LAB. DI ALGORITMI

APPUNTI A CURA DI: RICCARDO LO IACONO

Università degli studi di Palermo
a.a. 2023-2024

Indice.

– 1 – Strutture dati astratte: alberi.

Tra le varie strutture dati astratte, gli alberi sono sicuramente quelli maggiormente utilizzati e di maggior importanza. Di questi, ne esistono molteplici varianti, ciascuna con uno scopo ben preciso; fatto sta che tutte queste varianti sono accomunate dall'efficienza.

– 1.1 – BST: binary search trees.

Prima di procedere con il discutere varianti di alberi più complesse, si procede a fare un richiamo al concetto di albero binario di ricerca. Questi si ricorda essere una tipologia di albero binario che, dato S un insieme di elementi ordinati, memorizza gli stessi in un nodo dell'albero in modo tale che, posto $x \in S$, si abbia

- per ogni altro y nel sotto-albero sinistro con radice x, si abbia

  $$key[y] \leq key[x]$$

  cioè, ogni elemento del sotto-albero sinistro deve avere un valore minore o uguale, a quello della radice del sotto-albero stesso;

- per ogni ogni altro y nel sotto-albero destro radicato in x, si abbia

  $$key[y] > key[x]$$

  cioè, ogni elemento del sotto-albero sinistro deve avere un valore maggiore, a quello della radice del sotto-albero stesso.

Si ricorda brevemente che, posto h l'altezza dell'albero, le operazioni di inserimento, ricerca e cancellazione sono tutto di costo $\mathcal{O}(h)$. Si ha quindi che, poiché

$$h = \begin{cases} \log_2(n) \text{ , se l'albero è perfettamente bilanciato;} \\ \log_2(n) \leq k \leq n \text{ , se l'albero non è perfettamente bilanciato;} \\ n \text{ , se completamente sbilanciato,} \end{cases}$$

nel caso pessimo si ha un costo di $\mathcal{O}(n)$.

– 1.2 – AVL trees: Adelson-Velsky-Landis trees.

Gli AVL sono una tipologia di alberi binari di ricerca bilanciati in altezza. Nello specifico, si dice che un AVL è bilanciato se questi ha, per ogni sotto-albero, un *fattore di bilanciamento* B_f minore o uguale ad uno.

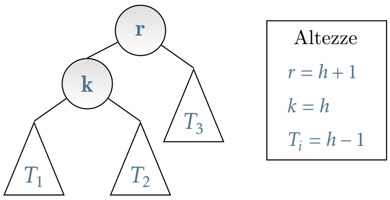
Qui per fattore di bilanciamento si intende la differenza in modulo tra l'altezza dei due sotto-alberi.

Per quel che riguarda le operazioni: essendo, come detto, che gli AVL sono dei BST, e poiché essa non modifica la struttura dell'albero, la ricerca è analoga a quella dei BST; inserimento e cancellazione viceversa, proprio perché modificano la struttura dell'albero, e rischiano di sbilanciarlo, sono modificate in modo tale che a seguito di esse l'albero risulti ancora bilanciato. Tale modifica consiste nelle operazioni di rotazione descritte a seguito.

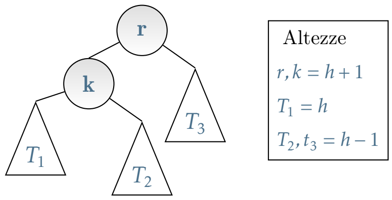
Osservazione. Sebbene l'aggiunta delle operazioni di rotazione nel caso di inserimento e cancellazione, ciascuna delle tre operazioni richiede al più tempo $\mathcal{O}(\log_2(n))$: questo perché proporzionali all'altezza dell'albero, e poiché il costo di ogni rotazione è $\mathcal{O}(1)$.

– 1.2.1 – Ribilanciamento di un AVL.

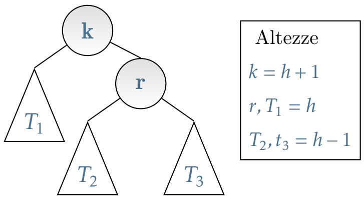
Alla base del processo di bilanciamento vi sono le operazioni di rotazione a sinistra e a destra. Per comprendere tali operazioni, si faccia riferimento a *Figura 1.a*. Si supponga di aggiungere ad T_1 un nodo, portando così a uno sbilanciamento dell'albero (*Figura 1.b*). In questo caso si dimostra sufficiente una singola rotazione a destra, a seguito della quale l'albero risulta bilanciato (*Figura 1.c*). L'operazione appena descritta prende il nome di rotazione destra-destra, a questa si aggiungono la rotazione sinistra-sinistra (simmetrica alla rotazione destra-destra) e le rotazioni sinistra-destra, destra-sinistra tra loro simmetriche.



