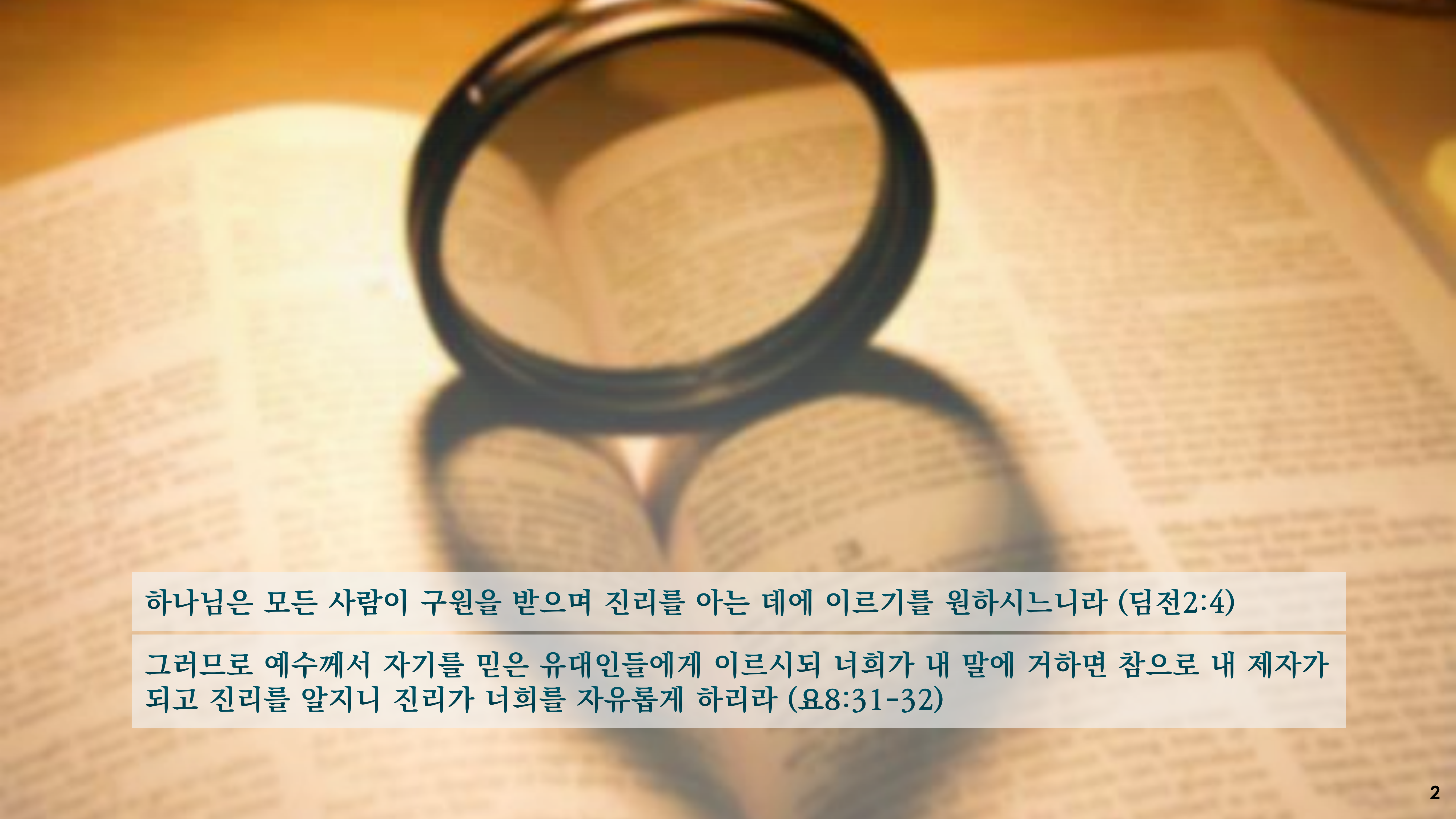
Data Structures
Chapter 3

1. Stack
2. **Queue**
   - Concepts and ADT
   - Queue Implementations
     - STL Queue
     - STL Deque (Double-ended queue)
     - Circular Queue

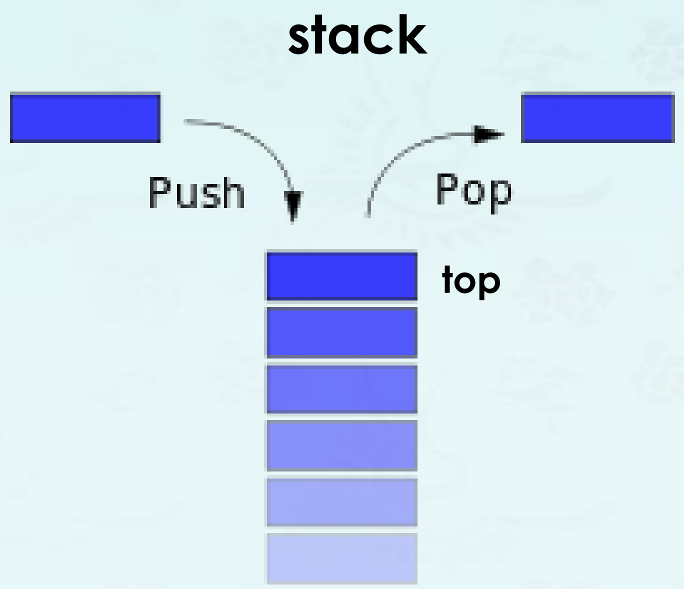


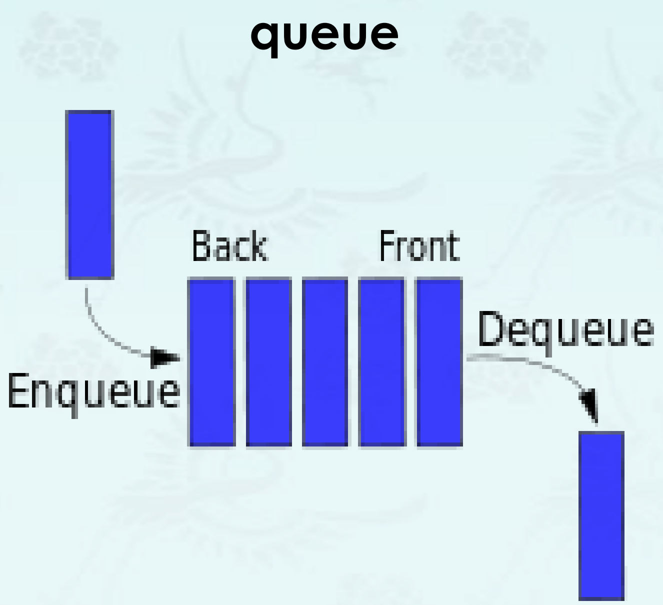
하나님은 모든 사람이 구원을 받으며 진리를 아는 데에 이르기를 원하시느니라 (딤전2:4)

