommands++;
    cout <<  " Command number " << numCommands << " completed."
         << endl;
    inFile >> command;

  };
```

```
  cout << "Testing completed."  << endl;
  inFile.close();
  outFile.close();
  return 0;
} // End main()
```

QueType.in

```
IsEmpty
Enqueue 5
Enqueue 7
Enqueue 6
Enqueue 9
Dequeue
Dequeue
Dequeue
Dequeue
Enqueue 2
Enqueue 3
Enqueue 3
Enqueue 4
Dequeue
Dequeue
Dequeue
Dequeue
Enqueue 5
Dequeue
Enqueue 3
Enqueue 7
Dequeue
Dequeue
IsEmpty
Enqueue 2
IsEmpty
Dequeue
IsEmpty
Dequeue
Enqueue 2
Enqueue 3
Enqueue 3
Enqueue 4
IsFull
Enqueue 1
IsFull
Enqueue 2
Dequeue
Enqueue 2
Dequeue
Dequeue
Dequeue
Dequeue
Dequeue
Dequeue
Quit
```

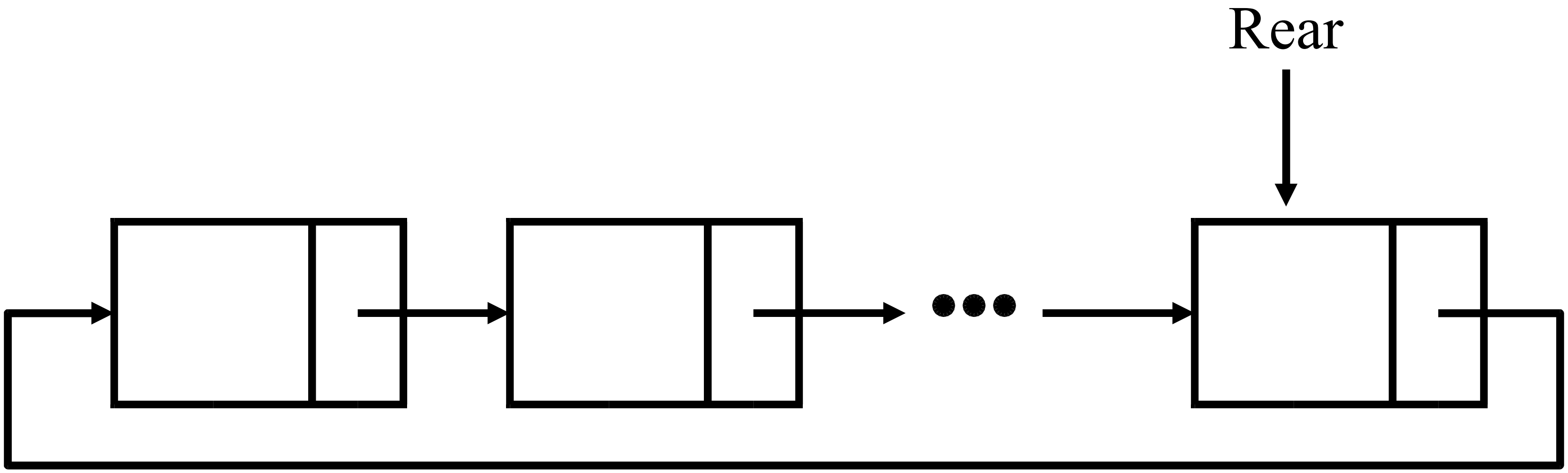
QueType.out

