Lecture 7b
Queue ADT

A First-In-First-Out Data Structure

Lecture Overview

- Queue
  - Introduction
  - Specification
  - Implementations
    - Array Based
    - Linked List Based
  - Application
    - Palindrome checking

What is a Queue

- Real life examples
  - A **queue** for movie tickets, Airline reservation **queue**, etc.

- First item added will be the first item to be removed
  - Has the **F**irst **I**n **F**irst **O**ut (**FIFO**) property

- Major Operations
  - **Enqueue:** Items are added to the **back of the queue**
  - **Dequeue:** Items are removed from the **front of the queue**
  - **Get Front:** Take a look at the first item

Queue: Illustration

A **queue** of 3 persons

Enqueue a new person to the **back of the queue**

Dequeue a person from the **front of the queue**

Queue ADT: C++ Specification

```
template<typename T>
class Queue {
public:
  Queue();

  bool isEmpty() const;
  int size() const;

  void enqueue(const T& newItem);
  void dequeue();
  void getFront(T& queueTop) const;

private:
  // Implementation dependant
  // See subsequent implementation slides
};
```

Queue ADT: Design Considerations

- How about the common choices?
  - Efficiency of **array based implementation**
    - Removing item at the head is the worst case
    - Adding item at the back is the best case
  - Efficiency of **singly linked list implementation**
    - Removing item at the head is the best case
    - Adding item at the back is the worst case
- Is it possible to have both efficient ***`enqueue()`*** and ***`dequeue()`*** operations?

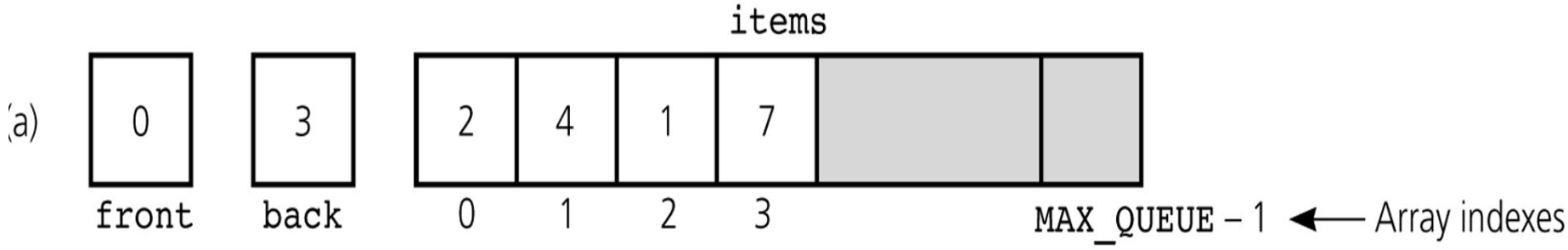
Queue ADT using Array

Array Implementation Issues

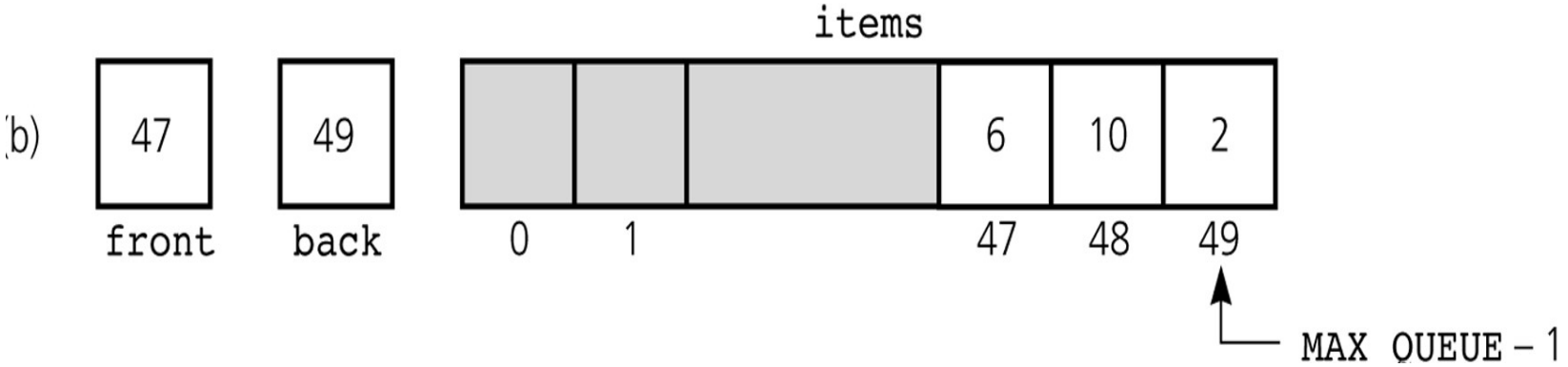
- Removing item from the front is inefficient
  - **Shifting items is too expensive**
- Basic Idea
  - The reason for shifting is
    - Front is assumed to be at index 0
  - Instead of shifting items
    - **Shift the front index**
- So, we have two indices
  - **Front:** index of the queue front
  - **Back:** index of the queue back