그러므로 예수께서 자기를 믿은 유대인들에게 이르시되 너희가 내 말에 거하면 참으로 내 제자가 되고 진리를 알지니 진리가 너희를 자유롭게 하리라 (요8:31-32)

Queue

- Queue is an ordered list in which **enqueues** (push or insert) at the rear and **dequeues** (pop or delete) take place at different end or front.
- It is also known as a **Fist-in-first-out(FIFO)** list since it removes the item **least** recently added.

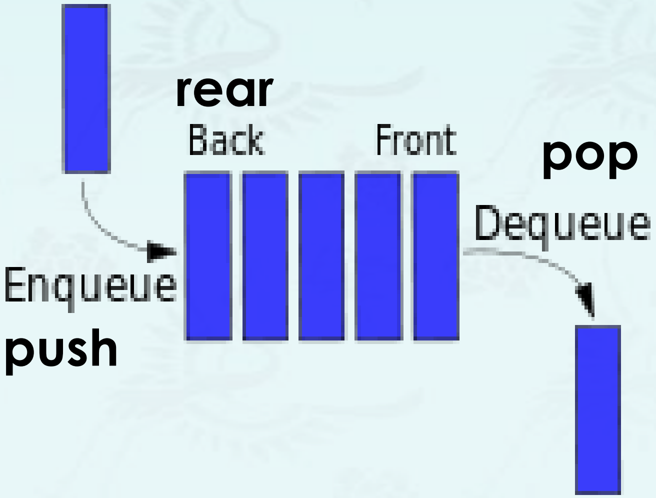




Queue

- Queue is an ordered list in which **enqueues** (push or insert) at the rear and **dequeues** (pop or delete) take place at different end or front.
- It is also known as a **Fist-in-first-out(FIFO)** list since it removes the item **least** recently added.

queue



❖ Items can only be added at the **back (rear)** of the queue and the only item that can be removed is the one at the **front** of the queue.

Queue Applications

- In a computer OS, requests for services come in unpredictable order and timing, sometimes faster than they can be serviced .
  - print a file
  - need a file from the disk system
  - send an email
  - job scheduling

Queue - ADT

- Objects: a finite ordered list with zero or more elements
- Functions:

STL	`#include <queue>`	queue class in C++ STL
	`stack<value_type>`	creates an empty queue of <value_type>
`void`	`push(value_type& item)`	inserts a new element at the end of the queue
`void`	`pop()`	removes the "oldest" element in the queue
`const_reference&`	`front()`	returns a reference to the front or "oldest element
`const_reference&`	`back()`	returns a reference to the last or "newest" element
`bool`	`empty()`	test whether container is empty
`int`	`size()`	returns the number of items in the queue

Queue - ADT

- `#include <queue>`
- `stack<value_type>`
- `push(value_type& item)`
- `pop()`
- `front()`
- `back()`
- `empty()`
- `size()`

```
// queue::front, back
int main() {
  queue<int> que;

  que.push(12);
  que.push(75);

  que.back() -= que.front();

  cout << "back() is " << que.back() << endl;

  return 0;
}
```

c++11

```
reference& back();
const_reference& back() const;
```

Returns **a reference** to the last element in the queue.
This is the "**newest**" element in the queue (i.e. the last element pushed into the queue).

Queue - ADT

- `#include <queue>`
- `stack<value_type>`
- `push(value_type& item)`
- `pop()`
- `front()`
- `back()`
- `empty()`
- `size()`

```
// queue::front, back
int main() {
  queue<int> que;

  que.push(77);
  que.push(16);

  que.front() -= que.back();

  cout << "front() is " << que.front() << endl;

  return 0;
}
```

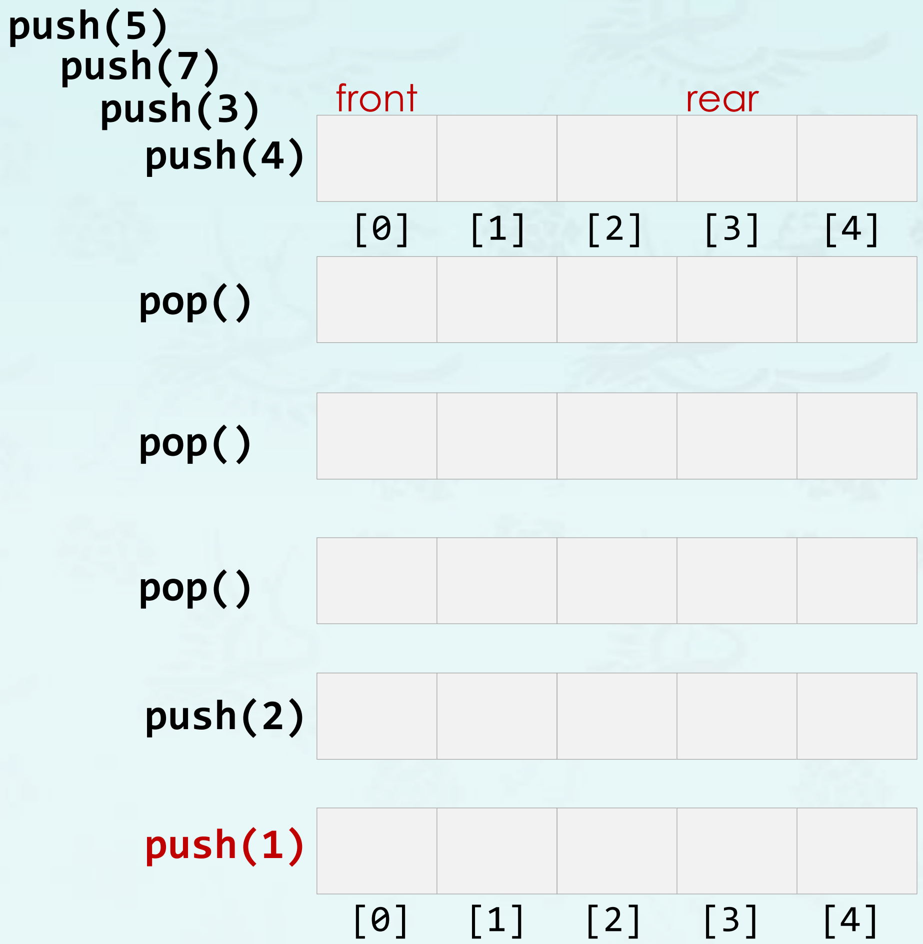
c++11

```
reference& front();
const_reference& back() front;
```

