sourcemind

Data Structures and Algorithms

Lionel KITIHOUN

Session 5

Basic Data Structures

sourcemind

Objectives

- Learn the basic data structures and implement them.
- Visualize general concepts of data structures, differences and similarities.

Summary

- Introduction
- Linked lists
- Stacks
- Queues

Method

Introduce the data structures with examples that show typical use cases to help understand their behaviour and usefulness.

Introduction

Typically, programs need to store the data they need to process in a form that allows rapid processing.

Basic data structure requirements

Whatever the form in which data is stored, the structure must at least allow the following operations:

- Add an item.
- Return the next item to process.
- Get how many items are currently stored.

Basic data structures

Depending on the requirements of an application, we may need to process data in some order:

- Sequentially (one at a time).
- In chronological order (first in, first out).
- In reverse chronological order (last in, last out).
- Based on some priority (most urgent first).

Linked list

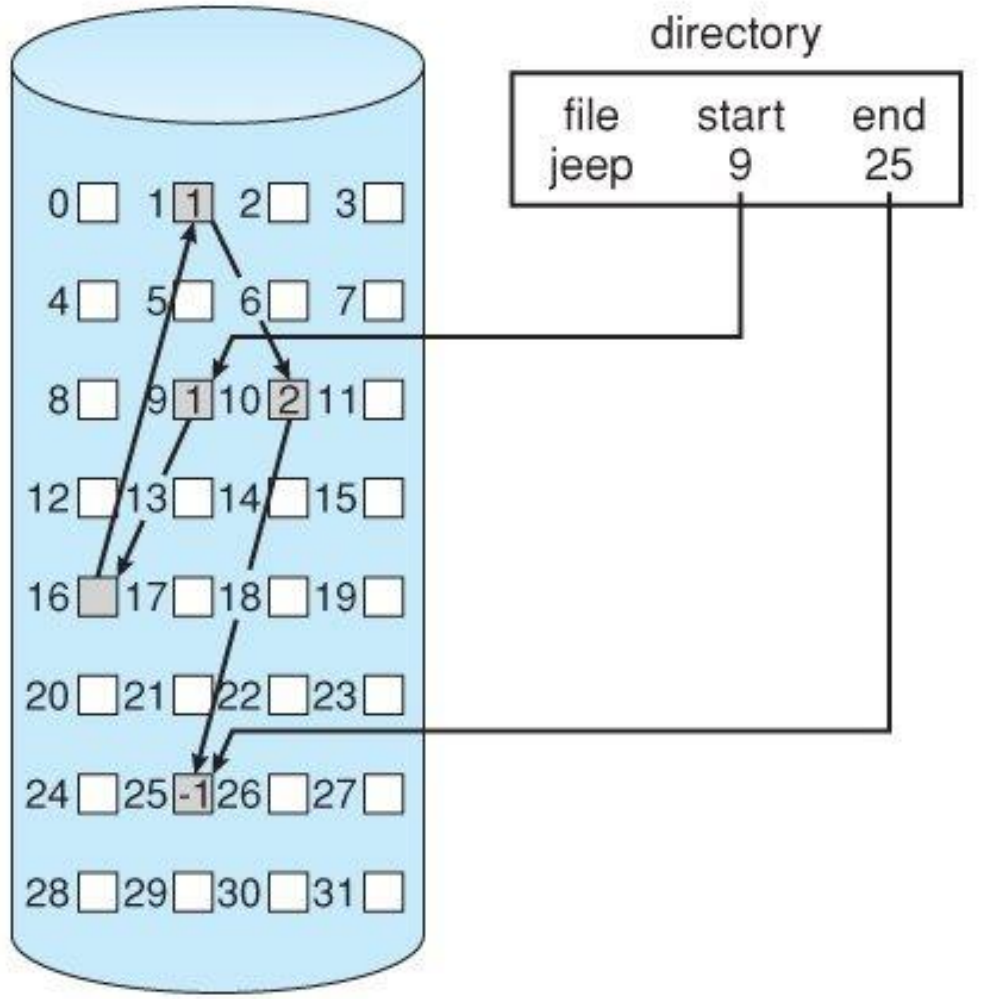
sourcemind

Problem: filesystem storage allocation

You are writing a program to manage disk space allocation for files.

