## Hash Map

Salva i dati con coppie Chiave - Valore. Le Funzioni di Hash accettano una chiave e ritornano un output unico che è associato SOLO a quella chiave.

## Collisioni di Hash

Le collisioni si verificano quando una funzione di hash ritorna lo stesso output per due input diversi. Le collisioni si risolvono con il Separate Chaining, dove gli elementi vengono inseriti in una lista (quando si verifica una collisione) Oppure con il Linear Probing; in questo caso inseriamo la chiave, se siamo in presenza di una collisione, nella prima posizione utile successiva all'index di hash.

## Clustering

Un cluster è un blocco di elementi, aggiungendo delle chiavi potremmo andare ad incrementare la grandezza di un cluster.

Quando elimino nel Linear Probing DEVO effettuare il re-Hashing di tutte le chiavi rimanenti.

## Resize - S.C.

L'obbiettivo è mantenere $N/M$ = Costante

- Raddoppio la grandezza M quando $N/M >= 8$
- Dimezzo // quando $N/M <= 2$

→ Rieffettuo l'Hashing

## Resize - L.P.

L'obbiettivo è mantenere $N/M <= \frac{1}{2}$

- Raddoppio M quando $N/M >= \frac{1}{2}$
- Dimezzo M quando $N/M <= \frac{1}{8}$

→ Rieffettuo l'hashing

## Time Complexity

- indexing: O(1)
- Ricerca: O(1)
- inserimento: O(1)

} Le Hash Map sono estremamente efficienti!

# Informazioni importanti Algoritimi e Strutture

## Arrays

Gli array salvano i dati degli elementi basandosi su un indice, solitamente che inizia da 0. Questo tipo di struttura è una delle più vecchie ancora più usata.

### time Complexity

- indexing O(1)
- ricerca O(n)
- inserimento O(1)

## Resize Arrays

Quando effettuiamo il resize, non aumentiamo la grandezza di 1, ma **Raddoppiamo** l'array iniziale. Lo stesso si applica quando dobbiamo ridurlo:

- Push(): raddoppiamo quando è pieno
- Pop(): dimezziamo quando è pieno per 1/4.

→ lo stack è sempre pieno tra il 25% e 100%.

## Linked List

Salva i dati attraverso dei Nodi che puntano ad altri Nodi.


### time Complexity

- indexing: O(n)
- ricerca: O(n)
- inserimento: O(n)

} Nota come una Struttura Lenta

## Albero Binario

E' un tipo di struttura dati ad ALBERO, dove ogni nodo ha al massimo DUE figli. Questi due figli sono posti a sinistra e a destra del nodo padre.

Gli alberi sono progettati per la ricerca ed ordinamento. Gli alberi sono inoltre usati per creare i BST, ovvero un albero che effettua dei confronti, nell'inserimento, per decidere se aggiungere l'elemento a Sx o Dx di ogni nodo.

### Time Complexity

- indexing: $O(\log n)$
- ricerca: $O(\log n)$
- inserimento: $O(\log n)$

} BST, le operazioni sono basate sulla search(), che è $O(\log n)$

## Ricerca - Breadth First Searc BFS

E' un algoritmo che ricerca all'interno di un albero (o grafo) ricercando prima in ogni Livello, partendo dalla root.
Questo alg. trova ogni nodo sullo stesso livello; facendo ciò traccia i nodi figli dei nodi del livello corrente.

Quando ha finito di esaminare un livello si sposta sul nodo più a sinistra del prossimo livello.

La BFS usa una Queue per salvare le informazioni sull'albero; per questo motivo usa più memoria della DFS.

## Ricerca - Depth First Search DFS

La DFS, al contrario della BFS, ricerca andando in profondità: arriva prima al nodo più a sinistra, quando raggiunge la fine del ramo, torna indietro ricorsivamente provando gli eventuali nodi destro, per prima, e poi sinistro. Il nodo più a destra è esaminato per ultimo.

### BFS - time Complexity

- Ricerca BFS: $O(|E|+|V|)$

### DFS - time Complexity

- Ricerca DFS: $O(|E|+|V|)$

### Pro e Cons

La BFS viene usata per alberi che sono più 'larghi' che profondi, questo perchè controlla livello per livello.
La DFS, invece, in caso di alberi molto profondi, potrebbe andare in profondità inutilmente.

## DFS & BFS

| | Tempo | Spazio |
|---|---|---|
| DFS | O(E+V) | O(Altezza) |
| BFS | O(E+V) | O(Lunghezza) |

Altezza: Altezza max dell'albero
Lunghezza: Numero numero di nodi in un singolo livello.

### DFS

- Migliore quando l'obbiettivo è più vicino alla root
- Stack -> LIFO
- Ricerca Preorder, inorder Postorder
- Va in profondità
- Ricorsiva
- Veloce

### BFS

- Migliore quando l'obbiettivo è lontano
- Queue -> FIFO
- Ricerca Level order
- Va in larghezza
- iterativa
- Lenta

## Binary Search Big O

| | Tempo | Spazio |
|---|---|---|
| BS | O(logn) | O(1) |

## Heaps

Gli heaps possono essere di due tipi: **MinHeap** e **MaxHeap**, la differenza tra i due è che il primo elemento può essere il più piccolo o più grande della collezione.

In un **MinHeap** tutti gli elementi sono più piccoli dei loro figli. Quindi, scendendo lungo l'albero, gli elementi diventano sempre più grandi.
Questo però non ci dice che la struttura sia ordinata, sappiamo infatti che la **root** è l'elemento più piccolo, ma non se gli altri elementi sono ordinati.