Returns **a reference** to the next element in the queue. The next element is the "**oldest**" element in the queue and the same element that is popped out from the queue when `queue::pop` is called.

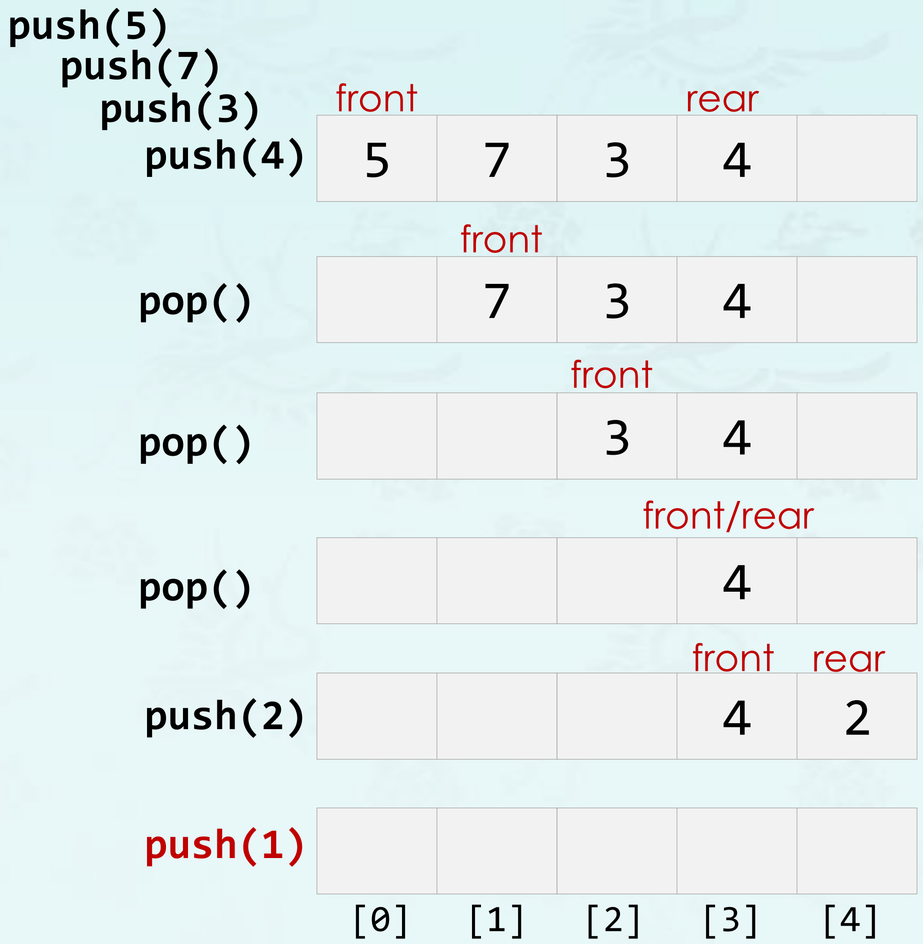
Queue Implementation

- Implementing a fixed size array has a memory problem.
  - Shift all elements by one toward the front
  - How dynamic memory allocation?