When a user needs to save a file, you have to find enough blocks on the disk to store the data. Ideally, the blocks should be “contiguous”.

But sadly, it is not usually the case and the blocks holding the data of a file can be spread on the disk.



Source: https://www.quora.com/How-do-I-describe-traditional-linked-allocation

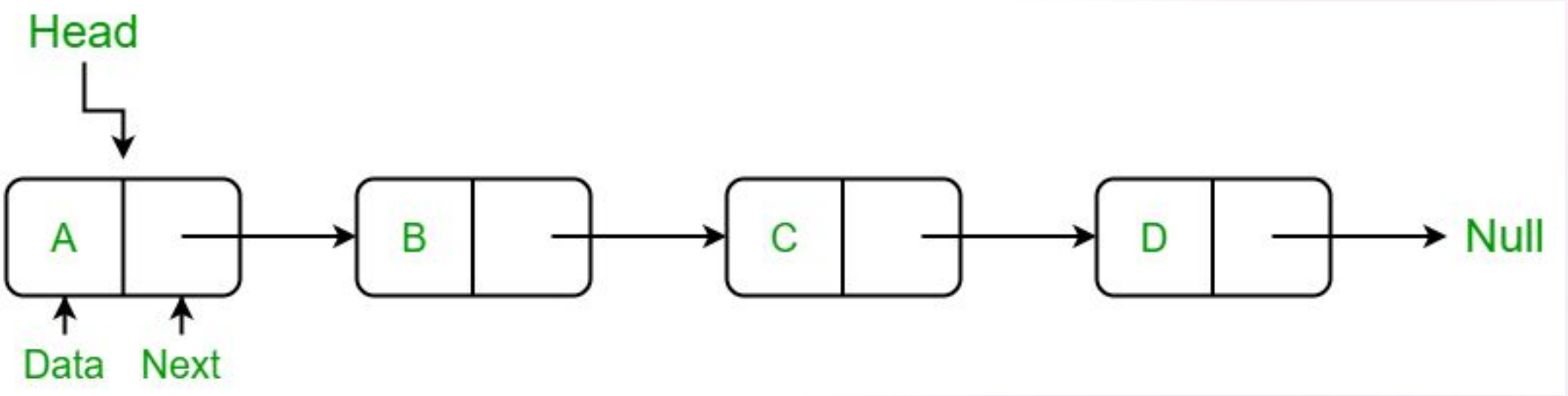
Problem: filesystem storage allocation

How can you model this situation to enable your program to handle its duty in an efficient way?

Introducing linked lists

A linked list is a collection of “nodes” connected together via links.

These nodes consist of the data to be stored and a pointer to the address of the next node within the linked list.



https://www.geeksforgeeks.org/data-structures/linked-list/singly-linked-list

Linked list to model disk blocks

- Each node will represent a block of data on the disk.
- The “next” attribute will be a pointer to the next node (block).

```
public class File {

   String  path;
   Block   start;
   int     mask;

   static class Block {
      static final int BLOCK_SIZE = 512;

      private int    address;
      private byte[] bytes;
      private Block  next;
   }
}
```

Standard lib implementation of linked list

Linked list is available in the Java standard library.

- `java.util.LinkedList`
- Generic class

We will however see how to implement it.

```
class LinkedList<T> {
    private Node<T> head, last;
    static class Node<T> {
        public T       data;
        public Node<T> next;
        public Node(T data) {
            this.data = data;
            this.next = null;
        }
    }
    public          LinkedList();
    public void     insert(T data);
    public Node<T> getHead();
    public Node<T> getLast();
}
```

Insert an element in a list

```
public void insert(T data) {
   Node node = new Node(data);

   if (this.head == null) {
      this.head = node;
   } else {
      this.last.next = node;
   }
   this.last = node;
}
```

