ALGORITHMS & DATA STRUCTURES

Lecture 2: Elementary Data Structures

Arrays, Linked Lists, Stacks, and Queues
Based on Sedgwick's Algorithms, 4th Edition

Duration: 4 hours (with breaks)

Lecture: 2 of 12

Textbook Reference: Chapters 1.2 – 1.3

Prerequisites: Lecture 1 — Analysis of Algorithms

Learning Objectives

By the end of this lecture, students will be able to:

1. **Define** the concept of an abstract data type (ADT) and distinguish between an ADT's interface and its implementation.
2. **Explain** the trade-offs between array-based and linked-list-based storage, including time complexity and memory overhead.
3. **Implement** resizing (dynamic) arrays and analyze their amortized cost using the doubling strategy.
4. **Implement** singly and doubly linked lists with insert, delete, and traversal operations.
5. **Implement** stacks using both arrays and linked lists, and apply stacks to expression evaluation (Dijkstra's two-stack algorithm).
6. **Implement** queues using circular buffers and linked lists, and explain their applications in BFS and scheduling.
7. **Describe** the iterator pattern and explain why iterable collections lead to cleaner, more modular code.
8. **Select** the appropriate data structure for a given problem by comparing operations, complexities, and use cases.

1. Abstract Data Types

~30 minutes

1.1 What Is an Abstract Data Type?

DEFINITION

An **abstract data type (ADT)** is a mathematical model for a data type defined by the set of *operations* (behavior) that can be performed on it, rather than by how those operations are implemented. An ADT specifies *what* operations are available and *what* they do, but not *how* they do it.

This distinction is the foundation of good software engineering. When you use a `Stack` in your program, you care that `push()` adds an item and `pop()` removes the most recently added item.

You do *not* care whether the stack uses an array or a linked list internally. This separation of concerns is what ADTs provide.

Interface vs. Implementation

Aspect	Interface (API)	Implementation
Defines	What operations exist; their signatures and contracts	How those operations work internally
Visible to	Client code (users of the ADT)	Only the implementer
Can change?	Rarely (breaking change)	Freely, without affecting clients
Example	`void push(Item item)`	Store `item` at `a[n++]`

1.2 Sedgwick's API Design Principles

Sedgwick advocates a disciplined approach to API design. Each data structure in the textbook is introduced first as an API (a table of method signatures with brief descriptions), and then one or more implementations follow. Key principles include:

1. **Specify the API precisely.** List every public method with its signature and a one-line description of its behavior.
2. **Keep the API minimal.** Include only the operations that are essential. Avoid "kitchen sink" APIs that try to do everything.
3. **Ensure wide applicability.** Design for generality so the same ADT can be used in many contexts (achieved via generics).
4. **Avoid side effects where possible.** A method named `size()` should not modify the data structure.
5. **Separate interface from implementation.** Clients should depend on the API, never on implementation details.

> **SEDGWICK'S APPROACH**
>
> Throughout *Algorithms, 4th Edition*, every major data structure is introduced with a crisp API table before any code appears. This forces us to think about the *contract* before worrying about the *mechanism*.

1.3 Generics and Type Parameters

Early data structure implementations used `Object` types and required casting, which was error-prone. Modern languages (Java 5+, C#, TypeScript, Rust, etc.) support **generics** (also called *type parameters* or *parametric polymorphism*), allowing a single implementation to work with any data type while preserving type safety.

Generic Stack API (Sedgwick style)

```
public class Stack<Item>
-----------------------------------------------------------
         Stack()            create an empty stack
    void push(Item item)    add an item to the top
    Item pop()              remove and return the top item
    Item peek()             return the top item (no removal)
 boolean isEmpty()          is the stack empty?
     int size()             number of items in the stack
```

The type parameter `<Item>` is a placeholder. When a client writes `Stack<String>`, every occurrence of `Item` in the implementation is conceptually replaced by `String`. This gives us:

- **Compile-time type checking** — no `ClassCastException` at runtime.
- **Code reuse** — one implementation works for `String`, `Integer`, `Transaction`, or any reference type.
- **Self-documenting code** — the type makes the intent clear.

> **JAVA CAVEAT**
>
> Java generics do not work with primitive types. You cannot write `Stack<int>`; you must use the wrapper class `Stack<Integer>`. Java's autoboxing handles conversions automatically, but this incurs a small performance overhead due to object creation.

2. Arrays and Dynamic Arrays

~45 minutes

2.1 Fixed-Size Arrays

An **array** is the most fundamental data structure: a contiguous block of memory where each element is the same size and can be accessed by an integer index in **O(1) time**. This constant-time access is possible because the address of element `a[i]` is computed as:

```
address(a[i]) = base_address + i * element_size
```

Properties of Fixed-Size Arrays

Property	Detail
Access by index	O(1)
Search (unsorted)	O(n)
Insert at end (if space)	O(1)
Insert at position *i*	O(n) — must shift elements
Delete at position *i*	O(n) — must shift elements
Memory	Exactly *n* * (element size); no overhead per element
Size	Fixed at creation; cannot grow

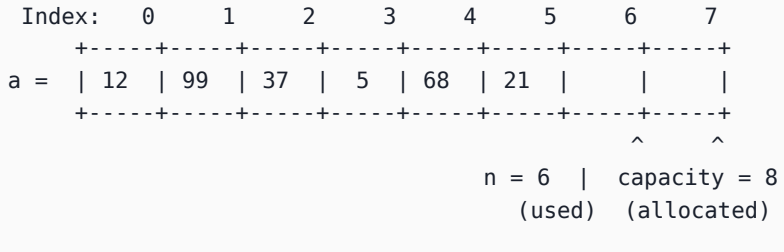

Figure 2.1: A fixed-size array with capacity 8 and 6 elements in use.

The central limitation is obvious: what happens when the array is full and we need to add another element? We must allocate a *new, larger* array and copy all elements over. This is the motivation for **dynamic arrays**.

2.2 Resizing Arrays: The Doubling Strategy

A **resizing array** (also called a *dynamic array*, and known as `ArrayList` in Java or `vector` in C++) automatically grows and shrinks as elements are added or removed. The key question is: *by how much should we grow?*

Strategy 1: Increment by 1 (Naive)

Every time the array is full, allocate a new array of size $n + 1$ and copy all n elements. After inserting n elements starting from an empty array, the total number of copy operations is:

```
1 + 2 + 3 + ... + n = n(n+1)/2 = O(n^2)
```

This means the *average* cost per insertion is O(n). Unacceptable.

Strategy 2: Double the Capacity (Sedgwick's Approach)

When the array is full, allocate a new array of size $2n$ and copy all n elements. Resizing events happen at sizes 1, 2, 4, 8, 16, ..., so the total copy cost after inserting n elements is:

```
1 + 2 + 4 + 8 + ... + n = 2n - 1 = O(n)
```

The *average* cost per insertion is O(n)/n = **O(1) amortized**.