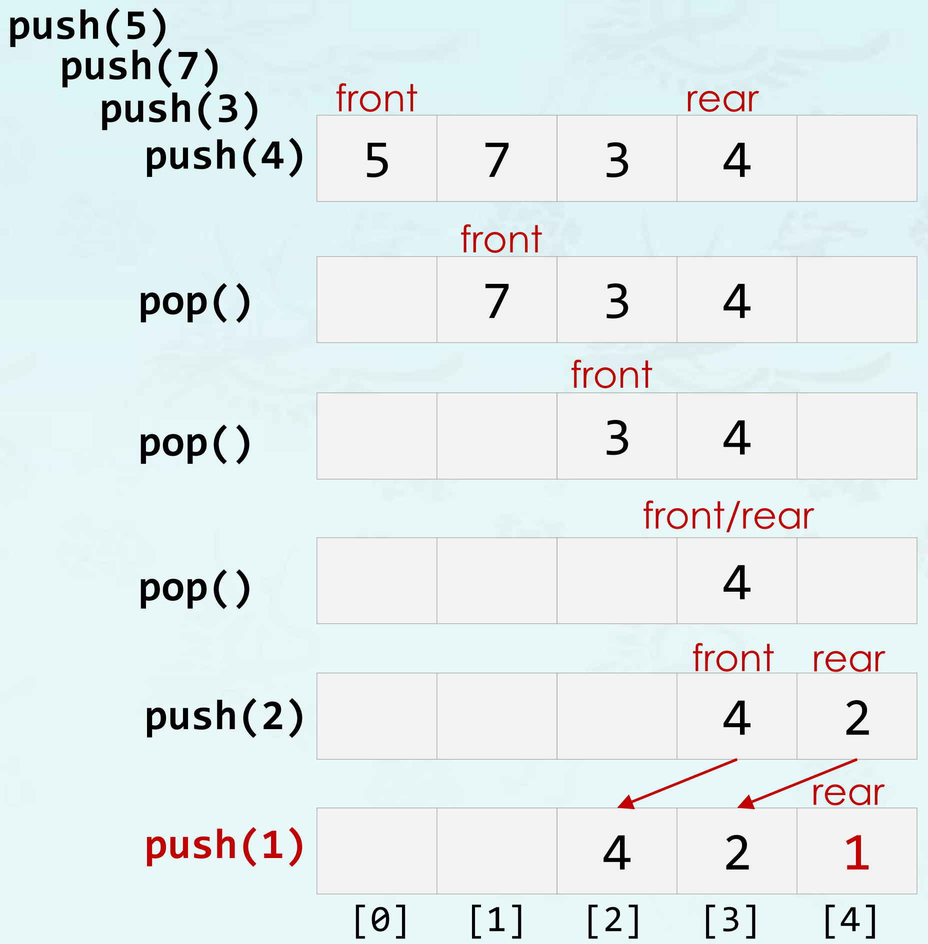
Queue Implementation

- Implementing a fixed size array has a memory problem.
  - Shift all elements by one toward the front
  - How dynamic memory allocation?



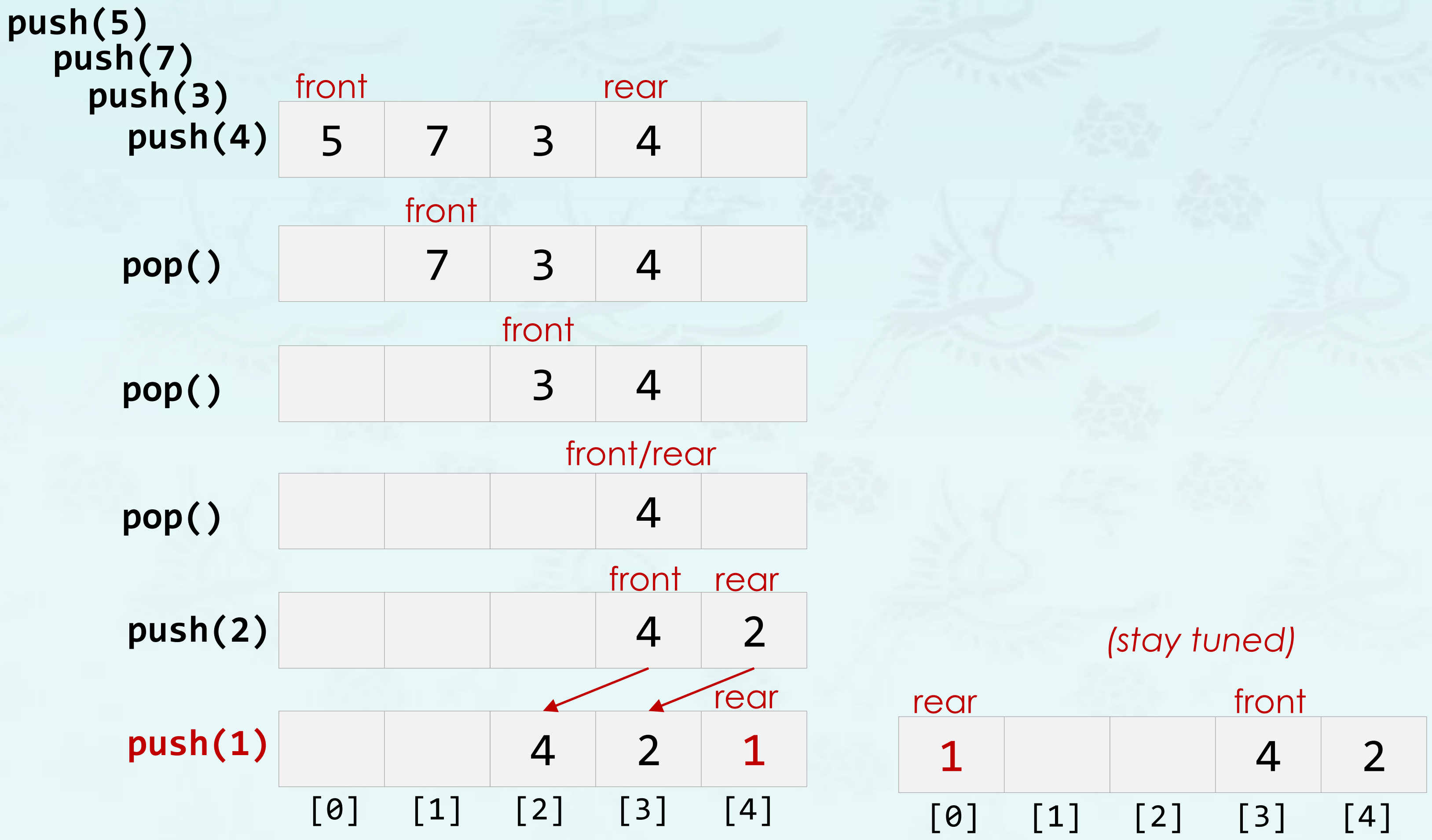
Queue Implementation

- Implementing a fixed size array has a memory problem.
  - Shift all elements by one toward the front
  - How dynamic memory allocation?



Queue Implementation

- Implementing a fixed size array has a memory problem.
  - Shift all elements by one toward the front
  - How dynamic memory allocation?

