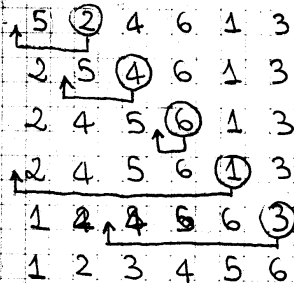


INSERTION SORT

- Efficiente nel caso si debba ordinare un piccolo numero di elementi.
- Ordinamento "in loco"

ESEMPIO:



EFFICIENZA:

- Caso migliore: array già ordinato $\rightarrow O(n)$
- Caso peggiore: array ordinato inversamente $\rightarrow O(n^2)$

MERGE SORT

- Segue in modo diretto il paradigma divide-et-impera.
- Struttura ricorsiva.
- Passi dell'algoritmo:
 - Divide la sequenza di n elementi in 2 sottosequenze di $n/2$ elementi.
 - Ordina ricorsivamente le due sottosequenze.
 - Fonde le due sottosequenze:
 - 1 + indice i sul primo elem. della prima sottoseq.
 - 2 + indice j sul primo elem. della seconda sottoseq.
 - 3 + si prende il valore minore, si colloca nel nuovo array, e si incrementa il rispettivo indice. Si ripete l'operazione.
 - 4 + quando uno dei due indici va in overflow, si copia tutta la restante sottoseq. nell'array ordinato.

- Caso base di MERGE-SORT: sequenza di un solo elemento.
- Eq. di ricorrenza:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n=1 \\ 2 T(n/2) + \Theta(n) & \text{se } n > 1 \end{cases}$$

EFFICIENZA

- Caso peggiore: $\Theta(n \log_2 n)$

TEOREMA PER LA RISOLUZIONE DELLE EQUAZIONI DI RICORRENZA

Sia $p(m^k)$ un polinomio di grado k

$$\text{se } T(m) = \begin{cases} C_0 & \text{se } m = 0 \\ a T(m/b) + p(m^k) & \text{se } m > 0 \end{cases}$$

per $a, b \geq 1$ e $p(m^k)$ un polinomio di grado k

Allora:

Caso 1: $T(m) = \Theta(m^{\log_b a})$ se $a > b^k$

Caso 2: $T(m) = \Theta(m^k \log_2 m)$ se $a = b^k$

Caso 3: $T(m) = \Theta(m^k)$ se $a < b^k$

HEAP SORT

- Ordinamento in loco
- Sfrutta le proprietà degli HEAP
- Algoritmo ricorsivo

HEAP:

- Albero binario quasi completo: l'albero è riempito completamente su tutti i livelli, tranne eventualmente, il più basso che è riempito da sinistra in poi
- Proprietà dell'ordin. parziale dello HEAP: \forall indice $i \neq 1, A[\text{PARENT}(i)] \geq A[i]$

Un HEAP si può rappresentare mediante un array, con due attributi:

LENGTH[A] e HEAP-SIZE[A], dove $\text{HEAP-SIZE}[A] \leq \text{LENGTH}[A]$

FIGLIO SX: $2 \cdot i$ FIGLIO DX: $2 \cdot i + 1$ PADRE: $\lfloor i/2 \rfloor$ (parte intera)

Si definisce ALTEZZA di un nodo in un albero il numero di archi sul più lungo cammino semplice discendente che va dal nodo a una foglia, e si definisce altezza dell'albero l'altezza della sua radice.

La radice dell'HEAP è l'elemento massimo.

- Passi dell'algoritmo:

- 1- Dato un array A, costruisce uno HEAP (BUILD-HEAP)
- 2- Sambia il primo elemento (la radice = il massimo) con l'ultimo.
- 3- Decrementa di 1 la grandezza dell'HEAP
- 4- Ripristina la proprietà dell'HEAP (HEAPIFY)

5 - Ripete dal passo 2 fino a che non si ottiene uno HEAP di dimensione 2

HEAPIFY: complessità $O(\log n)$

- la procedura viene chiamata su un nodo, ed assume che gli alberi binari con radici in $LEFT(i)$ e $RIGHT(i)$ siano HEAP, ma che il nodo su cui è stato chiamato possa essere più piccolo dei suoi figli.
- Fa assumere il nodo in questione nello HEAP, in modo da riottenere di nuovo uno HEAP.
- la procedura è ricorsiva.

BUILD-HEAP: complessità $O(n)$

- Si appoggia ad HEAPIFY
- Chiama HEAPIFY a partire dal primo nodo non foglia fino alla radice.
- Nota Bene: da $\lfloor n/2 \rfloor + 1$ fino ad n , sono presenti solo foglie. Una foglia possa singola di per se rispetta la proprietà di HEAP.

EFFICIENZA:

- Caso peggiore di HEAPIFY: ad ogni chiamata la dimensione dell'input si riduce da n a $2/3 n$.
- Complessità di HEAP-SORT: $O(n \log_2 n)$

ALTRI METODI:

- EXTRACT-MAX: complessità $O(\log_2 n)$
- INSERT: complessità $O(\log_2 n)$

CODE CON PRIORITA':

Una coda con priorità è una struttura di dati per mantenere un insieme S di elementi, ognuno con un valore associato ~~chiamato~~ ~~chiamato~~ chiamato chiave.

METODI:

- INSERT (S, x)
- MAXIMUM (S)
- EXTRACT-MAX (S)

QUICK SORT

- Strategia Divide - et - Impera
- Ordinamento in loco
- Tempo di esecuzione nel caso peggiore $\Theta(n^2)$
- Tempo di esecuzione nel caso medio $\Theta(n \log_2 n)$, e i fattori costanti nascosti dalla notazione Θ sono abbastanza piccoli

Divide: $A[p \dots r]$ è ripartito (e riciccolato) in due sottoarray non vuoti in modo che ogni elemento del primo sottoarray sia minore o uguale ad ogni elemento del secondo.

Impera: i due sottoarray sono ordinati ricorsivamente.

Combina: nessuna operazione da effettuare: $A[p \dots r]$ è ordinato.

- + **Paso base:** il sottoarray ha un solo elemento, è dunque già ordinato
- + **Paso ricorsivo:** calcola il pivot, riciclando l'array (PARTITION) e applica QUICKSORT ai due sottoarray.

PARTITION (A, p, r)

- 1 + il primo elemento del sottoarray, $A[p]$, viene scelto come pivot $\rightarrow x$
- 2 + le due regioni di sinistra e destra vengono ricostituite per stack.
Si usano due indici $i \leftarrow p-1$, $j \leftarrow r+1$
- 3 + j viene spostato verso sinistra finché non trova un elem. minore del pivot. i viene spostato verso destra finché non trova un elem. maggiore del pivot.
- 4 + Se i e j non si sono sovrapposti, scambio i due elementi.
- 5 + Ripeto dal passo 3 fino a che i due indici non si sovrappongono.
- 6 + Restituire l'indice j , delimitatore dei due sottoarray.