> **AMORTIZED ANALYSIS (INFORMAL)**
>
> **Amortized cost** is the average cost per operation over a worst-case sequence of operations. Even though a single `push` may cost O(n) when it triggers a resize, if we spread that cost over all the cheap O(1) pushes that preceded it, the average is O(1). Think of it like paying a small "tax" on each cheap insertion to save up for the occasional expensive one.

Shrinking: Halve When One-Quarter Full

To avoid *thrashing* (repeatedly growing and shrinking at the boundary), Sedgwick recommends halving the array when it becomes **one-quarter full**, not one-half full. This ensures that after a halving, the array is half full, so a sequence of alternating inserts and deletes at the boundary does not trigger repeated resizes.

INVARIANT

With the double-when-full / halve-when-quarter-full strategy, the array is always between 25% and 100% full. Memory usage is always within a factor of 4 of what is strictly needed.

2.3 Pseudocode: Resizing Array Stack

ResizingArrayStack<Item>

```
class ResizingArrayStack<Item>:
    a = new Item[1]     // underlying array
    n = 0               // number of elements

    function push(item):
        if n == a.length:
            resize(2 * a.length)    // double capacity
        a[n] = item
        n = n + 1

    function pop():
        if isEmpty(): throw "Stack underflow"
        n = n - 1
        item = a[n]
        a[n] = null                 // avoid loitering
        if n > 0 and n == a.length / 4:
            resize(a.length / 2)    // halve capacity
        return item

    function isEmpty():
        return n == 0

    function size():
        return n

    function resize(capacity):
        copy = new Item[capacity]
        for i = 0 to n - 1:
            copy[i] = a[i]
        a = copy
```

LOITERING

After popping an element, we set `a[n] = null`. Without this, the array still holds a reference to the popped object, preventing Java's garbage collector from reclaiming it. This is called **loitering**. Always null out references you no longer need.

2.4 Amortized Cost Analysis (Detailed)

We can prove the amortized O(1) bound rigorously using the **accounting method**:

1. Charge each `push` operation **3 units** of work (the actual cost is 1 unit for the insertion itself).
2. Bank the extra 2 units as "credit" stored with the element.
3. When a resize from capacity k to $2k$ occurs, we need to copy k elements. Each of these k elements was inserted since the *last* resize (or since the array was created), and each contributed 2 credits. Total credits available: $2k \geq k$. This covers the copy cost.

Since we charge exactly 3 units per push and never go into "debt," the amortized cost per push is O(1).

Operation	Best Case	Worst Case	Amortized
push	O(1)	O(n)	**O(1)**
pop	O(1)	O(n)	**O(1)**
peek	O(1)	O(1)	O(1)
size / isEmpty	O(1)	O(1)	O(1)

3. Linked Lists

~60 minutes

3.1 Motivation

Arrays provide O(1) indexed access but O(n) insertions and deletions at arbitrary positions (due to shifting). Linked lists offer a complementary trade-off: **O(1) insertion and deletion** at any position (given a reference to the node), but **O(n) access** to an arbitrary element (because we must traverse from the head).

3.2 Singly Linked Lists

> **DEFINITION**
>
> A **singly linked list** is a sequence of nodes where each node contains a *data element* and a *pointer (reference)* to the next node in the sequence. The last node's next pointer is `null`.

Node Structure

```
class Node:
    Item item       // the data
    Node next       // reference to the next node
```

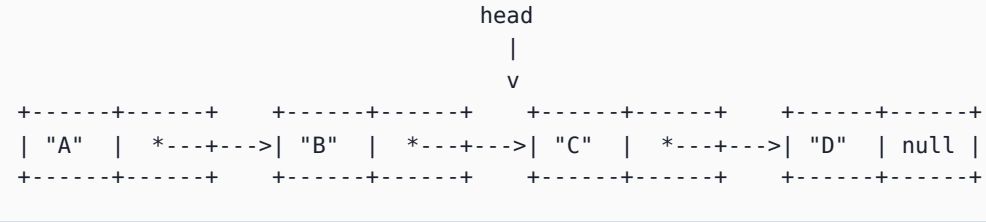

Figure 3.1: A singly linked list with four nodes. Each box has an item and a next pointer.

Operations on Singly Linked Lists

Insert at the front (O(1)):

```
function insertFront(item):
    oldHead = head
    head = new Node()
    head.item = item
    head.next = oldHead
    n = n + 1
```

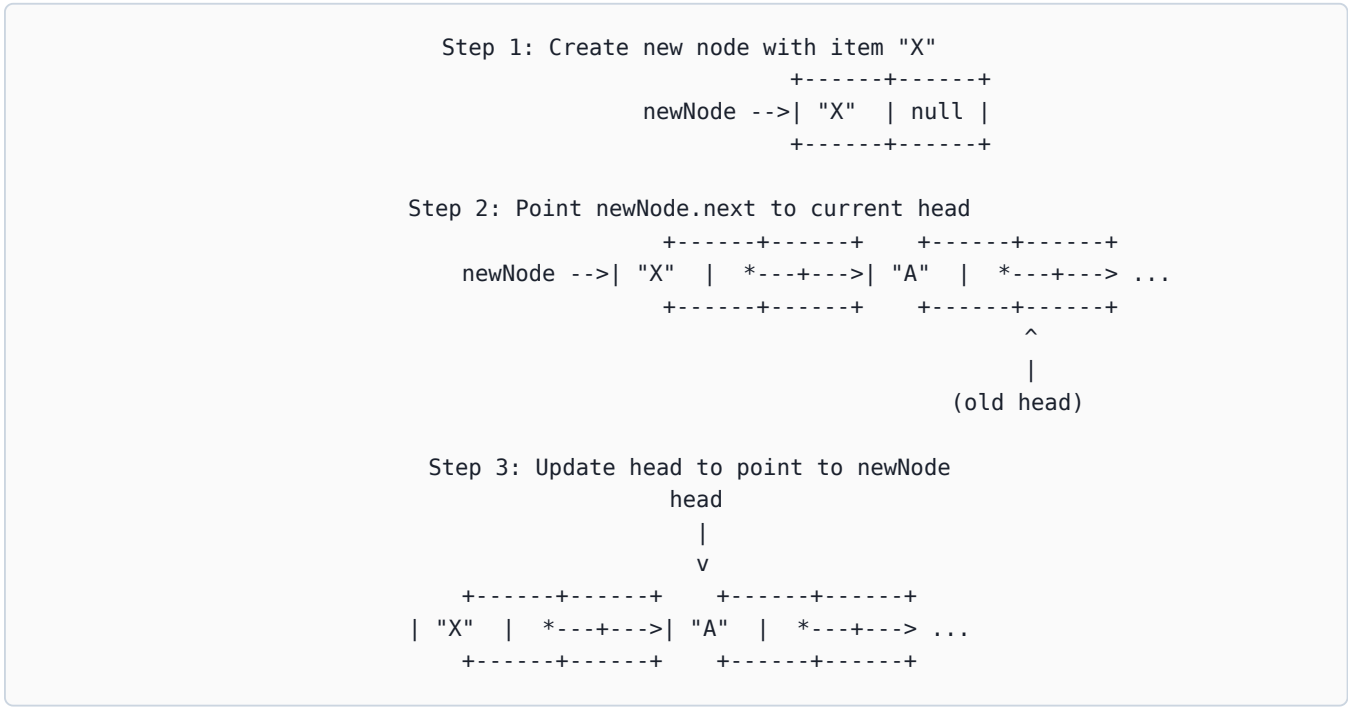

Figure 3.2: Inserting "X" at the front of a singly linked list.