Incorrect Implementation

- At the beginning, with 4 items queued



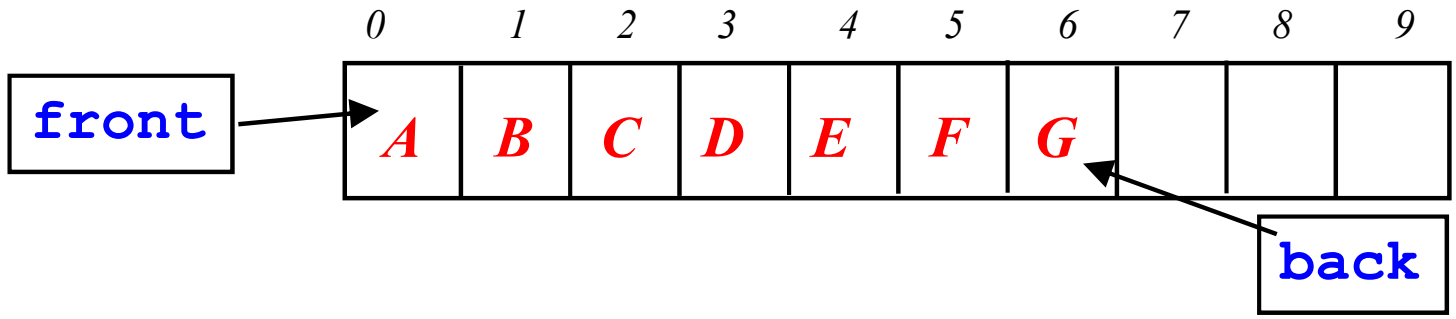
- After many queue operations



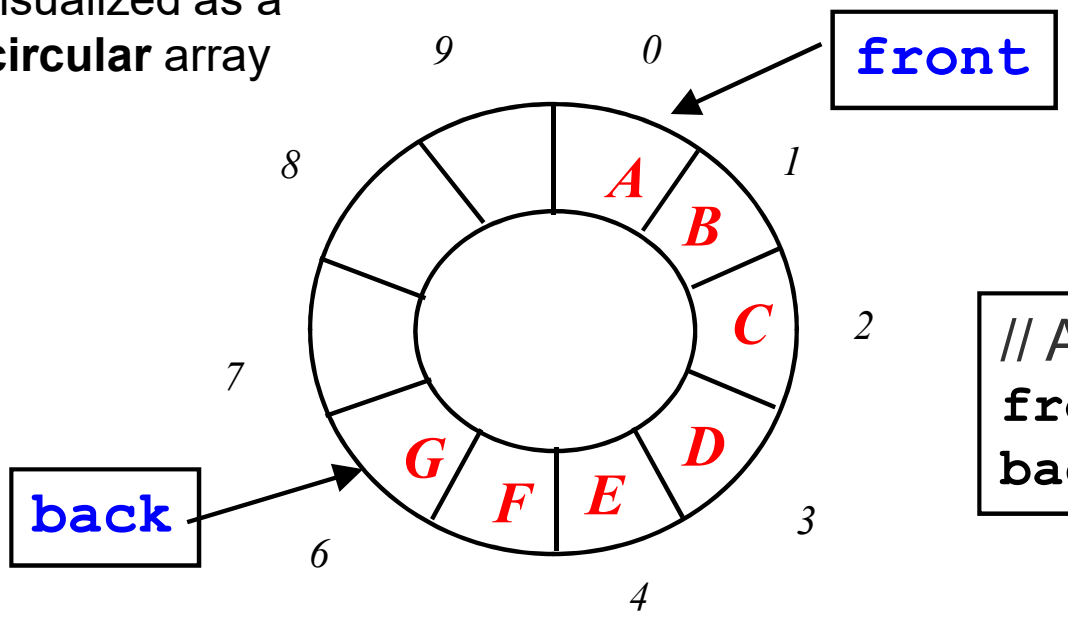
- The front index will drift to the right,
  - Most array locations empty and unusable

Circular Array

- Allow both indices to "wrap" back to index 0 when they reached the end of array
  - Effectively making the array "circular"



Visualized as a **circular** array




```
// Advancing front and back  index
front = (front+1) % maxsize;
back = (back+1) % maxsize;
```

Queue ADT (Array): C++ Specification

```
const int MAX_QUEUE = 50; // here is the main problem of array
                          // our queue cannot be that large

template<typename T>
class Queue {
public:
  Queue();
  bool isEmpty() const;
  int size() const;

  void enqueue(const T& newItem);
  void dequeue();
  void getFront(T& queueFront) const;

private:
  T _items[MAX_QUEUE];
  int _front, _back, _count;
};
```

QueueA.h

Implement Queue ADT (Array): 1/2

```
#include <string>
using namespace std;
const int MAX_QUEUE = 50;
template<typename T>
class QueueA {
public:
  QueueA() {
    _front = 0;
    _back = MAX_QUEUE-1;
    _count = 0;
  }
  bool isEmpty() const { return _count == 0; }
  int size() const { return _count; }
  void enqueue(const T& newItem) {
    if (_count == MAX_QUEUE)
      throw string("Queue is full");
    else {
      _back = (_back+1) % MAX_QUEUE;
      _items[_back] = newItem;
      ++_count;
    }
  }
```