EFFICIENZA

- Costo di PARTITION: $O(n)$
- Costo di QUICKSORT:
 - + Caso peggiore: 1. l'array è ordinato in senso decrescente
2. l'array è già ordinato
- Complessità: $\Theta(n^2)$
 - + Caso medio: - Complessità: $\Theta(n \log_2 n)$
 - + Caso migliore: 1. l'array è diviso a metà (con uno scarto di ± 1).
- Complessità: $\Theta(n \log_2 n)$

ESEMPIO DI ESECUZIONE DI PARTITION

$$\begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ < 5, & 3, & 2, & 6, & 4, & 1, & 3, & 7 > \end{matrix}$$
1. Calcolo del pivot $\rightarrow x = 5$

$$2. i = 0, j = 9$$

$$\begin{array}{cccccccc} 5 & 3 & 2 & 6 & 4 & 1 & 3 & 7 \\ \uparrow & & & & & & & \uparrow \\ i & & & & & & & j \end{array}$$
3. $j-1, A[j]$ è min o uguale a $x \rightarrow 7 \leq 5$? FALSO, RIPETI 3 $j-1, j=7; A[j]=3; 3 \leq 5$? SI, STOP4. $i+1, i=1; A[i]=5; 5 \geq 5$? SI, STOP5. i è minore di j ? SI, PROSEGUI6. Scambia $A[i]$ con $A[j] \rightarrow$

$$\begin{array}{cccccccc} 3 & 3 & 2 & 6 & 4 & 1 & 5 & 7 \\ \uparrow & & & & & & \uparrow \\ i & & & & & & j \end{array}$$
7. $j-1, j=6; A[j]=1; 1 \leq 5$? SI, STOP8. $i+1, i=2; A[i]=3; 3 \geq 5$? NO, RIPETI $i+1, i=3; A[i]=2; 2 \geq 5$? NO, RIPETI $i+1, i=4; A[i]=6; 6 \geq 5$? SI, STOP9. $i < j$? $i=4, j=6, 4 < 6$? SI, PROSEGUI10. Scambia \rightarrow

$$\begin{array}{cccccccc} 3 & 3 & 2 & 1 & 4 & 6 & 5 & 7 \\ \uparrow & & & \uparrow \\ i & & & j \end{array}$$
11. $j-1, j=5; A[j]=4; 4 \leq 5$? SI, STOP12. $i+1, i=5; A[i]=4; 4 \geq 5$? NO, PROSEGUI $i+1, i=6; A[i]=6; 6 \geq 5$? SI, STOP13. $i < j$? $6 < 5$? NO, ESCI E RESTITUISCI j

$$\begin{array}{ccccccc} 3 & 3 & 2 & 1 & 4 & 6 & 5 & 7 \\ \uparrow & & & \uparrow \\ j & & & i \end{array}$$

Prossime chiamate del quicksort

- QUICKSORT(A, 1, 5)

- QUICKSORT(A, 6, 8)

COUNTING SORT

- Ipotesi di base: la lunghezza dell'array è dello stesso ordine di grandezza del range dei valori da ordinare K .
- Ordinamento non in loco

L'idea di base è di determinare, per ogni elemento x di input, il numero di elementi minori di x . Questa informazione può essere utilizzata per porre l'elemento x direttamente nella sua posizione nell'array di output.

- Array di input A , Array di output B , Array di appoggio C
- Passi dell'algoritmo: COUNTING-SORT (A, B, K)
 - 1+ inizializza il vettore C contenente le occorrenze di ciascun elemento, con tutti zeri.
 - 2+ inizializza le posizioni di C con il numero di elementi uguali all'indice di C .
 - 3+ somma progressivamente il numero di elementi minori dell'indice i di C , per ogni indice.
 - 4+ per ogni $A[j]$, metti $A[j]$ nel vettore di output B , alla posizione data da $C[A[j]]$, a partire da $j = \text{length}[A]$.
 - 5+ decrementa di 1 il valore di $C[A[j]]$.
 - 6+ ripeti dal punto 4.

- Proprietà del COUNTING-SORT: stabilità \rightarrow elementi con lo stesso valore compaiono nell'array di output nello stesso ordine in cui compaiono in quello di input.

ESEMPIO

$A = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 3 & 6 & 4 & 1 & 3 & 4 & 1 & 4 \end{matrix}$

Passo 1: $C = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}$

Passo 2: $C = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 0 & 2 & 3 & 0 & 1 \end{matrix}$

Passo 3: $C = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 2 & 4 & 7 & 7 & 8 \end{matrix}$

Passi 4-6: $A[j] = 4$, $B = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ & & & & & & 4 & \end{matrix}$, $C = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 2 & 4 & 6 & 7 & 8 \end{matrix}$

$j-1 = 7 \rightarrow A[j] = 1$, $B = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ & & & & & & 4 & 1 \end{matrix}$, $C = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 4 & 6 & 7 & 8 \end{matrix}$

$j-1 = 6 \rightarrow A[j] = 4$, $B = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ & & & & & & 4 & 4 \end{matrix}$, $C = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 4 & 5 & 7 & 8 \end{matrix}$

$j-1 = 5 \rightarrow A[j] = 3$, $B = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ & & & & & & 4 & 4 & 3 \end{matrix}$, $C = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 5 & 7 & 8 \end{matrix}$

$j-1 = 4 \rightarrow A[j] = 1$, $B = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ & & & & & & 4 & 4 & 3 & 1 \end{matrix}$, $C = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 2 & 3 & 5 & 7 & 8 \end{matrix}$

$j-1 = 3 \rightarrow A[j] = 4$, $B = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ & & & & & & 4 & 4 & 3 & 4 \end{matrix}$, $C = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 2 & 3 & 4 & 7 & 8 \end{matrix}$

$j-1 = 2 \rightarrow A[j] = 6$, $B = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ & & & & & & 4 & 4 & 3 & 4 & 6 \end{matrix}$, $C = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 2 & 3 & 4 & 7 & 7 \end{matrix}$

$j-1 = 1 \rightarrow A[j] = 3$, $B = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ & & & & & & 4 & 4 & 3 & 4 & 6 & 3 \end{matrix}$, $C = \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 0 & 2 & 2 & 4 & 7 & 7 \end{matrix}$

PILE

- l'elemento cancellato dall'insieme è quello inserito più di recente. Politica di gestione LIFO (Last In - First Out).
- per una implementazione, è possibile usare un array con un attributo, $TOP[S]$, che è l'indice dell'elemento inserito più di recente.

