LINFO1121: Algorithms and Data Structures — Summary

Pierre Schaus & Guillaume Derval
December 24, 2025

Disclaimer & Learning Resources

These slides are a **summary** only. They do not contain the full course material required for the exam. The codes are Java-like pseudo-codes (most of them won't compile).

Official Course Content:

- **Theory:** The official website (slides, chapters to read, theoretical exercises and links to inginious exercises)
  `https://pschaus.github.io/LINFO1121/`
- **Practice:** All **INGInious** exercises:
  `https://inginious.info.ucl.ac.be/course/LINFO1121`
- **Reference:** Related chapters of *Algorithms, 4th Edition* by **Sedgewick & Wayne**.

Course Material = Website + INGInious + Book Chapters

Contributions are welcome! Submit a Pull Request at:
`https://github.com/pschaus/LINFO1121-summary`

Special thanks to Gemini 3.0 Pro for assistance with formatting and the generation of Java-like pseudo-code.

The Iterable and Iterator Pattern

The Interfaces

- **Iterable<T>**: A collection that can be the target of a "for-each" loop. Requires method `iterator()`.
- **Iterator<T>**: The object that tracks the state of the traversal.
- **Why Inner Classes?** The iterator needs access to the private array/data of the container.
- Although *delete* method is part of the *Iterator* interface, we generally do not implement it (not trivial and optional).

```
public class MyList<T> implements Iterable<T> {
    private T[] items;
    private int size;

    @Override
    public Iterator<T> iterator() {
        return new ListIterator();
    }

    private class ListIterator implements Iterator<T> {
        private int cursor = 0;

        public boolean hasNext() {
            return cursor < size;
        }

        public T next() {
            if (!hasNext()) throw new NoSuchElementException();
            return items[cursor++];
        }
    }
}
```

Iteration Strategies: Fail-Fast vs. Fail-Safe

The Problem: Structural changes (add/remove) during iteration can lead to undefined behavior.

1. **Fail-Fast (frequent in Java collections)**
   - Throws `ConcurrentModificationException` immediately if the collection changes.
   - **Pros:** Detects bugs early; no memory overhead.
   - **Cons:** Unsuitable for highly concurrent environments.
2. **Fail-Safe**
   - Works on a **snapshot/copy** of the data.
   - **Pros:** No exception thrown; thread-safe.
   - **Cons:** Overhead of copying; might not see recent updates.

```java
public class MyList<E> {
    // Structural version of the list
    protected int modCount = 0;

    public Iterator<E> iterator() {
        return new Itr(); // Snapshot taken here!
    }

    private class Itr implements Iterator<E> {
        // Snapshot of modCount at creation
        int expectedModCount = modCount;

        public E next() {
            checkForComodification();
            // ... return next element
        }

        final void checkForComodification() {
            // Compare current state vs creation state
            if (modCount != expectedModCount)
                throw new ConcurrentModificationException();
        }
    }
}
```

Available implementations of ADTs in Java langage

Arrays (non-dynamic)

Usefulness	Storage of a known number of elements
Constraint	Fixed number of elements
Instantiation	`new Type[size]` (e.g.: `new int[8]`)

Methods

Name	*Java*	*Complexity*
Access element `i`	`array[i]`	$\Theta(1)$
Assign `x` to element `i`	`array[i] = x`	$\Theta(1)$

Arrays (dynamic)

Usefulness	Storage of elements, random access
Instantiation	`new ArrayList<Type>()`

Methods on an object `o` of size n

Name	*Java*	*Complexity*
Access element `i`	`o.get(i)`	$\Theta(1)$
Assign `x` to element `i`	`o.set(i, x)`	$\Theta(1)$
Add at the end	`o.add(x)`	$\Theta(1)$ amortized
Remove from the end	`o.remove(n-1)`	$\Theta(1)$ amortized
Add anywhere	`o.add(i, x)`	$\mathcal{O}(n)$
Remove from anywhere	`o.remove(i)`	$\mathcal{O}(n)$
Contains `x`?	`o.contains(x)`	$\mathcal{O}(n)$

Doubly linked lists

Usefulness	Storage of elements, sequential access
Instantiation	`new LinkedList<Type>()`

Methods on an object `o` of size n

Name	*Java*	*Complexity*
Access first element	`o.getFirst()`	$\Theta(1)$
Add to front	`o.addFirst(x)`	$\Theta(1)$
Remove from front	`o.removeFirst()`	$\Theta(1)$
Access last element	`o.getLast()`	$\Theta(1)$
Add to back	`o.addLast(x)`	$\Theta(1)$
Remove from back	`o.removeLast()`	$\Theta(1)$
Access element `i`	`o.get(i)`	$\mathcal{O}(n)$
Assign `x` to element `i`	`o.set(i, x)`	$\mathcal{O}(n)$
Add anywhere	`o.add(i, x)`	$\mathcal{O}(n)$
Remove from anywhere	`o.remove(i)`	$\mathcal{O}(n)$
Contains `x`?	`o.contains(x)`	$\mathcal{O}(n)$

Queue

Usefulness	First-in, first-out (FIFO)
Instantiation	`Queue<Type> o = new LinkedList<>()`
	(`Queue` is an interface in Java!)

Methods on an object `o` of size n

Name	*Java*	*Complexity*
Access first element	`o.element()`	$\Theta(1)$
Add (at the end)	`o.add(x)`	$\Theta(1)$
Remove (from the front)	`o.remove()`	$\Theta(1)$

When using the Queue ADT, you should NEVER have to use other methods (except `size` and `empty`).

Be careful, these functions can throw exceptions if the Queue is empty.