1.a

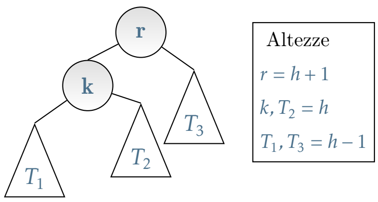


1.b

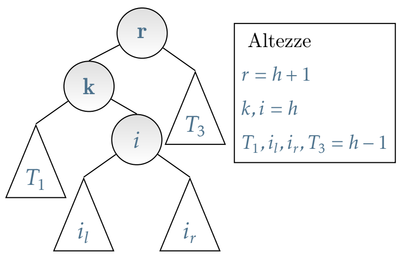


1.c

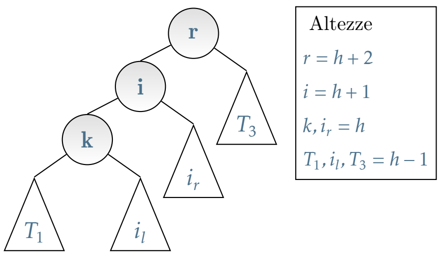
Sia ora considerata l'operazione di rotazione sinistra-destra: prendendo in riferimento l'albero di *Figura 1.a*, si supponga di aggiungere al sotto-albero T_2 un nodo; come evidente da *Figura 2.a* l'albero risulta ora sbilanciato. Poiché si osserva banal-



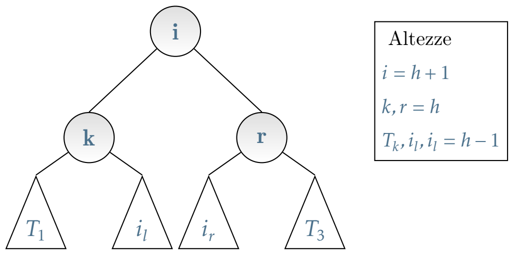
2.a



2.b



2.c



2.d

mente che una sola operazione di rotazione non è sufficiente; sia considerata la radice del sotto-albero, sia questa i (*Figura 2.b*), in tal modo eseguendo dapprima una rotazione a sinistra (*Figura 2.c*) e successivamente una a destra (*Figura 2.d*), l'albero risulta nuovamente bilanciato.

– 1.3 – 2-3 Trees.

Definizione: un albero che in ogni suo nodo interno abbia due o tre figli, tali che

- se il nodo è un 2-nodo, questi abbia due link, rispettivamente sinistro e destro, tali che, il figlio sinistro abbia chiave minore del nodo e il figlio sinistro chiave maggiore;
- se il nodo è un 3-node, in aggiunta ai link precedenti ne presenta un centrale in cui figli hanno chiave compresa tra il link sinistro e destro;

si dice essere un albero 2-3.

Come visto per gli alberi binari, anche nel caso degli alberi 2-3 si dimostra esservi un legame tra altezza dell'albero e numero di nodi; in particolare si ha il seguente teorema.

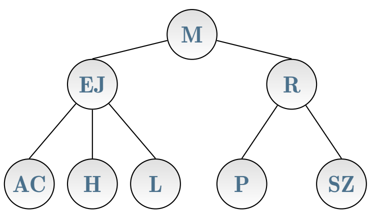
Teorema 1.1. *Sia* T *un albero 2-3. Posti* n *il numero di nodi,* f *il numero di foglie e* h *l'altezza dell'albero, si ha*

$$2^{h+1} - 1 \leq n \leq \frac{3^{h+1} - 1}{2}$$
$$2^h \leq f \leq 3^h$$

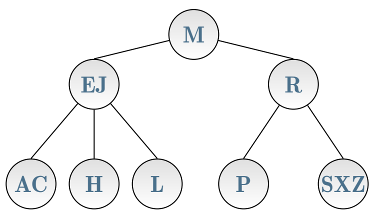
Dimostrazione: segue banalmente procedendo per induzione su h.

Parlando delle operazioni: la ricerca è analoga a quella dei BST; inserimento e cancellazione, similarmente a quanto detto per gli AVL, rischiano di sbilanciare l'albero, sono per tale ragione modificate così da ripristinare la condizione di albero 2-3. In questo caso, il bilanciamento è effettuato eseguendo opportunamente la procedura addSon: sostanzialmente, nel caso dell'inserimento, se questi è da eseguire su un 2-nodo, non si hanno problemi; se si inserisce in un 3-nodo, si divide il 4-nodo venuto a formarsi. Tale divisione è dipendente dal genitore p del 4-nodo, se infatti p è un 2-nodo, si aggiunge a questi l'elemento centrale del 4-nodo; se invece p è a sua volta un 3-nodo, si procede ricorsivamente eventualmente creando una nuova radice.

