CIT 5940

Sequential Data Structures

Agenda

1. Why Arrays?
2. Stack Implementation
3. Queue Implementation
4. ArrayDeque
5. ArrayList
6. Comparison & Use Cases

C I T 5 9 4 0

Why Arrays?

Arrays as Building Blocks

- Memory is fundamentally sequential
- Arrays provide O(1) access to any position
- Fixed size forces us to think about space efficiency
- Basis for more complex data structures

Sequential Access Pattern

All our structures will share:

- Elements stored contiguously in memory
- Direct access to positions via indices
- Need strategies for:
  - Insertion
  - Deletion
  - Growth

Different Access Patterns

Structure determines where operations happen:

Structure	Insert Location	Remove Location
Stack	One end only	Same end
Queue	One end	Other end
ArrayDeque	Either end	Either end
ArrayList	Any position	Any position

C I T
5 9 4 0

Stacks

Managing Array Access

To build a stack with an array, we need:

- A way to track where elements end
- A strategy for insertion/removal

Simplest approach: single index tracking "top"

- Points to next free position
- All operations happen at this position

Implementation Basics

```
public class ArrayStack<E> {
    private E[] elements;
    private int top;  // points to next free spot

    @SuppressWarnings("unchecked")
    public ArrayStack(int capacity) {
        elements = (E[]) new Object[capacity];
        top = 0;
    }
}
```

What Operations Make Sense?

Given a top index, what can we do?

- Add element at top index (push)
- Remove element at (top-1) (pop)
- Look at element at (top-1) (peek)
- Check if any elements exist (isEmpty)
- Count elements (size)

The Stack ADT

Core operations:

- `push(e)`: Add element to top
- `pop()`: Remove and return top element
- `peek()`: Return but don't remove top

Helper operations:

- `isEmpty()`: Check if stack empty
- `size()`: Return number of elements

Stack Behavior

The single-index implementation leads to Last-In-First-Out (**LIFO**):

- Most recently pushed element must be first to be popped
- Elements pop out in reverse order of pushing
- Only one element accessible at a time

Examples:

- Web browser history
- Function call stack
- Syntax checking

Push Operation

```
public void push(E elem) {
    if (top == elements.length) {
        throw new IllegalStateException(
            "Stack is full");
    }
    elements[top] = elem;
    top++;
}
```

Pop Operation

```
public E pop() {
    if (top == 0) {
        throw new NoSuchElementException(
            "Stack is empty");
    }
    top--;
    E elem = elements[top];
    elements[top] = null;
    return elem;
}
```

Stack Runtime Analysis

Operation	Runtime	Reason
push	O(1)	Array append
pop	O(1)	Array removal from end
peek	O(1)	Array access
isEmpty	O(1)	Check top
size	O(1)	Return top

All operations are constant time!

Activity: Peek at Distance k

Write a method that looks at element k positions from top:

```
public static <E> E peekK(Stack<E> stack, int k) {
    // k = 0 means look at top element
    // k = stack.size()-1 means look at bottom element
    // Throw exception if k invalid
}
```

Requirements:

- Stack should be unchanged when method returns
- Must work for any Stack implementation

Solution: Using Helper Stack

Runtime: $\Theta(k)$ to pop/push k elements

```java
public static <E> E peekK(Stack<E> stack, int k) {
    if (k < 0 || k >= stack.size()) {
        throw new IllegalArgumentException();
    }

    Stack<E> helper = new Stack<>();
    // Pop k elements onto helper
    for (int i = 0; i < k; i++) {
        helper.push(stack.pop());
    }

    // Peek at next element
    E result = stack.peek();

    // Restore stack
    while (!helper.isEmpty()) {
        stack.push(helper.pop());
    }

    return result;
}
```

CIT 5940

QUEUES

Changing the Order of Pop & Push

Stacks provide constant time access to the element at the top—which is nice—but are fundamentally LIFO.

💡 Idea: LIFO implies the existence of **FIFO** (First-In-First-Out), a more natural ordering for humans

Examples:

- Waiting in line at a restaurant
- Preparing & serving ingredients from older shipments first
- Sending jobs to a printer

FIFO: Two Pointers

We can make a simple modification to an *ArrayStack* to make it behave in a FIFO manner: add an additional pointer.

