

Linked List

La linked list salva i dati in nodi collegati ad altri nodi.

①

| Time | Access | Search | Spazio: $O(n)$ |
|--------|--------|--------|----------------|
| worst: | $O(n)$ | $O(n)$ | |
| best: | $O(1)$ | $O(1)$ | |

Stack

Lo Stack è una struttura di tipo LIFO

Access: $O(n)$ Insert e delete: $O(1)$

Search: $O(n)$

Queue: da queue è una struttura di tipo FIFO

- Stesso time complexity dello Stack.

Binary Search

La binary search divide a metà l'array ad ogni passaggio e confronta la key con il mid.

Best case: $O(1)$ Space: $O(\log n)$

Avg case: $O(\log n)$

Union Find

La struttura UF supporta le op union e find su insiemi.

Unione: $O(n)$ se abbiamo m op e n el, con $n=m \Rightarrow$
ricerca: $O(1)$ \Rightarrow time complexity: $\underline{n^2}$

Selection Sort - in place

Il Selection Sort seleziona l'el più piccolo ad ogni passaggio e lo pone in-place

Worst: $O(n^2)$ Space: $O(1)$

Insertion Sort - in place

Funziona bene con array parzialmente ordinati. Simile al Selection, ma inserisce l'elemento corrente TRA gli elementi precedenti.

Worst: $O(n^2)$ Space: $O(1)$

Best: $O(n)$

Shell Sort - in place

L'algoritmo inizia ordinando coppie di elementi lontane tra di loro, accorciando il gap ad ogni iterazione.

Worst: $O(n^2)$ Space: $O(1)$

Best: $O(n \log n)$

Shuffle Sort - in place

Lo shuffle sort attribuisce ad ogni elemento un indice random(r) compreso tra 0 e i e scambiamo le pos i, r.

time compl: $O(n)$

Space: $O(1)$

Merge Sort

Il merge è un algoritmo ricorsivo che divide sempre a metà un sotto array e lo ordina ricorsivamente. Ogni due sottoarray effettua il merge dei due sottoarray.

Best: $n \log n$

Space: $O(n)$ ← not good!
non in-place

Worst: $n \log n$

Quick Sort in-place

È un algoritmo ricorsivo in-place. Nel QS ad ogni passaggio l'array viene partizionato in modo da porre il pivot in una posizione tale che tutti gli elementi alla sua sx sono più piccoli e tutti quelli a dx più grandi.

Partition: d'algoritmo si basa sul metodo partition() che fa scorrere i due indici i (da low-1) e j (da low) ed incrementa i quando l'el corrente è minore del pivot.

Average: $n(\log n)$

Worst: $O(n^2)$ → per questo motivo si effettua lo shuffle

Space: $O(\log n)$ ← Quasi Costante!

Per ogni 'iterazione' se $j < \text{pivot}$, incrementiamo i e scambiamo i con j . Ci fermiamo quando $j > \text{len-1}$ (pivot index). Alla fine del for scambiamo il pivot con $i+1$, ora il pivot è nella posizione finale.

Priority Queue

Le PQ sono collezioni di dati implementate solitamente ad albero o con un array il cui scopo principale è quello di rimuovere sempre l'elemento più grande o più piccolo.

insert del max max

Array non ordinato

1

n

n

Arr Ordin. obiettivo

\textcircled{n}
 $\log n$

1

$\log n$

1

$\log n$

Per via dell'ordinamento
da fare ogni volta.

Binary Heap

Un Btt è una struttura basata su un Albero binario. (2)

L'heap ha due condizioni:

- I figli di ogni nodo devono essere più piccoli (o grandi) e quindi avere meno (o più) priorità dei genitori.
 - Ogni livello dell'heap deve essere COMPLETO tranne per l'ultimo.
- Insert, worst: $O(\log n)$ \leftarrow per via del Bubble up
Insert, average: $O(1)$

HeapSort in-place

L'heapsort è un algoritmo in-place che sfrutta l'heap per ordinare gli elementi.

Siccome dobbiamo copiare gli elementi nella struttura, effettuare il poll ogni volta e controllare che TUTTI gli elementi siano in ordine, per ogni iterazione, l'HS non è molto efficiente.

best case: $O(n \log n)$ Space: $O(1)$

worst case: $O(n \log n)$

Binary Search tree

Gli alberi binari di ricerca sono simili agli Heaps, ma l'idea principale è che tutti i nodi a Sx di un nodo root sono più piccoli della root, e tutti quelli a dx sono più grandi.