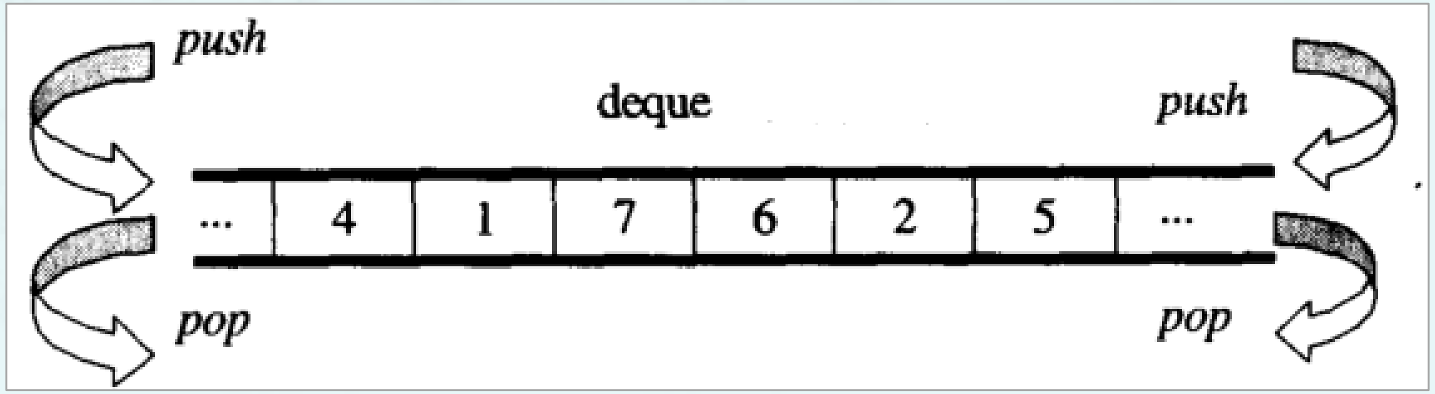


Queue Variations: Deque

- The standard queue data structure has the following variations:
  - Deque
  - Circular Queue
- **Double-ended queue** or **Deque**(pronounced “**deck**”)
  - The element can be inserted and deleted from both the front and back of the queue.
  - It is a dynamic array that is implemented so that it can **grow in both directions.**
  - So, inserting elements at the end and at the beginning is fast. However, inserting elements in the middle takes time because elements must be moved.
  - We can also implement stacks and queues using deque.

Queue Variations: Deque

- Double-ended queue or Deque(pronounced "**deck**")
- Basic Deque Operations
  - `push_back()`
  - `push_front()`
  - `insert()`
  - `pop_back()`
  - `pop_front()`
  - `empty()`
  - `size()`

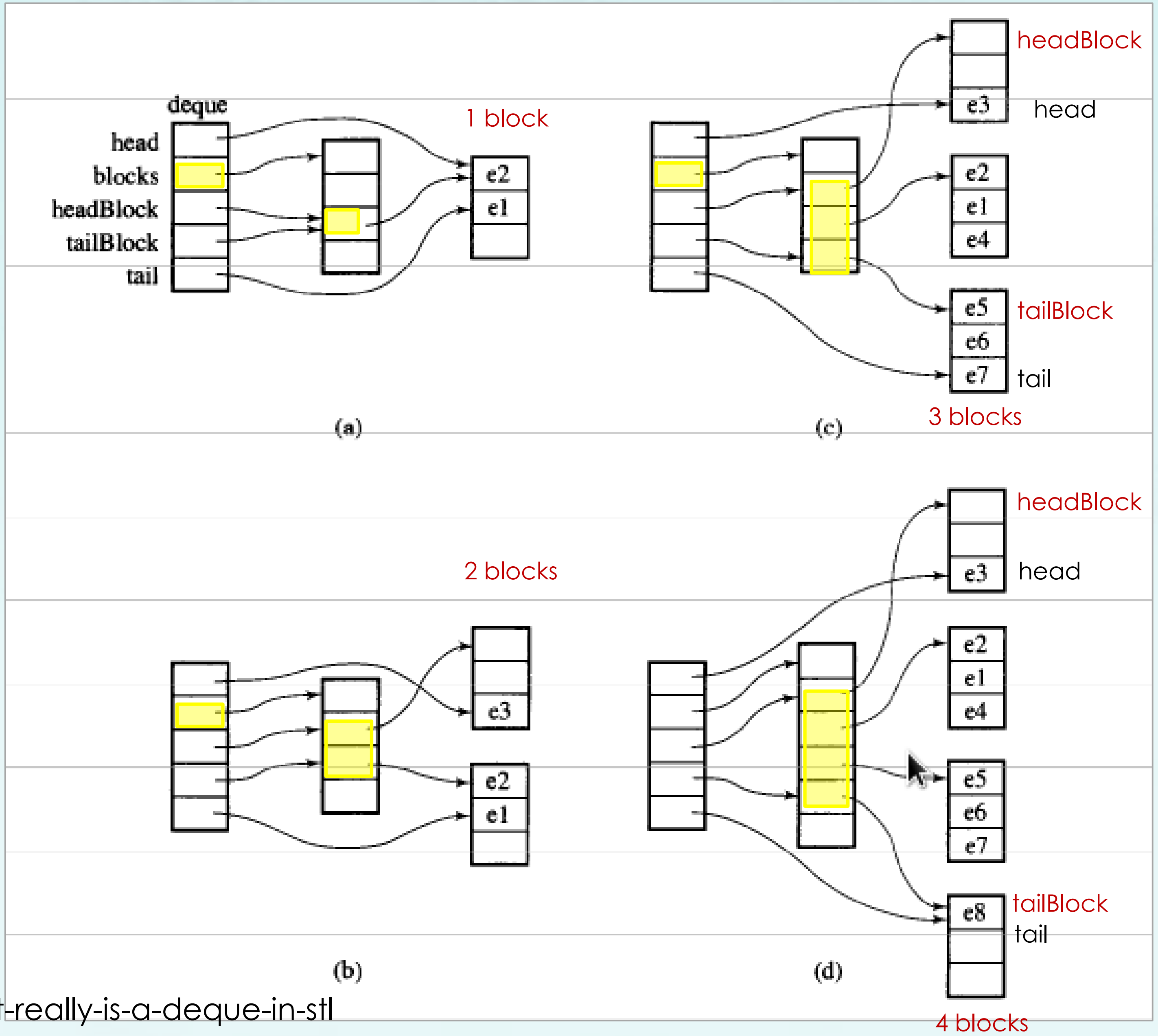


Queue Variations: Deque

- The storage of a STL `deque` is automatically expanded and contracted as needed.
  - Expansion of a **deque** is cheaper than the expansion of a `std::vector` because it does **not** involve copying of the existing elements to a new memory location.
  - An STL `deque` is not implemented as a linked list but as **an array of pointers to blocks** or arrays of data. The number of blocks changes **dynamically** depending on storage needs, and the size of the array of pointers changes accordingly.
- The complexity (efficiency) of common operations on deques is as follows:
  - Random access - **constant *O(1)***
  - Insertion or removal of elements at the end or beginning - **constant *O(1)***
  - Insertion or removal of elements in the middle - **linear *O(n)***

Queue Variations: Deque

- STL Deque Implementation for your reference.
  - An image is worth a thousand words.