QueueA.h, expanded

Implement Queue ADT (Array): 2/2

```
void dequeue() {
  if (isEmpty())
    throw string("Queue is empty");
  else {
    _front = (_front+1) % MAX_QUEUE;
    --_count;
  }
}

void getFront(T& queueFront) const {
  if (isEmpty())
    throw string("Queue is empty");
  else
    queueFront = _items[_front];
}
```

QueueA.h, expanded

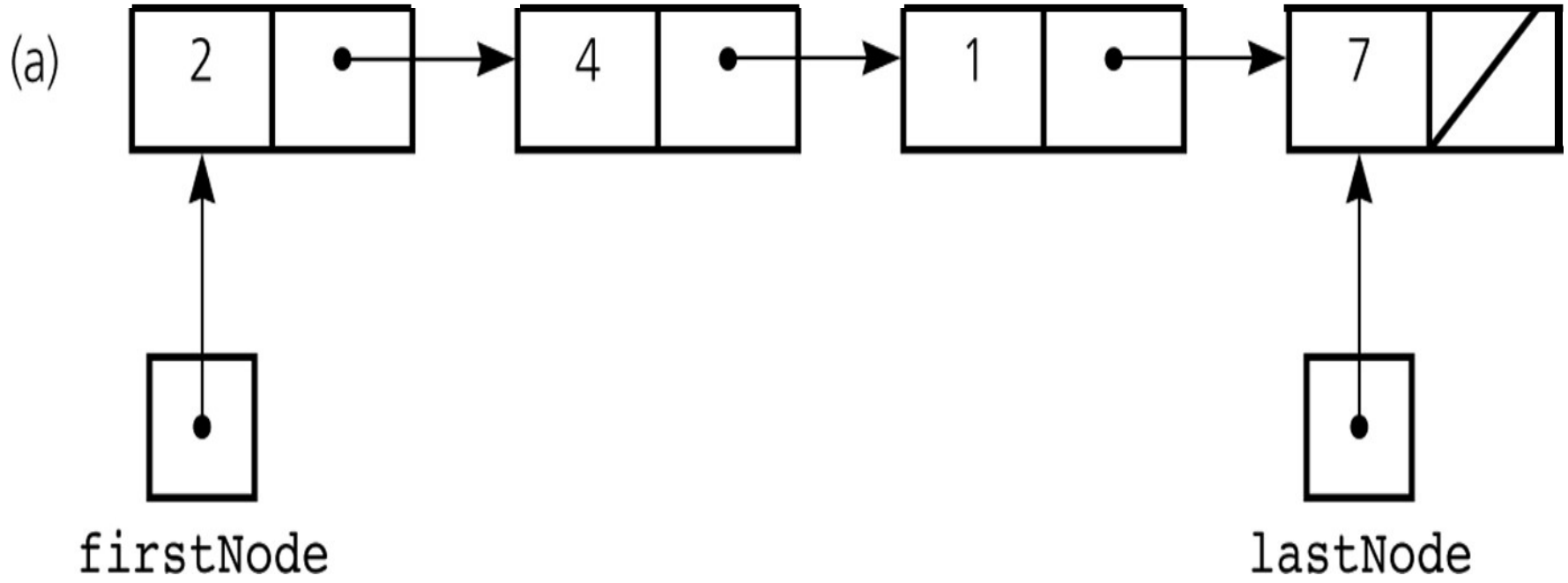
Queue ADT using Modified Linked List

Conceptual Discussion Only

Improving the Singly Linked List

- Singly linked list performs badly for ***`enqueue()`***
  - Need to traverse all the way to the last node
  - Takes longer time as the queue grows
- How to avoid the traversal to the last node?
  - Easy: Just need to "know" where the last node is all the time
- Solutions
  - Keep an additional pointer to the last node, OR
  - Circular linked last with a tail pointer

