**INDICE**

1. **Algoritmi e Complessità**
   1. Teoria e Masther Theorem 2
2. **Sorting**
   1. Insertion Sort - Selection Sort 3
   2. Bubble Sort – Counting Sort – Binary Heap 4
   3. Heap, Merge, Quick, Radix Sort 5
3. **Alberi e liste**
   1. Teoria – Alberi Binari – BST 6
   2. AVL e bilanciamento – BTREE 7
4. **HashMap**
   1. Teoria 10
5. **Teoria dei Grafi**
   1. Teoria 11
   2. Kruskal - Prim – Dijkstra – Flusso 12
   3. Rete residual – Tagli 13
6. **Programmazione Dinamica**
   1. Teoria ed esempio 14
   2. Esempi 15
7. **Algoritmi di Compressione**
   1. Teoria - RLE – VLE 16
   2. Huffman – Shannon Fano 17
   3. LZ77 – LZ78 18
   4. LZW 19

**Istanza di un algoritmo**: è l’insieme di dati che devono essere processati dall’algoritmo, un algoritmo si dice corretto se l’output soddisfa le specifiche del problema per tutte le possibili istanze.

**Istruzione atomica**: è un’istruzione macchina a basso livello che ha tempo di esecuzione costante, O(n)

**Running Time**: è il numero di istruzioni atomiche eseguite, dipende dalla dimensione dell’input

Ordini di grandezza degli algoritmi



* Big-oh O() : indica il caso pessimo, limite superiore della funzione,
* Big-omega Ω(): indica il caso migliore, limite inferiore della funzione
* Theta Θ(): indica il caso medio, (non sempre facile da determinare)

T(n) è la funzione che determina il tempo di esecuzione

Proprietà

1. Le costanti possono essere trascurate, in tutti e 3 i casi
2. Proprietà transitiva: *se T(n) è O(f(n)) e f(n) è O(g(n)) allora T(n) è O(g(n))*
3. **(Solo per Theta)** proprietà di simmetria: *f(n) è Θ(f(n)) allora g(n) è Θ(f(n))*
4. **(Per O(n) e Ω(n))** proprietà di trasposizione simmetrica: *f(n) è O(g(n)) allora g(n) è Ω(f(n))*

**Complessità delle istruzioni**

Le istruzioni semplici hanno complessità O (1)

For loop: un ciclo avente una funzione F all’interno ha complessità

* O(n) se F è un’istruzione semplice
* O(n\*f(n)) se F ha un running time uguale a O(f(n))

While e Do-while: non si conosce il numero esatti di iterazioni quindi il caso pessimo è uguale a quello dei for loops

If-else: if(C) then B else D, si avranno le seguenti complessità

* C normalmente è O (1)
* se B e D sono istruzioni semplici allora avranno O (1)
* se B ha running time (RT) O(f(n)) e D ha RT O(g(n)) allora la complessità sarà O(max(f(n), g(n))

Funzioni: si sommano le complessità delle varie funzioni

**MASTER THEOREM**

Funzioni ricorsive: si dovrebbe trovare la funzione per induzione oppure, scelta migliore, si usa il master theorem

Caso in cui l’input viene decrementato di una costante

a>0, numero di chiamate ricorsive nella funzione

b>0, decremento della dimensione della dimensione di input

f(n)=O(f(n)), rappresenta la complessità della parte non ricorsiva

Casi

* se a=1 allora
* se a>1 allora

k è nella complessità della parte non ricorsiva

Caso in cui l’input viene diviso per una costante

a≥1, numero di chiamate ricorsive nella funzione

b>0, costante divisoria della dimensione della dimensione di input

f(n)=O(nklogp(n)), rappresenta la complessità della parte non ricorsiva

Casi:

* se logb(a)>k allora
* se logb(a)=k allora
  + se p > -1:
  + se p = -1:
  + se p < -1:
* se logb(a)<k allora
  + se p ≥ 0 :
  + se p < 0 :

k e p sono nella complessità non ricorsiva

**Sorting**

Un algoritmo di ordinamento consente l’ordinamento di elementi, esso può essere:

* In place: non utilizza memoria ausiliaria (es. array temporaneo)
* Stabile: gli elementi sono ordinati secondo un ordine relativo (es. [Ar,Br,An,Cr]->[Ar,An,Br,Cr]) gli elementi sono si ordinati, ma seguendo l’ordine di input

**Algoritmi:**

**Insertion-Sort:**

* Complessità: O(n2)
* In place: Sì
* Stabile: sì

Procedimento

1. Parto dal secondo elemento e lo controllo con quello precedente e se minore SWAP
2. Vado al terzo elemento e lo controllo col secondo, se minore SWAP. Una volta swappato tengo ancora in considerazione l’elemento che sto spostando e lo controllo anche col primo elemento dell’array
3. Ora vado alla quarta posizione e faccio la stessa cosa (prima controllo col 3, se swappo controllo col 2, eccetera)

In generale: partendo da un indice i=1 (secondo elemento) controllo l’elemento con tutti quelli precedenti (questo è l’indice **j** che si decrementa), swappando coppia per coppia ogni volta che minore e fermandomi quando quello precedente è maggiore. Quando passo all’elemento successivo da controllare, parto dalla posizione successiva da cui ero partito (i+1)

**Selection-Sort:**

* Complessità: O(n2)
* In place: Sì
* Stabile: no, ma esiste una versione stabile dell’algoritmo

L’array si divide in 2, parte ordinata e non ordinata, si seleziona il primo elemento della parte non ordinata e si cerca il minimo, elemento più piccolo, e si scambiano di posizione. Si va avanti con il nuovo elemento della partizione non ordinata e si rieseguono le stesse operazioni fino alla fine dell’array.

**Bubble-Sort:**

* Complessità: O(n2)
* In place: Sì
* Stabile: sì

Si basa sugli swap tra elementi, si scansionano gli elementi da sinistra verso destra se il successivo è minore si scambiano. L’array viene ridotto dal fondo e termina quando non ci sono più swap da fare.

(può essere fatto anche dal fondo dell’array in modo inverso)

**Counting-Sort:**

* Complessità: O(n)
* In place: no
* Stabile: sì
* Gli elementi devono essere interi compresi in un range che va da 0 a k

Si scansiona l’array e si contano le frequenze (array di supporto inizializzato a 0) per ogni valore incrementando l’elemento nella posizione letta

Si sommano alla posizione i-esima il valore dell’elemento della pozione precedente (es: [0,1,2,0] => [0,1,3,3]) il valore massimo indica la dimensione dell’array di output che è uguale a quella di input

Parto leggendo gli elementi dell’array di input uno ad uno e vado alla posizione di quell’elemento nell’array di supporto. Quel numero mi indica la posizione in cui inserire l’elemento nell’array di output. Una volta inserito, decremento il valore di quella cella dell’array di support.