OPERAZIONI:

- $STACK-EMPTY(S)$: verifica se lo stack è vuoto (controlla quindi $TOP[S]$, se è zero) e restituisce TRUE se è vuoto, FALSE altrimenti.
- $PUSH(S, x)$: inserire un elemento: 1 + incrementa $TOP[S]$ di uno
2 + inserisce l'oggetto all'indice $TOP[S]$
- $POP(S)$: rimuovere un elemento: 1 + se lo stack è vuoto \rightarrow errore underflow
2 + decrementa di uno $TOP[S]$
3 + restituisce l'elemento in posizione $TOP[S] + 1$

EFFICIENZA:

- Ogni operazione richiede tempo $O(1)$

CODE

- l'elemento cancellato è quello che permanece nell'insieme da più tempo. Politica di gestione FIFO (First In - First Out)
- è possibile usare un array con due attributi, $HEAD[S]$ che punta alla testa della coda, e $TAIL[S]$ che è l'indice della prossima locazione in cui l'ultimo arrivato sarà inserito nella coda.
- Uso della struttura in modo circolare. Se $HEAD[S] = TAIL[S]$, la coda è vuota. Se $HEAD[S] = TAIL[S] + 1$, la coda è piena.

OPERAZIONI:

- $ENQUEUE(S, x)$: inserire un elemento: 1 + inserisce l'elemento x alla posizione indicata da $TAIL[S]$
2 + se $TAIL[S] = LENGTH[S]$, $TAIL[S] = 1$, altrimenti si incrementa $TAIL[S]$ di uno.
- $DEQUEUE(S)$: rimuovere l'elemento in cima: 1 + rimuove l'elemento puntato da $HEAD[S]$
2 + se $HEAD[S] = LENGTH[S]$, $HEAD[S] = 1$, altrimenti si incrementa $HEAD[S]$ di uno.

EFFICIENZA:

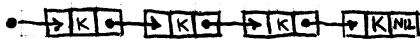
- Ogni operazione richiede tempo $O(1)$.

LISTE CONCATENATE

- Insieme di elementi di dimensione massima non prefissata.
- Elementi collocati secondo un ordine lineare.
- Possono essere implementate tramite array o tramite puntatori.

LISTE CONCATENATE SEMPLICI

- Un elemento x della lista è un oggetto con almeno due campi
 - + $KEY[x]$: campo con la chiave
 - + $NEXT[x]$: puntatore al successore dell'oggetto nella lista.
- Una lista L è un oggetto con un campo $HEAD[L]$: puntatore al primo elem. x della lista.
- Lista vuota: $HEAD[L] = NIL$

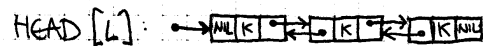
$HEAD[L]$: 

OPERAZIONI

- $EMPTY(L)$: svuota la lista: inizializza $HEAD[L]$ a NIL
- $NULL(L)$: restituisce $TRUE$ se la lista è vuota (ovvero se $HEAD[L] = NIL$), $FALSE$ altrimenti
- $INSERT(L, x)$: restituisce una nuova lista L' con in cima l'elemento x e a seguire la lista L concatenata (dunque $HEAD[L'] = x$ e $NEXT[x] = HEAD[L]$).
- $REST(L)$: restituisce una nuova lista L' ottenuta rimuovendo il primo elemento della lista L .
- $FIRST(L)$: restituisce $HEAD[L]$.
- $SEARCH(L, k)$: restituisce il puntatore all'oggetto x con $KEY[x] = k$
 - + se la lista è vuota, restituisce NIL
 - + altrimenti assegna ad x il puntatore all'oggetto successivo
 - + ripeti il punto 2 fino a che x non diventa NIL oppure $KEY[x]$ non è uguale a k
 - + restituisci x
- $DELETE(L, x)$: elimina l'elemento x dalla lista
 - + se la lista è vuota, non c'è nulla da cancellare
 - + se x è il primo elem. della lista, rimuovilo con $REST(L)$ e assegna a $HEAD[L]$ il valore $NEXT[x]$.
 - + altrimenti, a partire da $FIRST[L]$, ricerca il predecessore di x nella lista, chiamato y
 - + assegna al $NEXT[y]$ il valore di $NEXT[x]$, in tal modo x viene escluso dalla lista.

LISTE CONCATENATE DOPPIE

- Un elemento X della lista ha almeno 3 campi:
 - + $KEY[X]$: come nelle liste semplici
 - + $PREV[X]$: puntatore al predecessore dell'oggetto nella lista.
 - + $NEXT[X]$: come nelle liste semplici
- La lista è un oggetto con un campo $HEAD[L]$: primo elemento della lista

OPERAZIONI PRINCIPALI

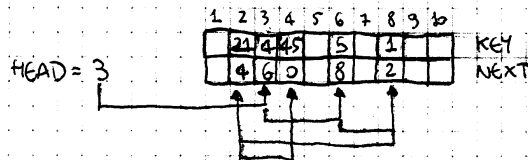
- $SEARCH(L, K)$:
 - 1+ memorizza in X l'inizio della lista
 - 2+ fino a che X è diverso da NIL e $KEY[X]$ è diverso dalla chiave di ricerca K
 - 3+ passa ad esaminare il successore di X
 - 4+ restituisce X (se restituisce NIL , la ricerca ha avuto esito negativo)
- $INSERT(L, X)$:
 - 1+ memorizza in $NEXT[X]$ la $HEAD[L]$
 - 2+ se la lista non era vuota, fai puntare il $PREV[]$ del "vecchio" primo elemento ad X
 - 3+ la "nuova" $HEAD[L]$ punta ad X
 - 4+ $PREV[X] = NIL$
- $DELETE(L, X)$:
 - 1+ se il predecessore di X esiste (è diverso da NIL), fai puntare il $NEXT[]$ del predecessore di X al successore di X
 - 2+ altrimenti (ovvero se X è il primo elemento), fai puntare ad $HEAD[L]$ il successore di X
 - 3+ se il successore di X è diverso da NIL (X non è l'ultimo elemento), fai puntare il $PREV[]$ del successore di X al predecessore di X .

SENTINELLE

- La cancellazione di un elemento dalla lista sarebbe più semplice se si potesse evitare la gestione dei casi limite relativi alla testa o alla coda della lista.
- Una sentinella è un oggetto fittizio, che rappresenta NIL ma che ha tutti i campi degli altri elementi della lista. (campo $NEXT[]$ e campo $PREV[]$)
- La sentinella trasforma la lista bidirezionale in una lista circolare.
- Vantaggi nell'uso delle sentinelle:
 - + possiamo ridurre i fattori costanti dei tempi di esecuzione.
 - + miglioriamo la gestione e la leggibilità del codice.
- Svantaggi: + richiediamo memoria supplementare.

