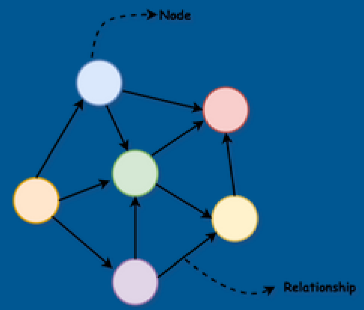
ALGORITHMS

AND

DATA STRUCTURES



Eimard Sobrino Zurera
Software Engineer

www.eimardsobrinozurera.com
linkedin.com/in/eimardsobrinozurera
https://gitlab.com/esobrinoz

JS

TS

Array List

Advantages

- ArrayList optimized for **retrieving items**.
- Simple **creation and usage**
- Foundational **building block** for complex **data structures**

Drawbacks

- **Expensive to manipulate** (insert/delete or resequence values)
- **Inefficient to sort**
- **Fixed size**

Applications

- Basic spreadsheets
- Within complex structures such as hash tables

Time Complexity

Average

Access	Search	Insertion	Deletion
0 (1)	0 (n)	0 (n)	0 (n)

Worst

Access	Search	Insertion	Deletion
0 (1)	0 (n)	0 (n)	0 (n)

Space Complexity

Worst 0 (n)

TS

Linked List

Advantages

- LinkedList **optimized for manipulation**
- Inserts and deletes are just changes of the pointer in nodes, **no collapsing or expanding needed**
- **Less complex than restructuring an array**

Drawbacks

- **Expensive to retrieve data**
- **Uses more memory than arrays**
- **Inefficient to traverse the list backward**

Applications

- Best used when data must be added and removed in quick succession from unknown locations
- Ypu can add or remove items from the beginning of the list in constant time. For specifics applications, this can be useful

Time Complexity

Average

Access	Search	Insertion	Deletion
0 (n)	0 (n)	0 (1)	0 (1)

Worst

Access	Search	Insertion	Deletion
0 (n)	0 (n)	0 (1)	0 (1)

Space Complexity

Worst 0 (n)

TS

Binary Search Tree

A binary tree is a tree in which each node has up to two children. Not all trees are binary trees. A node is called 'leaf' node if it has no children.

A binary search tree is a binary tree in which every node fits a specific ordering property: ***all left descendents <= n < all right descendents***. This must be true for each node n.

The definition can vary slightly with respect to equality.

Considerations

- **Fast for searh |** Length is only O (height)
- Notion of order
- **Optimized for adding and finding items** because of notion of order (O(log n))
- **Not optimized for sorted lists** (become O(n))
- Binary search trees **are not as fast as the more complicated hash table**

Applications

- Storing hierarchical data such as a file location
- Binary search trees are excellent for tasks needing searching or ordering of data

Time Complexity

Average

Access	Search	Insertion	Deletion
0 (log(n))	0 (log(n))	0 (log(n))	0 (log(n))

Worst

Access	Search	Insertion	Deletion
0 (n)	0 (n)	0 (n)	0 (n)

Space Complexity

Worst 0 (n)

TS

Binary Heap

A binary heap is very **similar to a binary tree with special rules around implementation**. A binary heap will be as **compact** as possible which means that **all the children of each node are as full as they can be and left children are filled out first**.

We can implement Binary Heap in two forms: **Max Binary Heap** and **Min Binary Heap**.

Considerations *(Max Binary Heap)*

- Every node have at most two children
- Each **parent node's value will be always greater** than its children and there is no guarantee in its children's order.
- **Root** node value is ***always*** greater than ***all*** the nodes
- Useful **only for very specific scenarios**
- You **cannot extrac a node other than the maximum**

Applications

- A Binary Heap data structure is a **specialized structure used to save the information** and has very **specific** use cases like **sorting** and **priority queue.**
- **Emergency ward system** with high critical tasks to carry out
- Task scheduling based on priority

Time Complexity

Average

Access	Search	Insertion	Deletion
0 (log(n))	0 (log(n))	0 (log(n))	0 (log(n))

Worst

Access	Search	Insertion	Deletion
0 (log(n))	0 (log(n))	0 (log(n))	0 (log(n))

Space Complexity

Worst 0 (n)

Time complexity to find minimum or maximum of all elements is alway constant: 0 (1)

TS

Quicksort | Mergesort | Heapsort

Quicksort

In quicksort we **pick a random element and partition the array**, such that all numbers that are less than the partitioning element come before all elements that are greater that it

Best	Average	Worst	Memory	inplace?	is stable?
0 (n log(n))	0 (2n log(n))	0 (n^2 / 2)	0 (log(n))	YES	NO

remarks

- N log N probalistic guarantee **fastest in practice.**
- **39% more compars than mergesort**
- **Faster than mergesort** in practice bease of less data movement
- **Random shuffle** guarantee against worst case

Mergesort

Merge sort **divides the array in half, sorts eacf of those halves**, and then merges them back together. Each of those halves has the same algorithm applied to it.