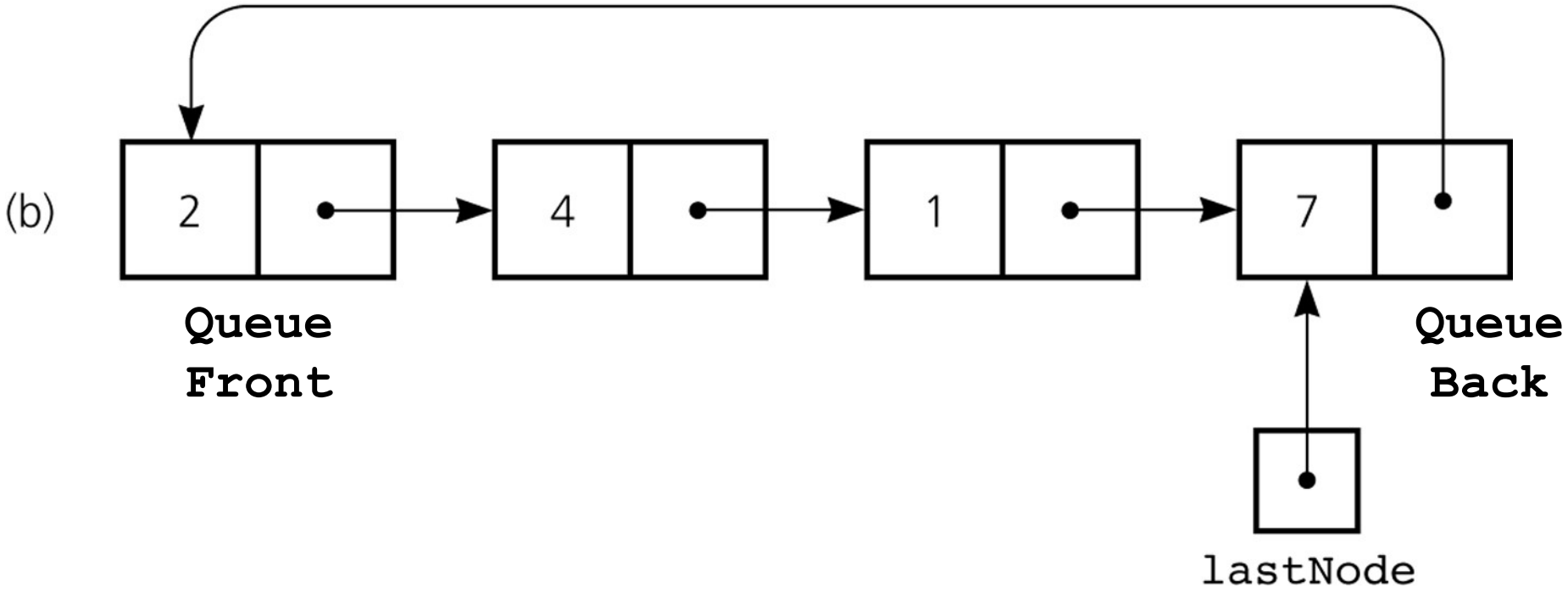
Linked List: with “head” and “tail”



Queue Front

Queue Back

Circular Linked List



- Only keep track of `lastNode` (tail) pointer
  - `firstNode` pointer can be set when needed
    - `firstNode = lastNode->next;`
- Will use circular linked list for subsequent discussion

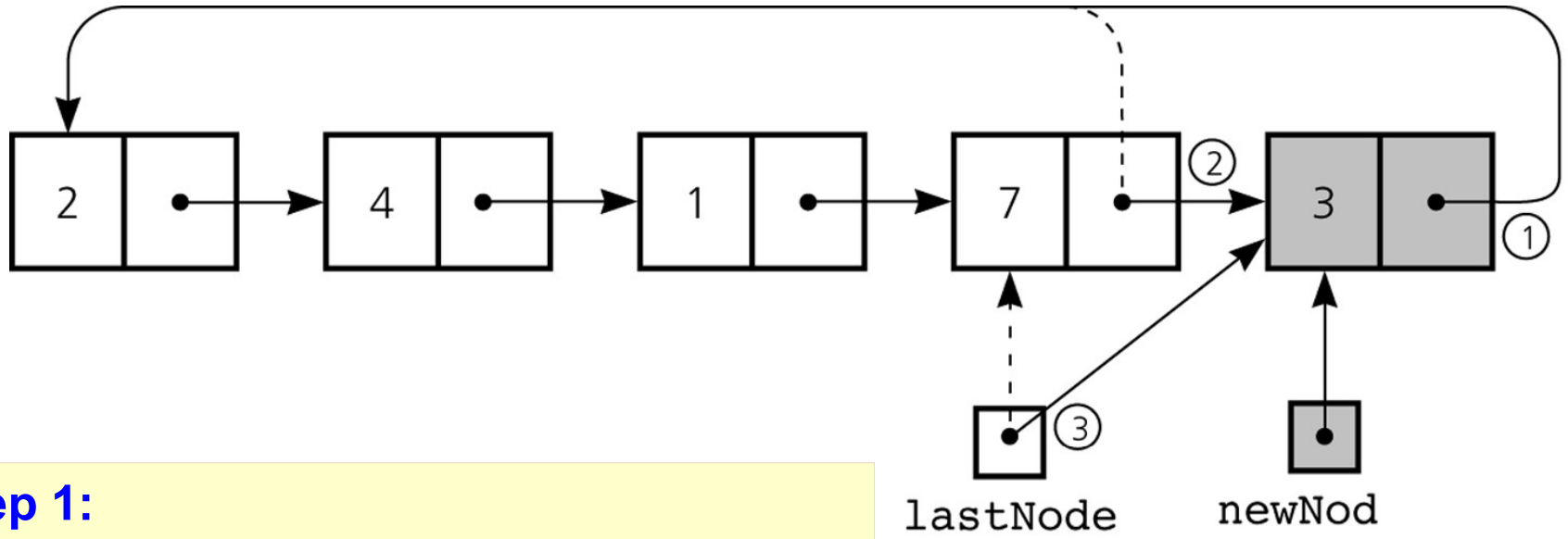
Queue ADT: C++ Specification

```
template<typename T>
class Queue {
public:
  Queue();
  ~Queue();

  bool isEmpty() const;
  int size() const;

  void enqueue(const T& newItem);
  void dequeue();
  void getFront(T& queueTop) const;

private:
  struct QueueNode {
    T item;
    QueueNode *next;
  };
  int _size;
  QueueNode *_lastNode;
};
```