LISTE SEMPLICI MEDIANTE ARRAY

- Per rappresentare una lista si possono usare due (o più) array.
- Nel caso di due array:
 - + array KEY, contenente i valori delle chiavi
 - + array NEXT, contenente i riferimenti ai successivi
- Presenza di un campo HEAD: indice del primo elem. della lista
- Nota bene: un riferimento è un indice dell'array.



- Dal momento che, per eseguire le operazioni REST e INSERT, è necessario conoscere le memorie delle posizioni libere, per far ciò si utilizza, all'interno dello stesso array, una lista libera, che collega tutte le posizioni libere.
- La struttura ha dunque un'ulteriore componente: FREE, l'indice della prima posizione libera.

OPERAZIONI:

- **EMPTY (A):** inizializza una lista libera: + HEAD = 0 (nessun elem. nella lista)

HEAD = 0

FREE = 1

1	2	3	4	5	6	7	8	9	
2	3	4	5	6	7	8	9	0	

KEY
NEXT

- + viene inizializzato FREE a 1, e programmi = vamente tutti i campi NEXT di ciascun elem. all'indice successivo
- + nel campo NEXT dell'ultimo elem. viene inserito zero

- **ALLOCATE (A):** una posizione viene prelevata dalla lista libera e viene restituito l'indice corrispondente:
 - + se FREE = 0, la lista è piena
 - + altrimenti memorizza in X la prossima posizione libera (X = FREE)
 - + memorizza in FREE l'indice della prossima cella libera (FREE = NEXT[X])
 - + restituisci X
- **FREE (A, X):** una posizione viene liberata e inserito nella lista libera
- **INSERT (A, X):**
 - + allora lo spazio per X (tramite ALLOCATE)
 - + inserire X nella lista (modifica campi KEY[i] e NEXT[i])
 - + assegna ad HEAD l'indice dell'elem. appena inserito.
- **DELETE-FIRST (A):** cancella il primo elem. della lista e libera la posizione corrispondente, che viene inserito nella lista libera (tramite FREE).

TABELLE HASH

- Utili per implementare dizionari
- Basate sul concetto di coppie chiave + valore.

TABELLE AD INDIRIZZAMENTO DIRETTO

- Funzionano bene quando l'universo U delle chiavi è ragionevolmente piccolo
- Dato $U = \{0, 1, 2, \dots, m-1\}$, per rappresentare l'insieme dinamico si usa un array $T[0, \dots, m-1]$, in cui ogni slot corrisponde ad una chiave nell'universo U .

OPERAZIONI:

- DIRECT-ADDRESS-SEARCH (T, K): restituisce il puntatore all'elem. con chiave K
- DIRECT-ADDRESS-INSERT (T, x): inserisce x nella tabella
- DIRECT-ADDRESS-DELETE (T, x): cancella x dalla tabella } x puntatore all'elem.

Tutte le operazioni vengono eseguite con tempo $O(1)$

TABELLE HASH

- Una funzione HASH definisce una corrispondenza tra l'universo U delle chiavi e gli indici della tabella hash $T[0, \dots, m-1]$
- L'elemento con chiave $K \in U$ si trova dunque nella posizione $h(K)$ di T .
- La funzione $h()$ è adottata per ridurre l'intervallo degli indici che devono essere gestiti: dunque per avere $m \ll |U|$
- $h()$ deve essere deterministica, e deve essere scelta in modo che le chiavi vengano distribuite in modo uniforme sugli m indici a disposizione
- Riducendo m rispetto al numero effettivo delle chiavi, è possibile che a due chiavi possano corrispondere la stessa posizione nella tabella (COLLISIONE).
- Una buona funzione HASH dovrebbe minimizzare le collisioni, ma dato che $m < |U|$, è impossibile non averne.

RISOLUZIONE DELLE COLLISIONI PER CONCATENAZIONE

- Si mettono tutti gli elementi che collidono nella stessa posizione in una lista concatenata.
- La posizione j contiene un puntatore alla testa della lista, o NIL se non vi sono elementi memorizzati in quella posizione.

OPERAZIONI

- CHAINED-HASH-SEARCH (T, K): cerca un elem. con chiave K nella lista $T[h(K)]$
- CHAINED-HASH-INSERT (T, x): inserisce l'elem. x in testa alla lista $T[h(KEY[x])]$
- CHAINED-HASH-DELETE (T, x): cancella x dalla lista $T[h(KEY[x])]$

EFFICIENZA:

- INSERT: $O(1)$ nel caso peggiore
- DELETE: $O(1)$ se le liste sono bidirezionali, $O(m)$ se semplici (m lunghezza lista)
- SEARCH: $O(m)$ (m lunghezza lista)

ULTERIORI CONSIDERAZIONI SULL'EFFICIENZA

- Ipotesi di base: uniformità semplice della funzione HASH: qualunque elemento possa corrispondere in modo equamente probabile a una delle m posizioni, indipendentemente dalla posizione degli altri elementi.
- Si definisce $\alpha = m/m$, fattore di carico, numero medio di elementi memorizzati in ogni lista concatenata.
- Se si suppone che $h(k)$ sia calcolata in tempo $O(1)$, allora sia una ricerca con successo che una senza successo sono portate a termine con tempo $O(1 + \alpha)$
- Inoltre, fissato un universo di chiavi U e una relativa tabella hash, se il num. di posizioni della tabella è proporzionale al num. di elem. nella tabella ($m = O(n)$), si ha: $\alpha = m/m = O(n/m) = O(1)$. Dunque la ricerca richiede in media tempo costante.

FUNZIONI HASH

- Una buona funzione hash è tale quando, applicata ad una chiave, genera dei numeri pseudocasuali uniformemente distribuiti nell'intervallo $1, \dots, m$
- Una buona funzione hash è tale quando, per chiavi simili, vengono generati indici diversi, indipendenti dalla configurazione delle chiavi.

METODO DIVISIONE

- $h(k) = k \bmod m$
- Da evitare:
 - + m potenza di 2, quando le chiavi sono binarie
 - + m potenza di 10, quando le chiavi sono numeri decimali
- Da auspicare: m numero primo lontano da potenze di 2 o 10

METODO MOLTIPLICAZIONE

- $h(k) = \lfloor m(k \cdot A - \lfloor k \cdot A \rfloor) \rfloor$, con $A \in (0, 1)$, fattore costante
- di solito $m = 2^p$, in tal modo è facile implementare questa funzione su calcolatori.