Stack

Usefulness	Last-in, first-out (LIFO)
Instantiation	`Stack<Type> o = new Stack<>()`

Methods on an object `o` of size n

Name	*Java*	*Complexity*
Access the element on top of the stack	`o.peek()`	$\Theta(1)$
Add (on top)	`o.push(x)`	$\Theta(1)$
Remove (from top)	`o.pop()`	$\Theta(1)$

When using the Stack ADT, you should NEVER have to use other methods (except `size` and `empty`).

You can (but are not *required* to) also take a look at the Deque interface (double-ended queue, which implements both the Stack and Queue ADTs) and its implementations: `LinkedList` and `ArrayDeque`.

NEVER iterate (foreach or `iterator`) over a stack. The elements are returned in reverse order...

Sorting in Java: Arrays and Collections

1. Primitive Arrays

- Use `Arrays.sort(arr)`.
- Implementation of a variant of quicksort in $O(N \log N)$.

2. Objects & Collections

- Use `list.sort()` or `Collections.sort(list)`. It requires objects to implement `Comparable` (defines "natural order").
- You can also pass a custom comparator in argument (more flexible).
- Implementation of variant of quicksort in $O(N \log N)$ (stable).

3. Custom Order (Comparators) *Using Lambda expressions:*

```
// Standard Lambda notation
list.sort((a, b) -> a.val - b.val);

// Using Comparator factory methods
list.sort(Comparator.comparing(Node::getVal));

// Lexico-graphic: criterion one, then second in case of ties
list.sort(Comparator.comparing(Node::getCategory)
                    .thenComparing(Node::getVal));
```

TreeSet: Self-balancing tree (set)

Usefulness	Set, sorted
Instantiation	`new TreeSet<Type>`
What is it?	Behind the scenes, it's a red-black tree
	No duplicates (it's a set)
	Elements must be comparable

Methods on an object o of size n

Name	*Java*	*Complexity*
Add	`o.add(x)`	$\mathcal{O}(\log n)$
Remove	`o.remove(x)`	$\mathcal{O}(\log n)$
Contains?	`o.contains(x)`	$\mathcal{O}(\log n)$
Find element below (or =)	`o.floor(x)`	$\mathcal{O}(\log n)$
Find element above (or =)	`o.ceil(x)`	$\mathcal{O}(\log n)$
Find element strictly below	`o.higher(x)`	$\mathcal{O}(\log n)$
Find element strictly above	`o.lower(x)`	$\mathcal{O}(\log n)$

Another useful method: `subset`, which allows extracting a part of the tree at constant cost.
`descendingIterator` is also very practical.

HashSet: hash table (set)

Usefulness	Set
Instantiation	`new HashSet<Type>`
What is it?	A hash table
	No duplicates (it's a set)
	Elements must be hashable (!!!)
	Unsorted

Methods on an object o of size *n*

Name	*Java*	*Complexity*
Add	`o.add(x)`	$\mathcal{O}(1)$ amortized
Remove	`o.remove(x)`	$\mathcal{O}(1)$ amortized
Contains?	`o.contains(x)`	$\mathcal{O}(1)$ amortized

Under the constraint that the hashes are well distributed!

TreeMap: self-balancing tree (dictionary)

Usefulness	Dictionary
Instantiation	`new TreeMap<TypeKey,TypeValue>`
What is it?	A red-black tree (or equivalent)
	No duplicate keys (it's a dictionary)
	Keys must be comparable

Methods on an object `o` of size n

Name	*Java*	*Complexity*
Add (key `x`, value `y`)	`o.put(x, y)`	$\mathcal{O}(\log n)$
Get key	`o.get(x)`	$\mathcal{O}(\log n)$
Remove key	`o.remove(x)`	$\mathcal{O}(\log n)$
Contains key?	`o.containsKey(x)`	$\mathcal{O}(\log n)$

TreeSet functions are also available for keys (`ceilingKey`, `subMap`...).

See later slide for a tip on iterating over Maps and especially TreeMaps.

Even if there are methods to search for values (`containsValue`, ...), they are generally in $\mathcal{O}(n)$.

HashMap: hash table (dictionary)

Usefulness	Set
Instantiation	`new HashMap<TypeKey,TypeValue>`
What is it?	A hash table
	No duplicate keys (it's a dictionary)
	Keys must be hashable (!!!)
	Unsorted

Methods on an object `o` of size n

Name	*Java*	*Complexity*
Add (key `x`, value `y`)	`o.put(x, y)`	$\mathcal{O}(1)$ amortized
Get key	`o.get(x)`	$\mathcal{O}(1)$ amortized
Remove key	`o.remove(x)`	$\mathcal{O}(1)$ amortized
Contains key?	`o.containsKey(x)`	$\mathcal{O}(1)$ amortized

Under the constraint that the hashes are well distributed!

See later slide for a tip on iterating over Maps and especially TreeMaps.

Even if there are methods to search for values (`containsValue`, ...), they are generally in $\mathcal{O}(n)$.

PriorityQueue: priority queue

Usefulness	Queue, but with priorities.
Instantiation	`new PriorityQueue<Type>`
What is it?	A binary heap based on an array
	Values must be comparable

Methods on an object `o` of size n

Name	*Java*	*Complexity*
Add	`o.add(x)`	$\Theta(\log n)$
See first element	`o.peek()`	$\Theta(1)$
Remove first element	`o.poll()`	$\Theta(\log n)$

If you use any other method (`remove`, `iterator`, ...), a PQ is not the right structure. It's usually $\mathcal{O}(n)$.

The iterator on PQs does not return the elements in order.

Which ADT/implementation to choose for your problem?

Which ADT/implementation to choose?