Just like a **`ListNode`** structure, yes we can use inheritance but ListLL.h in Lecture6 is an SLL

`QueueLL.h`

Insertion: Non-Empty Queue



Step 1:

```
newNode = new QueueNode;
newNode->next = lastNode->next;
newNode->item = 3;
```

This value is just an example only

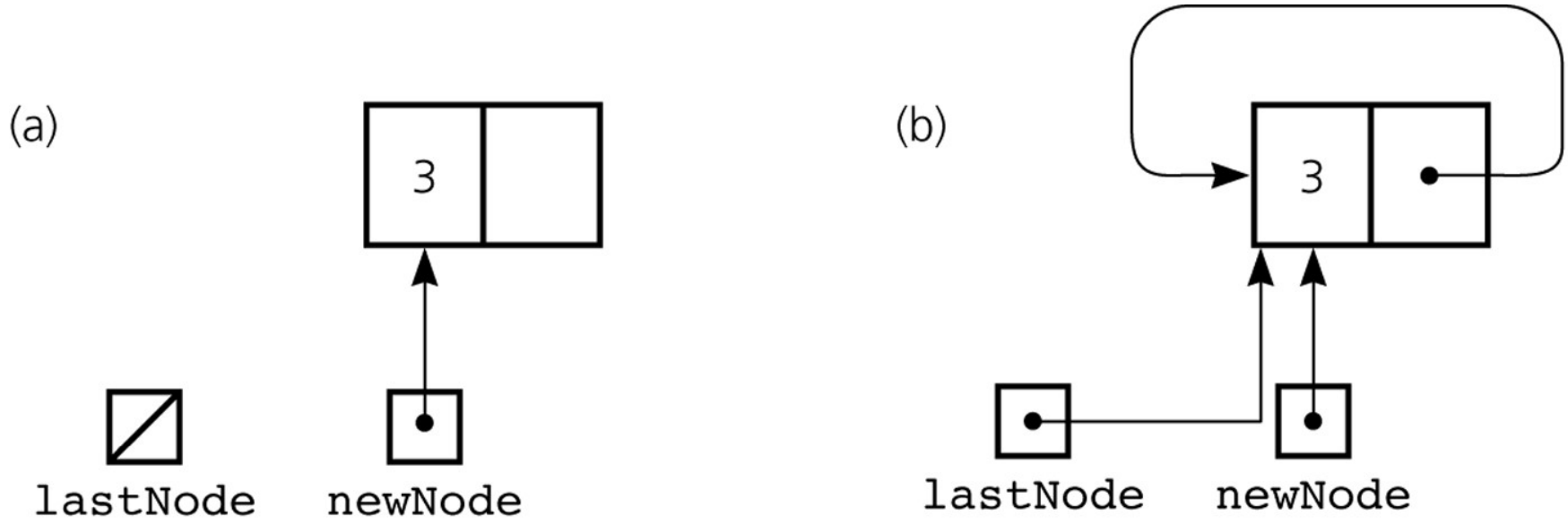
Step 2:

```
lastNode->next = newNode;
```

Step 3:

```
lastNode = newNode;
```

Insertion: Empty Queue



Step (a):