HASHING UNIVERSALE

- Si definisce un insieme di funzioni hash e a run-time se ne sceglie una. L'algoritmo dunque, per uno stesso input, può comportarsi diversamente per ogni esecuzione.
- Sia \mathcal{H} un insieme di funzioni $U \rightarrow \{0, 1, \dots, m\}$. \mathcal{H} è universale se $\forall (x, y) \in U$, il numero di funzioni hash $h \in \mathcal{H}$ tali che $h(x) = h(y)$ è pari a $|\mathcal{H}|/m$
- La probabilità di una collisione tra x e y è $1/m$ se $x \neq y$

TABELLE HASH CON INDIRIZZAMENTO APERTO

- Questa rappresentazione non fa uso di puntatori
- il fattore di carico α è sempre minore o uguale a 1
- Tutto il modo è contenuto nell'array T
- le collisioni vengono gestite memorizzando gli elementi in altri punti della hash stessa.
- di solito queste tabelle sono usate quando non sono previste cancellazioni

OPERAZIONI

- **HASH-INSERT** (T, K): inserisce un elem. con chiave K : + esamina la posizione $h(K, i=0)$
 - + se è vuota, inserisce l'elemento e restituisce l'indice
 - + se è piena, incrementa di uno i e ricomincia, fino a che $i \neq m$
 - + quando $i = m$, allora la tabella è piena \rightarrow overflow
- **HASH-SEARCH** (T, K): cerca un elem. con chiave K . Simile all'inserzione, solo che l'algoritmo termina non appena trova un campo NIL (se la chiave ci fosse stata, essendo la posizione vuota, sarebbe stata inserita proprio in quella posizione).
- **HASH-DELETE** (T, K): cancella l'elem. con chiave K . la posizione viene marcata con DELETED e non con NIL. Si dovrebbe riavviare il metodo di INSERT per usare anche questa componente.

TECNICHE DI SCANSIONE

- **SCANSIONE LINEARE**: $h(K, i) = (h'(K) + i) \bmod m$, per $i = 0, 1, \dots, m-1$
 - + non molto valida: fenomeno di agglomerazione primaria
 - + non una buona funzione hash uniforme
- **SCANSIONE QUADRATICA**: $h(K, i) = (h'(K) + C_1 \cdot i + C_2 \cdot i^2) \bmod m$, con $C_1, C_2 \neq 0$
- **HASHING DOPPIO**: $h(K, i) = (h_1(K) + i \cdot h_2(K)) \bmod m$ dove h_1 e h_2 sono due funzioni hash. Comportamento vicino allo schema "ideale" della funzione hash uniforme.

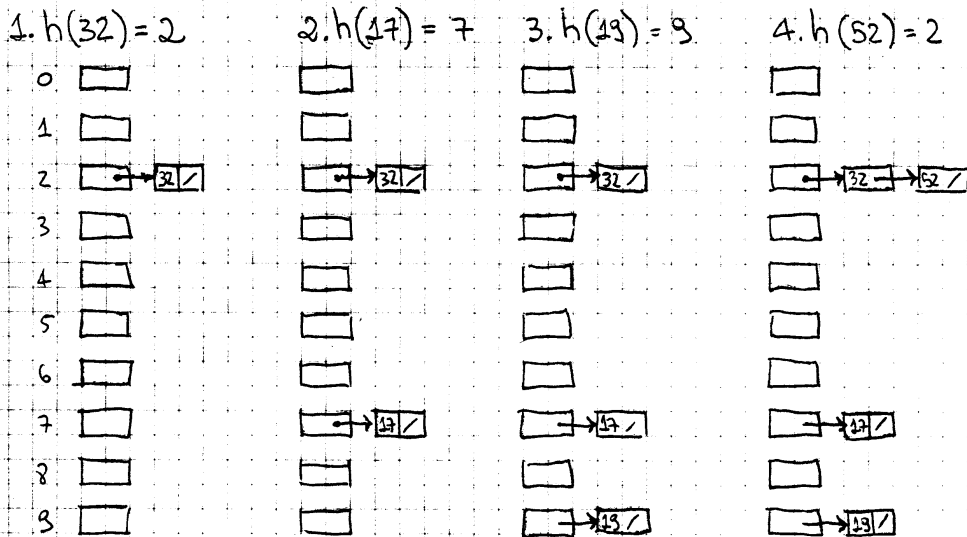
EFFICIENZA:

- Ricerca senza successo: num. medio di accessi $1/(1-\alpha)$, se $\alpha < 1$
- Ricerca con successo: num. medio di accessi $(1/\alpha) \cdot \ln(1/(1-\alpha))$, se $\alpha < 1$

ESEMPIO - TABELLE HASH CON CONCATENAZIONE

$m = 10$, $h(k) = k \bmod m$, chiavi = $\{32, 17, 19, 52, 33, 15, 38, 46\}$

Illustrare l'inserimento delle chiavi in una Tabella Hash con concatenazione

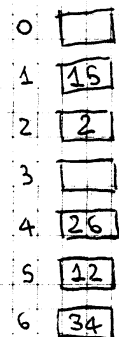


ESEMPIO - TABELLE HASH CON INDIRIZZAMENTO APERTO - HASHING DOPPIO

$m = 7$, chiavi = $\{34, 12, 15, 26, 2, 32, 40\}$, $h_1(k) = k \bmod 7$, $h_2(k) = 1 + (k \bmod 6)$

Quindi $h(k, i) = [(k \bmod 7) + i(1 + k \bmod 6)] \bmod 7$

1. $K = 34$, $h(34, 0) = [6] \bmod 7 = 6$. E' LIBERO? SI
 2. $K = 12$, $h(12, 0) = [5] \bmod 7 = 5$. E' LIBERO? SI
 3. $K = 15$, $h(15, 0) = [1] \bmod 7 = 1$. E' LIBERO? SI
 4. $K = 26$, $h(26, 0) = [5] \bmod 7 = 5$. E' LIBERO? NO $\rightarrow i++$
 $h(26, 1) = [5 + 1 \cdot (3)] \bmod 7 = 1$. E' LIBERO? NO $\rightarrow i++$
 $h(26, 2) = [5 + 2 \cdot (3)] \bmod 7 = 4$. E' LIBERO? SI
 5. $K = 2$, $h(2, 0) = [2] \bmod 7 = 2$. E' LIBERO? SI
-

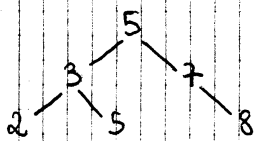


ALBERI BINARI DI RICERCA

- Organizzato ad albero binario. Un tale albero può essere rappresentato da una struttura di dati con puntatori in cui ciascun nodo è un oggetto.
- Ciascun nodo contiene i campi:
 - + KEY[x]: il valore memorizzato nel nodo
 - + LEFT[x]: puntatore al figlio sinistro (NIL se assente)
 - + RIGHT[x]: puntatore al figlio destro (NIL se assente)
 - + P[x]: puntatore al padre (NIL solo nella radice)
- Le chiavi sono sempre memorizzate in modo che sia soddisfatta la proprietà dell'albero binario di ricerca:

Per ogni nodo x dell'albero:

- + tutti i nodi del sottoalbero SX di x hanno chiave minore o uguale a quella di x.
- + tutti i nodi del sottoalbero DX di x hanno chiave maggiore o uguale a quella di x.