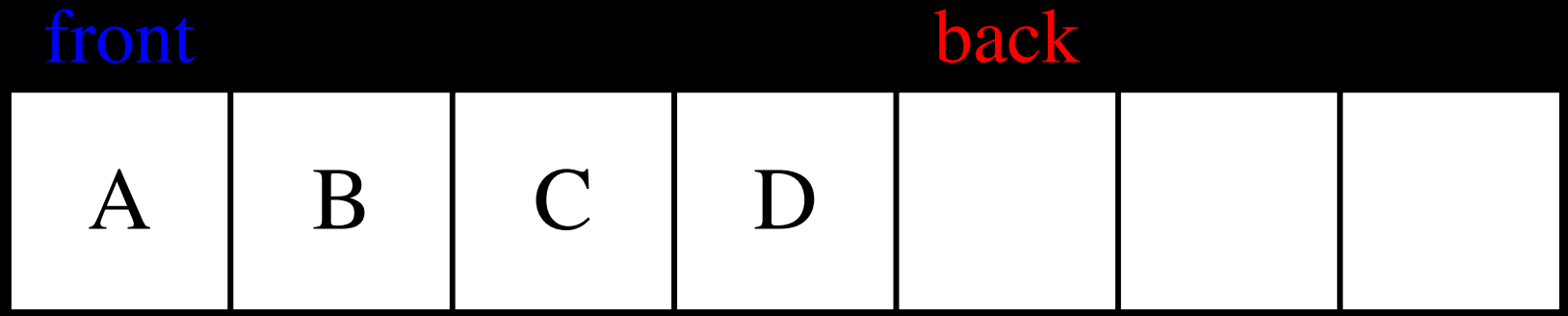
- one to refer to the "back" (previously the "top"), where elements are added
- another to refer to the "front", where elements are removed

A linear sequence where elements are added to the `front` and removed from the `back` is called a **queue.**

Two-Pointer: Structure

```
public class ArrayQueue<E> {
    private E[] elements;
    private int front = 0;  // removal index
    private int back = 0;   // insertion index
    private int size = 0;
}
```

Example state after enqueueing A, B, C, D:



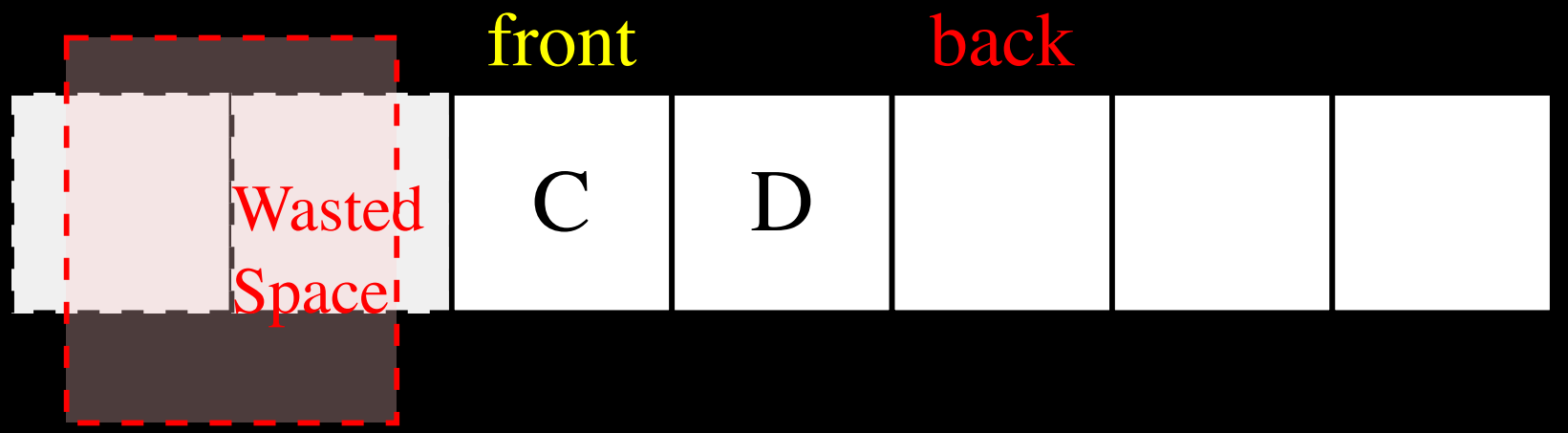
Two-Pointer: Enqueue

```
public void enqueue(E elem) {
    if (back == elements.length) {
        throw new IllegalStateException("Queue full");
    }
    elements[back] = elem;
    back++;
    size++;
}
```