**Attenzione! ->**  l’array di support indica la posizione n-esima, non l’indice i! quindi fare -1 se lavori con indici.

**Binary-Heap:**

Sfrutta un albero binario quasi completo (facile da rappresentare in un array), ogni livello è pieno. I nodi devono soddisfare la heap-property

Max-heap property: il valore di ogni nodo figlio è minore del padre, se la max-heap property non è rispettata allora l’albero non è un heap-tree[viene usata per heap sort]

Min-heap property: il valore di ogni nodo figlio è maggiore o uguale a quello del nodo padre

Rappresentare uno heap con un array:

* il primo elemento è la radice
* i padri sono fino a **N/2** (se ho 10 elementi, i primi 5 sono root e padri)
* gli indici dei figli sinistri sono a **2\*i** (i parte da 1, esempio -> la root ha indice 1 con figlio sx indice 2)
* gli indice dei figli destri sono a **2\*i+1** (i parte da 1, esempio -> la root ha indice 1 con figlio dx indice 3)

tutti i nodi possono raggiungere i figli e il padre

Heapify (o BuildHeap) è l’operazione che ripristina la heap-property, è un’operazione ricorsiva O(n\*log(n))

(nel caso di Heapify con MaxHeapProperty)

1. Si parte dal primo nodo non foglia dal basso a destra dell’ultimo livello (o nell’array l’ultimo padre)
2. Si verifica se esso è maggiore dei figli, altrimenti si fa SWAP con figlio maggiore
3. Si passa al prossimo nodo non foglia (a sinistra nel grafo, oppure a sinistra nell’array)
4. Si verifica anche qui se è maggiore dei figli e nel caso si fa lo swap
5. Se si fa lo SWAP bisogna continuare ricorsivamente a controllare nei figli per “farlo scendere il più possibile”

**Heap-Sort:**

* Complessità: O(n\*log(n))
* In place: si
* Stabile: no

L’algoritmo consiste nella costruzione dello heap usando la max-heap property. L’ordinamento (sfrutta l’heapify) avviene prendendo l’elemento più grande, che si trova nel nodo radice (primo elemento del vettore), e viene scambiato con l’ultimo elemento del vettore. La dimensione del vettore viene decrementata in modo da congelare gli elementi allo stesso tempo si effettua heapify escludendo l’elemento congelato. Si ripete fino alla riduzione a 0 della dimensione del vettore così che si sono congelati tutti gli elementi.

**Divide and Conquer**

È un paradigma di programmazione per risolvere problemi ricorsivi,

(Divide)dividi il problema in parti più piccole->(Conquer)risolvi i problemi->(Combine)combina le soluzioni:

merge sort e quick sort sfruttano questo paradigma

**Merge-Sort:**

* Complessità: O(n\*log(n))
* In place: no
* Stabile: si

Procedimento:

1. Divido l’array ricorsivamente a metà (in sotto array) fino ad arrivare agli elementi singoli (tipo albero)
2. Partendo dalle foglie controllo e ordino gli elementi in coppia e li metto in un array (di due posti)
3. Abbasso le foglie dal livello superiore (che non avevo quindi ancora confrontato)
4. Confronto gli array a coppie (L e R) partendo dal primo elemento di entrambi gli array
5. Inserisco in un nuovo array sottostante l’elemento minore tra i due confrontati e avanzo di indice nell’array in cui ho abbassato l’elemento
6. Ripeto ricorsivamente il punto 5 finche non ho inserito tutti gli elementi di L e R nel nuovo array
7. Continuo a confrontare tutti gli L e R che si generano fino a da arrivare ad un unico array, ordinato.

**Quick-Sort:**

* Complessità: Teta(n\*log(n)) caso pessimo O(N2)
* In place: si
* Stabile: si

Il processo chiave è quello di partition ossia ri-organizzare l’array in 2 sotto-array.

Seleziona il pivot (Ultimo elemento), confronta gli elementi con il pivot se sono maggiori non fare niente se sono minori allora scambiali con l’ultimo elemento maggiore trovato. Scambia il pivot con il primo elemento maggiore del pivot. Ripetere l’operazione di partition sia per gli elementi minori, sia per i maggiori fin quando non si riduce l’array al minimo (1elemento).

La funzione quick-sort ha due indici uno low e uno high si continua a fare le operazioni di partion e quick-sort ricorsivo sui due array (uno con valori maggiori e uno con valori minori del pivot) fin quando low<high, nelle chiamate ricorsive high viene decrementato.

**Radix-Sort:**

* Complessità: O(n+k)
* In place: si
* Stabile: richiede un algoritmo di ordinamento stabile di supporto

Consiste nell’ordinare chiavi numeriche basandosi sulle cifre meno significative andando verso quelle più significative

Ordina i valori in base alle cifre meno significative, ordina i valori in base alle seconde cifre meno significative, … , ordina i valori in base alle cifre più significative.

L’algoritmo stabile serve per mantenere l’ordine delle cifre meno significative.

**Strutture dinamiche (Alberi e liste)**

Le liste sono strutture dati ricorsive, ossia hanno al loro interno riferimenti a elementi dello stesso tipo. Esse possono essere linkate a una via, punta solo a successivo, o linkate doppiamente, punta sia al successivo che al precedente. Le operazioni sulle liste sono di complessità O(n)

*Alberi*

Un albero è una struttura dati ricorsiva nella quale sono presenti nodi connessi ad altri nodi. Le operazioni di inserimento, ricerca cancellazione hanno complessità lineare O(n).

Il nodo più alto è chiamato root ed è l’unico nodo che non dipende da altri nodi; i nodi che non hanno figli sono dette foglie. La profondità di un nodo è il cammina che va dal nodo radice al nodo stesso. L’altezza di un albero è la profondità del nodo più basso.

**Albero binario:**

è un albero nel quale nodo ha massimo due figli

Caso pessimo di profondità N-1, esempio albero degenere (elementi tutto su uno stesso ramo es. tutti a dx o sx.