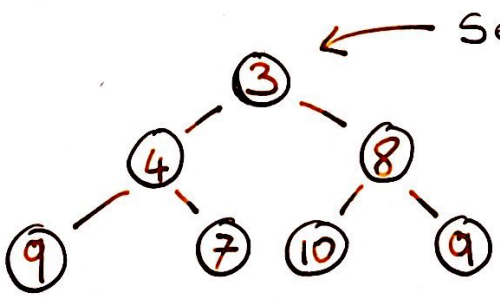
Abbiamo un ordinamento completo solo quando effettuiamo il pop() dell'elemento minimo, questo perché dopo ogni pop() la struttura deve essere 'riarrangiata', portando il min in prima pos.

### Inserimento

Quando inseriamo un elemento, esso viene sempre posto nel primo posto libero alla fine della struttura. Se l'elemento non rispetta l'ordinamento (deve essere più grande del suo genitore) allora si effettua un'operazione chiamata **Bubble-Up**, che fa' salire l'elemento fino alla posizione che rispetta l'ordinamento.

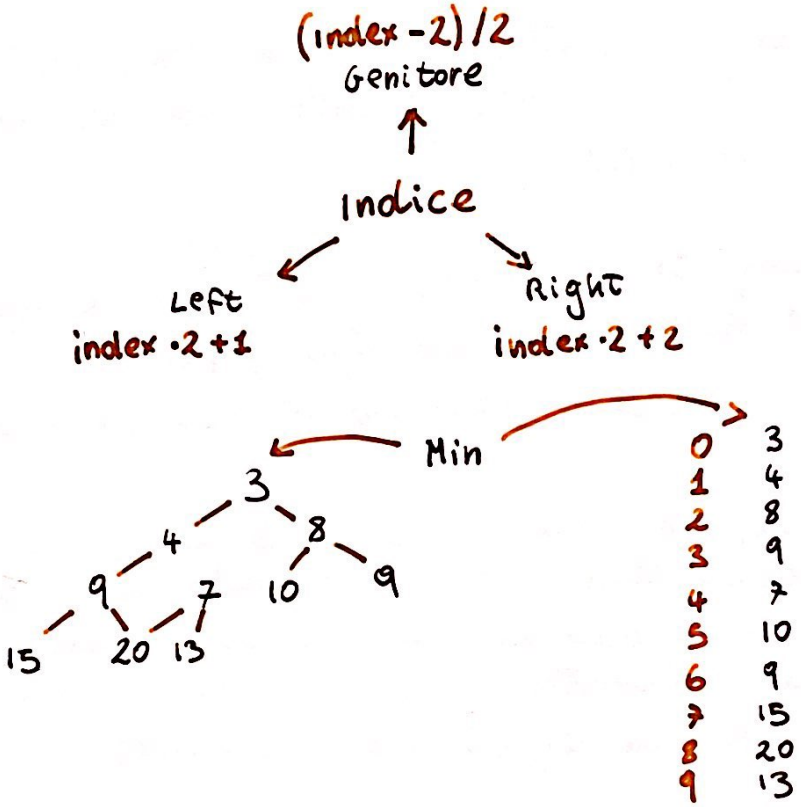
### Rimozione

Per rimuovere un elemento scambiamo la root con l'ultimo nodo, rimuoviamo l'ultimo nodo (la vecchia root) e successivamente effettuiamo il **Bubble-Down** della root. L'operazione fa 'scendere' l'elemento finché non rispetta la condizione.



## Implementazione

Potremmo pensare di implementare la struttura usando gli oggetti, e creando una classe Node, ma esiste un modo migliore per farlo: Guardando la struttura ci risulta semplice capire che gli elementi (padre e figli) si troveranno sempre in una posizione precisa, e possiamo quindi determinare la posizione di ogni nodo con una semplice equazione:



## Binary Search

La BS usa la struttura dati Array che è stata precedentemente ordinata. L'idea generale è quella di suddividere l'array ad ogni passaggio a metà, usando solo la metà dove potrebbe essere l'elemento che cerchiamo.

### Algoritmo

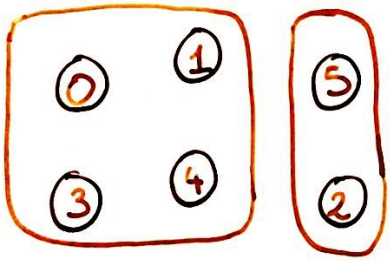
- Confronto la key con l'el mid dell'array
- Se la key è maggiore cerco a destra
- Se la key è minore cerco a sinistra
- Se la key è uguale l'ho trovata.

### Time complexity

Siccome dividiamo l'array ad ogni passaggio, l'algoritmo è molto efficiente con un tempo di esecuzione pari a $O(\log n)$.

## Union Find

Immaginiamo di avere dei gruppi di oggetti, un esempio potrebbe essere il seg:

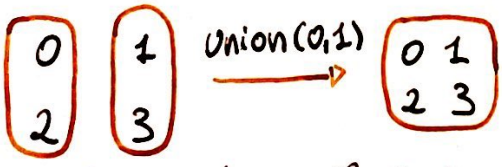



Una volta che abbiamo raggruppato gli oggetti, potremmo voler sapere alcune proprietà degli oggetti, come ad esempio

- Quanti gruppi ci sono?
- Due oggetti appartengono allo stesso gruppo?

La struttura supporta due operazioni:

1. Union (x, y) → unisce i gruppi contenenti x e y.




2. Find (x) → trova il gruppo a cui x appartiene.

### Utilizzi

L'union-Find può essere usata in molte applicazioni, tra cui:

- Trovare le componenti connesse in un grafo.
- Reti di amici in un Social Network.
- Diversi pixel dello stesso colore in un'immagine.