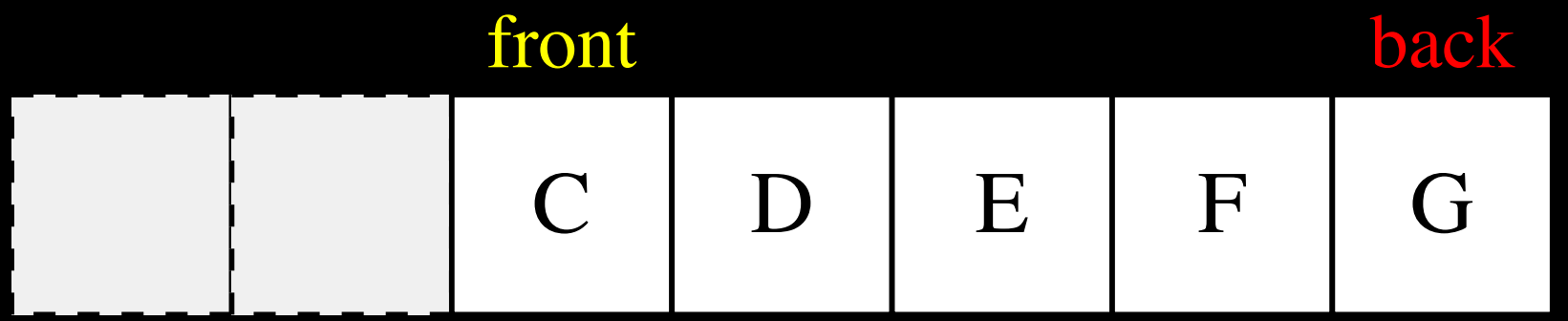
Two-Pointer: Dequeue

```
public E dequeue() {
    if (size == 0) {
        throw new NoSuchElementException("Queue empty");
    }
    E result = elements[front];
    elements[front] = null;
    front++;
    size--;
    return result;
}
```

After dequeueing A,B:



Then enqueueing E,F,G:

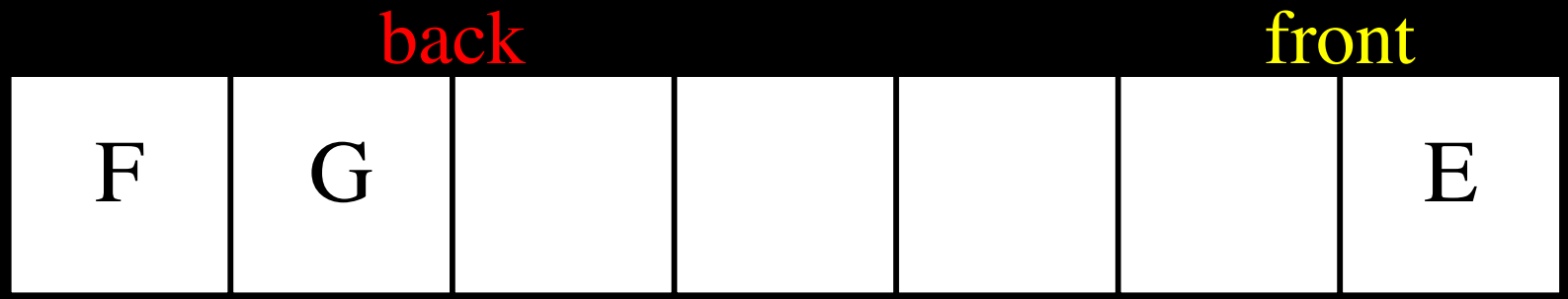


✗

Can't add

Circular Array Solution

Key insight: When we hit the end, wrap around!



- Use modulo arithmetic for indices
- Reuse space at beginning of array
- Need to track size separately from front/back

Implementation: Setup

```
public class CircularArrayQueue<E> {
    private E[] elements;
    private int front = 0;
    private int back = 0;
    private int size = 0;

    @SuppressWarnings("unchecked")
    public CircularArrayQueue(int capacity) {
        elements = (E[]) new Object[capacity];
    }
}
```

Circular Arithmetic

For an array of size n:

- Next position after i: `(i + 1) % n`
- Previous position before i: `(i - 1 + n) % n`
    - not actually relevant yet—why?