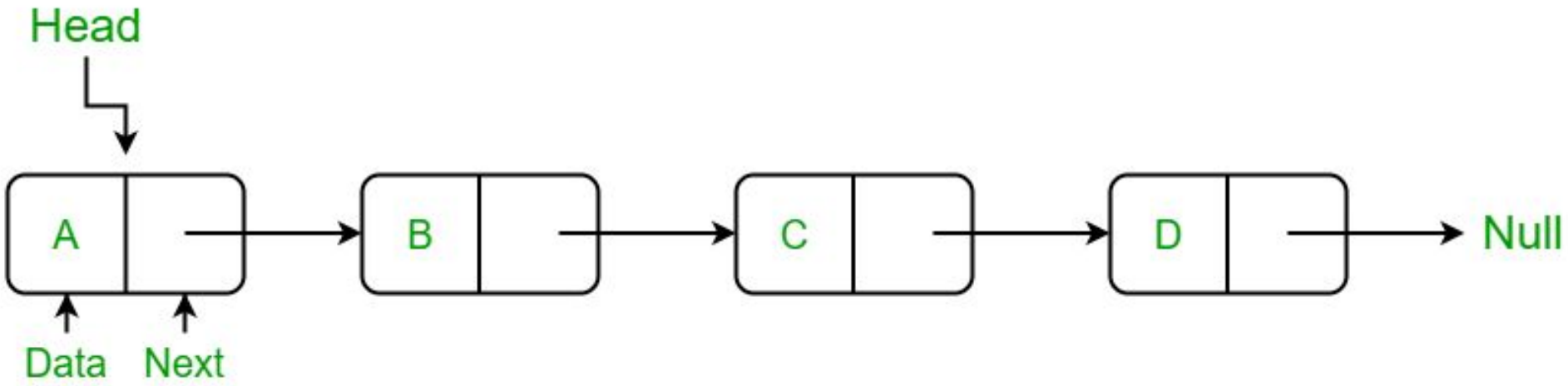
Iterate over a list

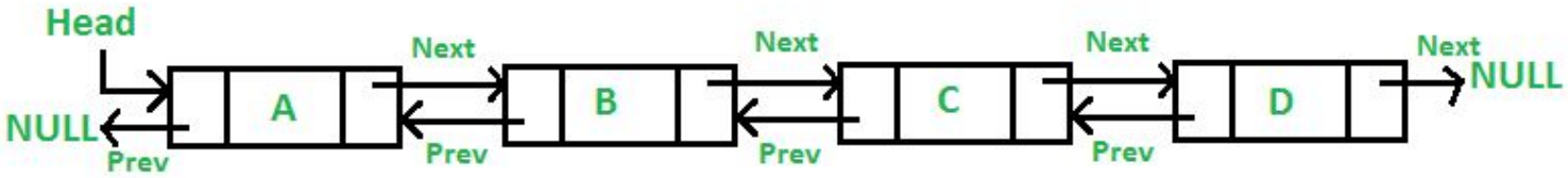
```
var node list.getHead();
while (node != null) {
   var data = node.getData();
   // Process the data here...
   node = node.next;
}
```

Single ou double

There are two types of linked lists.

- Singly linked list, where each node contains a link to the next node.
- Doubly linked list, where each node contains links to the previous and next nodes.


Head
A
B
C
D
Null
Data
Next


Head
NULL
Prev
A
Next
Prev
B
Next
Prev
C
Next
Prev
D
Next
NULL

Single vs double

```
public class List<T> {

   static class Node<T> {

      private T         data;

      private Node<T>   next;
   }
}
```

```
public class List<T> {

   static class Node<T> {

      private T         data;

      private Node<T> next;

      private Node<T> previous;
   }
}
```

Performance of linked list operations

- Add an element: **O(N)** but can be done in **O(1)** if we keep a reference to the last node.
- Remove an item: **O(N)**.
- Iterate: **O(N)**.
- Size: **O(N)** but can be done in **O(1)** if we keep track of the size.