Delete from the front (O(1)):

```
function deleteFront():
    if head == null: throw "List is empty"
    item = head.item
    head = head.next
    n = n - 1
    return item
```

Insert after a given node (O(1) given the node):

```
function insertAfter(node, item):
    if node == null: return
    newNode = new Node()
    newNode.item = item
    newNode.next = node.next
    node.next = newNode
    n = n + 1
```

Delete after a given node (O(1) given the node):

```
function deleteAfter(node):
    if node == null or node.next == null: return null
    target = node.next
    node.next = target.next
    n = n - 1
    return target.item
```

Traverse (O(n)):

```
function traverse():
    current = head
    while current != null:
        process(current.item)
        current = current.next
```

Search (O(n)):

```
function search(key):
    current = head
    while current != null:
        if current.item == key:
            return current
        current = current.next
    return null     // not found
```

3.3 Doubly Linked Lists

A singly linked list has a critical limitation: given a node, we can only move *forward*. Deleting a node requires a reference to the *previous* node, which means traversing from the head. A **doubly linked list** solves this by adding a `prev` pointer.

Node Structure

```
class DNode:
    Item item       // the data
    DNode prev      // reference to the previous node
    DNode next      // reference to the next node
```

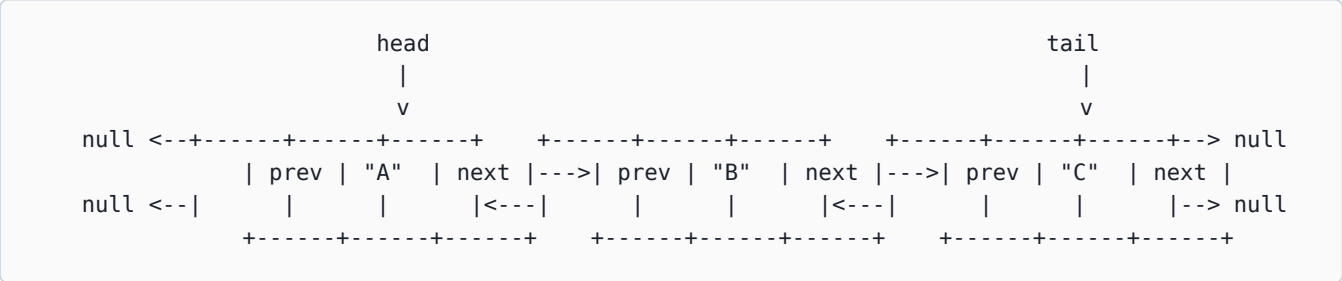

Figure 3.3: A doubly linked list with three nodes.

Insert at front (O(1)):

```
function insertFront(item):
    newNode = new DNode()
    newNode.item = item
    newNode.prev = null
    newNode.next = head
    if head != null:
        head.prev = newNode
    head = newNode
    if tail == null:
        tail = newNode
    n = n + 1
```

Insert at back (O(1)):

```
function insertBack(item):
    newNode = new DNode()
    newNode.item = item
    newNode.next = null
    newNode.prev = tail
    if tail != null:
        tail.next = newNode
    tail = newNode
    if head == null:
        head = newNode
    n = n + 1
```

Delete a given node (O(1) given the node):

```
function deleteNode(node):
    if node.prev != null:
        node.prev.next = node.next
    else:
        head = node.next           // deleting the head
    if node.next != null:
        node.next.prev = node.prev
    else:
        tail = node.prev           // deleting the tail
    n = n - 1
    return node.item
```

> **SENTINEL NODES**
>
> A common technique to simplify doubly linked list code is to use **sentinel** (dummy) nodes at the head and tail. These sentinel nodes are always present, never removed, and hold no real data. They eliminate all the `null` checks in insert/delete operations because every real node always has both a valid `prev` and a valid `next`. Sedgwick mentions this approach but typically uses the simpler null-checking version in the textbook.

3.4 Memory Analysis: Arrays vs. Linked Lists

Understanding memory usage is critical for making informed implementation choices.

Aspect	Array	Singly Linked List	Doubly Linked List
Per-element overhead	None (just the data)	1 pointer (next)	2 pointers (next, prev)
Total for n elements (64-bit pointers, 8-byte data)	$8n$ bytes	$8n + 8n = 16n$ bytes + object overhead	$8n + 16n = 24n$ bytes + object overhead
Java object overhead	~24 bytes for the array	~40 bytes per node (16 header + 8 item ref + 8 next ref + 8 padding)	~48 bytes per node
Wasted space	Up to 75% (resizing array at quarter-full)	0% (allocate per node)	0% (allocate per node)
Cache locality	**Excellent** (contiguous memory)	Poor (nodes scattered in heap)	Poor (nodes scattered in heap)

PRACTICAL CONSIDERATION

In modern hardware, **cache performance** often dominates. Arrays, being contiguous in memory, enjoy excellent spatial locality — accessing one element brings neighboring elements into the CPU cache. Linked list nodes are scattered across the heap, causing frequent cache misses. For this reason, arrays are often faster in practice than linked lists, even for operations where linked lists have better theoretical complexity.

4. Stacks

~45 minutes

4.1 The Stack ADT

> **DEFINITION**
>
> A **stack** is a collection based on the **last-in, first-out (LIFO)** policy. The element most recently added is the first one to be removed. Think of a stack of plates: you add to the top and remove from the top.

Stack<Item> API (Sedgwick)

```
public class Stack<Item> implements Iterable<Item>
-----------------------------------------------------------
         Stack()            create an empty stack
    void push(Item item)    add an item to the top
    Item pop()              remove and return the top item
    Item peek()             return the top item (no removal)
 boolean isEmpty()          is the stack empty?
     int size()             number of items in the stack
```

4.2 Array-Based Implementation

We use the resizing array from Section 2. The "top" of the stack is at index `n - 1`, where `n` is the number of elements. Push appends to the end; pop removes from the end. Both are O(1) amortized.