Circular Enqueue

```
public void enqueue(E elem) {
    if (size == elements.length) {
        throw new IllegalStateException("Queue full");
    }
    elements[back] = elem;
    back = (back + 1) % elements.length;
    size++;
}
```

Wrap `back` pointer using modulo—enqueuing always moves `back` to the right.

Circular Dequeue

```
public E dequeue() {
    if (size == 0) {
        throw new NoSuchElementException("Queue empty");
    }
    E result = elements[front];
    elements[front] = null;
    front = (front + 1) % elements.length;
    size--;
    return result;
}
```

Wrap front pointer using modulo—dequeuing always moves `front` to the right.

Circular Queue Runtime

Operation	Runtime	Reason
enqueue	O(1)	Array append + modulo
dequeue	O(1)	Array removal + modulo
peek	O(1)	Array access at front
size	O(1)	Return field
isEmpty	O(1)	Check size field

All operations constant time due to:

- No shifting of elements
- Modulo arithmetic is $O(1)$
- Direct array access is also $O(1)$

Activity: Peek at Position k (Queue)

Write a method that looks at element k positions from front:

```
public static <E> E peekK(Queue<E> q, int k) {
    // k = 0 means look at front element
    // k = q.size()-1 means look at back element
    // throw exception if k invalid
}
```

Requirements:

- Queue should be unchanged when method returns
- Must work for any Queue implementation

Solution: Using Helper Queue

```
public static <E> E peekK(Queue<E> q, int k) {
    Queue<E> helper = new ArrayDeque<>();
    // Dequeue k elements to helper, then look at result.
    for (int i = 0; i < k; i++) {
        helper.add(q.remove());
    }
    E result = q.peek();
    // Restore the queue by clearing it then putting everything back.
    while (!q.isEmpty()) {
        helper.add(q.remove());
    }
    while (!helper.isEmpty()) {
        q.add(helper.remove());
    }
    return result;
}
```

CIT 5940

ArrayDeque

Deque Operations

A **double-ended queue**, or **deque** ("deck") allows:

- Add/remove at front
- Add/remove at back
- Peek at either end

Stack? Queue? *¿Por qué no los dos?*

```
void addFirst(E e)      void addLast(E e)
E removeFirst()         E removeLast()
E getFirst()           E getLast()
```

Circular Array + Two Pointers

Key ideas from Queue implementation:

- Use circular array with modulo
- Track size separately
- Both ends can move either direction

Implementation Setup

```
public class ArrayDeque<E> {
    private E[] elements;
    private int front;
    private int back;
    private int size;

    @SuppressWarnings("unchecked")
    public ArrayDeque(int capacity) {
        elements = (E[]) new Object[capacity];
        front = 0;
        back = 0;
    }
}
```

Adding At Front

```
public void addFirst(E e) {
    if (size == elements.length) {
        throw new IllegalStateException("Deque is full");
    }
    front = (front - 1 + elements.length) % elements.length;
    elements[front] = e;
    size++;
}
```

To add at front:

- Decrement `front` (with wrap-around)—moves to the "left"
- Place element at new `front` position

Adding At Back

```
public void addLast(E e) {
    if (size == elements.length) {
        throw new IllegalStateException("Deque is full");
    }
    elements[back] = e;
    back = (back + 1) % elements.length;
    size++;
}
```

Same as Queue's enqueue operation:

- Place element at `back` position—moves to the "right"
- Increment `back` (with wrap-around)

Removing From Front

```
public E removeFirst() {
    if (size == 0) {
        throw new NoSuchElementException("Deque is empty");
    }
    E result = elements[front];
    elements[front] = null;
    front = (front + 1) % elements.length;
    size--;
    return result;
}
```

Same as Queue's dequeue operation:

- Get element at `front`
- Increment `front` (with wrap-around)—moves to the "right"

Removing From Back

```
public E removeLast() {
    if (size == 0) {
        throw new NoSuchElementException("Deque is empty");
    }
    back = (back - 1 + elements.length) % elements.length;
    E result = elements[back];
    elements[back] = null;
    size--;
    return result;
}
```

To remove from back:

- Decrement `back` (with wrap-around)—moves to the "left"
- Get element at new `back` position

Runtime Analysis

Operation	Runtime	Reason
addFirst	O(1)	Array write + modulo
addLast	O(1)	Array write + modulo
removeFirst	O(1)	Array read + modulo
removeLast	O(1)	Array read + modulo
peekFirst	O(1)	Array access
peekLast	O(1)	Array access

All operations constant time with circular array!