Non-exhaustive and heuristic list obviously...

- I just want to store elements...
  - And do sequential access → `LinkedList`
  - And do random access...
    - And add elements as I go → `ArrayList`
    - And I know the size from the start → array `type[]`
- I want to maintain a set (without duplicates) → `HashSet` (hashable) or `TreeSet` (comparable)
- I want to use contains → `HashSet` (hashable) or `TreeSet` (comparable)
- I want to maintain a sorted set → `TreeSet`
- I want to map a key to a value...
  - sorted → `TreeMap`
  - unsorted → `HashMap`
- I want to add/remove/get, only at the beginning and at the end → `LinkedList`, `Queue` or `Stack` depending on the case
- I have priorities to manage, always the highest priority item first → `PriorityQueue`
- I want to iterate over the content → anything except `Stack`, `Queue`, `PQ`, `UF`, ...

Sorting algorithms ADT relying on comparisons

Binary Search

- In a sorted array, let's say we are looking for an element x.
- Take the middle of the array, y. If $x = y$, we have won.
- If $x < y$, then x is necessarily on the left; otherwise, it is on the right.
- Recursively call on the left or right half of the array accordingly.

Complexity:
$\mathcal{O}(\log n)$

```
public static int rank(int[] a, int x) {
    int lo = 0, hi = a.length - 1;
    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;
        if      (x < a[mid]) hi = mid - 1;
        else if (x > a[mid]) lo = mid + 1;
        else return mid;
    }
    return -1;
}
```

Sorting properties

- **In-place sort**: An algorithm that sorts items within the same storage space as the original data, using only a small, constant amount of additional memory (usually $O(1)$ or $O(\log n)$ auxiliary space). It modifies the input array directly without creating a new array to store the sorted elements.
- **Stable sort**: A sorting algorithm that preserves the relative order of elements that have equal keys. If two items have equal keys, a stable sort ensures their original relative order is maintained in the sorted output. This is important when sorting data on multiple criteria.

Algorithm	In-place?	Stable?
Selection Sort	Yes	No
Insertion Sort	Yes	Yes
Merge Sort	No	Yes
Quick Sort	Yes	No
Heap Sort	Yes	No

Selection Sort

Algorithm Logic

- **Search:** Find the smallest element in the unsorted portion.
- **Swap:** Swap it with the first unsorted element.
- **Invariant:** Items to the left of i are sorted and in their final positions.
- **Performance:** Always $\mathcal{O}(n^2)$ comparisons, but only $\mathcal{O}(n)$ swaps.

```java
public static void selectionSort(int[] a) {
    int n = a.length;
    for (int i = 0; i < n - 1; i++) {
        int min = i;
        // Search for the smallest item
        for (int j = i + 1; j < n; j++) {
            if (a[j] < a[min]) {
                min = j;
            }
        }
        // Swap into correct position
        int temp = a[i];
        a[i] = a[min];
        a[min] = temp;
    }
}
```

Insertion Sort

Algorithm Logic

- **Process:** Pick the next element and "insert" it into the sorted prefix by sliding it left.
- **Adaptive:** Very fast for **nearly sorted** data ($O(n)$ best case).
- **Stable & In-place:** Maintains relative order and uses $O(1)$ extra space.
- **Complexity:** $\mathcal{O}(n^2)$ comparisons and swaps in the worst case.

```java
public static void insertionSort(int[] a) {
    int n = a.length;
    for (int i = 1; i < n; i++) {
        // Slide a[i] into position in a[0..i]
        for (int j = i; j > 0 && a[j] < a[j-1]; j--) {
            swap(a, j, j - 1);
        }
    }
}

private static void swap(int[] a, int i, int j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

Merge sort

- Divide the array into two halves.
- Recursively sort each half.
- Merge the two sorted halves.
- Time complexity $\mathcal{O}(n \log n)$

```
public static void mergeSort(int[] a, int[] aux, int lo, int hi) {
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);        // Sort left half
    sort(a, aux, mid + 1, hi);    // Sort right half
    merge(a, aux, lo, mid, hi);   // Merge results
}

private static void merge(int[] a, int[] aux, int lo, int mid, int hi) {
    // 1. Copy to aux[]
    for (int k = lo; k <= hi; k++) {
        aux[k] = a[k];
    }

    // 2. Merge back to a[]
    int i = lo, j = mid + 1;
    for (int k = lo; k <= hi; k++) {
        if      (i > mid)          a[k] = aux[j++];
        else if (j > hi)           a[k] = aux[i++];
        else if (aux[j] < aux[i]) a[k] = aux[j++];
        else                       a[k] = aux[i++];
    }
}
```

Quick sort

- Partition the array so that for some 'j'
- the entry 'a[j]' is in place (it won't move any more)
- no larger entry to the left of 'j'
- no smaller entry to the right of 'j'
- Sort each piece recursively.
- $\mathcal{O}(n^2)$ in the worst-case (when the array is sorted) (shuffle the array first to have $\mathcal{O}(n \log n)$ expected)
- We have seen a variant of the quick-sort for computing the median.