Stack

sourcemind

Introduction

Suppose you are a pioneer in CS and you are creating a new language.

As part of your work, you need an algorithm that parses and evaluates arithmetic expressions that consists only of integers and the four binary operators.

Example

5 + 6 * 8 + 2 - 8 / 4

Algorithm

Instead of reinventing the wheel, you do some research and find an article of your professor who describe an algorithm to do that.

The algorithm runs in two rounds and uses something called a **bag**.

First round

Scan the expression from left to right.

If the current token is a number, addition, or subtraction operator, put it in the **bag**.

If the token is a multiplication or division operator, retrieve the last number you put in our **bag**, get the token that follows the operator, and put the result of the operation in the bag.

Second round

At this point, there should be only numbers, addition and subtraction operators in the bag.

While there is more than item in your bag, get the last three items you put in the bag. These should be two number and an operator. Compute the result and put the result back in the bag.

Result

The only number remaining in the bag at the end of this is the result of the expression.

Simulation

5 + 6 * 8 + 2 - 8 / 4

Simulation

5 + 6 * 8 + 2 - 8 / 4

Simulation

5 + 6 * 8 + 2 - 8 / 4

Simulation

5 + 6 * 8 + 2 - 8 / 4

Simulation

5 + 6 * 8 + 2 - 8 / 4

Simulation

5 + 6 * 8 + 2 - 8 / 4

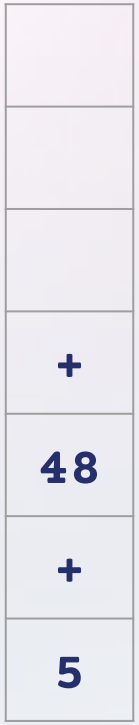
6 * 8 = 48

Simulation

5 + 6 * 8 + 2 - 8 / 4

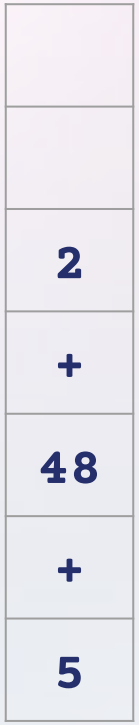
Simulation

5 + 6 * 8 + 2 - 8 / 4



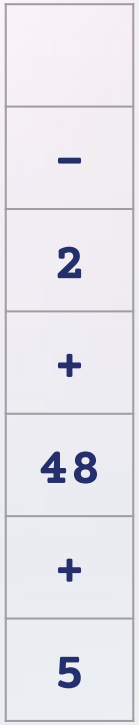
Simulation

5 + 6 * 8 + 2 - 8 / 4



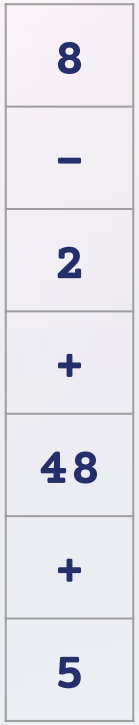
Simulation

5 + 6 * 8 + 2 - 8 / 4



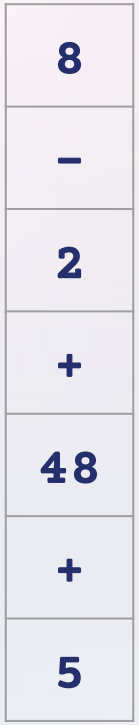
Simulation

5 + 6 * 8 + 2 - 8 / 4



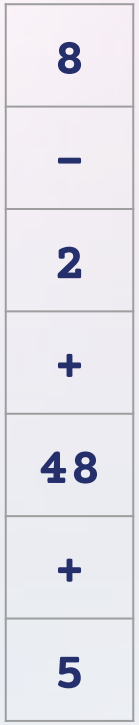
Simulation

5 + 6 * 8 + 2 - 8 / 4



Simulation

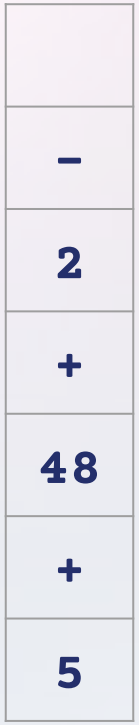
5 + 6 * 8 + 2 - 8 / 4



Simulation

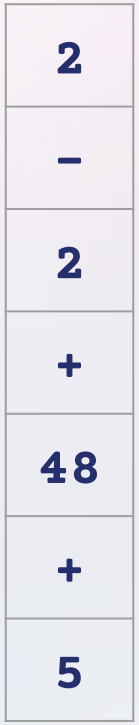
5 + 6 * 8 + 2 - 8 / 4

8 / 4 = 2



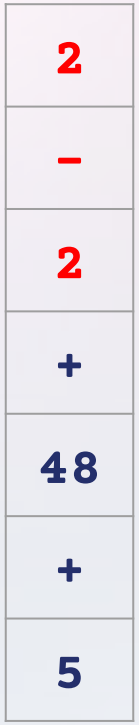
Simulation

5 + 6 * 8 + 2 - 8 / 4



Simulation

2 - 2 = 0