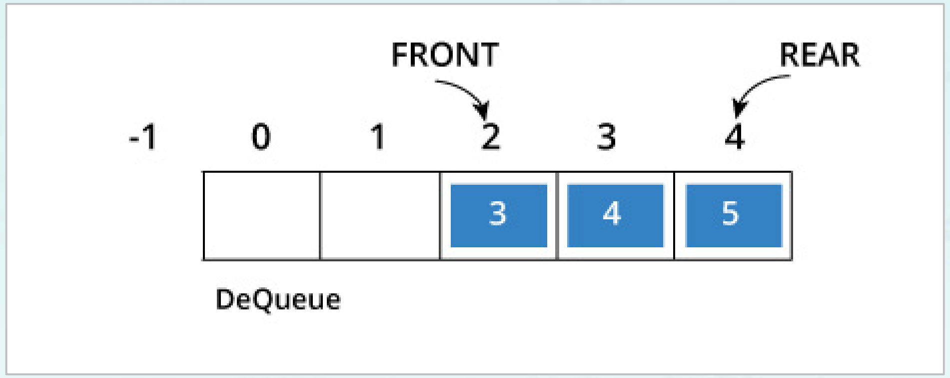


Notice that the arrays in the middle is the array of pointers to the data (chunks on the right), and it is dynamically changing.

Reference - https://stackoverflow.com/questions/6292332/what-really-is-a-deque-in-stl

Queue Variations: Circular Queue

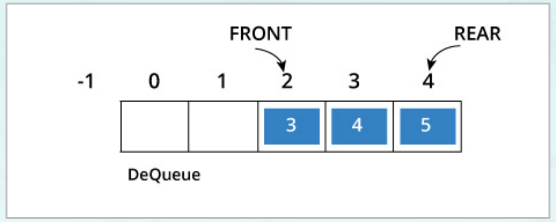
- The standard queue data structure has the following variations:
  - Deque
  - Circular Queue
- **Circular queue** avoids the wastage of space in a regular queue implementation using arrays.

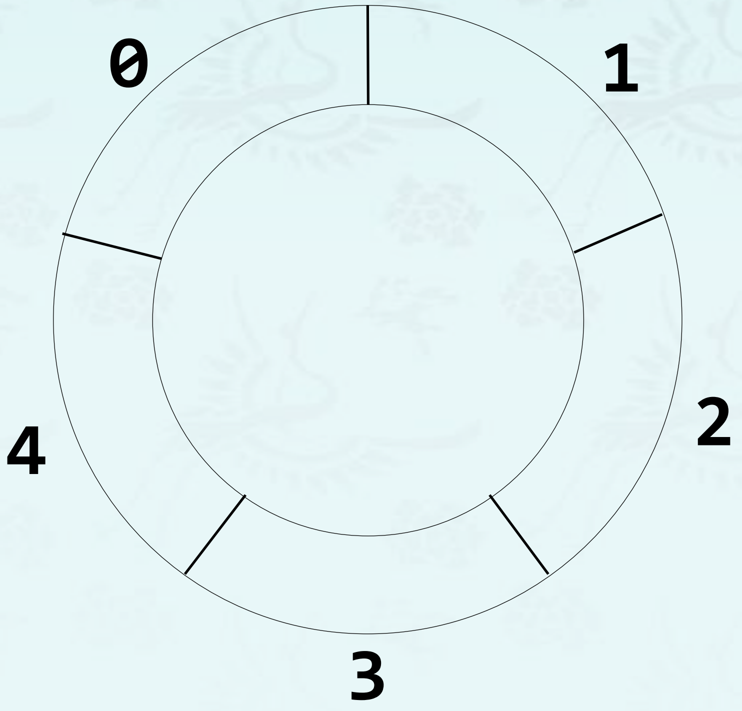


  - As you can see in the above image, after a bit of enqueueing and dequeueing, the size of the queue has been reduced.
  - The indexes 0 and 1 can only be used after the queue is reset when all the elements have been dequeued.