**Algoritmi di attraversamento**

* **Pre-order**: processa il nodo; chiama pre-order sul nodo sinistro; chiama pre-order sul nodo destro
* **In-order:** chiama in-order sul nodo sinistro; processa il nodo; chiama in-order sul nodo destro
* **Post-order:** chiama post-order sul nodo sinistro; chiama post-order sul nodo destro; processa il nodo

*Con processa si intende leggi valore ad esempio.*

**Alberi di ricerca binaria (BST):**

sono alberi binari dove il figlio destro solitamente ha valore maggiore mentre il sinistro ha valore minore rispetto al padre, quindi il padre è sempre maggiore del figlio sinistro e minore del figlio destro.

Esempio: Se si va a sinistra e del nodo root e si trova un valore più grande del nodo radice allora non si ha un albero binario

La profondità media di un nodo è O(Log(N)), la profondità massima è O(N), altezza di un albero binario Log(N) con N numero di nodi.

Tutte le operazioni sono operazioni ricorsive (Ricerca, attraversamento, inserimento, cancellazione)

Inserimento: si esplora l’albero fin quando non si trova un nodo foglia senza figlio

Eliminazione: si presentano 3 casi poiché l’albero deve rispettare la condizione di albero binario:

1. Si elimina una foglia, nessuna operazione aggiuntiva si elimina direttamente
2. L’elemento ha un solo figlio, si sostituisce il puntatore del padre sul figlio dell’elemento da eliminare e si elimina l’elemento
3. L’elemento ha 2 figli, si sostituisce con il figlio più piccolo del ramo **destro** con l’elemento da eliminare, si aggiustano i puntatori e si elimina l’elemento poiché è diventato una foglia.

Costruzione da sequenza (se non bilanciato): data la sequenza la scorro in ordine inserendo gli elementi stando attento al fatto che nel figli sinistro di un padre avrò solo elementi minori (del padre) e a destra maggiori

**AVL, Alberi binari di ricerca bilanciati**

Sono degli alberi binari nei quali l’altezza è controllata e mantenuta in O(log(n)), l’altezza destra e sinistra non può essere più di uno per tutti i nodi.

Gli AVL vengono mantenuti tramite una proprietà dei nodi chiamata **Balance factor (BF)**

Dove H è l’altezza del nodo che è la lunghezza del cammino nodo alle foglie. Il BF deve essere { -1, 0, 1 } per far si che si abbia un AVL bilanciato

***[Domanda da esame Orale] Un AVL può avere 2h nodi con h altezza dell’albero, con n nodi si hanno log(n) livelli***

***(log in base 2)***

**Ribilanciamento di una AVL**

Per ribilanciare un AVL ci sono diversi casi (4):

1. Left rotation: si effettua se l’albero è sbilanciato verso destra destra (BF (nodo)= 2, BF (nodo\_dx) > 0)
2. Right rotation: si effettua se l’albero è sbilanciato verso sinistra sinistra (BF(nodo)=-2, BF(nodo\_sx)<0 )
3. Right Left rotation: si effettua se l’albero è sbilanciato a destra sinistra (BF (nodo)= 2, BF (nodo\_dx) < 0)
4. Left Right rotation: si effettua se l’albero è sbilanciato a sinistra destra (BF (nodo)= -2, BF (nodo\_sx) > 0)

Le operazioni si basano controllando il BF e sullo spostamento dei nodi, segue descrizione più dettagliata

**Right Right -> Left rotation**

Si prende il figlio sinistro del figlio destro del nodo con BF=2 e lo si pone come figlio destro nel nodo di BF=2. Successivamente si mette il padre del figlio che abbiamo spostato a padre del nodo che aveva BF=2 (quel nodo finisce nel ramo sinistro.

“Alzo il figlio destro (quindi l’ex padre diventa figlio sinistro) e il figlio sinistro del nodo che abbiamo alzato diventa figlio destro dell’ex padre”

**Left Left -> Right rotation**

Si prende il figlio destro del figlio sinistro del nodo con BF=-2 e lo si pone come figlio sinistro nel nodo di BF=-2. Successivamente si mette il padre del figlio che abbiamo spostato a padre del nodo che aveva BF=-2 (quel nodo finisce nel ramo destro.

“Alzo il figlio sinistro (quindi l’ex padre diventa figlio destro) e il figlio destro del nodo che abbiamo alzato diventa figlio sinistro dell’ex padre”

Gli altri due casi sono combinazioni dei primi due, bisogna valutare il BF. Tutte le operazioni sono O(n); ogni volta che si effettuano delle operazioni di inserimento e cancellazione, che seguono le stesse regole del BST, bisogna effettuare il ribilanciamento.

**BTREE**

Si usano per la ricerca di dati in database, la memorizzazione avviene in blocchi. La memorizzazione avviene tramite indici che identificano i blocchi; essi sono memorizzati in una struttura apposita. È possibile creare anche una struttura a livelli di indici.

Un BTREE è un albero di ricerca bilanciato (AVL) nei quali l’altezza è molto minore rispetto a un BST e ogni nodo può avere più figli. L’obbiettivo dei BTREE è quello di ridurre l’accesso al disco fisso poiché richiede molto tempo e, essendo caricati nella memoria principale, sono molto più veloci.

La complessità delle operazioni è in funzione dell’altezza (O(log n)) come per i BST, ma hanno il vantaggio che Hbtree<Hbst.

**Caratteristiche di un BTREE:**

* Tutte le foglie sono allo stesso livello
* Gli indici sono messi in ordine crescente
* Ogni nodo ha n+1 puntatori, se il nodo è una foglia il puntatore non ha significato
* Le chiavi dividono l’albero in sottoalberi
* Per contare le chiavi devo sommare il numero di chiavi dei nodi che conosco già e moltiplicare quanti nodi foglia che ho per il numero minimo (t-1) o massimo (2t-1) di possibili chiavi in essi

**Grado di un BTREE**

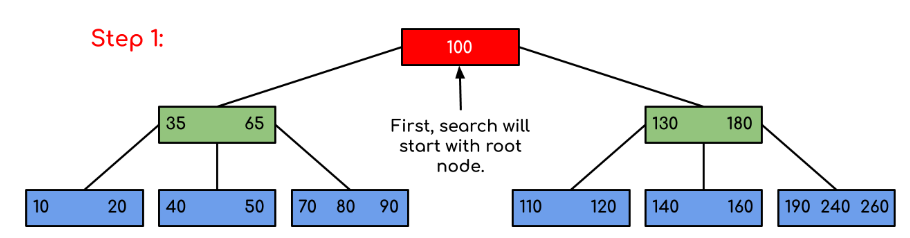
I nodi hanno un limite inferiore e un limite superiore di numeri di chiavi che possono contenere, t è l’ordine del BTREE (t è sempre maggiore o uguale di 2)