```
LinkedQueTypeTest
Queue is empty.
5 is enqueued.
7 is enqueued.
6 is enqueued.
9 is enqueued.
5 is dequeued.
7 is dequeued.
6 is dequeued.
9 is dequeued.
2 is enqueued.
3 is enqueued.
3 is enqueued.
4 is enqueued.
2 is dequeued.
3 is dequeued.
3 is dequeued.
4 is dequeued.
5 is enqueued.
5 is dequeued.
3 is enqueued.
7 is enqueued.
3 is dequeued.
7 is dequeued.
Queue is empty.
2 is enqueued.
Queue is not empty.
2 is dequeued.
Queue is empty.
EmtpyQueue exception thrown.
2 is enqueued.
3 is enqueued.
3 is enqueued.
4 is enqueued.
Queue is not full.
1 is enqueued.
Queue is not full.
2 is enqueued.
2 is dequeued.
2 is enqueued.
3 is dequeued.
3 is dequeued.
4 is dequeued.
1 is dequeued.
2 is dequeued.
2 is dequeued.
```

Queue ADT

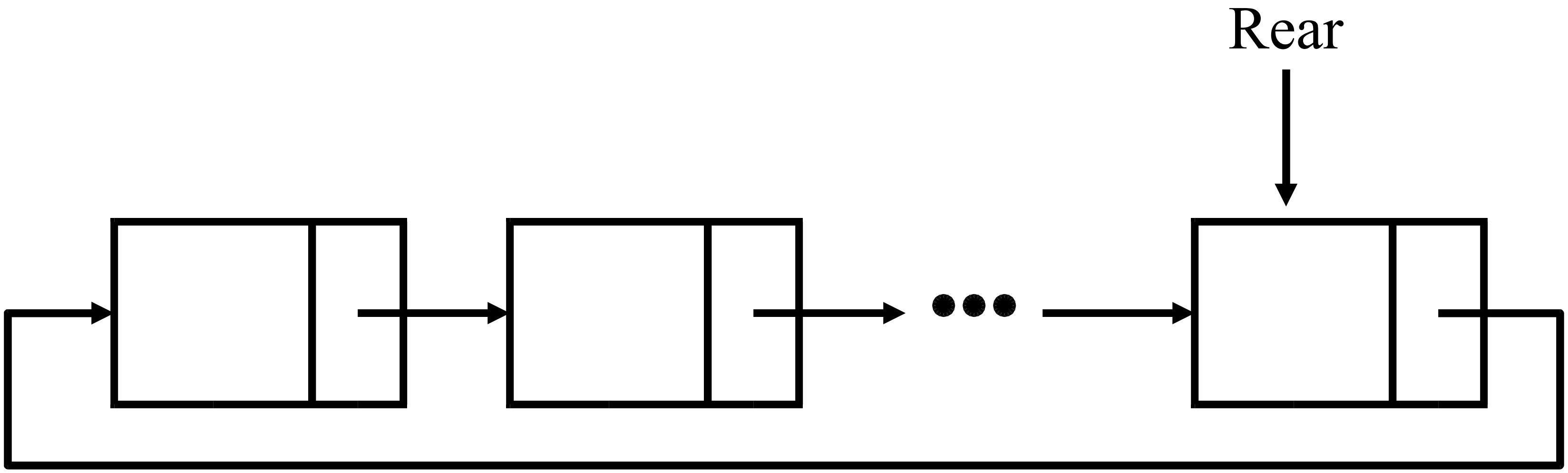
Circular Queue

Circular Linked Queue - 1



Where is the front of this queue?

Circular Linked Queue - 2

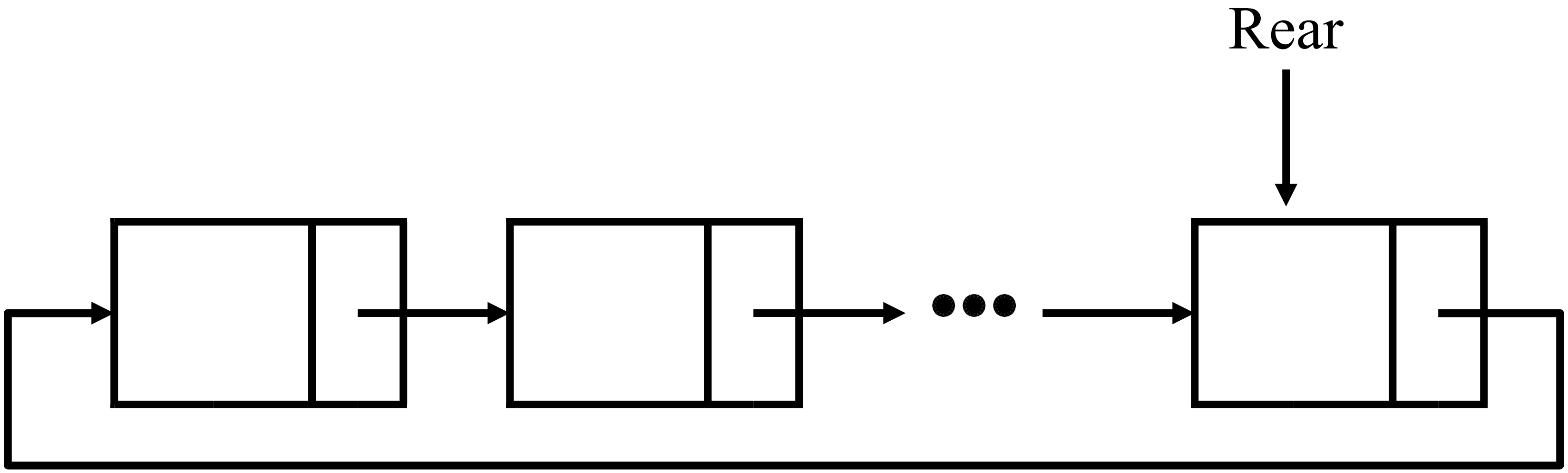


Where is the front of this queue?

Rear->next

How do you know if the queue is empty?

Circular Linked Queue - 3



Where is the front of this queue?

Rear->next

How do you know if the queue is empty?

Rear = NULL

Summary

- **Queues** are **First-In**, **First-Out** containers
- Several ways to implement a **Queue**
  - Arrays (static or dynamic)
    - With Fixed Front, Floating Front, or Circular
  - Linked, dynamically allocated nodes
  - Tradeoffs:
    - Array implementations are memory efficient but difficult to resize
    - Linked node implementation uses more memory per data element but allows size of container to vary based upon amount of data