Array-Based Stack (complete pseudocode)

```
class ArrayStack<Item>:
    a = new Item[1]
    n = 0

    function push(item):
        if n == a.length: resize(2 * a.length)
        a[n] = item
        n = n + 1

    function pop():
        if isEmpty(): throw "Stack underflow"
        n = n - 1
        item = a[n]
        a[n] = null                          // prevent loitering
        if n > 0 and n == a.length / 4:
            resize(a.length / 2)
        return item

    function peek():
        if isEmpty(): throw "Stack underflow"
        return a[n - 1]

    function isEmpty(): return n == 0
    function size():    return n

    function resize(capacity):
        copy = new Item[capacity]
        for i = 0 to n - 1: copy[i] = a[i]
        a = copy
```

4.3 Linked-List-Based Implementation

We use a singly linked list. The "top" of the stack is the head of the list. Push inserts at the front; pop removes from the front. Both are O(1) worst case (no amortization needed).

Linked-List-Based Stack (complete pseudocode)

```
class LinkedStack<Item>:
    head = null
    n = 0

    class Node:
        Item item
        Node next

    function push(item):
        oldHead = head
        head = new Node()
        head.item = item
        head.next = oldHead
        n = n + 1

    function pop():
        if isEmpty(): throw "Stack underflow"
        item = head.item
        head = head.next
        n = n - 1
        return item

    function peek():
        if isEmpty(): throw "Stack underflow"
        return head.item

    function isEmpty(): return head == null
    function size():    return n
```

Comparison of Stack Implementations

Operation	Array-Based	Linked-List-Based
push	O(1) amortized	O(1) worst case
pop	O(1) amortized	O(1) worst case
peek	O(1)	O(1)
Memory per element	~8 bytes (ref only)	~40 bytes (node object)
Worst-case single operation	O(n) (resize)	O(1)
Cache performance	Good	Poor

When to choose which: Use the array-based implementation when memory efficiency and cache performance matter (most situations). Use the linked-list-based implementation when you need guaranteed O(1) worst-case time per operation (e.g., real-time systems where occasional O(n) pauses are unacceptable).

4.4 Applications of Stacks

4.4.1 Expression Evaluation: Dijkstra's Two-Stack Algorithm

One of the most elegant applications of stacks is **Dijkstra's two-stack algorithm** for evaluating fully parenthesized arithmetic expressions. Sedgwick presents this as a compelling example in Section 1.3.

Algorithm: Maintain two stacks — one for *operators* and one for *operands* (values). Process the expression left to right:

1. **Value:** Push onto the value stack.
2. **Operator:** Push onto the operator stack.
3. **Left parenthesis:** Ignore.
4. **Right parenthesis:** Pop an operator and two values, apply the operator to the values, and push the result onto the value stack.

Dijkstra's Two-Stack Algorithm

```
function evaluate(expression):
    ops  = new Stack<String>()     // operator stack
    vals = new Stack<Double>()     // value stack

    for each token in expression:
        if token == "(":
            // ignore
        else if token is an operator (+, -, *, /):
            ops.push(token)
        else if token == ")":
            op = ops.pop()
            v = vals.pop()
            if op == "+": v = vals.pop() + v
            if op == "-": v = vals.pop() - v
            if op == "*": v = vals.pop() * v
            if op == "/": v = vals.pop() / v
            vals.push(v)
        else:
            vals.push(parseDouble(token))

    return vals.pop()
```

Trace example: Evaluate `( ( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) ) )`

Token	Value Stack	Operator Stack	Action
(			ignore
(			ignore
1	1		push value
+	1	+	push operator
(	1	+	ignore
(	1	+	ignore
2	1 2	+	push value
+	1 2	+ +	push operator
3	1 2 3	+ +	push value
)	1 5	+	pop +, pop 3 and 2, push 5
*	1 5	+ *	push operator
(	1 5	+ *	ignore
4	1 5 4	+ *	push value
*	1 5 4	+ * *	push operator
5	1 5 4 5	+ * *	push value
)	1 5 20	+ *	pop *, pop 5 and 4, push 20
)	1 100	+	pop *, pop 20 and 5, push 100
)	101		pop +, pop 100 and 1, push 101

Result: 101

WHY THIS WORKS

The algorithm relies on the expression being *fully parenthesized*. Each closing parenthesis tells us exactly when to apply the most recent operator. The LIFO nature of stacks ensures operators are applied in the correct order. Extending this to handle operator precedence and non-parenthesized expressions leads to the *shunting-yard algorithm* (also by Dijkstra).

4.4.2 Function Call Stack

Every running program uses a stack to manage function calls. When a function is called, its **stack frame** (containing local variables, parameters, and the return address) is pushed onto the call stack. When the function returns, its frame is popped. This naturally handles recursion: each recursive call gets its own stack frame.

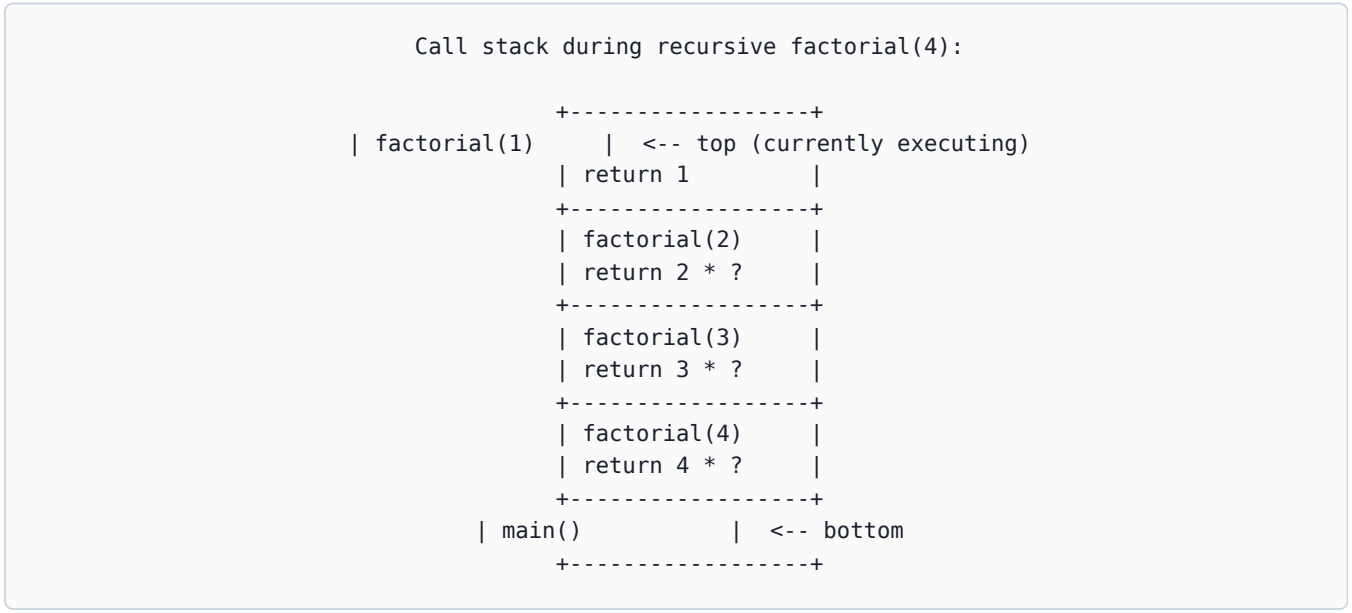

Figure 4.1: The call stack during execution of factorial(4).