sourcemind

Simulation

Simulation

48 + 0

Simulation

48 + 0 = 48

Simulation

Simulation

5 + 48 = 53

Simulation

5 + 48 = 53

53

sourcemind

Simulation

There is only one number remaining.

53

Simulation result

The algorithm gave the correct answer.

The key element of the algorithm is the **bag** from which we must always extract the last element we inserted.

LIFO

The bag is LIFO (last-in, first-out) data structure and is called a stack.

Stacks

Stack are really useful in CS and have various applications.

- Evaluation of expression in language
- Function calls
- Undo/redo operations

Stack operations

Stacks offer two typical operations:

- **push**, to add an item at the top of the stack.
- **pop**, to retrieve the last element pushed on the stack, i.e. the top.

Implementation

A stack can simply be implemented with an array.

All operations take constant time.

```
class Stack<T> {
    private T[] items;
    private int maxSize;
    private int size;
    public Stack(int maxSize) {
        this.items = new T[maxSize];
        this.maxSize = maxSize;
        this.size = 0;
    }
    public T pop();
    public void push(T item) throws Exception;
}
```

Stack push implementation

```
public void push(T item) {

    if (this.size >= this.maxSize) {
        throw new Exception("Stack max size reached");
    }

    this.items[this.size] = item;
    this.size += 1;
}
```

Stack pop implementation

```
public T pop() {

    if (this.size == 0) {
        throw new Exception(“Stack is empty”);
    }

    this.size -= 1;

    T item = this.items[this.size];

    this.items[this.size] = null;

    return item;
}
```

Other stack operations

- `isEmpty()`
- `size()`
- `top()`

Complexity of stack operations

- Push **O(1)**
- Pop **O(1)**
- Get the size **O(1)**

Queues

Definition

Queue is a linear data structure, just like the stack, in which elements are inserted from one end called the **rear** (also called **tail**), and the removal of an element takes place from the other end called as **front** (also called **head**).

This makes queue a FIFO (First In First Out) data structure, which means that element inserted first will be removed first.