```
public void quickSort(int[] a, int lo, int hi) {
    if (hi <= lo) return;
    int j = partition(a, lo, hi);
    sort(a, lo, j - 1);
    sort(a, j + 1, hi);
}

private int partition(int[] a, int lo, int hi) {
    int i = lo, j = hi + 1;
    int v = a[lo]; // Pivot element
    while (true) {
        while (a[++i] < v)
            if (i == hi) break;
        while (v < a[--j])
            if (j == lo) break;
        if (i >= j) break;
        swap(a, i, j);
    }
    swap(a, lo, j); // Put pivot at j
    return j;
}
```

Quick select

- Find the kth largest element in an unsorted array
- Choose a pivot
- Put the elements smaller than the pivot to the left of the pivot, the others to the right
- If the pivot ends up at position k, stop (it is the sought-after element).
- If not, if $k <$ pivot position, recursively call only on the left, and on the right otherwise.
- Time complexity: $\mathcal{O}(n)$ average, $\mathcal{O}(n^2)$ in general

Heap sort

- Transform the array into a heap.
  - **Heapify:** $\mathcal{O}(n)$ (bottom-up)
  - *vs* Insert one-by-one: $\mathcal{O}(n \log n)$
- Depop the elements one by one. They are therefore sorted.
- Time complexity: $\mathcal{O}(n \log n)$.

```
public static void heapSort(Comparable[] a) {
    int N = a.length;
    // Phase 1: Heapify = Build a valid heap
    for (int k = N/2; k >= 1; k--) {
        sink(a, k, N);
    }
    // Phase 2: Sortdown
    while (N > 1) {
        exch(a, 1, N--);
        sink(a, 1, N);
    }
}

private static void sink(Comparable[] a, int k, int N) {
    while (2 * k <= N) {
        int j = 2 * k; // Left child
        if (j < N && less(a, j, j + 1)) j++;
        if (!less(a, k, j)) break;
        exch(a, k, j);
        k = j;
    }
}
```

Binary heaps (Priority Queues)

- Complete binary tree (except possibly the last layer, filled from left).
- **Property:** Children of a node are larger or equal than it (min-heap).
- **Array Impl:** Node i has children at $2i$ and $2i + 1$. (1-based indexing).
- **Insertions:** Add to end. Swim up (exchange with parent) until property holds.
- **Min Deletion:** Replace root with end element. Sink down.
- Complexity: $\mathcal{O}(\log n)$ min deletion and insertions, $\mathcal{O}(1)$ to just retreive the min.

```
public void insert(Key x) {
    pq[++N] = x;
    swim(N);
}

private void swim(int k) {
    while (k > 1 && greater(k/2, k)) {
        exch(k, k/2);
        k = k/2;
    }
}
```

Min-max heap

- Variant of the binary heap, which maintains both the minimum and the maximum and allows them to be removed in $\mathcal{O}(\log n)$
- Property: on the "even" layers (level 0 (root), 2, 4, ...) the min-heap property applies (all nodes below are larger). For odd layers, it is the max-heap property.
- Necessarily, the minimum is the root, and the maximum is the maximum of the first two children.
- The rules for swim/sink of the values must be adapted (a sheet of paper and a small drawing should be enough to find the rule, a little too long to put in this slide).

$\mathcal{O}(\log n)$ for everything, $\mathcal{O}(1)$ to see the smallest/largest element.

Counting Sort & String Sorting

Counting Sort Logic

- **Pre-condition:** Values are integers in a fixed range $[0, R)$.
- **Frequency:** Count occurrences of each value.
- **Accumulate:** Transform counts into indices (cumulative sum).
- **Distribute:** Move items to their correct position in a temp array.

Complexity: $\mathcal{O}(N + R)$

- N: Number of items.
- R: Size of alphabet/range.

Key Idea for Strings

- View characters as integers ($a = 97, b = 98 \ldots$).
- **LSD (Least Significant Digit):** Sort strings from right-to-left.
- **Stability is Mandatory:** Counting sort is stable, so a sort on digit i doesn't break the order from digit $i + 1$.

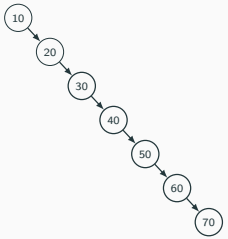
LSD String Sort

```
for (int d = W-1; d >= 0; d--) {
    int[] count = new int[R + 1];
    // 1. Count frequencies of char at d
    for (int i = 0; i < N; i++)
        count[s[i].charAt(d) + 1]++;
    // 2. Compute cumulates
    for (int r = 0; r < R; r++)
        count[r+1] += count[r];
    // 3. Move to aux array
    for (int i = 0; i < N; i++)
        aux[count[s[i].charAt(d)]++] = s[i];
    // 4. Copy back
    for (int i = 0; i < N; i++) s[i] = aux[i];
}
```

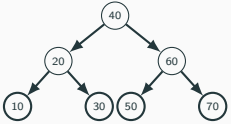
Binary Search Trees (BST)

- **Ordering property:** For any node x, all keys in the left subtree are $< x$ and all keys in the right subtree are $> x$ (no duplicate keys).
- **Recursive search:** Compare the key with the current node and recursively move left or right.
- **Best case:** The tree is **balanced**, with height $O(\log n)$, yielding $O(\log n)$ search time.
- **Worst case:** Inserting sorted keys produces a degenerate tree (linked list), giving $O(n)$ search time.

Worst case (degenerate BST)



Best case (balanced BST)



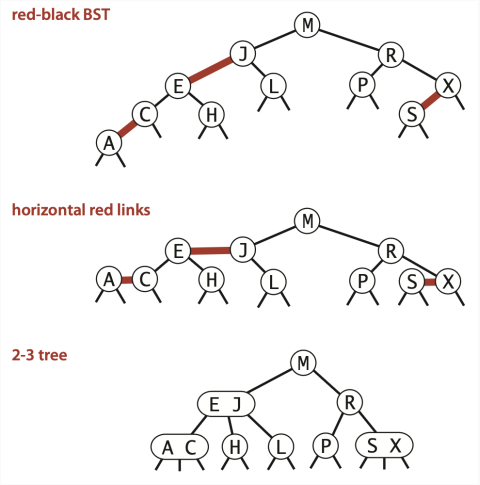
2-3 tree

- Tree containing either 2-nodes (2 children) or 3-nodes (3 children).
- The 2-nodes work like search trees. The 3-nodes have two values. To the left smaller than the first, in the middle between the first and the second value, to the right larger than the second value.
- 2-3 trees are always complete and therefore balanced.
- The addition rules are complex to list, BUT intuitive. We go down to where the value should be placed, we add it to the node. If the node becomes a 4-node (three values) we must divide it by moving a value up to the parent node.

Everything is in $\mathcal{O}(\log n)$ as a result.

Left-Leaning Red-Black-Trees (LL-RBT) and 2-3 Tree Equivalence

- **Search Structure:** A LL-RBT is a BST where red links are only on the left.
- **One-to-One Correspondence:**
  - 2-nodes → standard black node.
  - 3-nodes → two nodes connected by a red link.
- **Black Balance:** Every path from root to null link has the same number of black links (perfect black balance).
- **Height:** Guaranteed $O(\log N)$ height; no more than $2 \times$ height of 2-3 tree.
- ⚠ **Note:** Some textbooks allow right-leaning red links (standard RBT), which are more complex (implementation) to balance.



Hash and Maps

Hash tables - Separate Chaining

- **Principle:** Map keys to indices via hashing. Store pairs in "buckets."
- **Separate Chaining:** Buckets are linked lists handling collisions.
- **Dynamic Resizing:**
  - When n/m gets too high, we increase m.
  - **Rehashing:** Every item must be re-processed because indices change ($hash \pmod m$).

Operations are $\mathcal{O}(1)$ **amortized**.