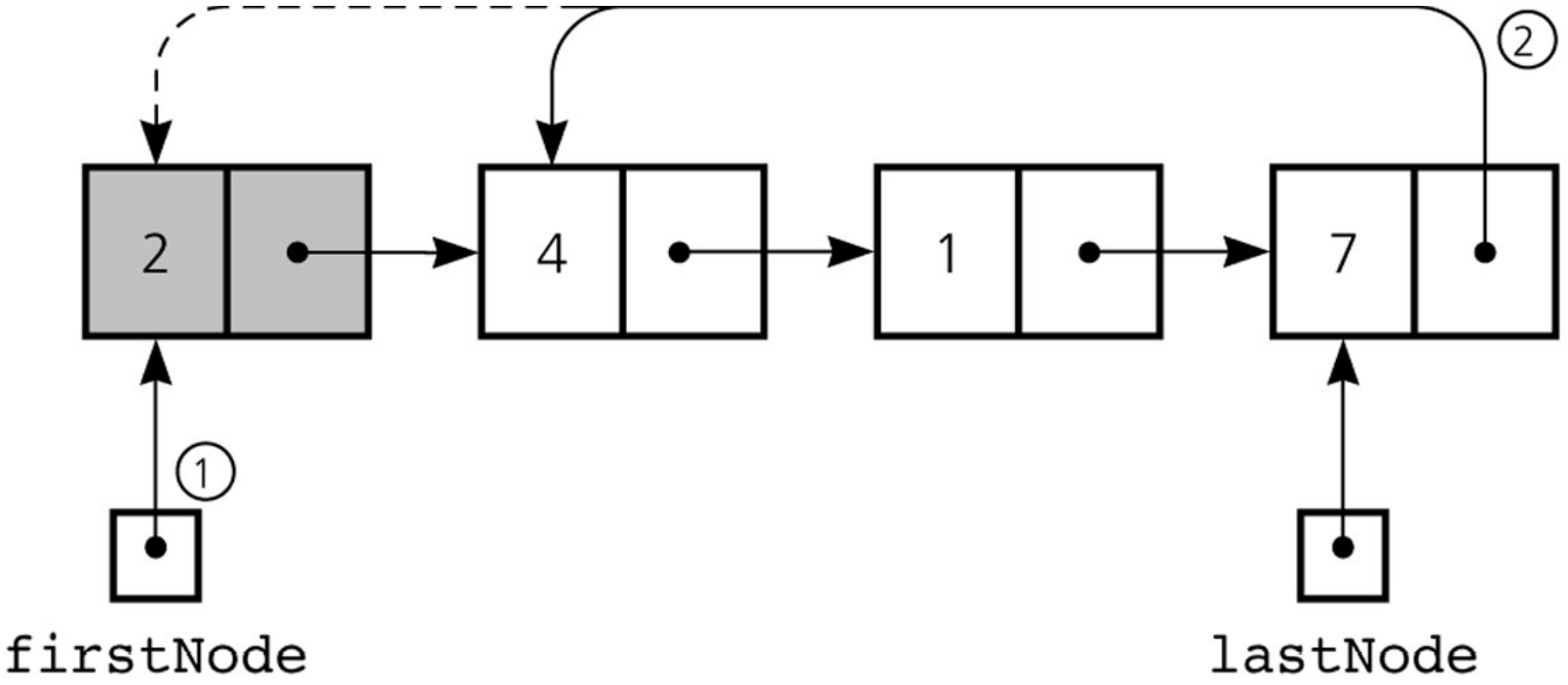
```
newNode = new QueueNode;
newNode->item = 3;
```

Step (b):

```
newNode->next = newNode;
lastNode = newNode;
```

Set up the “loop”

Deletion: Queue size larger than one



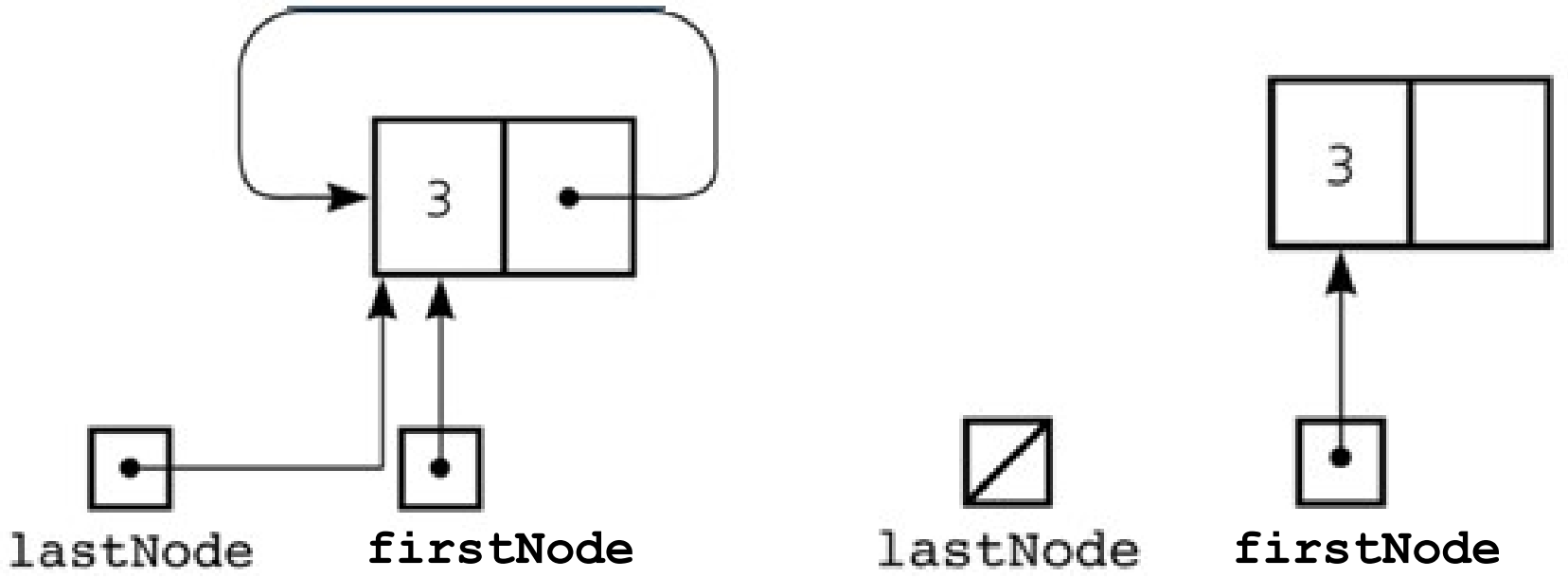
Step 1:

```
QueueNode* firstNode = lastNode->next;
```

Step 2:

```
lastNode->next = firstNode->next;
delete firstNode;
```

Deletion: Queue size equal to one?