OPERAZIONI

- INORDER-TREE-WALK(x): stampa le chiavi in modo ordinato, eseguendo una visita in ordine simmetrico, a partire dal nodo x.
- ITERATIVE-TREE-SEARCH(x, k): ricerca un elemento con chiave k in un ABR, dove x è il puntatore alla radice, e restituisce il puntatore al nodo, se esiste; NIL altrimenti.
 1. + x esiste e la sua chiave è diversa da k?
 2. + se sì, confronta la sua chiave con k
 3. + se k è minore di KEY[x], proseguì la ricerca nel sottoalbero sinistro (x ← LEFT[x])
 4. + altrimenti proseguì la ricerca nel sottoalbero destro (x ← RIGHT[x])
 5. + torna al punto 1
 6. + restituisce x
- TREE-SEARCH(x, k): versione ricorsiva di ricerca.
- TREE-MINIMUM(x): riporta un puntatore al minimo elemento dell'ABR, dove x è la radice dell'albero. (il minimo è il nodo più a sinistra senza figli sinistri)
- TREE-MAXIMUM(x): riporta un puntatore al massimo elemento dell'ABR, dove x è la radice dell'ABR. (il massimo è il nodo più a destra senza figli destri)

- **TREE-INSERT (T, z)**: restituire T modificato con l'aggiunta della foglia z .
 - + se T è vuoto: crea l'albero con l'unico nodo z
 - + altrimenti, se z ha chiave minore della radice di T , inserirlo nel sottoalbero sinistro, altrimenti inserirlo nel sottoalbero destro.
 - + il caso base della ricorrenza è quando l'albero sinistro o destro, dove dovrebbe essere inserito il nodo, è NIL.
 - + in tal caso, inserisci il nodo.
- **ITERATIVE-TREE-INSERT (T, z)**: versione iterativa dell'inserimento.
- **TREE-DELETE (T, x)**: cancella il nodo puntato da x dall'ABR T , e modifica T affinché mantenga le proprietà ABR.
 - + se x ha al massimo un figlio, si cancella x , bypassandolo
 - il puntatore ad x del padre di x viene sostituito con il puntatore al figlio di x
 - + se x ha due figli, x viene sostituito dal suo successore y (che è l'elemento minimo del sottoalbero destro di x)
 - y non ha figlio sinistro: viene cancellata, bypassandola
 - il contenuto (chiave e dati satellite) di x viene sostituito dal contenuto di y .
- **TREE-SORT (A)**: ordina l'array A sfruttando la visita simmetrica degli ABR.
 - + costruire un ABR T con gli elementi di A
 - + visita T in ordine simmetrico e copia gli elementi in A

EFFICIENZA:

- Tempo di esecuzione proporzionale all'altezza dell'albero h .
- Caso peggiore: albero completamente sbilanciato $\rightarrow h = n$ (numero di nodi)
- Caso migliore: $h = \log_2 n$
- Caso medio come caso migliore, cambiando dei fattori costanti.

RB - ALBERI

- Un RB-albero è un albero binario che mantiene un buon bilanciamento
- È un ABR con in più un campo binario in ogni nodo.
- Struttura di un nodo:

+ KEY [x]	} come negli ABR
+ LEFT [x]	
+ RIGHT [x]	
+ P [x]	
+ COLOR [x]: colore del nodo: rosso o nero	

Un albero di ricerca binario è un RB-albero se soddisfa le 4 proprietà RB:

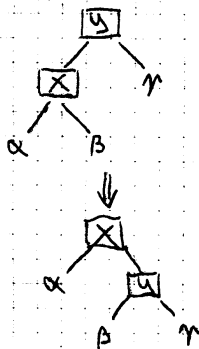
1. ciascun nodo è rosso o nero
2. ciascuna foglia (NIL) è nera
3. se un nodo è rosso, allora entrambi i figli sono neri
4. ogni cammino semplice da un nodo ad una foglia ausiliaria contiene lo stesso numero di nodi neri.

- la b-altezza di un nodo è il numero di nodi neri su un cammino da un nodo x, non incluso, ad una foglia.
- Un RB-albero con n nodi interni ha, al più, un'altezza di $2 \lg(n+1)$ (DHL)
- se h è la profondità della foglia meno profonda, la foglia più profonda si trova al massimo a profondità $2h+1$
- Un sottoalbero con radice in un qualsiasi nodo x contiene almeno $2^{b_h(x)} - 1$ nodi interni.

OPERAZIONI

- TREE-INSERT (T, x) } modificano la struttura dell'RB-albero, che deve essere ripristinata
- TREE-DELETE (T, z) }
- LEFT-ROTATE (T, x) } mantengono le proprietà ABR (visite in ordine simmetrico)
- RIGHT-ROTATE (T, x) }

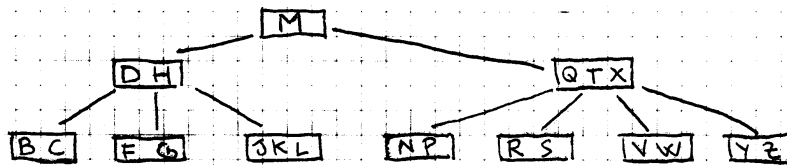
Nel caso di RIGHT-ROTATE: • y è maggiore di tutti gli elementi di sinistra, quindi deve trovarsi nella parte destra



- tutti gli elementi di b sono maggiori di x ma minori di y quindi b diventa sottoalbero sinistro di y
- x diventa la radice dell'albero.