1. Ogni nodo, eccetto il nodo root, devono avere almeno t-1 chiavi, ed ha sempre almeno t figli
2. Tutti i nodi possono contenere massimo 2t-1 chiavi quindi 2t figli
3. Se root ha m elementi deve avere almeno m+1 figli, e viceversa
4. La root è l’unico nodo che può avere un solo elemento

**Teorema:** Se l’altezza h del BTREE, T di grado t soddisfa

*Inserimento di una chiave*, complessità

* Si attraversa l’albero fino alle foglie come i BST
* L’elemento se la foglia non è piena si mette in modo ordinato all’interno del nodo. Altrimenti le chiavi si inseriscono nei nodi foglia
* Non si può creare una foglia sotto un'altra foglia per inserire un elemento: spazi liberi si generano splittando le altre foglie (vedi punto successivo)
* Se il nodo foglia è pieno, viene diviso in due nodi e l’elemento mediano **sale di un livello** (consente la crescita dell’albero) con i due nuovi nodi connessi a sinistra e a destra dell’elemento mediano
* Se splitto la root, il mediano sale in un nuovo nodo ad ha due figli. Se splitto un nodo che non è la root, il mediano che sale finisce nel nodo superiore inserito ordinatamente.

Le chiavi hanno due figli: a sinistra quelli minori del valore, a destra quelli maggiori del valore

*Eliminazione di una chiave*

Si possono eliminare le chiavi da qualsiasi nodo, una volta eliminata bisogna riorganizzare l’albero per far mantenere la proprietà del BTREE.

Caso 1) La chiave da cancellare si trova nella foglia.

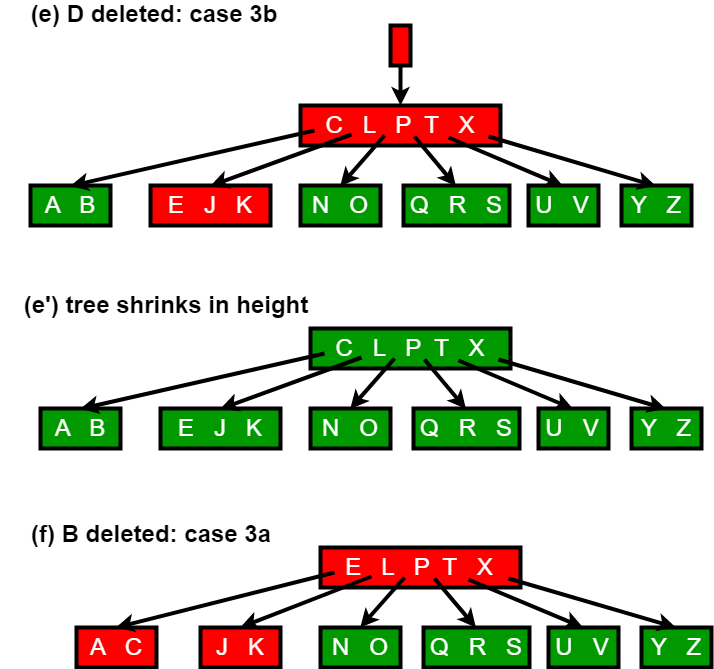
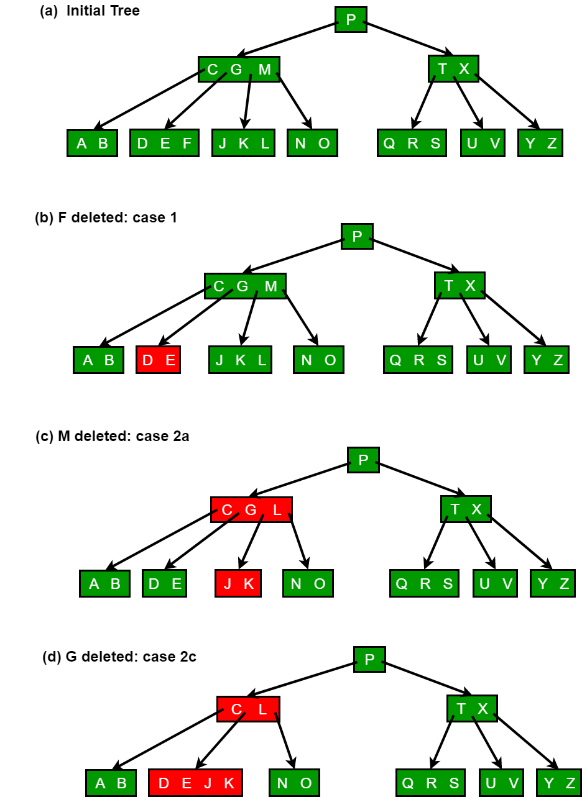
1. La cancellazione della chiave non viola la proprietà del numero minimo di chiavi che un nodo deve possedere.
2. La cancellazione della chiave viola la proprietà del numero minimo di chiavi che un nodo deve possedere. In questo caso, prendiamo in prestito una chiave dal nodo fratello immediatamente vicino nell'ordine da sinistra a destra.
   1. Per prima cosa, visitiamo il nodo fratello di sinistra
   2. Se il nodo fratello di sinistra ha più di un numero minimo di chiavi, allora prendiamo in prestito una chiave da questo nodo.
   3. Altrimenti, controlla di prendere in prestito dal nodo fratello immediato di destra
3. Se la cancellazione della chiave porta a una riduzione del numero di nodi che viola il fatto che un nodo padre con m chiavi debba avere almeno m+1 figli, allora si fa salire nel padre un valore dal nodo affianco a quello che stiamo cancellando, e si fa scendere una chiave dal nodo padre a quello in cui si trovava la chiave che abbiamo cancellato.
4. Se siamo nel caso 3, ma il nodo affianco ha solo 1 chiave allora faccio il merge di questi due nodi figli, faccio scendere in mezzo la chiave dal padre e posso poi quindi cancellare la chiave che volevo.

Caso 2) Se la chiave da cancellare si trova nel nodo interno, si verificano i seguenti casi.

1. Se le foglie (se non sono foglie, scorri tutto il sotto albero e prendi le foglie interne) di questa chiave hanno la “possibilità” di “perdere” un elemento, allora scambio il mio elemento da eliminare con uno dei due elementi più interni delle foglie che sto valutando (o quello maggiore del figlio sx o quello minore del figlio dx). Una volta scambiati posso cancellare l’elemento sceso.
2. Se nessuna delle due foglie può perdere elementi, allora faccio il merge di queste due foglie e cancello direttamente la chiave dal nodo in cui si trova.