CIT 5940

ArrayList

Fixed Array + Size

Starting point: Array with size counter

```
public class FixedArrayList<E> {
    private E[] elements;
    private int size;

    @SuppressWarnings("unchecked")
    public FixedArrayList(int capacity) {
        elements = (E[]) new Object[capacity];
        size = 0;
    }
}
```

What Operations Make Sense?

These operations cost the same:

- Access/modify any position
- Count elements

These operations can't always be constant time—not working at a known position each time:

- Insert at any position up to size
- Remove from any position

List ADT

Core positional operations:

```
void add(int index, E element)  // Insert at position
E remove(int index)              // Remove at position
E get(int index)                 // Access at position
E set(int index, E element)      // Modify at position
```

Helper operations:

```
boolean isEmpty()
int size()
void add(E element)  // Append to end
```

Adding Elements (Fixed Array)

```
public void add(int index, E element) {
    if (index < 0 || index > size) {
        throw new IndexOutOfBoundsException();
    }
    if (size == elements.length) {
        throw new IllegalStateException("List is full");
    }

    // Shift elements right
    for (int i = size; i > index; i--) {
        elements[i] = elements[i-1];
    }
    elements[index] = element;
    size++;
}
```

Removing Elements

```
public E remove(int index) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    }

    E removed = elements[index];

    // Shift elements left
    for (int i = index; i < size - 1; i++) {
        elements[i] = elements[i+1];
    }
    elements[size-1] = null;
    size--;

    return removed;
}
```

Get and Set

```
public E get(int index) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    }
    return elements[index];
}

public E set(int index, E element) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    }
    E old = elements[index];
    elements[index] = element;
    return old;
}
```

Fixed List Runtime

Operation	Runtime	Reason
add(E)	O(1)	Array append
add(i,E)	O(n)	Shift elements
remove(i)	O(n)	Shift elements
get(i)	O(1)	Array access
set(i,E)	O(1)	Array access

Problem: Fixed size limits usability

Making it Dynamic

Solution: Grow array when it fills up

```
@SuppressWarnings("unchecked")
private void grow() {
    E[] newElements = (E[]) new Object[elements.length * 2];
    for (int i = 0; i < size; i++) {
        newElements[i] = elements[i];
    }
    elements = newElements;
}

private void ensureCapacity() {
    if (size == elements.length) {
        grow();
    }
}
```

Dynamic Add Operation

```
public void add(int index, E element) {
    if (index < 0 || index > size) {
        throw new IndexOutOfBoundsException();
    }
    ensureCapacity();  // Only change from fixed version

    // Shift elements right
    for (int i = size; i > index; i--) {
        elements[i] = elements[i-1];
    }
    elements[index] = element;
    size++;
}
```

Growth Example

Starting with n = 4:

Total copy cost for 9 elements: 4 + 8 = 12 copies

Average cost per add: 12/9 ≈ 1.33 copies

Growth Analysis

When does array growth happen?

- Start with size n
- Double size when full
- Copy all elements to new array

Growth sequence: n, 2n, 4n, 8n, ...

Cost sequence: n, 2n, 4n, 8n, ...

Amortized Analysis

For m insertions starting with size n:

- Total copying cost: n + 2n + 4n + ...
- Geometric series sum ≤ 2m
- Average cost per operation: O(1)

Even though individual operations might be expensive, the average cost per operation is constant.

Amortized Cost Visualization

Operation #	Array Size	Copy Cost
1-n	n	0
n+1	2n	n
n+1 to 2n	2n	0
2n+1	4n	2n
2n+1 to 4n	4n	0

Most operations cost O(1), occasional O(n)

Comparing Implementations

Operation	Stack	Queue	ArrayDeque	ArrayList
Add First	O(1)	—	O(1)	O(n)
Add Last	—	O(1)	O(1)	O(1)*
Add at i	—	—	—	O(n)
RemoveFirst	O(1)	O(1)	O(1)	O(n)
RemoveLast	—	—	O(1)	O(1)
Remove at i	—	—	—	O(n)
Get/Set	O(1)	O(1)	O(1)	O(1)

* Amortized

When to Use Each?

Stack:

- LIFO access pattern

Queue:

- FIFO access pattern

ArrayDeque:

- Need efficient operations at both ends

ArrayList:

- Random access needed, position-based insertions/removals