4.4.3 Other Stack Applications

- **Undo/Redo in editors:** Each action is pushed onto an "undo" stack. Undoing pops the last action and pushes it onto a "redo" stack.
- **Bracket matching:** Push opening brackets; on each closing bracket, pop and verify it matches. If the stack is empty at the end with no mismatches, brackets are balanced.
- **Back/Forward in browsers:** The back button pops from the history stack and pushes the current page onto the forward stack.
- **Postfix (RPN) evaluation:** Values are pushed; operators pop their operands, compute, and push the result.

- **Depth-first search (DFS):** An iterative DFS uses an explicit stack to track vertices to visit.

5. Queues

~45 minutes

5.1 The Queue ADT

> **DEFINITION**
>
> A **queue** is a collection based on the **first-in, first-out (FIFO)** policy. The element that has been in the queue the longest is the first one to be removed. Think of a line at a ticket counter: people join at the back and are served from the front.

Queue<Item> API (Sedgwick)

```
public class Queue<Item> implements Iterable<Item>
-----------------------------------------------------------
         Queue()              create an empty queue
    void enqueue(Item item)   add an item to the back
    Item dequeue()            remove and return the front item
    Item peek()               return the front item (no removal)
 boolean isEmpty()            is the queue empty?
     int size()               number of items in the queue
```

5.2 Linked-List-Based Implementation

With a linked list, we maintain pointers to both the `head` (front) and `tail` (back). Enqueue appends at the tail; dequeue removes from the head. Both are O(1).

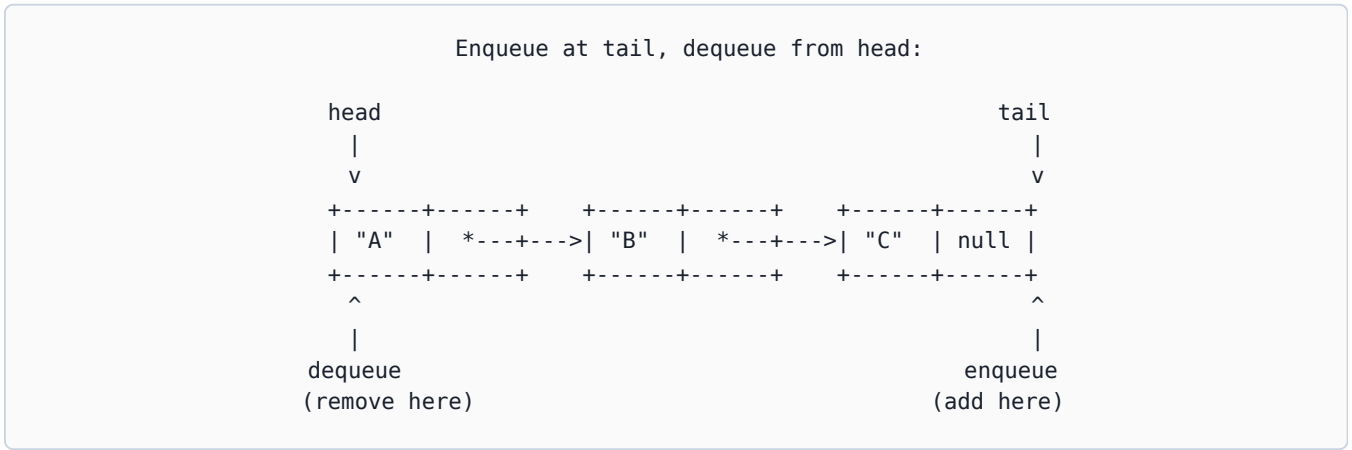

Figure 5.1: A linked-list queue. Items enter at the tail and exit from the head.

Linked-List Queue (complete pseudocode)

```
class LinkedQueue<Item>:
    head = null         // front of queue
    tail = null         // back of queue
    n = 0

    class Node:
        Item item
        Node next

    function enqueue(item):
        oldTail = tail
        tail = new Node()
        tail.item = item
        tail.next = null
        if isEmpty():
            head = tail         // special case: first element
        else:
            oldTail.next = tail
        n = n + 1

    function dequeue():
        if isEmpty(): throw "Queue underflow"
        item = head.item
        head = head.next
        n = n - 1
        if isEmpty():
            tail = null         // special case: last element removed
        return item

    function peek():
        if isEmpty(): throw "Queue underflow"
        return head.item

    function isEmpty(): return head == null
    function size():    return n
```

5.3 Array-Based Implementation: Circular Buffer

A naive array-based queue that always enqueues at index `n` and dequeues from index 0 would require shifting all elements on each dequeue — O(n) per operation. The **circular buffer** (ring buffer) solves this elegantly.

DEFINITION

A **circular buffer** uses a fixed-size array with two indices: `head` (front) and `tail` (back). When either index reaches the end of the array, it wraps around to index 0 using modular arithmetic: `index = (index + 1) % capacity`.