Caso 3) In questo caso, l'altezza dell'albero si riduce. Se la chiave di destinazione si trova in un nodo interno, e la cancellazione della chiave porta ad un numero inferiore di chiavi nel nodo (cioè meno del minimo richiesto), allora cerca il predecessore e il successore in ordine. Se entrambi i figli contengono un numero minimo di chiavi, allora il prestito non può avvenire. Questo porta al caso II(3), cioè alla fusione dei figli.

Di nuovo, cerca il fratello per prendere in prestito una chiave. Ma, se anche il fratello ha solo un numero minimo di chiavi, allora, fondete il nodo con il fratello insieme al genitore. Disponi i figli di conseguenza (ordine crescente).



*Ricerca*

Si leggono le chiavi e si naviga nell’albero in base alla scansione. Lo si fa tramite una funzione ricorsiva.

Complessità

**Hash Map:**

Sono strutture dati per la creazione di dizionari, (key,value).

Il tempo di ricerca in una hash table è costante, come per le linked list O(1).

Le associazioni sono unidirezionali ovvero che dalla chiave è possibile ottenere l’elemento, ma **non** viceversa.

Quando due chiavi indicano allo stesso elemento si dice che si ha una **collisione**.

L’obbiettivo delle funzioni di hashing è quello di minimizzare il numero di collisioni.

In caso di collisione una tecnica utilizzata è quella del **chaining**: questo metodo consiste nell’avere nella lista delle chiavi delle liste bidirezionali come elementi così che se una chiave punta a più elementi è possibile risalire ad esso leggendo la lista dell’indice. Questo tipo di tabelle sono dette **open hash table.**

Il tempo di ricerca in questo tipo di liste però è O(n).

Il rapporto tra chiavi immagazzinate e dimensione della tabella è chiamato load factor.

***Funzioni di hash per interi***

Si ha t = dimensione hash table

*Metodo divisione*:

*Metodo moltiplicazione:* , A è una costante e

La moltiplicazione non è sensibile alla dimensione della tabella a differenza della divisione

*Metodo Mid-square*: h(k) = k2 => prendi il numero di bits centrali pari al numero di cifre del seme, il seme va scelto all’inizio.

*Hashing di stringhe:*

Si converte la stringa in intero dividendola in k caratteri per ogni blocco di k caratteri, sommano i risultati, si esegue l’hashing della somma.

con POS(c)= per esempio codice ascii del carattere

applica hashing intero su z

*Altri metodi di chainging*

In questi casi si dice che la hash table è **closed** perché andiamo a cercare un indice libero nella tabella

*Linear probing*

; p=1,2, 3….; h(k) è la funzione con la collisione

Itera fin quando non si trova un indice libero; p può diventare molto grande, si tende a rimanere intorno al cluster

*Quadratic probing*

c1 e c2 sono due valori costanti, ci si allontana in modo rapido dal cluster

*Pseudo-random probing*

il seme della funzione random deve essere sempre lo stesso

*Double hasing*

h1 e h2 sono funzioni di hash ausiliarie. Non si producono cluster

**Teoria dei Grafi**

Un grafo è uno strumento matematico che consente la rappresentazione di decisioni e problemi.

Esso si compone di N nodi e E lati(archi).

Un digrafo è un grafo nel quale gli archi hanno un orientamento, indicano le direzioni che si possono percorre da ogni nodo.

Un grafo può essere rappresentato tramite:

* **Matrice di adiacenza**: è una matrice NxN nella quale ogni cella contiene la presenza o l’assenza di un arco con l’eventuale peso, le righe e le colonne sono i nodi. Sulla diagonale, se non ci sono cicli sullo stesso nodo, tutti 0. Per i grafi non direzionati è simmetrica.

Essa è utile per grafi densi poiché veloce da elaborare, al prezzo di occupare parecchio spazio O(N2).

* **Liste di adiacenza:** sono delle liste che rappresentano l’adiacenza dei lati. Utile per grafi sparsi.

All’atto pratico è un array, di dimensione N che indica il numero di nodi, di liste, ogni lista contiene gli archi che si connettono al nodo; se vi sono dei pesi il peso è contenuto nell’elemento della lista. Lo svantaggio è nella ricerca degli archi poiché bisogna scorrere le liste O(V).

Terminologie:

* **Cammino**: è una sequenza consecutiva ordinata di archi e vertici
* **Ciclo**: è un cammino che parte da un nodo e finisce nello stesso
* **Grafo connesso**: è un grafo nel quale è possibile raggiungere tutti i nodi con un cammino
* **Digrafo** fortemente connesso: è un grafo nel quale esiste un cammino nel quale è possibile attraversare tutti i nodi
* **Albero:** è un grafo nel quale non ci sono cicli
* **Grafo completo**: ogni nodo ha almeno un arco
* **Ciclo Hamiltoniano**: cammino che tocca tutti i nodi una sola volta
* **Ciclo Euleriano:** cammino che attraversa tutti gli archi una sola volta
* **Taglio**: è un insieme di lati che se rimossi causano una disconnessione
* **Grafo bipartito**: è un grafo separabile in 2 gruppi N1 e N2 e c’è un arco che collega ogni nodo N1 con N2
* **Spanning tree**: grafo che tocca tutti i nodi

**Algoritmi**

**DFS (Depth First Search)**

È un algoritmo di esplorazione del grafo, esso visita i nodi connessi al nodo che si sta analizzando in modo ricorsivo.

* Complessità: O(N+E)

Da solo il DFS ha poca utilità, ma con delle modifiche può essere usato per verificare se il grafo ha determinate caratteristiche.

**Calcolo delle componenti connesse**

Calcola il numero di elementi connessi a un nodo, si sfrutta l’esplorazione del grafo contando i nodi connessi al grafo.

**BFS (Breadth First Search):**

è simile al DFS, ma visita i nodi più vicini alla sorgente. Mette i nodi connessi in una coda FIFO e la usa per gestire quale nodo va analizzato. Complessità ()

**Albero ricoprente con costo minimo (KRUSKAL)**

Sfrutta il BFS, sia G un grafo pesato, ogni arco ha un peso. L’obbiettivo è quello di trovare lo spanning tree di costo minimo. Per trovare lo spanning tree di costo minimo si usa l’**algoritmo di Kruskal:**

* Complessità: O (E\*log(N))

1. Ordina tutti i lati in modo crescente in base al peso
2. Prendi il lato più piccolo e controlla se forma un ciclo nello spanning tree, se c’è un ciclo escludi il lato e vai al successivo
3. Ripeti pt. 2 fin quando ci sono N-1 lati nell’albero spanning

Il costo minimo si ottiene sommando il peso degli archi dello spanning tree

**Algoritmo di Prim:**

è un’alternativa all’algoritmo di Kruskal, l’idea è quella di avere due liste di nodi X e Y: X sono i nodi già raggiunti e Y i nodi da raggiungere.