```
public void put(Key key, Value val) {
    if (n >= m) resize(2 * m);

    int i = hash(key);
    for (Node x = st[i]; x != null; x = x.next) {
        if (key.equals(x.key)) {
            x.val = val;
            return;
        }
    }
    // Prepend to chain
    st[i] = new Node(key, val, st[i]);
    n++;
}
private void resize(int newM) {
    Node[] temp = new Node[newM];
    for (int i = 0; i < m; i++) {
        for (Node x = st[i]; x != null; x = x.next) {
            // Recompute index for new M
            int idx = (x.key.hashCode() & 0x7fffffff) % newM;
            // Prepend to new chain
            temp[idx] = new Node(x.key, x.val, temp[idx]);
        }
    }
    this.st = temp;
    this.m = newM;
}
```

Hash tables - Linear Probing

- **Collision Resolution:** If ‘table[i]‘ is occupied, try ‘i+1‘, ‘i+2‘, etc. (wrapping around).
- **Insertion:** Find the first empty slot in the sequence.
- **Search:** Follow the probe sequence until the key is found or an empty slot is reached.
- **The Deletion Problem:**
  - You cannot simply set a slot to ‘null‘.
  - This would break the probe chain for items inserted *after* it in the same cluster.
  - **Fix:** Use "tombstones" or re-insert subsequent items in the cluster.

```
public class LinearProbingHashST<Key, Value> {
    private int n;           // number of key-value pairs
    private int m = 16;      // size of linear probing table
    private Key[] keys;      // the keys
    private Value[] vals;    // the values

    public void put(Key key, Value val) {
        if (n >= m/2) resize(2*m);

        int i;
        for (i = hash(key); keys[i] != null; i = (i + 1) % m) {
            if (keys[i].equals(key)) {
                vals[i] = val; // Update existing key
                return;
            }
        }
        keys[i] = key;
        vals[i] = val;
        n++;
    }

    public Value get(Key key) {
        for (int i = hash(key); keys[i] != null; i = (i+1)%m) {
            if (keys[i].equals(key)) return vals[i];
        }
        return null;
    }
}
```

LRU Cache Implementation

An LRU (Least Recently Used) Cache is a HashMap with a fixed capacity that tracks access order using a doubly linked list. Every time an entry is accessed (get/put), it is moved to the front (Most Recently Used). If an insertion exceeds capacity, the tail (Least Recently Used) entry is evicted.

Goal: $\mathcal{O}(1)$ for both 'get' and 'put'. We combine two data structures:

- **Doubly Linked List (DLL):**
  - Maintains usage order.
  - **Head:** Most Recently Used (MRU).
  - **Tail:** Least Recently Used (LRU) - candidate for eviction.
- **Hash Map:**
  - Maps 'Key' $\rightarrow$ the specific 'DLL Node'.
  - Allows jumping directly to a node in the middle of the list without traversal.

```java
public class LRUCache<K, V> {
    private class Node { K key; V val; Node prev, next;}
    private Map<K, Node> map = new HashMap<>();
    private Node head, tail; // Sentinels
    private int capacity;
    public V get(K key) {
        Node node = map.get(key);
        if (node == null) return null;
        remove(node); // Move to front (MRU)
        addFirst(node); // Not shown, easy
        return node.val;
    }
    public void put(K key, V val) {
        Node node = map.get(key);
        if (node != null) {
            node.val = val;
            remove(node); // Not shown, easy
        } else {
            node = new Node();
            node.key = key; node.val = val;
            map.put(key, node);
            if (map.size() > capacity) {
                Node lru = tail.prev;
                remove(lru);
                map.remove(lru.key);
            }
        }
        addFirst(node); // Still easy
    }
}
```

String Algorithms

Rabin-Karp Algorithm

Goal: efficient search for fixed-length (M) patterns in a text.

- **Preprocessing:** Hash all patterns and store in a `Map<Long, String>`.
- **Rolling Hash:** Update the hash in $\mathcal{O}(1)$ as the window slides.
- **Verification:** If hash matches, compare actual characters to rule out collisions.

The Math (Polynomial Hash):

$$H_i = \sum_{j=0}^{M-1} x_{i+j} R^{M-1-j}$$

Rolling: To move window $i \to i+1$:

1. Remove leading char x_i ($x_i R^{M-1}$).
2. Shift left (multiply by R).
3. Add new trailing char x_{i+M}.

$$H_{i+1} = (H_i - x_i R^{M-1}) \cdot R + x_{i+M}$$