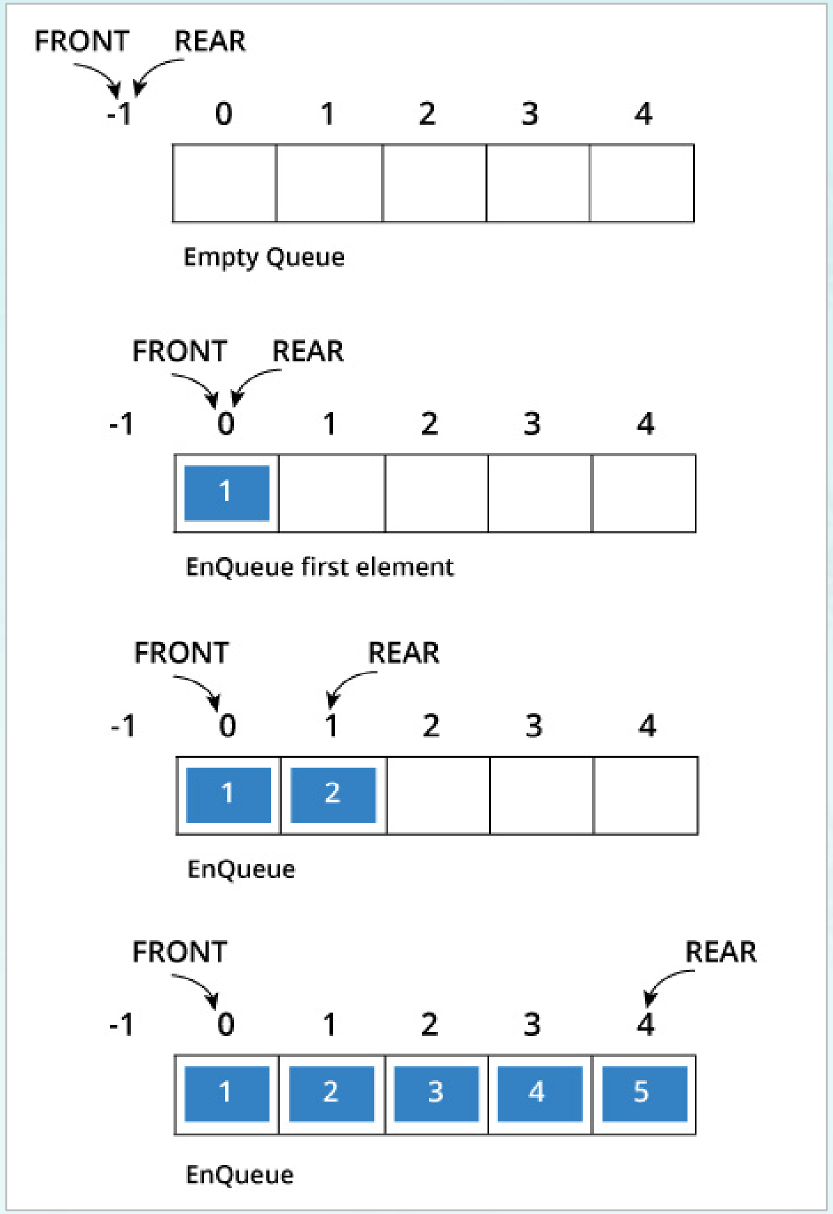
Queue Variations: Circular Queue

- **How Circular Queue Works**
  - Circular Queue works by the process of circular increment i.e. when we try to increment any variable and we reach the end of queue, we start from the beginning of queue by modulo division with the queue size.
  - `if REAR + 1 == 5 (overflow!), New REAR = (REAR + 1) % 5 (start of queue)`