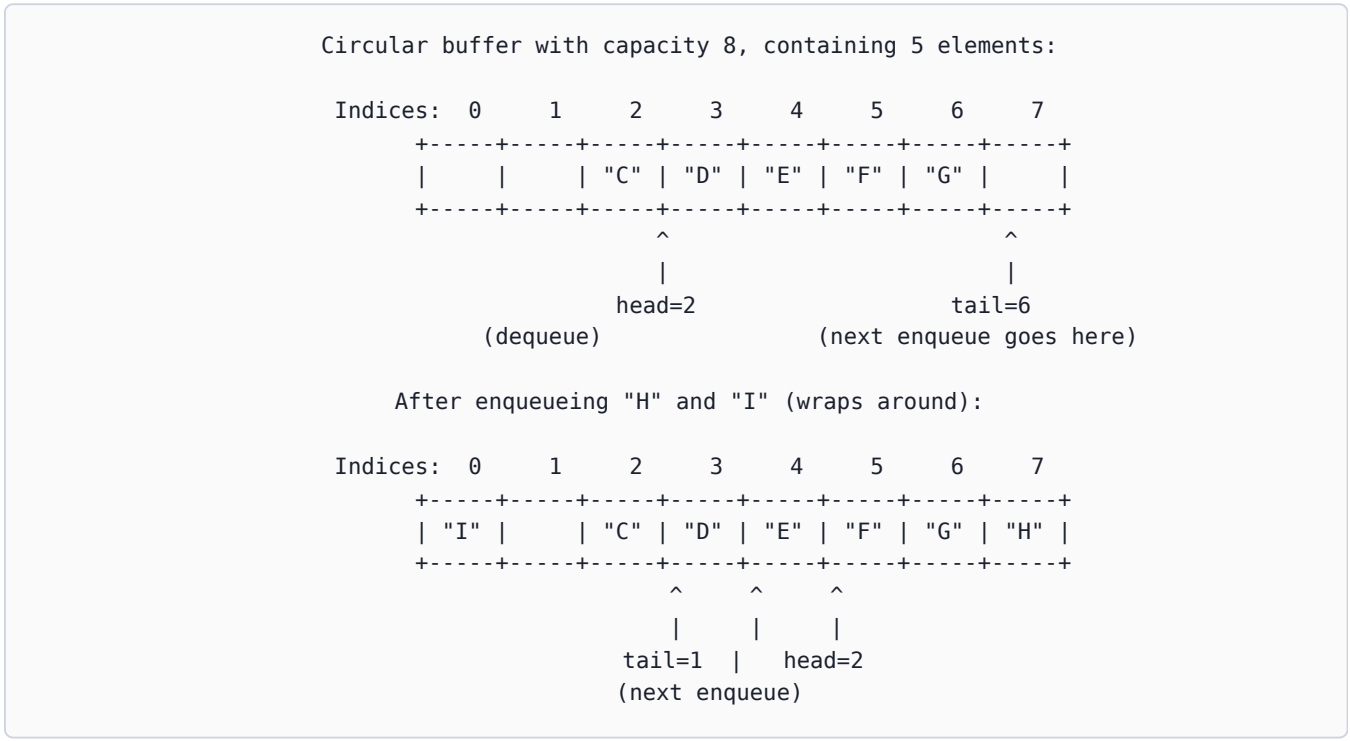

Figure 5.2: A circular buffer wrapping around the end of the array.

Circular Buffer Queue (complete pseudocode)

```
class CircularQueue<Item>:
    a = new Item[8]       // underlying array
    head = 0              // index of front element
    tail = 0              // index of next insertion point
    n = 0                 // number of elements

    function enqueue(item):
        if n == a.length: resize(2 * a.length)
        a[tail] = item
        tail = (tail + 1) % a.length
        n = n + 1

    function dequeue():
        if isEmpty(): throw "Queue underflow"
        item = a[head]
        a[head] = null                // prevent loitering
        head = (head + 1) % a.length
        n = n - 1
        if n > 0 and n == a.length / 4:
            resize(a.length / 2)
        return item

    function peek():
        if isEmpty(): throw "Queue underflow"
        return a[head]

    function isEmpty(): return n == 0
    function size():    return n

    function resize(capacity):
        copy = new Item[capacity]
        for i = 0 to n - 1:
            copy[i] = a[(head + i) % a.length]
        a = copy
        head = 0
        tail = n
```

RESIZE LOGIC FOR CIRCULAR BUFFERS

When resizing a circular buffer, we cannot simply copy the array — elements may wrap around. The resize function must "unwrap" the elements by copying them in logical order (from `head` to `tail`) into the new array starting at index 0. After resizing, `head = 0` and `tail = n`.

5.4 Applications of Queues

5.4.1 Breadth-First Search (BFS)

The canonical application of queues in algorithms is **breadth-first search** on a graph. BFS explores all vertices at distance k before any at distance $k+1$. The queue maintains the "frontier" of vertices to explore.

```
function BFS(graph, source):
    visited = new Set()
    queue = new Queue()
    visited.add(source)
    queue.enqueue(source)

    while not queue.isEmpty():
        v = queue.dequeue()
        for each neighbor w of v:
            if w not in visited:
                visited.add(w)
                queue.enqueue(w)
                // process edge v -> w
```

5.4.2 Other Queue Applications

- **Operating system scheduling:** Processes are placed in a ready queue and served in FIFO order (round-robin scheduling).
- **Print spooling:** Print jobs are queued and processed in order of arrival.
- **Buffering:** Data arriving faster than it can be processed is stored in a queue (e.g., keyboard buffer, network packet buffer).
- **Simulation:** Event-driven simulations use priority queues (a generalization we will study in Lecture 6) to process events in time order.
- **Level-order tree traversal:** Visiting tree nodes level by level uses a queue.

5.5 Deques (Double-Ended Queues)

> **DEFINITION**
>
> A **deque** (pronounced "deck") is a generalization that supports insertion and deletion at *both* ends. It can act as either a stack or a queue.

Deque<Item> API

```
public class Deque<Item> implements Iterable<Item>
-----------------------------------------------------------
         Deque()                create an empty deque
    void addFirst(Item item)    add to the front
    void addLast(Item item)     add to the back
    Item removeFirst()          remove from the front
    Item removeLast()           remove from the back
    Item peekFirst()            examine the front
    Item peekLast()             examine the back
 boolean isEmpty()              is the deque empty?
     int size()                 number of items
```

Implementations:

- **Doubly linked list:** All four add/remove operations are O(1) worst case. This is the most natural implementation.
- **Circular array:** All four operations are O(1) amortized. Use the same modular arithmetic as the circular queue, but allow both `head` to move backward and `tail` to move forward.

When a singly linked list falls short: A singly linked list can efficiently add/remove at the front (head) and add at the back (tail), but `removeLast()` requires traversing the entire list to find the second-to-last node — O(n). This is why deques typically use doubly linked lists or circular arrays.

Comparison of Queue Implementations

Operation	Linked List Queue	Circular Buffer Queue	Deque (Doubly Linked)
enqueue / addLast	O(1)	O(1) amortized	O(1)
dequeue / removeFirst	O(1)	O(1) amortized	O(1)
addFirst	O(1)*	O(1) amortized	O(1)
removeLast	O(n)*	O(1) amortized	O(1)
peek	O(1)	O(1)	O(1)
Memory per element	~40 bytes	~8 bytes	~48 bytes

* Singly linked list; a doubly linked list achieves O(1) for all operations.

6. Iterators and Iterable Collections

~15 minutes

6.1 The Iterator Pattern

When working with collections, we often need to process every element. The **iterator pattern** provides a standard way to traverse a collection without exposing its internal structure. In Java (and Sedgwick's framework), this is the `Iterable` and `Iterator` interface pair.