```java
public int rabinKarpSearch(String txt, Set<String> patterns, int M) {
    int N = txt.length();
    long Q = 997; // Large prime for modulo
    int R = 256;  // Alphabet size
    long RM = 1;  // R^(M-1) % Q
    Map<Long, String> hashes = new HashMap<>();

    for (int i = 1; i <= M - 1; i++) RM = (R * RM) % Q;

    // 1. Preprocess patterns
    for (String p : patterns) hashes.put(hash(p, M, Q), p);

    // 2. Initial hash for first window
    long h = hash(txt.substring(0, M), M, Q);

    // 3. Roll through text
    for (int i = 0; i <= N - M; i++) {
        if (hashes.containsKey(h)) {
            if (txt.substring(i, i + M).equals(hashes.get(h)))
                return i; // Match found
        }
        // Roll to next window
        if (i < N - M) {
            h = (h + Q - txt.charAt(i) * RM % Q) % Q;
            h = (h * R + txt.charAt(i + M)) % Q;
        }
    }
    return -1;
}
```

Huffman compression

- Start with a set of trees, each corresponding to a character's frequency.
- Use a Min-Priority Queue to store these trees.
- Repeat until only one tree remains:
- Extract the two smallest frequency trees (x and y).
- Merge them into a new tree with frequency $x.freq + y.freq$.
- Insert the new tree back into the PQ.

$\mathcal{O}(n \log n)$ where n is the number of characters.

```
public Node buildTree(int[] freq) {
    PriorityQueue<Node> pq = new PriorityQueue<>();
    for (char c = 0; c < 256; c++) { // Creat a leaf nodes for each character
        if (freq[c] > 0)
            pq.add(new Node(c, freq[c], null, null));
    }
    while (pq.size() > 1) { // Merge smallest nodes until unique tree
        Node left  = pq.poll();
        Node right = pq.poll();
        // Create internal node (char '\0' or similar)
        Node parent = new Node('\0', left.freq + right.freq,
                               left, right);
        pq.add(parent);
    }
    return pq.poll();
}
private static class Node implements Comparable<Node> {
    private final char ch;
    private final int freq;
    private final Node left, right;
    Node(char ch, int freq, Node left, Node right) {
        this.ch = ch; this.freq = freq;
        this.left = left; this.right = right;
    }
    public int compareTo(Node that) {
        return this.freq - that.freq;
    }
}
```

Graph Algorithms

A quick reminder about graphs

- Graph: set of nodes and edges connecting these nodes. ($G =< V, E >$)
- *Weighted* graph: the edges have an associated weight. Mathematically, usually, we define this via a function $f : E \to \mathbb{R}$ which gives a weight when given an edge.
- *Simple* graph: there is, between any pair of nodes, at most one edge. This implies that $E \leq \frac{V \cdot (V-1)}{2} \in \mathcal{O}(V^2)$ and therefore that $\mathcal{O}(\log E) \subseteq \mathcal{O}(\log V^2) = \mathcal{O}(2 \log V) = \mathcal{O}(\log V)$.
- *Directed* graph: the edges have a direction. Generally, we then call these edges *arcs*.
- Degree of a node: number of edges touching a node.
- In-degree (resp. out-degree): number of arcs of which the node is the destination (resp. origin).
- Sum of the degrees of the nodes $= 2E$
- *Acyclic* graph: without a cycle (directed or not, depending on the case).

Graph ADT: Adjacency Matrix vs. List

1. Adjacency Matrix

- A 2D array `adj[V][V]` where `adj[u][v] == 1` if edge (u, v) exists.

```
boolean[][] adj = new boolean[V][V];
void addEdge(u, v) { adj[u][v] = true; }
```

2. Adjacency List

- An array of V lists; `adj[u]` contains all neighbors of u.

```
List<Integer>[] adj = new List[V];
void addEdge(u, v) { adj[u].add(v); }
```

Pros, Cons & Complexity

Operation	Matrix	List
Space	$O(V^2)$	$O(V + E)$
Add Edge	$O(1)$	$O(1)$
Check Edge	$O(1)$	$O(degree(u))$
Iterate neighbors	$O(V)$	$O(degree(u))$

When to use?

- **Matrix: Dense** graphs ($E \approx V^2$) or when you need constant-time edge lookups.
- **List: Sparse** graphs (most real-world cases). Memory efficient and faster for traversing neighbors.

Breadth-First Search (BFS)

Concepts

- **Layer-by-layer:** Visits all nodes at distance d before moving to $d + 1$.
- **What it computes:**
  - In unweighted graphs, BFS finds the minimum number of hops from source s to any reachable v.
  - `dist[v]`: Shortest path length of $s \to v$.
  - `prev[v]`: The node that "discovered" v.
- **Path Reconstruction:** Following `prev[]` pointers backwards from v yields the shortest path to s.

Complexity: $\mathcal{O}(V + E)$