Esempio: sia considerato l'albero di *Figura 3.a*, e a questi di aggiungere X. Si è in tal modo creato un 4-nodo (SXZ di *Figura 3.b*).



3.a: Esempio di albero 2-3 bilanciato.



3.b: Esempio di albero 2-3 sbilanciato.

Circa il costo, si dimostra che tutte le operazioni sono $\mathcal{O}(\log_2(n))$.

– 1.4 – R-B Trees.

I Red-Black trees sono una variante degli alberi binari di ricerca, che assicurano che il costo delle operazioni sia $\mathcal{O}(\log_2(n))$. In particolare, ad ogni nodo interno si attribuisce una colorazione rossa o nera; colorazione che è assegnata dal nodo padre. Nella loro versione classica, radice e foglie sono colorati di nero, e nessun nodo rosso può avere un figlio rosso.

– 1.4.1 – Inserimento.

Sfruttando le operazioni di rotazione a sinistra e a destra descritte per gli AVL, e considerando che vi è una corrispondenza 1 a 1 tra R-B Trees e 2-3 Trees, si ha che: se l'inserimento è relativo un 2-nodo, si inserisce come in un BST, si colora il link di rosso e se quest'ultimo è a desta, si effettua una rotazione a sinistra; se l'inserimento è invece relativo un 3-nodo, si effettua un eventuale rotazione per bilanciare il 4 nodo venuto a crearsi, si commutano i colori per passare il link rosso verso l'alto e, in fine, si esegue, se necessario, una rotazione per mantenere il link rosso a sinistra.

– 1.5 – B-Trees.

Definizione: sia $M = 2h, h > 0$, si definisce B-Tree di ordine M un albero con k nodi interni o un k-nodo, tali che

- ogni cammino radice-nodo esterno ha la stessa lunghezza;
- per la radice $2 \leq k \leq M - 1$;
- per ogni altro nodo $M/2 \leq k \leq M - 1$.

Osservazione. Si può pensare ai B-Tree come ad una generalizzazione degli alberi 2-3.

Qui $*$ rappreseta una chiave sentinella, la cui chiave è minore di ogni altra.

Esempio: sia $M = 4$. Un B-Tree di ordine 4 è quello in *Figura 4*.

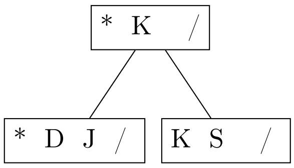


Figura 4: Esempio di B-Tree.

Circa le operazioni, si dimostra che, dato un B-Tree di ordine M con N chiavi, queste richiedono un tempo compreso tra $\mathcal{O}(\log_{M-1}(N))$ e $\mathcal{O}\left(\log_{M/2}(N)\right)$.
Osservazione. I B-Tree sono generalmente utilizzati per la gestione della memoria. Nello specifico per operazioni di I/O. Segue da questo che scegliendo blocchi proporzionali alla dimensione dei blocchi di memoria B, la complessita si riduce a $\mathcal{O}(\log_2(N))$.

– 1.5.1 – Ricerca.

L'operazione di ricerca è banale; questa infatti consiste nel trovare di volta in volta l'intervallo per la chiave, e seguendo il link relativo giungere a una foglia.

– 1.5.2 – Inserimento.

Banale tanto quanto la ricerca, consiste nel ricercare la chiave, chiave che se non è trovata viene inserita. Risalendo, se presenti, procede a separare i nodi con M chiavi.

– 2 – String sorting.

Si assume noto che relativamente al sorting, esiste un lower-bound di $\mathcal{O}(n\log_2(n))$ per quel che riguarda l'ordinamento di dati primitivi, presupposto l'esistenza di una relazione d'ordine tra gli stessi. Relativamente le stringhe (così come per altri dati non elementari), tale lower-bound risulta non essere corretto. Ciò segue da una semplice osservazione: siano s_1, s_2 due stringhe da ordinare, si nota banalmente che è necessario confrontare almeno l'*lcp* tra le due, ossia è necessario confrontare almeno il prefisso comune alle due. Generalizzando a n stringhe vale il seguente teorema.