Java Iterator Interface

```
public interface Iterator<Item> {
    boolean hasNext();      // is there another element?
    Item next();            // return the next element
    void remove();          // (optional) remove the last returned element
}
```

Making a Collection Iterable

```
public interface Iterable<Item> {
    Iterator<Item> iterator();  // return an iterator over the elements
}
```

When a class implements `Iterable`, Java's enhanced for-each loop works:

```
Stack<String> stack = new Stack<>();
stack.push("A");
stack.push("B");
stack.push("C");

// Clean, readable iteration (no index, no hasNext/next calls)
for (String s : stack) {
    System.out.println(s);   // prints C, B, A (LIFO order)
}
```

6.2 Implementing an Iterator for a Linked-List Stack

Stack Iterator (linked-list version)

```
class LinkedStack<Item> implements Iterable<Item>:
    // ... (push, pop, etc. as before) ...

    function iterator():
        return new ListIterator(head)

    class ListIterator implements Iterator<Item>:
        current = null

        ListIterator(first):
            current = first

        function hasNext():
            return current != null

        function next():
            if not hasNext(): throw "No such element"
            item = current.item
            current = current.next
            return item
```

6.3 Implementing an Iterator for a Resizing-Array Stack

Stack Iterator (array version)

```
class ArrayStack<Item> implements Iterable<Item>:
    // ... (push, pop, etc. as before) ...

    function iterator():
        return new ArrayIterator()

    class ArrayIterator implements Iterator<Item>:
        i = n    // start from top of stack

        function hasNext():
            return i > 0

        function next():
            if not hasNext(): throw "No such element"
            i = i - 1
            return a[i]       // traverse in LIFO order
```

6.4 Why Iterators Matter

- **Decoupling:** Client code does not need to know whether the collection uses an array, linked list, or any other structure. The iteration protocol is uniform.
- **Encapsulation:** The internal representation remains hidden. Clients cannot accidentally corrupt the data structure by manipulating indices or pointers.
- **Composability:** Iterators enable functional-style operations (map, filter, reduce) that chain together without materializing intermediate collections.
- **Multiple simultaneous traversals:** Each call to `iterator()` returns a new, independent iterator object, allowing nested iterations over the same collection.

> **CONCURRENT MODIFICATION**
>
> Modifying a collection while iterating over it leads to undefined behavior (or, in Java, a `ConcurrentModificationException` if using `java.util` classes). If you need to remove elements during iteration, use the iterator's own `remove()` method, not the collection's.

7. Summary and Key Takeaways

7.1 The Big Picture

This lecture covered the four most fundamental data structures in computer science. Every more complex structure we study later (binary search trees, hash tables, graphs) is built on top of these primitives. The key insight is that there is no "best" data structure — there are only *trade-offs*.

7.2 Comprehensive Comparison Table

Data Structure	Access by Index	Insert/Delete Front	Insert/Delete Back	Insert/Delete Middle	Memory Overhead	Primary Use Case
Array (fixed)	O(1)	O(n)	O(1)*	O(n)	None	Random access, known size
Resizing Array	O(1)	O(n)	O(1) amort.	O(n)	0–75% waste	Dynamic collections with index access
Singly Linked List	O(n)	O(1)	O(1)†	O(1)‡	1 ptr/node	Stacks, frequent front operations
Doubly Linked List	O(n)	O(1)	O(1)	O(1)‡	2 ptrs/node	Deques, frequent insert/delete at known positions
Stack (LIFO)	Top only	push/pop at top: O(1)			Varies	Undo, DFS, expression eval
Queue (FIFO)	Front only	enqueue at back, dequeue at front: O(1)			Varies	BFS, scheduling, buffering
Deque	Ends only	add/remove at both ends: O(1)			Varies	Sliding window, work stealing

* O(1) only if space is available.

† O(1) insert at back requires maintaining a tail pointer; O(n) delete at back (need predecessor).

‡ O(1) given a reference to the node; O(n) if you must first find the node.

7.3 Key Takeaways

1. **ADTs separate what from how.** Define the interface first, then choose the implementation.
2. **Arrays excel at random access** and have superior cache performance, but are inflexible in size and costly for mid-insertions.
3. **Linked lists excel at dynamic insertions/deletions** at known positions, but sacrifice random access and cache locality.
4. **The doubling strategy** gives resizing arrays O(1) amortized insertions, making them practical for most applications.
5. **Stacks (LIFO) and Queues (FIFO)** are restricted-access data structures that trade generality for simplicity and clarity of purpose.
6. **Choose the right tool for the job:** Need LIFO? Use a stack. Need FIFO? Use a queue. Need both ends? Use a deque. Need random access? Use an array.
7. **Iterators** provide clean, uniform traversal without exposing internal structure.

7.4 Looking Ahead

In Lecture 3, we will use these elementary data structures as building blocks to study **sorting algorithms** (elementary sorts: selection sort, insertion sort, and shellsort). Stacks and queues will reappear throughout the course — in graph algorithms (Lectures 8–9), in symbol tables (Lectures 5–6), and in string processing (Lecture 11).

8. Practice Problems

Problem 1: Stack Operations Trace

Given the following sequence of stack operations (starting from an empty stack), what are the contents of the stack after all operations, and what values were returned by the `pop` operations?

```
push(4), push(7), pop(), push(3), push(9), pop(), pop(), push(1)
```

Solution

Trace step by step:

```
push(4):    Stack: [4]              (bottom to top)
push(7):    Stack: [4, 7]
pop():      Returns 7.  Stack: [4]
push(3):    Stack: [4, 3]
push(9):    Stack: [4, 3, 9]
pop():      Returns 9.  Stack: [4, 3]
pop():      Returns 3.  Stack: [4]
push(1):    Stack: [4, 1]
```

Final stack: [4, 1] (1 is on top). **Returned values:** 7, 9, 3.

Problem 2: Queue Operations Trace

Given the following sequence of queue operations, what are the contents of the queue after all operations, and what values were returned by the `dequeue` operations?

```
enqueue("A"), enqueue("B"), dequeue(), enqueue("C"), enqueue("D"), dequeue(), enqueue(
```

Solution

```
enqueue("A"):  Queue: [A]            (front to back)
enqueue("B"):  Queue: [A, B]
dequeue():     Returns "A". Queue: [B]
enqueue("C"):  Queue: [B, C]
enqueue("D"):  Queue: [B, C, D]
dequeue():     Returns "B". Queue: [C, D]
enqueue("E"):  Queue: [C, D, E]
```

Final queue: [C, D, E] (C is at the front). **Returned values:** "A", "B".

Problem 3: Resizing Array Analysis

A resizing array starts with capacity 1. How many times will the array be resized (doubled) when inserting the first 33 elements? What is the total number of element copies performed across all resize operations?