Best	Average	Worst	Memory	inplace?	is stable?
0 (n log(n))	0 (n log(n))	0 (n log(n))	0 (n)	NO	YES

remarks

- N log N guarantee, **stable**
- **No in-place sorting, linear extra space**

Heapsort

Heapsort **divides its input into a sorted and an unsorted region**, and it iteratively shrinks the unsorted region by **extracting** the **largest** element from it and **inserting** it **into the sorted region**.

Best	Average	Worst	Memory	inplace?	is stable?
0 (n log(n))	0 (2n log(n))	0 (2n log(n))	0 (1)	YES	NO

remarks

- **Optimal for both time and space, but**
- **Inner loop longer than quicksort**
- **Makes poor use of cache memory**
- **Not stable**

TS

Disjktra

Disjktra algorithm is a way to **find out the shortest path between two points in a weighted directed graph** (which might have cycles). All edges must have positive values.

Disjktra's algorithm finds the minimum weight path from a start node s to every node on the graph.

Implementation and complexity

The **runtime** of this algorithm **depends** heavily **on** the **implementation** of the priority queue. Assume you have v vertices and e nodes:

- Implement priority queue **with array**: Each operation would take **0 (V)** . Additionally you would update the values of the paths weight per each edge, so that's **0 (V)** . Therefore, the total runtime is **0 (V^2)**
- If you implement the priority queue **with min heap**, then remove calls per each vertex will take **0 (v log(v))** . Additionally on each edge you might call one update/insert, so that's **0 (e log(v))** . Therefore the total runtime is **0 ((v+e) log v)**

Which is better?

If the **graphs has a lot of edges**, then v^2 will be close to e. In this case you might be **better off with the array** implementation, as **0 (V^2)** is better than **0 (V + V^2)**

However, if the **graph is sparse**, then e is much less than v^2. In this case, the **min heap** implementation **may be better**

Applications

- **Representing shortest path between two locations,** as GPS system does.
- **Represent cities** and edges representing travel time
- **Task scheduling** based on priority

BFS && DFS | Graph Search

The two most common ways to search a graph are deep-first search and breadth-first search

Breadth-first Search

We start at the root (or arbitrary node) and explore each neighbour nefore going to any of their children. **We go deep before we go wide.**

- If we want to **find the shortest path** between nodes, **BFS** is generally a **better solution.**
- BFS is **not recursive**. It **uses a queue.**
- Often **used as building block** in other algorithms
- Some applications: **Social Networking** websites, **Peer to Peer** Networks, **Crawlers** in search engineers
- Complexity **0 (V + E)**

Deep-first Search

We start at the root (or arbitrary node) and explore each branch completely before moving on to the next branch. **We go deep first we go deep.**

- BFS tends to be used in different scenarios. **DFS is often preferred if we want to visit every node in the graph.**
- BFS is **not recursive**. It **uses a queue.**
- By itself, DFS is not too **useful** but it is **for** specifics tasks such as **count connected components**, determine **connectivity**, or find the bridges/**articulation points**, detects **cycles**, even generate **mazes**.
- Complexity **0 (V + E)**

TS

RB Tree | AVL Tree

These are two types of self balanced binary search trees.

RB Tree

Red-black trees (a type of self-balancing binary seacrh tree) do **not ensure quite as strict balancing**, but the balancing is still good enough to ensure **0 (log(n))** insertions, deletions and retrievals.

- **It requires a bit less memory**
- **Can rebalance faster**
- **Used in situation where the tree will be modified frequently**

Properties

- Every node is either **black** or **red**
- Root **is black**
- The leaves, which are NULL nodes, are considered **black**
- Every node must have two **black children.** That is, **red node** cannot have red children (although a **black node** can have **black children**)
- Every path from a node to its leaves must have the same number of **black children**

AVL Tree

An AVL tree is one of the two common ways to implement tree balancing.

- **AVL trees are also Binary Search Trees** (lesser items to left, greater items to the right)
- **used to keep tree as flat as possible**
- **Rigidly balance tree** => **Provide faster look-ups**
- **Insertion/Deletion is not that fast**
- **Only uses Rotations for balancing**

Properties

An AVL tree stores in each node the height of the subtrees rooted at this nide. Then, for any node, we can check if it is height balanced: that the height of the left subree and the height of the right subtree differ by no more than one. This prevents situations where the tree gets too lopsided.

balance(n) = n.left.height - n.right.heigh
-1 <= balance(n) <= 1

Time Complexity *(For RB Trees and AVL Trees)*

Average				Worst			
Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion
0 (log(n))	0 (log(n))	0 (log(n))	0 (log(n))	0 (log(n))	0 (log(n))	0 (log(n))	0 (log(n))

Space Complexity

Worst 0 (n)

TS