Accesso, best e worst: $O(\log n)$, $O(n)$

→ Floor: trovare la chiave che è minore uguale di una data key.

\nwarrow nel caso di albero sbilanciato maggiore

→ Ceiling: trovare la chiave minore maggiore uguale di una data key.

→ Rank: Quante chiavi ci sono che sono \leq di una data key?

→ Select: Trovare una chiave, da un index, in orolone (ad esempio la settima key partendo dalla root)

• BFS: Ricerca in ampiezza

$O(|V|)$
Tempo

$O(|V|)$
Spazio

• DFS: Ricerca in profondità

Binary Search

La binary search usa un array ORDINATO per trovare una key.

Se la key $<$ mid allora cerca a Sx, nel caso opposto a Dx.

$O(\log n)$

$\frac{O(1)}{\nwarrow}$

\nwarrow molto buono

Il problema è ordinare l'array!

Mappe

Una Hash Map e' una struttura dati che implementa un tipo di dato astratto ASSOCIAZIONE. Nelle mappe, infatti viene associata una CHIAVE ad un VALORE.

Le mappe sono molto efficienti in quanto COSTANTI nel caso medio. Caso medio: O(1)

Worst case: $O(n) \leftarrow$ nel caso tutti gli el sono mappati in un unico index.

- Hashing: L'hash code e' un int a 32 bit e per oggi definiti dall'utente viene calcolato inizializzando l'hash ad un int non 0 e poi moltiplicando l'hash per un seed e addizionando un campo primitivo.

Esempio $\text{hash} = \text{seed} * \text{hash} + \text{campo primitivo}$
Per ogni campo primitivo \rightarrow

- Separate Chaining

E' un modo per risolvere le collisioni. Viene usato un array di linked lists dove vengono inseriti gli elementi. In caso di collistone due o più elementi vengono inseriti nella linked list in ordine di inserimento.

Worst case: $O(n) \leftarrow$ tutti gli el sono su un'unica lista.

Quanto deve essere lungo l'array?

L'array dovrebbe essere lungo $\sim N/4$ in modo da mappare tutti gli elementi uniformemente.

M troppo grande: Sprechi spazio

M troppo piccolo: Troppi elementi mappati sulla stessa lista.

- Linear Probing

Quando due chiavi collidono, mi muovo in avanti fino a trovare una posizione libera, inserendo l'elemento in quella pos.

• Ricerca: Per trovare l'elemento effettivo l'hashing dell'oggetto e cerco l'elemento a quella posizione. Se non e' presente l'i, lo cerco nelle posizioni successive finche' non trovo una posizione null.

Sep.Ch. VS Lin. Pr.

All'atto pratico il Linear Probing e' meno efficiente; questo perche' quando devo eliminare un elemento e' necessario effettuare il re-hashing per tutti gli elementi, consumando spazio.

Alberi bilanciati

Negli alberi 2-3 possiamo avere due tipi di nodi:

- Nodi di tipo 2: Una chiave e due figli
- Nodi di tipo 3: Due chiavi e tre figli

Il sottoalbero a Sinistra ha tutte le chiavi che sono più piccole della prima chiave (K_1) il nodo centrale ha tutte le chiavi comprese tra K_1 e K_2 , e quello a Dx tutte quelle $> K_2$.

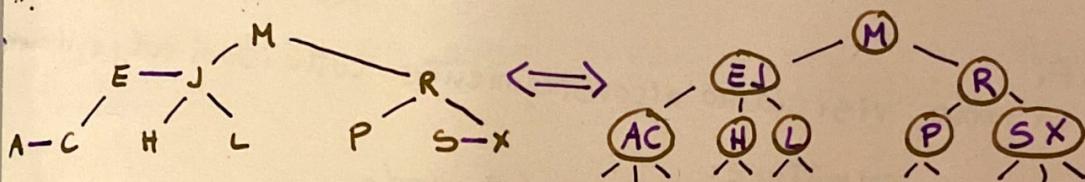
Questa Struttura garantisce un tempo: $O(\log n)$ ← buono!

Alberi Red-Black

Usiamo dei Link Rossi che possono comparire solo a Sinistra.