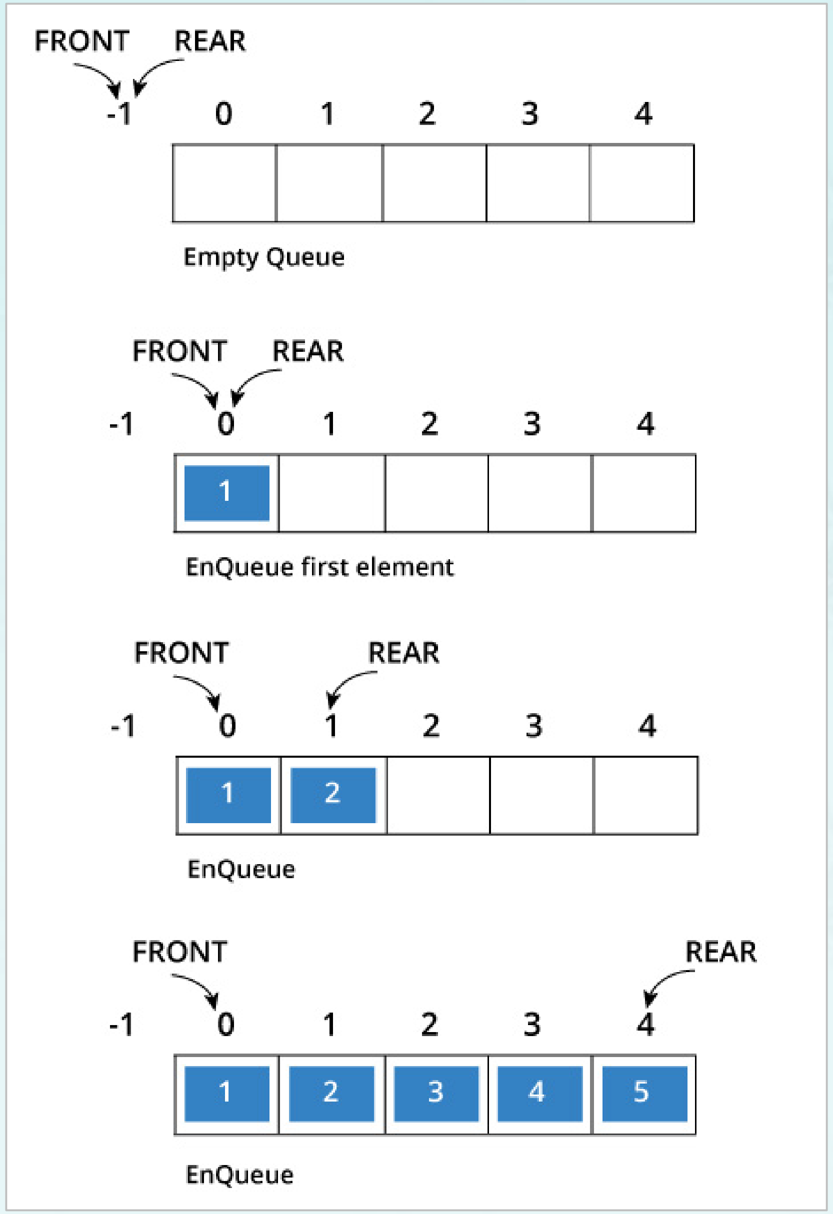


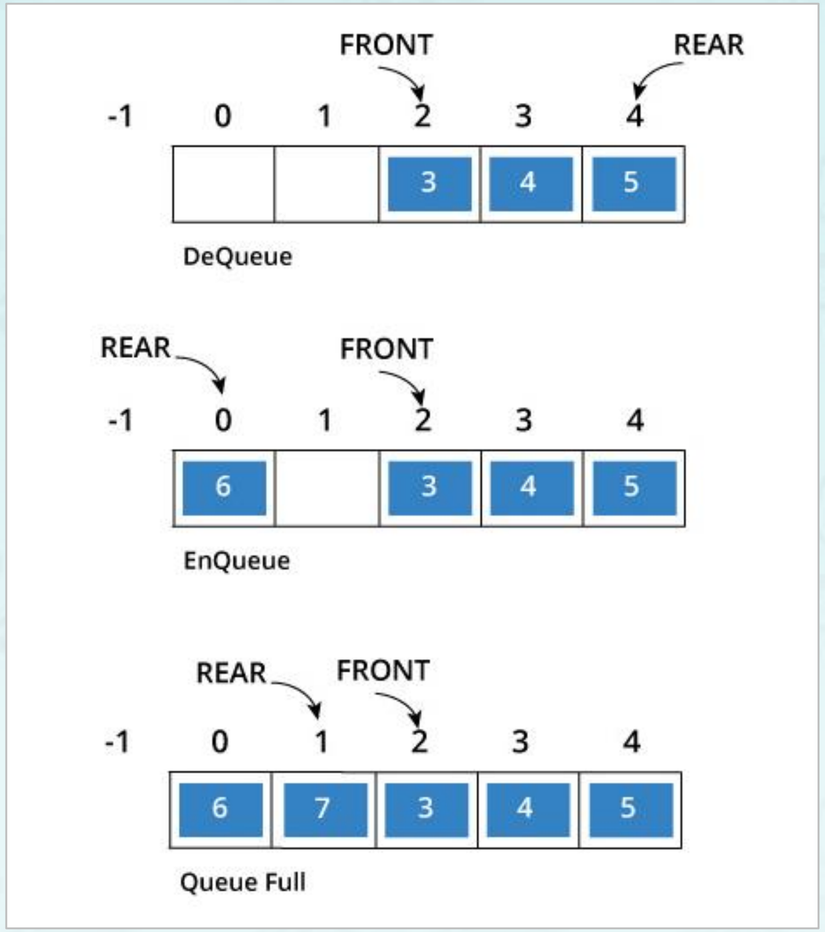
Queue Variations: Circular Queue



Case 1: `FRONT = 0 && REAR == SIZE - 1`

Queue Variations: Circular Queue





Case 2: `FRONT = REAR + 1`

Case 1: `FRONT = 0 && REAR == SIZE - 1`

Queue Variations: Circular Queue

- **Queue operations work as follows:**
  - Two pointers called FRONT and REAR are used to keep track of the first and last elements.
  - When initializing the queue, we set the value of FRONT and REAR to -1.
  - On enqueing an element, we circularly increase the value of REAR index and place the new element in the position pointed to by REAR.
  - On dequeueing an element, we return the value pointed to by FRONT and circularly increase the FRONT index.
  - Before enqueing, we check if queue is already full.
  - Before dequeuing, we check if queue is already empty.
  - When enqueing the first element, we set the value of FRONT to 0.
- The check for full queue has a new additional case:
  - Case 1: `FRONT = 0 && REAR == SIZE - 1`
  - Case 2: `FRONT = REAR + 1`
  - The second case happens when REAR starts from 0 due to circular increment and when its value is just 1 less than FRONT, the queue is full.

Queue Variations: Circular Queue

```
// size of circular queue, a magic number
#define SIZE 5

struct Queue {
  int items[SIZE], front, rear;
};
using queue = Queue *;

queue newQueue(){
  queue q = new Queue;
  q->front = -1;
  q->rear = -1;
  return q;
}

bool full(queue q){
  if (q->front == 0 && q->rear == SIZE - 1)
    return true;
  if (q->front == q->rear + 1) return true;
  return false;
}
```

```
bool empty(queue q){
  if (q->front == -1) return true;
  return false;
}

void enqueue(queue q, int element){
  if(full(q)){
    cout << "Queue is full" << endl;
  } else {
    if(q->front == -1) q->front = 0;
    q->rear = (q->rear + 1) % SIZE;
    q->items[q->rear] = element;
    cout << "enqueued: " << element << endl;
  }
}
```

Queue Variations: Circular Queue

```
int dequeue(queue q){
  int element;
  if (empty(q)){
      cout << "Queue is empty" << endl;
      return(-1);
  }
  else {
    element = q->items[q->front];
    if(q->front == q->rear){
        q->front = -1;
        q->rear = -1;
    } // q has only one element,
      // we reset the q after deleting it.
    else {
        q->front=(q->front + 1) % SIZE;
    }
    return element;
  }
}
```

```
void display(queue q) { // display queue status
  int i;
  if(empty(q))
    cout << endl << "Empty Queue" << endl;
  else {
    cout << "Front[" << q->front << "]\n";
    cout << "Items: ";
    for(i = q->front; i!=q->rear; i=(i+1)%SIZE)
        cout << q->items[i] << ' ';
    cout << q->items[i];
    cout << endl << " Rear["<< q->rear << "]\n";

  }
}
```

Queue Variations: Circular Queue

```
int main() {
    queue q = newQueue();
    dequeue(q);
    enqueue(q, 1);
    enqueue(q, 2);
    enqueue(q, 3);
    enqueue(q, 4);
    enqueue(q, 5);
    enqueue(q, 6);
    display(q);
    int elem = dequeue(q);
    if (elem != -1)
       cout << "dequeued: " << elem << endl;
    display(q);
    enqueue(q, 7);
    display(q);
    enqueue(q, 8);
    dequeue(q);
    dequeue(q);
    display(q);
    return 0;
}
```

Quiz:

- How many failures occurred?
- At the end of running this main(),
  (1) draw a diagram that shows the status of queue items as well as the locations of front and rear.
  (2) write elements in the queue from front to rear.

Data Structures
Chapter 3

1. Stack
2. **Queue**
   - Concepts and ADT
   - Queue Implementations
     - STL Queue
     - STL Deque (Double-ended queue)
     - Circular Queue