Solution

Resizes occur when the array is full. The capacity sequence is:

```
Capacity 1  -> full after 1 element   -> resize to 2  (copy 1)
Capacity 2  -> full after 2 elements  -> resize to 4  (copy 2)
Capacity 4  -> full after 4 elements  -> resize to 8  (copy 4)
Capacity 8  -> full after 8 elements  -> resize to 16 (copy 8)
Capacity 16 -> full after 16 elements -> resize to 32 (copy 16)
Capacity 32 -> full after 32 elements -> resize to 64 (copy 32)
```

Number of resizes: 6 (at sizes 1, 2, 4, 8, 16, 32).

Total copies: 1 + 2 + 4 + 8 + 16 + 32 = 63.

For 33 insertions, total work = 33 (insertions) + 63 (copies) = 96. Amortized cost per insertion = 96/33 ≈ 2.9, confirming the O(1) amortized bound.

Problem 4: Reverse a Linked List

Write pseudocode to reverse a singly linked list in place (i.e., without creating new nodes). Analyze the time and space complexity.

Solution

```
function reverse(head):
    prev = null
    current = head
    while current != null:
        next = current.next      // save next node
        current.next = prev      // reverse the pointer
        prev = current           // advance prev
        current = next           // advance current
    return prev                  // new head
```

Time complexity: O(n) — single pass through the list.

Space complexity: O(1) — only three pointer variables regardless of list size.

Visual trace for list A -> B -> C -> null:

```
Start:        null    A -> B -> C -> null
                ^      ^
              prev  current

Step 1:       null <- A    B -> C -> null
                      ^    ^
                    prev  current

Step 2:       null <- A <- B    C -> null
                           ^    ^
                         prev  current

Step 3:       null <- A <- B <- C
                               ^    null
                             prev  current

Return prev (C), which is the new head.
```

Problem 5: Balanced Parentheses

Using a stack, write pseudocode to determine whether a string of parentheses `(`, `)`, `[`, `]`, `{`, `}` is balanced. For example, `"({[]})"` is balanced but `"({[}])"` is not.

Solution

```
function isBalanced(str):
    stack = new Stack<Character>()

    for each char c in str:
        if c in {'(', '[', '{'}:
            stack.push(c)
        else if c in {')', ']', '}'}:
            if stack.isEmpty():
                return false     // closing bracket with no opener
            top = stack.pop()
            if not matches(top, c):
                return false     // mismatched bracket types
        // ignore non-bracket characters

    return stack.isEmpty()       // all openers must be closed

function matches(open, close):
    return (open == '(' and close == ')')
        or (open == '[' and close == ']')
        or (open == '{' and close == '}')
```

Time complexity: O(n) — single pass through the string.

Space complexity: O(n) — worst case, all characters are opening brackets.

Problem 6: Implement a Queue Using Two Stacks

Design a queue implementation that uses two stacks as its only internal data structures. Analyze the amortized time complexity of `enqueue` and `dequeue`.

Solution

```
class TwoStackQueue<Item>:
    inbox = new Stack<Item>()     // for enqueue
    outbox = new Stack<Item>()    // for dequeue

    function enqueue(item):
        inbox.push(item)          // always push to inbox

    function dequeue():
        if outbox.isEmpty():
            // transfer all elements from inbox to outbox
            // this reverses the order, giving FIFO behavior
            while not inbox.isEmpty():
                outbox.push(inbox.pop())
        if outbox.isEmpty():
            throw "Queue underflow"
        return outbox.pop()

    function isEmpty():
        return inbox.isEmpty() and outbox.isEmpty()

    function size():
        return inbox.size() + outbox.size()
```

Analysis: Each element is pushed and popped at most twice: once onto `inbox`, once from `inbox` to `outbox`, once from `outbox`. By the accounting method, charge each `enqueue` 3 units (1 for its own push, 2 saved for the future transfer). Every `dequeue` uses at most 1 pre-paid unit. Thus:

- `enqueue`: O(1) amortized
- `dequeue`: O(1) amortized (even though a single dequeue can be O(n) when it triggers a transfer)

Problem 7: Dijkstra's Two-Stack Algorithm

Use Dijkstra's two-stack algorithm to evaluate the expression: `( ( 3 + 2 ) * ( 8 - 3 ) )`. Show the contents of both stacks after processing each token.

Solution

Token	Value Stack	Operator Stack	Action
(			ignore
(			ignore
3	3		push value
+	3	+	push operator
2	3 2	+	push value
)	5		pop +, pop 2 and 3, compute 3+2=5, push 5
*	5	*	push operator
(	5	*	ignore
8	5 8	*	push value
-	5 8	* -	push operator
3	5 8 3	* -	push value
)	5 5	*	pop -, pop 3 and 8, compute 8-3=5, push 5
)	25		pop *, pop 5 and 5, compute 5*5=25, push 25

Result: 25

Problem 8: Memory Analysis

In a 64-bit Java environment (8-byte references, 16-byte object overhead, 8-byte alignment), calculate the total memory used by a linked-list stack containing $n = 1000$ `Integer` objects. Compare with a resizing array stack that is exactly half full (i.e., 1000 elements in an array of capacity 2000).

Solution

Linked-list stack:

- Each `Node` object: 16 (object overhead) + 8 (item ref) + 8 (next ref) = 32 bytes, padded to 40 bytes (inner class has 8-byte implicit reference to outer class).
- Each `Integer` object: 16 (object overhead) + 4 (int value) = 20 bytes, padded to 24 bytes.
- Stack object itself: 16 (overhead) + 8 (head ref) + 4 (n) + 4 (padding) = 32 bytes.
- **Total:** 32 + 1000 * (40 + 24) = 32 + 64,000 = **64,032 bytes (~62.5 KB)**.

Resizing array stack (half full):

- Array object: 16 (overhead) + 8 (length field) + 2000 * 8 (references) = 16,024 bytes, padded to 16,024.
- 1000 `Integer` objects: 1000 * 24 = 24,000 bytes.
- Stack object itself: 16 (overhead) + 8 (array ref) + 4 (n) + 4 (padding) = 32 bytes.
- **Total:** 32 + 16,024 + 24,000 = **40,056 bytes (~39.1 KB)**.

The array-based stack uses about 37% less memory, even when only half full. The 1000 unused array slots (8 bytes each = 8 KB) are still less wasteful than the 40-byte per-node overhead of the linked list.

References

1. Sedgwick, R. & Wayne, K. *Algorithms, 4th Edition*. Addison-Wesley, 2011. Sections 1.2 (Data Abstraction) and 1.3 (Bags, Queues, and Stacks).
2. Sedgwick, R. & Wayne, K. Companion website: https://algs4.cs.princeton.edu/home/
3. Cormen, T.H., Leiserson, C.E., Rivest, R.L., & Stein, C. *Introduction to Algorithms*, 4th Edition. MIT Press, 2022. Chapters 10–11 (Elementary Data Structures).

4. Knuth, D.E. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd Edition. Addison-Wesley, 1997. Section 2.2 (Linear Lists).