B-ALBERI

- Albero m -ari bilanciati di ricerca
- un nodo di un b -albero contiene più di una chiave: ogni nodo è un punto di divisione
- se un nodo X ha m chiavi allora ha $m+1$ figli
- con questa struttura si riducono il numero di accessi alla memoria

STRATEGIA GREEDY

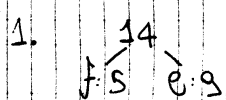
- Ad ogni passo si prende la decisione che sembra la migliore, la decisione localmente ottima.
- Problema: + un insieme S di oggetti
+ un insieme di vincoli relativi a sottoinsiemi di S
- Soluzione ammissibile: $A \subseteq S$ che soddisfa i vincoli.
- Soluzione ottima: soluzione ammissibile che massimizza o minimizza una funzione obiettivo
- Metodo GREEDY \rightarrow costruzione incrementale della soluzione:
 - + S viene ordinato secondo una misura di convenienza
 - + Si inizializza $A = \emptyset$
 - + Ad ogni stadio: si aggiunge ad A l'elemento di S più conveniente che non violi i vincoli, fino ad ottenere A ammissibile.
- Perché un problema di ottimizzazione possa essere risolto con una strategia greedy, deve avere le seguenti caratteristiche:
 - + Proprietà della scelta greedy: si può ottenere una soluzione ottima globale prendendo decisioni che sono ottimi locali.
 - + Sottostuttura ottima: una soluzione ottima del problema contiene al suo interno una soluzione ottima del sottoproblema.

CODICI DI HUFFMAN

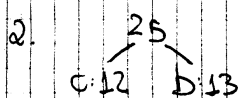
- Si basa su una strategia greedy
- Convenzioni della codifica:
 - + codice binario per caratteri: ogni carattere è rappresentato da una stringa binaria ("parola" del codice)
 - + codice a lunghezza variabile: i caratteri possono essere rappresentati da parole di diversa lunghezza
 - + codice prefisso: nessuna parola è un prefisso di qualche altra parola del codice.
- Per la codifica si utilizza un albero binario
- Il codice ottimali di un file è sempre rappresentato da un albero pienamente binario: ogni nodo interno ha esattamente due figli.

ESEMPIO

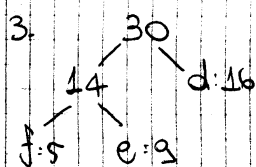
f:5 e:9 c:12 b:13 d:16 a:45 → ordinate secondo le occorrenze



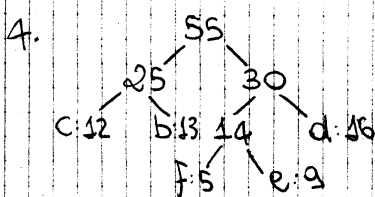
→ c:12 b:13 14 d:16 a:45



→ 14 d:16 25 a:45

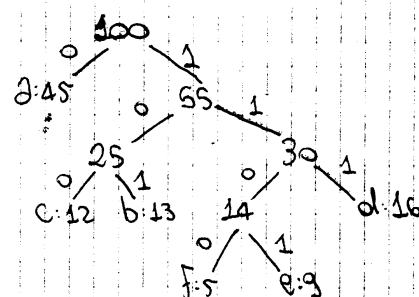


⇒ 25 30 a:45



⇒ a:45 55

5.



a:45 = 0
 b:13 = 101
 c:12 = 100
 d:16 = 111
 e:9 = 1101
 f:5 = 1100

SELEZIONE DI ATTIVITÀ

- Problema: massimizzare il numero di attività in competizione fra loro
 $S = \{J_1, J_2, \dots, J_m\}$ e che utilizzano una data risorsa.
- Soluzione ammissibile: un insieme di attività mutuamente compatibili.
- Obiettivo: massimizzare il numero di attività.
- Per ogni J_i è specificato un tempo di inizio s_i e un tempo di fine f_i , con $f_i > s_i$.
- J_i e J_j sono compatibili se $[s_i, f_i)$ e $[s_j, f_j)$ non si sovrappongono.
- Passi dell'algoritmo:
 - + ordinare le attività secondo valori crescenti di f_i :
 $f_1 \leq f_2 \leq f_3 \leq \dots \leq f_m$
 - + inizializzazione: $A = \{1\}$, last = 1 (last: ultima attività inserita)
 - + Per $i = 2, \dots, m$, considerare l' i -esima attività:
 - se è compatibile con le attività in A (se inizia dopo f_{last}) aggiungere i ad A e porre last = i .

GRAFI

- Un grafo orientato G è una coppia (V, E) dove V è un insieme finito ed E è una relazione binaria su V . L'insieme V è chiamato l'insieme dei vertici di G ed i suoi elementi sono chiamati vertici. L'insieme E è chiamato l'insieme degli archi di G . Sono possibili cappi, ovvero archi da un vertice a se stesso.
- In un grafo non orientato $G = (V, E)$, l'insieme degli archi E è costituito da coppie non ordinate di vertici, cioè un arco è un insieme $\{u, v\}$ dove $u, v \in V$ e $u \neq v$. I cappi sono proibiti.
- In un grafo orientato:
 - + archi uscenti da un nodo x : (x, y)
 - + archi incidenti su un nodo x : (y, x)
 - + nodi adiacenti a un nodo x o successori di x : $\{y : (x, y) \in E\}$
 - + predecessori di un nodo x : $\{y : (y, x) \in E\}$
 - + grado di ingresso di un nodo x : num. di archi incidenti su x
 - + grado di uscita di un nodo x : num. di archi uscenti da x
 - + sorgente: nodo senza predecessori (grado di ingresso = 0)
 - + pozzo: nodo senza successori (grado di uscita = 0)

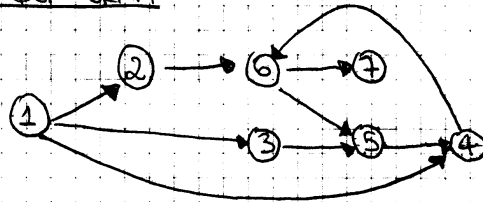
- Cammino da un nodo X a un nodo Y : sequenza di nodi (d_0, \dots, d_m) tali che:
 - 1+ $d_0 = X$
 - 2+ $d_m = Y$
 - 3+ per ogni $i = 0, \dots, m-1 \Rightarrow (d_i, d_{i+1}) \in E$
- la lunghezza m del cammino è il numero degli archi che compongono il cammino.
- Y è raggiungibile da X se esiste un cammino da X a Y
- Semicammino da X a Y come un cammino, cambia la condizione 3
 - 3+ per ogni $i = 0, \dots, m-1 \Rightarrow (d_i, d_{i+1})$ oppure $(d_{i+1}, d_i) \in E$
- Un grafo è fortemente connesso se e solo se per ogni $X, Y \in V$ esiste un cammino da X a Y
- Un grafo è debolmente connesso se e solo se per ogni $X, Y \in V$ esiste un semicammino da X a Y
- Cicli: cammino di lunghezza ≥ 1 che inizia e termina in uno stesso nodo.
 - + ciclo semplice: $(d_0, \dots, d_m), (d_1, \dots, d_m)$ sono tutti distinti
 - + lappio: ciclo di lunghezza 1
 - + Grafo semplice: senza lappi.
 - + Grafo aciclico: senza cicli.
- In un grafo NON ORIENTATO
 - + Ciclo: cammino (d_0, \dots, d_m) tale che:
 - $m \geq 3$
 - $d_0 = d_m$
 - d_1, \dots, d_{m-1} sono tutti distinti
 - + Grafo connesso: ogni coppia di vertici è connessa da un cammino.