Questi Link Servono a rappresentare un NODO di tipo 3

Dimostriamo la corrispondenza tra 2-3 e R-B considerando i link rossi come orizzontali



Rotazione a Sinistra

Nel caso in cui ho un link rosso con figlio Dx, devo portare il link rosso a Sinistra.

Grafi non Orientati

Un grafo è una struttura dati composta da un insieme di nodi (V) ed un insieme di archi (E).

Cammino: Una sequenza di nodi connessi da archi.

Ciclo: Un cammino i quali primo e ultimo nodo coincidono.

Rappresentazione

| | Spazio | Add arco | Esiste $e \rightarrow w$? | Iterazione |
|----------------|--------------------|----------|----------------------------|------------|
| lista di archi | (N) E | 1 | E | E |
| Matrice Adj | V V^2 | 1 | 1 | V |
| Lista Adj | $E + V$ | 1 | degree (v) | degree (v) |

DFS nei grafi

- Partiamo da un nodo, visitiamo ricorsivamente tutti i suoi adiacenti e li marchiamo.

Nella DFS abbiamo 2 strutture:

- **Marked:** Tutti i nodi raggiungibili da un dato nodo.
- **edgeTo:** Il predecessore di ogni nodo.

BFS nei grafi

La BFS NON usa la ricorsione, ma una CODA. Aggiungiamo gli elementi alla coda nell'ordine in cui essi vengono visitati (adiacenti al nodo corrente).

- ★ La BFS viene usata nei grafi: DIREZIONATI!

Grafi Orientati

- **Applicazioni BFS:** Percorso minimo da tutti i vertici del grafo a tutti gli altri vertici

Ordinamento topologico

Il T.S. dispone i vertici in modo che tutti gli archi puntino in un'unica direzione. Usa una struttura dati per salvare l'ordine degli elementi.

Sei All qui 1) Eseguiamo lo DFS

2) Ritorniamo i vertici contenuti nella struttura postorder; in reverse!

Greedy

MST: Connesso, Aciclico, Include tutti i vertici.
L'algoritmo effettua dei tagli che attraversano gli archi del grafo e lo dividono in due sottoalberi. Trova l'arco, di quelli toccati dal taglio, più piccolo e lo aggiunge all'MST.

Kruskal

E' usato per calcolare i MST su grafi NON orientati oventi archi con pesi NON negativi.

Tempo: $O(E \log v)$

Spazio: $O(E + v)$

Con questo Alg. Per prima cosa ordiniamo in ordine crescente gli archi, dopodiché gli esaminiamo. Se un arco non crea un CICLO lo aggiungiamo all'MST.

Per vedere se un arco crea un ciclo usiamo la struttura union-find

Prim

Esaminiamo inizialmente un vertice e prendiamo in cons. i suoi archi uscenti. Aggiungiamo all'MST l'arco con peso minore. Ci fermiamo quando abbiamo V-1 Archi.

Tempo: $O(E + v \log v)$

Kruskal VS Prim

Kruskal crea l'MST a machia di Leopardi, mentre Prim lo crea a machia d'olio.

Con Prim scegliamo l'arco più vicino all'albero.

Prim viene usato in GraFi NON orientati.

Dijkstra

\rightarrow GraFi Orientati pesati
In questo Algoritmo Teniamo traccie sia dei pesi che dei predecessori di ogni nodo. (`distTo[]` e `edgeTo[]`)

uso la struttura `distTo[]` per scegliere ad ogni iterazione il nodo di peso minimo.

Esaminando i suoi archi uscenti ed aggiorno le strutture dei nodi raggiunti. Se un nodo raggiunto è settato a $+\infty$ aggiorno peso e predecessore.

Se il nodo è già raggiunto da altri, con peso minore al nodo corrente non aggiorno.

Se il path corrente è minore del precedente allora aggiorno.

Tempo: $O(E + v \log v)$

Lower bound

L' L.B. è il tempo di esecuzione nel caso migliore. Il bound è infatti definito dall'input più semplice, esso è l'obiettivo per tutti i tipi di input.

Upper bound

L' U.B. è il tempo di esecuzione nel caso peggiore ed è determinato dall'input più difficile.

L' U.B. è di fatto una garanzia per tutti i tipi di input, infatti ci dice che un dato algoritmo NON avrà performance peggiori dello stesso lower bound.

Heap su array