La lista X parte con il primo nodo A mentre la Y ha tutti gli altri nodi BCD…

Devo cercare tra tutti i possibili archi che connettono un nodo di X a un nodo di Y quello che ha costo minore.

Trovato l’arco lo scrivo (e/o disegno) e sposto il nodo destinatario dell’arco da Y a X.

Ripeto l’operazione iterativamente fino a che tutti i nodi saranno passati da Y a X.

Tabella-> IT, X, Y, ARCO, PesoArco. Vedi esercizio allegato.  
  
**Algoritmo di Dijkstra**

Consente di trovare il cammino di più breve tra due nodi, è un esempio di programmazione dinamica, la complessità è O(N\*log(N)) con implementazione tramite code altrimenti O(N2)

* DIST = s B C D … t (indica il costo che abbiamo fin ora per arrivare a quel nodo)
* PRED = s B C D … t (indica il nodo da cui siamo arrivati)
* Tabella con Iterazione, DIST, PRED e Q
* DIST parte da “0 - - - - … -“, PRED parte da “- - - - … -“
* Q = s B C D … t e nella prima riga sottolineiamo s. Ad ogni iterazione sottolineiamo il primo elemento e quando passiamo all’iterazione successiva lo rimuoviamo.  
  L’elemento sottolineato indica l’elemento che devo controllare all’iterazione SUCCESSIVA. Se da quell’elemento arrivo a un nodo con un peso minore di quanto già scritto in DIST allora aggiorno sia DIST che PRED rispettivamente con il nuovo peso e con il nuovo nodo da cui arrivo

Vedi esercizio allegato.

**Flusso di reti:**

Tramite i grafi è possibile rappresentare delle reti con del flusso; infatti, ogni il peso di ogni arco rappresenta la capacità massima di esso (esempio di applicazione: reti idriche).

Sull’arco troviamo una notazione del tipo (a/b) dove **a** è la capacità, **b** è il flusso passante,  
Il flusso che scorre da **s** a **t** si trova o leggendo il flusso **uscente da s** o quello **entrante in t** (si equivalgono)

I nodi sorgente sono indicati con s mentre quelli di destinazione t

* Il flusso in un arco non può superare la capacità massima dell’arco stesso, se l’arco è pieno si dice che è saturo.
* Il flusso in entrata è uguale al flusso in uscita per tutti i nodi eccetto per s e t

Ogni nodo è associato un valore reale b che indica il comportamento del flusso nel nodo se b è:

* Positivo: nodo domanda, assorbe flusso
* Negativo: nodo offerta, genera flusso
* Nullo: nodo di transito, mantiene il flusso invariato

**Calcolo del flusso massimo**

* Immagine che contiene testo

  Descrizione generata automaticamente-f: partenza
* 0: transito
* f: destinazione

Le due sommatorie indicano tutto ciò che entra – tutto ciò che esce

**Ford-Fulkerson:**

Consente il calcolo del flusso massimo basandosi sulla **rete residuale**/complementare. Ogni arco ha una capacità residua che è pari a

Per trovare il cammino aumentante si può usare DFS o BFS nel grafo residuo

1. inizia con un flusso pari a 0
2. Fin quando c’è un ***cammino aumentante*** dalla sorgente alla destinazione(sink)
   1. Aggiungi il cammino al flusso
3. Ritorna il flusso

Complessità *O(max\_flow\*E)*, E numero di archi

***Cammino aumentante:*** è un cammino da s e t sulla **rete residuale**, se esiste si può incrementare il flusso pari alla capacità minima degli archi appartenenti al percorso

**Calcolare la rete residuale**

1. Prendere un cammino che vada da s e t
2. Prendere la capacità minore del cammino, sarà il flusso da cui partire, i nodi devono rimanere i equilibrio flow\_in=flow\_out.
3. Computa il grafo con il flusso scelto riducendo la capacità degli archi del cammino (la capacità utilizzata la segno con un arco opposto all’arco originale che avrà capacita pari alla capacità residua), *è possibile togliere flusso a un arco di ritorno, si annulla del flusso raggiungendo dello spazio all’arco originale.*
4. Cerca un altro cammino s -t e riesegui il punto 2 e 3 mantenendo le capacità degli archi aggiornate
5. Termina quando non è più possibile trovare un percorso s-t, gli archi sono saturi

**Taglio di una rete di flusso**

Si partiziona la rete in due sotto reti e nelle quali s si trova inmentre t si trova in .  
La **capacità** di un taglio è la somma degli archi che vanno da U a . Essa è uguale al flusso massimo della rete.  
Il flusso massimo è uguale alla capacità del taglio minimo (sulla rete di partenza).

1. Trova il grafo residuo con Ford-Fulkerson
2. Rimuovi gli archi del grafo residuo saturi nei quali la somma delle loro capacità è minima, quindi uguale al flusso massimo, essi devo disconnettere t dal grafo in cui c’è s.

Per trovare il taglio minimo, posso aiutarmi cercando di far passare il taglio attraverso il maggior numero possibile di archi saturati **SOLO SE USO UNA RETE RESIDUALE CON FLUSSO OTTIMO** (ovvero se ho già trovato tutti i possibili cammini aumentanti).

**Programmazione dinamica**

La programmazione dinamica consente la risoluzione di un problema in modo iterativo tramite la risoluzione di sotto problemi più piccoli. La maggior parte degli algoritmi di programmazione dinamica fa uso della ***memoization*** ossia che un sotto problema una volta risolto viene salvato il risultato in modo da riutilizzarlo se necessario nel proseguo delle iterazioni

Essa torna utile quando bisogna risolvere problemi combinatori.

**Step comuni per affrontare un problema con la programmazione dinamica**:

1. Caratterizzare la struttura della *soluzione ottima*
2. Trovare una soluzione ricorsiva dalla soluzione ottima
3. Calcolare il valora di una soluzione ottima, solitamente si usa bottom-up poiché più efficiente
4. Costruire una soluzione ottimale dalle informazioni precedenti

La soluzione ottima è data dalla somma di soluzioni di problemi più piccoli.

Solitamente i problemi sono risolti con complessità polinomiale.