Teorema 2.1. *Sia $R = \{s_1, \dots, s_n\}$ insieme di n stringhe. Allora, se utilizzato un algoritmo basato su confronti, ordinare R richiede $\Omega(\Sigma LCP(R) + n\log_2(n))$*

In questa sezione si discuteranno algoritmi e strutture dati studiati per ordinare efficientemente le stringhe.

– 2.1 – 3-Way quicksort.

Prima di descrivere il 3-way-quicksort, si procede a ricordare il quicksort nella sua versione base. Supposto S un insieme di dati da ordinare, dati sui quali esiste una relazione d'ordine, si procede scegliendo ricorsivamente uno degli elementi (il pivot) e sulla base di questi si suddivide l'insieme in due sottoinsiemi; il primo contenente quegli elementi di S tali che questi risultino minori o al più uguali al pivot, il secondo contenente quegli elementi che risultano invece maggiori. Si procede applicando la procedura per ciascuno dei sottoinsiemi, terminando quando i sottoinsiemi contengono un solo elemento.

Essendo una sua evoluzione, 3-way-quicksort procede similarmente al classico quicksort con una sola differenza: anziché formare due sole partizioni, se ne costruiscono tre, identificando pertanto i sottoinsiemi di elementi minori, uguali e maggiori al pivot scelto. Da un punto di vista implementativo, lo pseudo-codice è il seguente.

```
function 3_way_quicksort(Array R, int currPos)
if |R| ≤ 1 then return R;
R⊥ = {s ∈ R : |s| < currPos};
R = R − R⊥;
choose pivot x ∈ R;
R< = {s ∈ R : s[currPos] < x[currPos]};
R= = {s ∈ R : s[currPos] = x[currPos]};
R> = {s ∈ R : s[currPos] > x[currPos]};
R< = 3_way_quicksort(R>, currPos);
R= = 3_way_quicksort(R=, currPos + 1);
R> = 3_way_quicksort(R>, currPos);
return R⊥ · R< · R= · R>
```

Figura 5: Pseudo codice 3_way_quicksort(stringSort).

Osservazione. Qui con $|R|$ si indica il numero di stringhe rimaste.

– 2.2 – Radix sort.

Sebbene nato per l'ordinamento di interi, il radix sort, per il modo in cui opera, può essere inteso come un'algoritmo per l'ordinamento di stringhe.

Si distinguono

- *MSD radix sort:* per cui l'ordinamento è effettuato a partire dalla cifra più significativa;
- *LSD radix sort:* con cui si ordina a partire dalla cifra meno significativa.

Osservazione. Tutte le stringhe/numeri hanno lo stesso numero di cifre/caratteri.

– 2.2.1 – LSD radix sort.

Sia $R = \{s_1, \ldots, s_n\}$ insieme di n stringhe, sia $(0, \sigma)$ l'alfabeto su cui sono definite le s_i. Allora, è possibile ordinare R con LSD radix sort (*Figura 6*).

```
function LSD_radix_sort(Array R, int size)
for l = size - 1 to 1 do
    countingSort(R, l);
return R;
```

Figura 6: Pseudo codice LSD radix sort.

Si dimostra che LSD ha costo $\mathcal{O}(\|R\| + m\sigma)$ con $\|R\|$ lunghezza totale delle stringhe.

– 2.2.2 – MSD radix sort.

Sia $R = \{s_1, \ldots, s_n\}$ insieme di n stringhe, sia $(0, \sigma)$ l'alfabeto su cui sono definite le s_i. Allora, è possibile ordinare R con MSD radix sort (*Figura 7*).

```
function MSD_radix_sort
if |R| < σ then
    return quickSort(R, l);
R⊥ = {s ∈ R : |s| = l};
(R1,...,Rσ) = countingSort(R,l);
for i = 0 to σ-1 do
    Ri = MSD_radix_sort(Ri,l+1);
return R⊥ · R0 · R1 · ... · Rσ;
```

Figura 7: Pseudo codice MSD radix sort.

Si dimostra che MSD ha costo $\mathcal{O}(LCP(R) + n\log_2(\sigma))$.

– 2.3 – Trie.

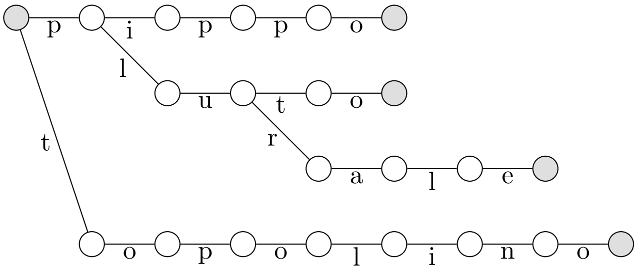
Un *trie* è una struttura ad albero che permette di rappresentare efficientemente le stringhe. Più precisamente, sono definiti come segue.