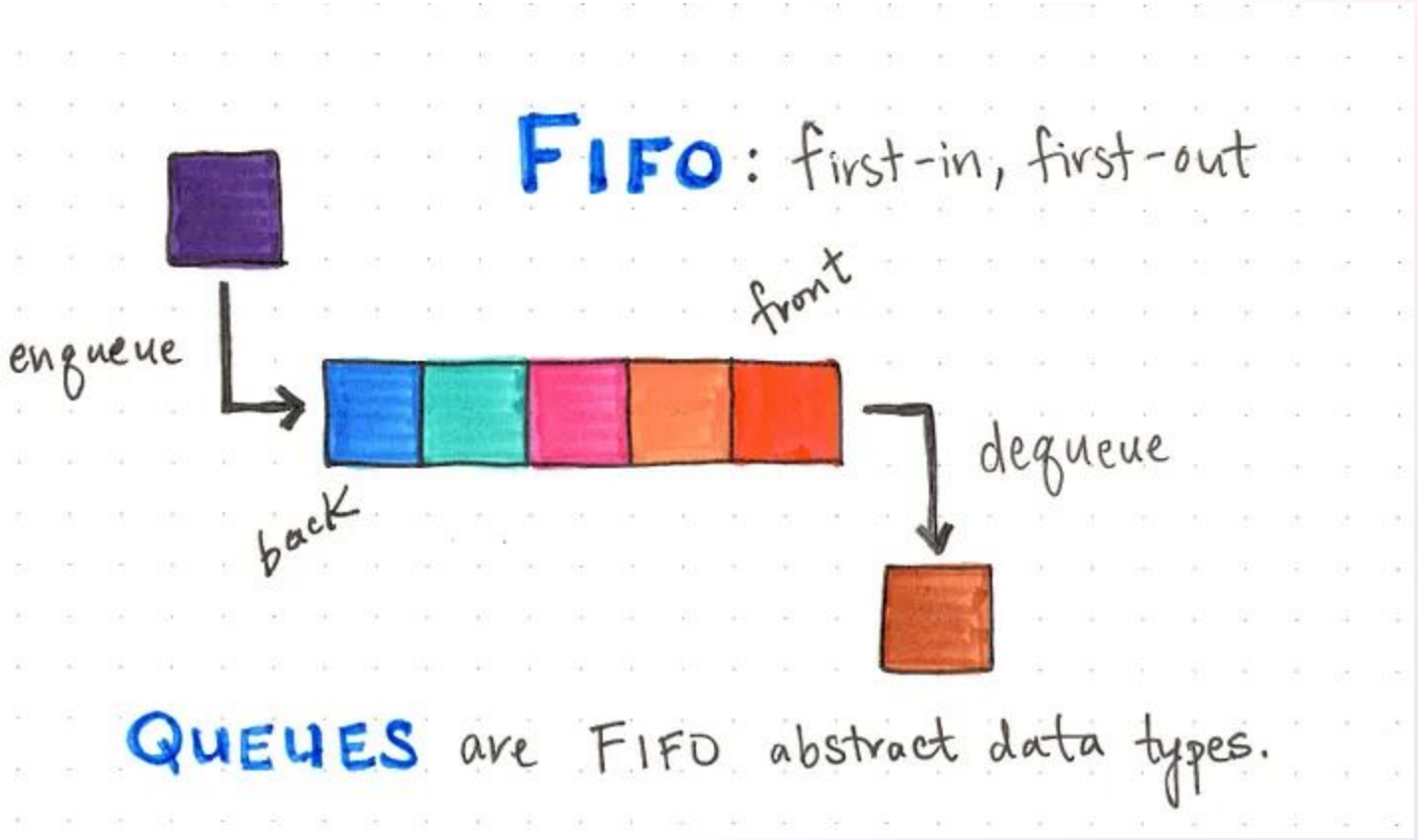
Queue in real-life

A queue is like a line waiting at the bank, where the first person in line is the first person served.

When you enter, you place yourself at the end of the queue and wait your turn.

Concepts

The operation of adding an element to the end of the queue is known as **enqueue**, and the operation of removing an element from the front is known as **dequeue**.



Source: https://medium.com/mlearning-ai/queue-data-structure-65cc3a1edc02

Queue operations

- `enqueue()`
- `dequeue()`
- `peek()`, `front()`
- `size()`
- `capacity()`

```
class Queue<T> {
    private T[] items;
    private int maxSize;
    private int size;
    private int front;
    private int back;

    public Queue(int maxSize) {
        this.items = new T[maxSize];
        this.maxSize = maxSize;
        this.size = 0;
        this.front= 0;
        this.back = 0;
    }
}
```

```
public void enqueue(T item) throws Exception {

      if (this.size === this.maxSize) {
            throw new Exception(“Queue is full”)
      }

      this.size += 1;
      this.items[this.back] = item;
      this.back = (this.back - 1) % this.maxSize;
}
```

```
public T dequeue() throws Exception {
        if (this.size === 0) {
                throw new Exception("Queue is empty")
        }

        T item = this.items[this.front];

        this.items[this.front] = null;
        this.size -= 1;
        if (this.size == 0) {
                this.front = 0;
                this.back = 0;
        } else {
                this.front = (this.front + 1) % this.maxSize;
        }

        return item;
}
```