Nella programmazione dinamica è possibile schematizzare la risoluzione dei sotto problemi tramite il grafo dei sotto problemi, che è in grado rappresentare i dati che necessita un sotto problema oltre che la dimensione dei sotto problemi da risolvere. È una riduzione dell’albero delle ricorsioni, e la dimensione del grafo aiuta a determinare la complessità di esecuzione dato che ogni nodo viene eseguito una sola volta.

* **Sottostruttura ottimale**: la soluzione ottima di un problema contiene la soluzione ottima di sotto problemi (Es. percorso minimo di un grafo, nota che il percorso massimo non ha una sotto struttura ottima)
* **Proprietà dei sotto problemi sovrapposti:** in alcuni problemi possono esserci sotto problemi che si ripresentano più volte, in questo caso è buona pratica usare la tabulazione o la memoization.

(es. Fibonacci, Ricerca binaria)

**Top down**L’approccio top-down usa metodi ricorsivi ma utilizzando valori salvati.

**Bottom up**L’approccio bottom-up ordina i problemi in base alla grandezza e sfrutta le soluzioni ai problemi più piccoli per risolvere quelli più grandi. (ciclando, iterando)

**Esempi di problemi**

***Cut rod***

Data una asta di n e un array di contenete i prezzi per ogni pezzo (segmento dell’asta) determinare il valore massimo tagliando l’asta e vendendo i pezzi.

Soluzione ottima ricorsiva

L’approccio con i metodi ricorsivi è inefficiente.

L’approccio bottom-up, quindi con cicli è più efficiente poiché costruisce la soluzione in modo incrementale. O(n2)

* Si mantiene un array r inizializzato a 0 nel quale verranno salvati i risultati massimi.
* Si inizia a percorre l’asta con lunghezza=1
* Per ogni lunghezza calcola prezzi[i]+r[lunghezza-i]
  + Se la somma è maggiore dell’elemento in r[lunghezza]: aggiorna il valore in r[lunghezza]
* Esegui fino al raggiungimento della lunghezza massima
* Ritorna r[lunghezza]

***Problema Sottosequenza comune più lunga (Genetica)***

Sono date due sequenze X e Y di lunghezza m e n

Si presentano 3 condizioni che descrivono la sottostruttura ottima, funzione ricorsiva lcs(x [], y[]):

1. Se x[m-1] == y[n-1] allora: ritorna 1+ lcs(x[0…m-2], y[0…n-2])
2. Se x[m-1]! = y[n-1] allora: ritorna MAX (lcs(x[0…m-1], y[0…n-2]), lcs(x[0…m-2], y[0…n-1]))
3. Se m=0 o n=0 ritorna 0

Gli elementi se disposti in una tabella dove le righe sono xi e le colonne sono yi , la prima riga e la prima colonna sono tutti 0; quando si ha una corrispondenza di un carattere si somma all’elemento superiore in diagonale 1, gli altri elementi si prende il massimo tra elemento superiore e quello sinistro. Si prosegue per tutte le righe il valore finale ultima cella dell’ultima riga sarà la lunghezza della sequenza più lunga.

*Complessità O(m\*n) versione bottom-up O(2n) versione ricorsiva (top down)*

***Problema Knapsack***

è dato uno zaino che può avere un peso massimo, si hanno N oggetti che a loro volta hanno un peso e un costo. L’obbiettivo è quello di trovare il valore massimo senza eccedere in peso.

Si può considerare il fatto se l’oggetto è inserito nello zaino oppure no tramite una variabile binaria xi:

* 1 presente nello zaino
* 0 altrimenti

La funzione obbiettivo è data da

Inoltre, bisogna rispettare il vincolo ; Pmax è il peso massimo dello zaino e wi è il peso di ogni oggetto.

La versione ricorsiva, senza memorization, ha complessità esponenziale O(2n) e ci sono sotto problemi ridondanti; Altrimenti O(N\*W), ma al costo di avere O(N\*W) anche in occupazione di spazio. La versione bottom-up, invece, ha complessità O (N\*W) si ha una complessità pseudo-polinomiale poiché W dipende dall’elemento in input.

Gli elementi se disposti in una tabella dove le righe sono gli elementi e le colonne rappresentano il peso, la prima riga e la prima colonna sono tutti 0;

quando si arriva al peso corrispondente dell’elemento si deve verificare se il valore dell’elemento della cella della riga superiore è maggiore del *valore\_elemento+rigaSup[w\_max-j],* con j indice colonna e *w\_max* peso massimo, ripeto il valore altrimenti metto la somma; altrimenti copia i valori della riga sopra.

Il risultato finale si trova nell’ultima cella dell’ultima riga.

**Algoritmi di compressione**

* **Messaggio**: dati da comprimere
* **Encoding e Decoding**: operazioni trasformare il messaggio in un messaggio più piccolo e viceversa
* **Compressione**: operazioni che riducono la dimensione del messaggio
* **Lossless**: quando si ricostruisce il messaggio in modo identico all’originale
* **Lossy**: quando si ricostruisce il messaggio si ha una tolleranza di perdita di informazioni (es: streaming video)

Non esiste un algoritmo che possa comprimere qualsiasi messaggio in ingresso

**Algoritmi di compressione**

**RLE (Run Length Encoding)**

Consente di comprimere messaggi con caratteri che non siano cifre.

Presa una stringa in input prende ogni simbolo e trascrive la frequenza di esso adiacente

Esempio: AAABBBBAAABBCCCCCCCAB => 3A4B3A2B7CAB

Solitamente si utilizzano dei caratteri di escape a bassa frequenza per codificare i contatori poiché occupano lo stesso spazio di un simbolo, per codificare l’escape si potrebbe inserire due volte il simbolo di escape.

Esempio: AAABBBBAAABBCCCCCCCAB => AAA&DBAAABB&GCAB; D and G encode the length (4 and 7)

***RLE di sequenze binarie***

Da una sequenza binaria si trascrivono le frequenze dei bit adiacenti in modo alternato

01111001010001100110011 => 1 4 2 1 1 1 3 2 2 2 2

Ha usi limitati come algoritmo. Si può impostare una lunghezza fissa di bits con cui codificare la frequenza, se non ci sta il numero bisogna separarlo mettendo il massimo, seguito da 0 bit del tipo opposto e successivamente la parte rimanente. La cosa fondamentale è sempre alternare 0 e 1

Esempio: (bit codifica 2) 0111100101 => 0,2,0,2,2,1,0,1

**VLE (Variable Length Encoding)**

Per codificare I simboli si usano:

* Più bit per i simboli meno frequenti
* Meno bit per i simboli più frequenti

L’algoritmo in fase di lettura della sequenza legge bit fin quando non trova una corrispondenza. I simboli devono essere codificati prima e devono avere una codifica univoca, ossia non devono essere prefissi di una sequenza finita(sequenza che dopo il valore del prefisso non ha altri bits).