Definizione: sia Σ un alfabeto, si definisce trie un albero radicato i cui nodi hanno al più $|\Sigma|$ figli; si deve avere inoltre che

1. ogni arco è etichettato con un simbolo $x \in \Sigma$;

2. per ogni coppia di nodi v, w distinti, ogni cammino da v verso una sua foglia è diverso da ogni cammino di w a una sua foglia.

Esempio: sia $S = \{pippo, pluto, plurale, topolino,\}$. Il suo trie è quello mostrato di seguito.



Osservazione. Qui ◯, oltre ad indicare la radice del trie, è utilizzata per indicare il termine di una stringa.

Per quanto concerne le operazioni di inserimento (di cui in *Figura 8* è riportata l'implementazione), cancellazione e ricerca, si dimostra che tutte richiedono tempo $\mathcal{O}(|S|)$ e spazio $\mathcal{O}(|S||\Sigma|)$.

```
procedure addToTrie(Trie root, String s)
    v = root;
    j = 0;
    while child(v, s[j]) ≠ null do
        v = child(v, s[j]);
        j++;
    while j < m do
        create node u;
        for each c∈Σ do
            child(u, c) = null;
        child(v, s[j]) = u;
        v = u;
        j++;
    mark v rappresentative of s;
```

Figura 8: Pseudo codice inserimento di una stringa in un trie.

– 2.3.1 – PATRICIA tree.

I *practical algorithm to retrive information coded in alphanumerics tree* o più semplicemente PATRICIA tree, sono una versione compatta dei trie. L'idea sostanziale è quella di non assegnare un arco ad ogni carattere della stringa, bensi utilizzare un'unico arco per prefissi comuni a più stringhe. Con tale definizione, il trie di dell'esempio precedente, si riduce a quanto mostrato in *Figura 9*.

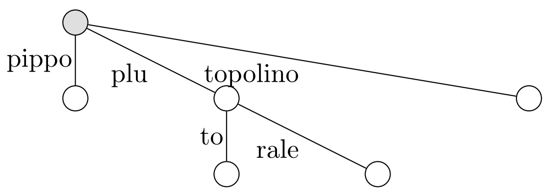


Figura 9: PATRICIA tree di *Figura 8*.

– 2.3.2 – Ternary search tree.

Un ternary search tree, è un trie in cui ogni nodo ha tre link: uno per sinistro che lo collega a stringhe di ordine inferiore, uno destro che lo collega a stringhe di ordine superiore e un link centrale che indica una relazione di uguaglianza tra nodi.

– 3 – Pattern matching.

Partendo dall'identificare il problema: sia T un testo di lunghezza n e sia P un pattern (una sotto-stringa) di lunghezza m da ricercare in T, con $n >> m$, in generale; di interesse è verificare se esista un qualche $T[i, i+m]$. Ossia, si è interessati a verificare se esista almeno un'occorrenza di P in T.

Soluzione più semplice, per tale ragione detta *naive*, consiste nel verificare per ogni carattere se i successivi m caratteri identificano un pattern. Risulta però ovvio che tale soluzione è inefficiente in termini di tempo, richiedendo infatti $\mathcal{O}(nm)$ confronti. Esiste però un algoritmo molto più efficiente, il *Knuth-Morris-Pratt* discusso a seguire.

– 3.1 – Algoritmo di Knuth-Morris-Pratt (KMP).

L'algoritmo, che deve il nome agli sviluppatori che lo hanno scoperto, parte da una semplice osservazione: ogni qualvolta si ha un mismatch, anziché ricercare il pattern dal carattere successivo, è più sensato riprendere la ricerca da quella porzione di patter trovato che risulti essere prefisso del pattern effettivo e suffisso di quello già trovato. Per quanto concerne l'implementazione, KMP parte col costruire un apposito automa a stati finiti, (*Figura 10*), e simula l'esecuzione dello stesso con il patter da ricercare.

```
void KMP_pattern_matching(String text, String pattern)
    int m = pattern.length;
    int** dfa = new int*[ALPHABET];
    for (int i = 0; i < ALPHABET; i++)
        dfa[i] = new int[m];
    dfa[pattern[0]][0] = 1;
    for (int x = 0, j = 1; j < m; j++)
        for (int c = 0; c < ALPHABET){
            dfa[c][j] = dfa[c][x];
            dfa[pattern[j]][j] = j++;
            x = dfa[pattern[j]][x];
        }
```

Figura 10: Codice C++ per la costruzione dell'automa per KMP.