```
Queue q = new Queue();
boolean[] visited = new boolean[G.V()];
int[] dist = new int[G.V()];
int[] prev = new int[G.V()]; // Path tree

visited[s] = true;
q.enqueue(s); // if you have multiple sources, add them here

while (!q.isEmpty()) {
    int u = q.dequeue();
    for (int v : G.adj(u)) {
        if (!visited[v]) {
            visited[v] = true;
            dist[v] = dist[u] + 1;
            prev[v] = u; // Discovery link
            q.enqueue(v);
        }
    }
}
```

Depth-First Search (DFS)

- **Intuition** Explores as far as possible along each branch before backtracking.
- **What it computes:**
  - Finds *a* path from s to v, but it is **not necessarily the shortest**.
  - `visited[v]`: Keeps track of nodes already explored.
  - `prev[v]`: Encodes the path tree (the "parent" node in the recursion).
- **Possible implems:** Uses the call stack (Recursion) or an explicit **Stack** (LIFO).
- **Path Reconstruction:** Following `prev[]` pointers backwards from v yields a path to s (not necessarily the shortest).
- **Complexity:** $\mathcal{O}(V + E)$.
- Can be used for directed or undirected graph.

DFS(Graph G Source s)

```
boolean[] visited = new boolean[G.V()];
int[] prev = new int[G.V()];
search(G, s) // Initial call

void search(Graph G, int u) {
    visited[u] = true;
    for (int v : G.adj(u)) {
        if (!visited[v]) {
            prev[v] = u; // Record discovery
            search(G, v); // Recursive dive
        }
    }
}
```

DFS Applications in $\mathcal{O}(V + E)$

1. (Directed) cycle detection in a (directed) graph

A cycle exists if we visit an **Open** node:

- **0 (Unseen):** Default.
- **1 (Open):** Node is on the **current** recursion stack.
- **2 (Closed):** Node & all descendants visited.

```
int[] state; // 0: UNSEEN, 1: OPEN, 2: CLOSED
public boolean hasCycle() {
    state = new int[V];
    for (int i = 0; i < V; i++) {
        if (state[i] == 0) {
            if (hasCycle(i)) return true;
        }
    }
    return false;
}
bool hasCycle(int u) {
    state[u] = 1; // Mark OPEN (on stack)
    for (int v : adj[u]) {
        if (state[v] == 0 && hasCycle(v)) return true;
        if (state[v] == 1) return true; // Cycle!
    }
    state[u] = 2; // Mark CLOSED (backtrack)
    return false;
}
```

2. Test if an undirected graph is bipartite

- **Logic:** Assign alternate colors (0 and 1).
- **Conflict:** A neighbor already has the **same** color as current node $\rightarrow$ not bipartite.

```
int[] color; // -1: NONE, 0: RED, 1: BLUE
public boolean isBipartite() {
    color = new int[V];
    Arrays.fill(color, -1);
    for (int i = 0; i < V; i++) {
        if (color[i] == -1) {
            if (!isBipartite(i, 0)) return false;
        }
    }
    return true; // All components successfully 2-colored
}
bool isBipartite(int u, int c) {
    color[u] = c; // Color this node
    for (int v : adj[u]) {
        if (color[v] == -1) { // Not visited
            if (!isBipartite(v, 1-c)) return false;
        } else if (color[v] == c) {
            return false; // Conflict found!
        }
    }
    return true;
}
```

Topological Sort

Objective

- For a **DAG**, find an ordering where for every edge (u, v), u comes before v.
- **Intuition:** "Unlock" nodes only after all their prerequisites are met.

In-degree Check

- If at the end the order contains fewer than V nodes, the graph has a **cycle**.

Complexity: $\mathcal{O}(V + E)$

```
// 1. Calculate in-degree for all nodes
int[] inDegree = new int[V];
for (int u = 0; u < V; u++)
    for (int v : adj[u]) inDegree[v]++;

// 2. Queue all nodes with no dependencies
Queue q = new Queue();
for (int i = 0; i < V; i++)
    if (inDegree[i] == 0) q.enqueue(i);

// 3. Process the queue
List order = new ArrayList();
while (!q.isEmpty()) {
    int u = q.dequeue();
    order.add(u);
    for (int v : adj[u]) {
        inDegree[v]--; // "Remove" edge (u, v)
        if (inDegree[v] == 0) q.enqueue(v);
    }
}
```

Union-Find (Weighted Quick-Union)

Useful data-strcuture to compute connected components in undiracted graph in $\mathcal{O}(E)$.
Idea: Maintain disjoint sets as trees using an array ‘parent‘.

- **Representative:** The root of the tree (‘parent[i] == i‘).
- **Weighted Union:** Always link the *smaller* tree to the *larger* tree (maintains balance).
- **Path Compression:** During ‘find‘, make nodes point directly to the root.

Complexity:
$\mathcal{O}(\log n)$ with weighted union per op.
$\mathcal{O}(\alpha(n))$ per op. with weighted union + path compression ($\mathcal{O}(1)$ practice).

```
public class UF {
    private int[] parent; // parent[i] = parent of i
    private int[] size;   // size[i] = #nodes in subtree
    private int count;    // number of components
    public UF(int n) {
        count = n;
        parent = new int[n];
        size = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }
    public int find(int i) {
        if (parent[i] == i) return i;
        return parent[i] = find(parent[i]); // Compression
    }
    public void union(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        if (rootP == rootQ) return;
        if (size[rootP] < size[rootQ]) { // Weithed
            parent[rootP] = rootQ;
            size[rootQ] += size[rootP];
        } else {
            parent[rootQ] = rootP;
            size[rootP] += size[rootQ];
        }
        count--;
    }
}
```

Minimum Spanning Tree: Prim & Kruskal Algorithms

Definitions

- **Spanning Tree:** Connected acyclic graph (n nodes $\Rightarrow n-1$ edges).
- **Minimum Weight:** No spanning tree exists with a smaller total edge sum.

Prim's (Vertex-Centric) *"Grow a single tree"*

- Start from an arbitrary node.
- Always pick the cheapest edge connecting the tree to a *new* node.

Prim(G)