È possibile posizionare i simboli in un albero binario come foglie di esso.

**Codifica di Huffman**Sulla base della VLE l’algoritmo crea le codifiche dei simboli sfruttando un albero binario basato sulle frequenze di ogni simbolo.Prima dell’algoritmo bisogna analizzare le frequenze nel messaggio O(n)Algoritmo di codifica:

1. Ordina le frequenze in ordine decrescente (# comparse / # caratteri totali)
2. Crea l’albero binario
   1. I simboli meno frequenti vengono aggregati e sommate le frequenze fino ad avere frequenza 1 (tutti i caratteri nella root)
   2. Leggendo da destro prendo il primo carattere (o gruppo) e lo separo dal resto della stringa:  
      Se nell’elenco originale (tabella) si trova sotto al resto della stringa, lo metto nel figlio di sinistra
   3. Proseguo fin a separare tutti i simboli nell’albero, le foglie dell’albero saranno i singoli caratteri
3. Per codificare, se vado a destra allora avrò 1 se vado a sinistra avrò 0
4. Tabella finale-> carattere **(C)**, frequenza **(P)**, # bit del Codice **(L**), **P \* L** (con in fondo alla colonna la somma di questi valori, che indica il **cammino ponderato**), **Codice**

Definizioni:

* **weighted path length**: di una foglia è la frequenza per la profondità (P\*L)
* **external path weight**: somma dei weighted path lengths (**cammino** **ponderato**)

Codifica canonica di Huffman:

l’idea è quella di inviare la lunghezza dei simboli codificati, devono seguire una codifca standard, segue l’algoritmo

Algoritmo di codifica canonica di Huffman

1. Ordina le sequenze codificate in ordine crescente, con algoritmo stabile
2. Assegna 0 al primo simbolo
3. Per i simboli con lunghezza uguale aggiungi 1
4. Se cambia la lunghezza somma 1 e shifta a sinistra

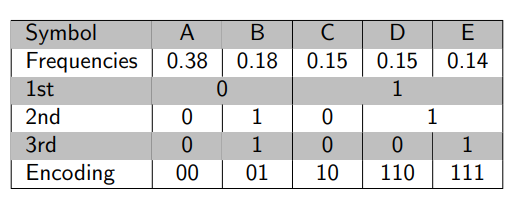
**Shannon Fano**

È un algoritmo che produce un codice prefisso usando le frequenze dei simboli.

I codici di Shannon-Fano sono subottimali, non sempre raggiungono la più bassa possibile lunghezza attesa del percorso esterno, ma hanno una lunghezza della parola chiave entro 1 bit dall'ottimale.

Algoritmo:

1. Calcola le frequenze
2. Ordina in ordina le frequenze in base ai simboli
3. Dividi la lista in 2 la somma delle frequenze della parte di sinistra deve essere il più vicina possibile a quella di destra
4. Assegna 0 alla parte sinistra e 1 a quella di destra
5. Applica ricorsivamente 3 e 4 per ogni metà fin quando non si hanno liste di lunghezza 1



**Codifiche basate sui dizionari**

Un dizionario è un riferimento per le sequenze ripetute di simboli, invece della sequenza viene inviato il riferimento del dizionario.

Dizionari:

* **Impliciti:** il dizionario è il messaggio stesso
* **Espliciti:** il dizionario è costruito durante la codifica/decodifica, i dizionari devono essere identici per sorgente e destinazione

**Algoritmi basati su dizionari impliciti**

**LZ77**

Una parte del messaggio è chiamata **sliding window**, e contiene la parte già codificata mentre il **lookahead buffer** è la porzione di messaggio da codificare.

La codifica segue il formato ***(B,L) C***

* **B**: numero di step da fare all’indietro nella sliding window per trovare l’inizio della sequenza da ripetere
* **L**: lunghezza della sequenza da ripetere
* **C**: prossimo carattere nel lookahead buffer

Leggi il messaggio e applica il formato (B,L) C per ogni sequenza, attenzione a eventuali sequenze già registrate.

La decodifica tiene traccia della sliding window (leggi la sliding window).

LZ77 è lento in fase di compressione.

**Algoritmi basati su dizionari espliciti**

**LZ78**

Il dizionario viene costruito durante la codifica e la decodifica, ogni simbolo è immagazzinato con un indice per ogni simbolo

Algoritmo:

1. Trova il match più lungo della stringa S nel dizionario
2. Invia l’indice di S e il simbolo successivo del messaggio c
3. Aggiungi Sc al dizionario se non esiste

Leggi carattere per carattere e controlla se la sequenza è già stata trovata, altrimenti invia la nuova sequenza.

Il dizionario non viene condiviso, ma viene aggiornato con l’invio dei dati ed eventualmente con nuove sequenze (da 0 fino a 255 caratteri sono noti).

Formato di invio **(indice, c)**, se indice=0 -> nuovo carattere

La decodifica segue la sequenza del dizionario, in ordine di arrivo e ricompone il messaggio, e aggiunge le sequenze al dizionario del destinatario.

**LZW**

È un miglioramento di LZ78 ed evita l’invio di caratteri addizionali c. Richiede un dizionario standard sia per la sorgente che per il destinatario, esso come LZ78 non deve essere inviato poiché noto ad entrambi le parti (da 0 fino a 255 caratteri sono noti).

**Encoding**:

\* PSEUDOCODE

1 Initialize table with single character strings

2 P = first input character

3 WHILE not end of input stream

4 C = next input character

5 IF P + C is in the string table

6 P = P + C

7 ELSE

8 output the code for P

9 add P + C to the string table

10 P = C

11 END WHILE

12 output code for P

**Decoding**:

\* PSEUDOCODE

1 Initialize table with single character strings

2 OLD = first input code

3 output translation of OLD

4 WHILE not end of input stream

5 NEW = next input code

6 IF NEW is not in the string table

7 S = translation of OLD

8 S = S + C

9 ELSE

10 S = translation of NEW

11 output S

12 C = first character of S

13 OLD + C to the string table

14 OLD = NEW

15 END WHILE

**Aggiungi dizionario**:

se esiste la corrispondenza nel dizionario:

corrispondenza precedente+ primo simbolo della nuova corrispondenza attuale

altrimenti:

corrispondenza precedente+ primo simbolo della nuova corrispondenza precedente