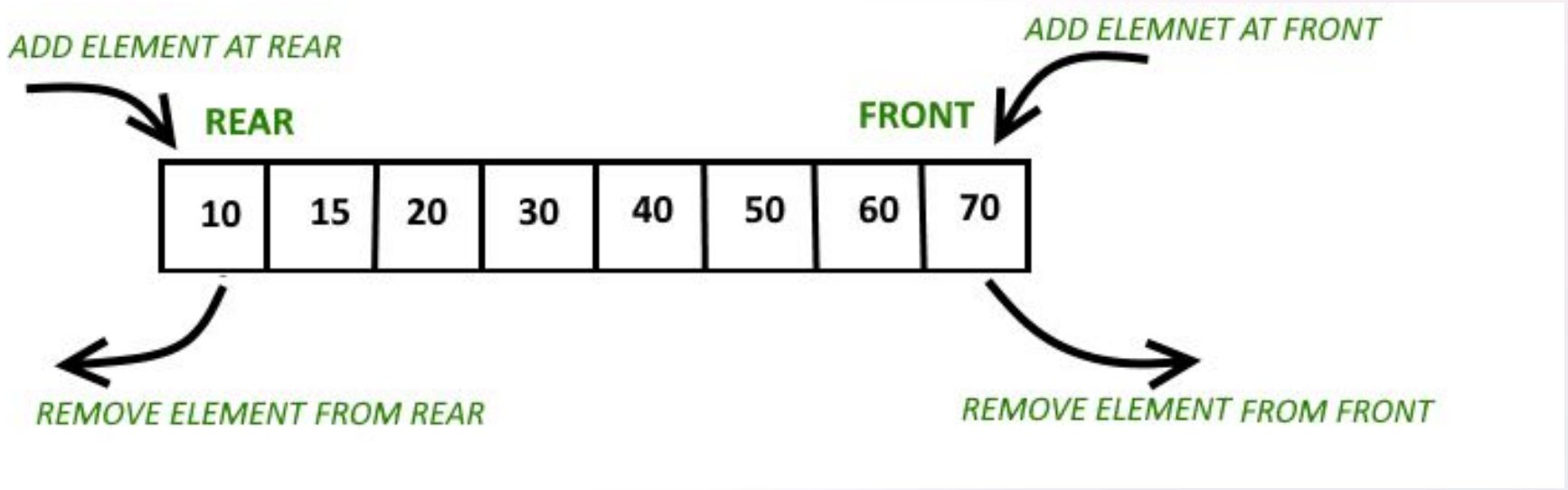
Other types of queue

- Deque (double-ended queue)
- Priority queues

Deque

In this special type of queue, elements can be added to or removed from either the front or back.

Deque



Source: https://media.geeksforgeeks.org/wp-content/uploads/anod.png

Deque operations

- `enqueueLeft()`
- `enqueueRight()`
- `dequeueLeft()`
- `dequeueRight()`
- `size()`
- `isEmpty()`

Deque in Java

Deque are available in Java through the `java.util.Deque` interface and classes that implement it.

Priority queue

In a priority queue, each element is associated with a priority.

Elements are arranged so that elements with higher priority values are typically retrieved before elements with lower priority values.

Like a regular queue, elements are added to one end and removed from the other.

Implementation

Priority queues are typically implemented using **heaps**, to allow fast insertion or retrieval of elements.

Priority queues in Java

Available through the `java.util.PriorityQueue` class.

Complexity of queue operations

Operation	Queue	Deque	Priority queue
Add	O(1)	O(1)	O(log N)
Remove	O(1)	O(1)	O(log N)
Get size	O(1)	O(1)	O(1)