Step 1:

```
QueueNode* firstNode = lastNode->next;
```

Step 2:

```
lastNode = null;
delete firstNode;
```

STL queue

You should have guessed it by now

STL has a built-in queue ADT

http://en.cppreference.com/w/cpp/container/queue

STL queue: Specification

```
template <class T>
class queue {
public:
  bool empty() const;
  size_type size() const;

  T& front();
  T& back();

  void push(const T& t);
  void pop();
};
```

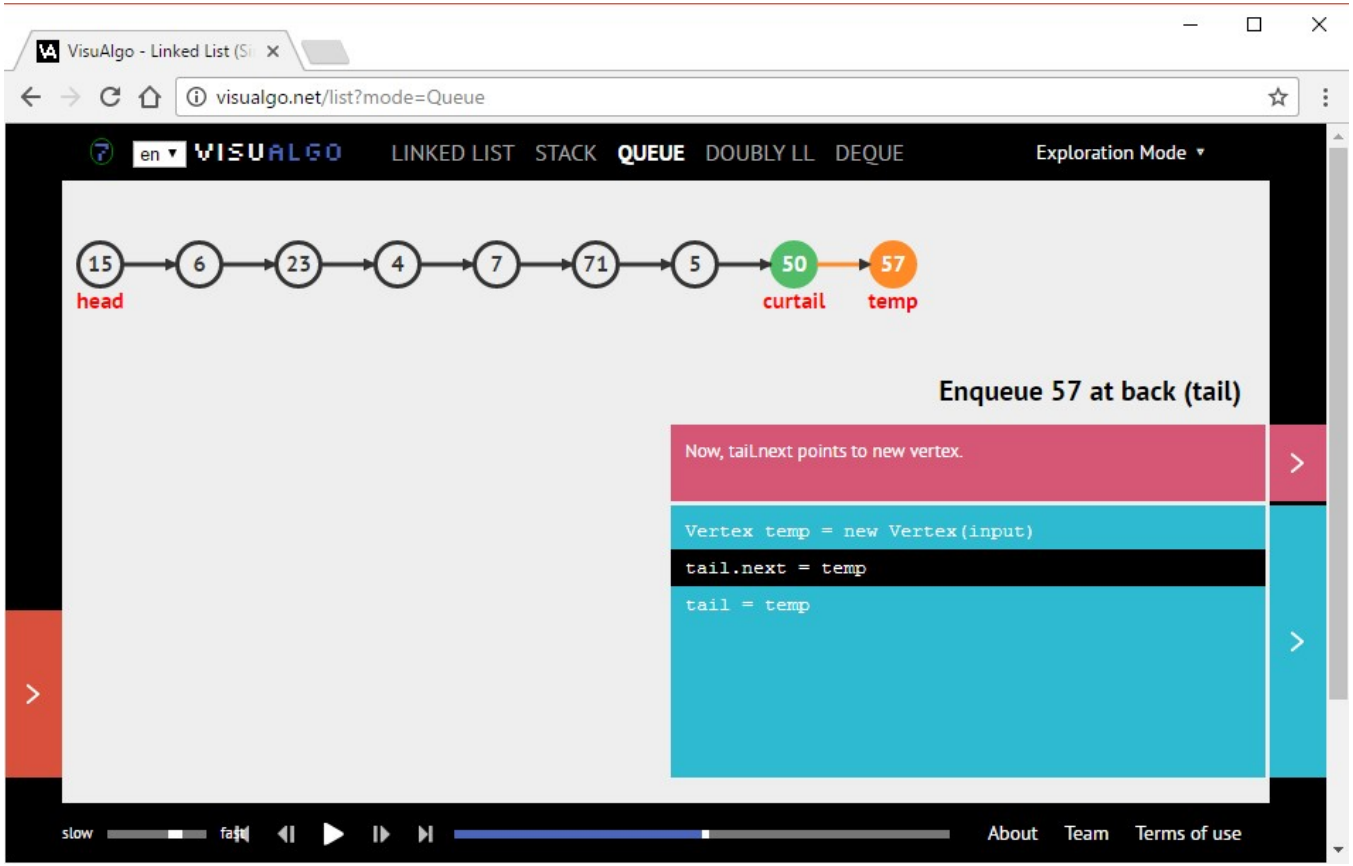
We can see both front and back

`enqueue()` is known as `push()` in STL Queue

This is the `dequeue()` equivalence

VisuAlgo

- http://visualgo.net/list?mode=Queue
- I use Tailed Linked List approach instead of Circular Linked List but the idea is the same