```
PriorityQueue pq;
pq.addEdges(startNode);
while (!pq.isEmpty()) {
    Edge e = pq.poll();
    if (e.to is visited) continue;

    add e to MST;
    visit(e.to);
    pq.addEdges(e.to neighbors);
}
```

Kruskal's (Edge-Centric) *"Merge separate forests"*

- Sort all edges by weight.
- Pick the cheapest edge that doesn't create a cycle (using **Union-Find**).

Kruskal(G)

```
sort edges by weight;
UnionFind uf = new UF(V);
for (Edge e : edges) {
    if (!uf.connected(e.u, e.v)) {
        uf.union(e.u, e.v);
        add e to MST;
    }
    if (MST.size == V-1) break;
}
```

Complexity for both: $\mathcal{O}(E \log V)$ or $\mathcal{O}(E \log E)$

Shortest Path: Dijkstra's Algorithm (Lazy Implementation: no priority update)

Concepts

- **Greedy Choice:** Always expand the node with the smallest current distance.
- **Relaxation:** If `dist[u] + w < dist[v]`, we found a better path.
- **Lazy Update:** Instead of updating a value in the PQ, we insert a *new* entry because the *PriorityQueue* of Java does not support priority update.
- **Condition:** We only process a node if the distance popped matches the `dist[]` array.
- ⚠ Dijstra does not work if negative edge weights.

Complexity: $\mathcal{O}(E \log V)$

Dijkstra(Graph G s)

```
dist[] = new int[V]; // Init with infinity
prev[] = new int[V];
PriorityQueue<Node> pq = new PriorityQueue<>();

dist[s] = 0;
pq.add(new Node(s, 0)); // if you have multiple sources, add them here

while (!pq.isEmpty()) {
    Node current = pq.poll();
    int u = current.id;
    int d = current.distance;

    // Skip if already found a shorter path to u
    if (d > dist[u]) continue;

    for (Edge e : G.adj(u)) {
        int v = e.to;
        if (dist[u] + e.weight < dist[v]) {
            dist[v] = dist[u] + e.weight;
            prev[v] = u;
            pq.add(new Node(v, dist[v]));
        }
    }
}
```

Shortest-Path: Bellman-Ford Algorithm

- Computes shortest paths with at most 1 edge, then 2, . . . , up to $V - 1$.
- Unlike Dijkstra, it handles negative edge weights correctly.
- If a distance can still be reduced in the V-th iteration, a **negative cycle** exists.

Complexity: $\mathcal{O}(V \times E)$.

BellmanFord(List<Edge> edges int s)

```
double[] dist = new double[V];
Arrays.fill(dist, Double.POSITIVE_INFINITY);
dist[s] = 0.0;

// Relax all edges V-1 times
for (int i = 1; i < V; i++) {
    for (Edge e : edges) {
        if (dist[e.u] + e.weight < dist[e.v]) {
            dist[e.v] = dist[e.u] + e.weight;
        }
    }
}

// Check for negative cycles
for (Edge e : edges) {
    if (dist[e.u] + e.weight < dist[e.v]) {
        throw new RuntimeException("Negative cycle!");
    }
}

return dist;
```

All-Pairs Shortest Paths: Floyd-Warshall Algorithm

- **Purpose:** Computes shortest distances between **all pairs** of nodes in a weighted directed graph.
- **Dynamic Programming:** $d[i][j]$ is the shortest path from i to j using only intermediate nodes from $\{0 \dots k\}$.
- **Complexity:** $\mathcal{O}(V^3)$ time and $\mathcal{O}(V^2)$ space.
- **Note:** Can handle negative edge weights, but not negative cycles.

```java
public double[][] floydWarshall(double[][] adj, int V) {
    double[][] dist = new double[V][V];

    // 1. Initialize distance matrix
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            dist[i][j] = adj[i][j];
        }
    }

    // 2. Main DP loop
    for (int k = 0; k < V; k++) {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                }
            }
        }
    }
    return dist;
}
```

Longest path

- It is conjectured that there is no polynomial time algorithm to find the longest path in an arbitrary graph (NP-Complete problem).
- If the graph is a DAG (directed acyclic graph) then there is a polynomial algorithm: take the opposite of the edge weights, and use Bellman-Ford.
- This works because there will be negative weights, but no negative cycles since the graph is acyclic.

Tips & tricks

Iterating over collections

Don't do this (because it takes $\mathcal{O}(n^2)$)...:

```
List<String> x = ...;
for(int i = 0; i < x.size(); i++) {
        String elem = x.get(i);
        //...
}
```

... use foreach instead:

```
for(String elem: x) {
        //...
}
```

or an iterator:

```
Iterator<String> itr = x.iterator();
while(itr.hasNext()) {
        String elem = itr.next();
        //...
}
```

Iterating over collections (2)

For dictionaries (`HashMap`, `TreeMap`), three reasonable possibilities:

```
HashMap<X, Y> hashmap = ...;
for(Map.Entry<X, Y> pair: hashmap.entrySet()) {
        X key = pair.getKey();
        Y value = pair.getValue();
        // ...
}
```

```
for(X key: hashmap.keySet()) {
        Y value = hashmap.get(key);
        // ...
}
```

Or with an iterator (on `hashmap.entrySet().iterator()` or `hashmap.keySet().iterator()`).

If you use a TreeMap, it's given in sorted order! Be careful that it's always $\mathcal{O}(n)$ except in the second case with the TreeMap, where it's $\mathcal{O}(n \log n)$.