Queue Application

Checking for Palindrome

Palindrome: Problem Description

- **Palindrome** is a string which reads the same either *left to right*, or *right to left*
  - **Palindromes:** "r a d a r" and "d e e d"
  - **Counter Examples (most random strings):** "d a t a"
- Many solutions
  - But for the sake of discussion, let's use the two newly learned ADTs
  - Highlight the difference of LIFO and FIFO property
- Main Idea
  - Use *stack* to reverse the input
  - Use *queue* to preserve the input
  - The two sequence should be the same for palindrome

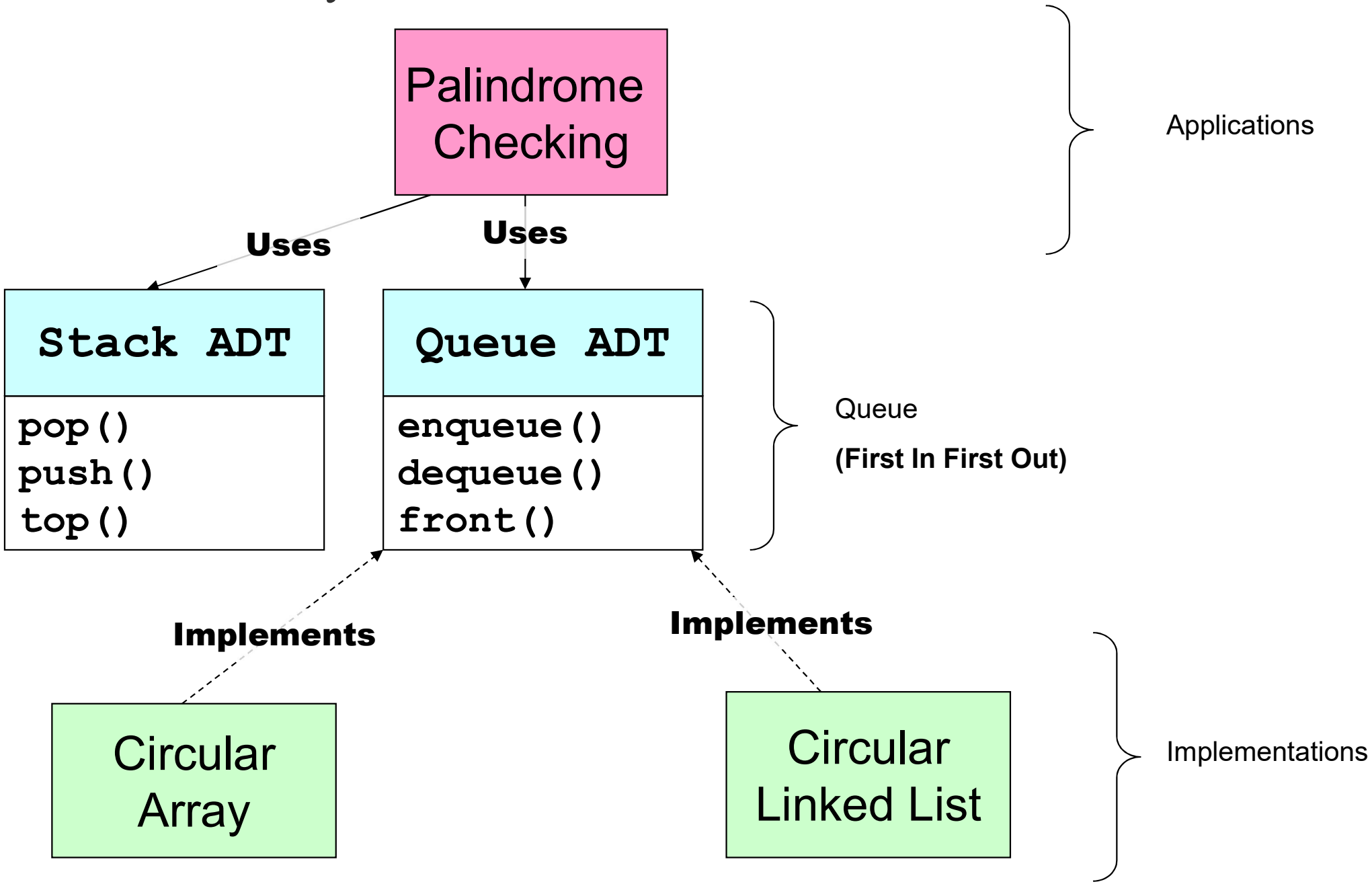
Palindrome: Implementation

```
#include <queue>
#include <stack>
using namespace std;

bool palindrome(string input) {
  stack<char> s ;
  queue<char> q ;

  for (int j = 0; j < input.size(); j++) {
    s.push(input[j]);
    q.push(input[j]);
  }
  while (!s.empty()) {
    if (s.top() != q.front())
      return false;
    s.pop();
    q.pop();
  }
  return true;
}
```

Push the same character into both queue and stack

Queue has the original sequence, Stack has the reversed. Compare to make sure they are the same

Summary


Palindrome Checking
Uses
Uses
Applications
Stack ADT
pop()
push()
top()
Queue ADT
enqueue()
dequeue()
front()
Queue
(First In First Out)
Implements
Implements
Circular Array
Circular Linked List
Implementations