VISITA IN PROFONDITÀ

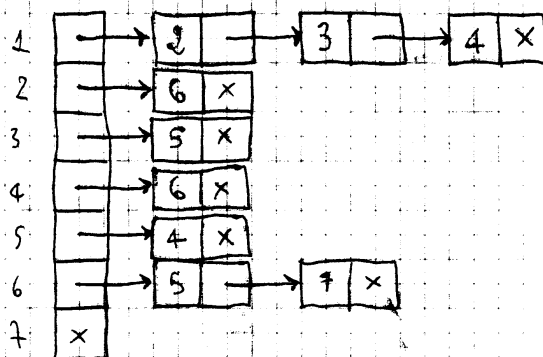
- Se il nodo di partenza $START$ non è già stato visitato, si analizza $START$ e, per ogni successore X di $START$, si visita, con lo stesso metodo, il grafo a partire da X , ricordando che $START$ è già stato visitato.
- Passi dell'algoritmo:
 - + parti dal nodo $START$
 - + analizza $START$ e marcalo "VISITATO"
 - + finché ci sono successori di $START$ non ancora visitati
 - sia X il prossimo
 - visita in profondità G a partire da X
- I successori di un nodo vengono visitati uno ad uno mediante chiamate ricorsive.
- I nodi in attesa di essere visitati sono memorizzati in una pila.

VISITA IN AMPIEZZA

- se il nodo di partenza START non è già stato visitato allora si analizza START, si visitano tutti i suoi successori, poi tutti i successori dei successori di START, e così via.
- Passi dell'algoritmo:
 - + visita il nodo START
 - + colora tutti i nodi di bianco
 - + inserisci START nella coda e coloralo di grigio
 - + finché la coda non è vuota:
 - estrai il primo elem. X dalla coda
 - analizza X
 - metti nella coda tutti i successori bianchi di X e colorali di grigio.
- i nodi in attesa di essere visitati sono collocati in una coda.

RAPPRESENTAZIONE DEI GRAFI

- Mediante LISTE DI ADIACENZA
 - + un array Adj di lunghezza $LENGTH[Adj] = V$
 - + Per $u \in V$: $Adj[u]$ è una lista con tutti i vertici v tali che $(u, v) \in E$
- Modo compatto per la rappresentazione di grafi sparsi: $|E| \ll |V|^2$
- Somma delle lunghezze di tutte le liste di adiacenza: $|E|$ per i grafi orientati, $2 \cdot |E|$ per i grafi non orientati.
- Memoria necessaria $O(V + E)$



- Mediante MATRICE di ADIACENZA

+ Matrice $A = (a_{ij})$ di dimensioni $|V| \times |V|$

$$+ a_{ij} = \begin{cases} 1 & \text{se } (i,j) \in E \\ 0 & \text{altrimenti} \end{cases}$$

- Memoria necessaria $\Theta(|V|^2)$

	1	2	3	4	5	6	7
1	0	1	1	1	0	0	0
2	0	0	0	0	0	1	0
3	0	0	0	0	1	0	0
4	0	0	0	0	0	1	0
5	0	0	0	1	0	0	0
6	0	0	0	0	1	0	1
7	0	0	0	0	0	0	0

GRAFI PESATI

- Ad ogni arco è associato un peso $w: E \rightarrow \mathbb{R}$
- Nelle liste di adiacenza il peso $w(u,v)$ dell'arco (u,v) è rappresentato insieme al modo v nella lista di adiacenza di u .
- Nelle matrici di adiacenza il peso è memorizzato al posto degli 1.

IMPLEMENTAZIONE DELLE VISITE MEDIANTE LISTE DI ADIACENZA.

- Operazione fondamentale: $\text{succ}(\text{Adj}, x)$: riporta un puntatore al primo elemento della lista dei successori del nodo x .
- Visita in AMPIEZZA: per tenere traccia dei nodi già inseriti nella coda e per segnalare i nodi già visitati, utilizziamo un array color

$$\text{color}[u] = \begin{cases} \text{WHITE} & \text{se } u \text{ non è stato visitato, ne è in coda} \\ \text{GRAY} & \text{se } u \text{ è nella coda, ma non visitato} \\ \text{BLACK} & \text{se } u \text{ è stato visitato} \end{cases}$$

- Passi dell'algoritmo a partire da S :
 - 1+ Colorare tutti i vertici di bianco, eccetto S (in grigio)
 - 2+ Inserimento di S nella coda Q
 - 3+ fino a che Q non è vuota:
 - estrai u dalla coda
 - per ogni vertice v adiacente a u :
 - se v è bianco, coloralo di grigio e mettilo in coda
 - Visitare u e colorarlo di nero

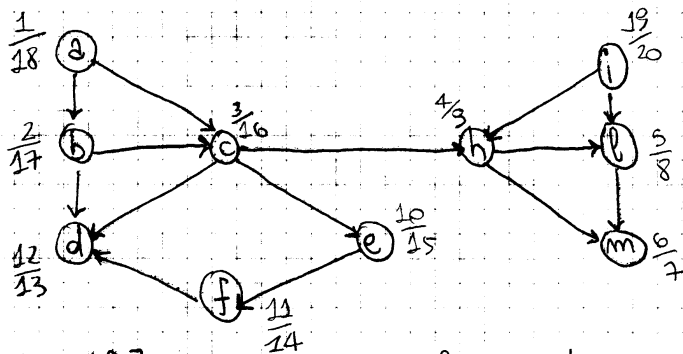
- Complessità della visita in ampiezza: $O(V+E)$
- Complessità della visita in profondità: $O(V+E)$

ORDINAMENTO TOPOLOGICO DI GRAFI ORIENTATI ACICLICI (DAG)

Un ordinamento topologico di un dag G è un ordinamento lineare dei vertici: c tale che se (u,v) è un arco in G , allora u viene prima di v

- Un algoritmo di ordinamento topologico si può basare sulla visita in profondità del grafo: per ciascun nodo viene registrato il tempo di inizio e fine visita.
- L'ordinamento è per tempi decrescenti di fine visita.
- Visita in profondità GENERALE:
 - + inizialmente i vertici sono tutti bianchi.
 - + quando un vertice u viene scoperto, diventa grigio: inizia la visita di u
 - + quando la visita di u termina (tutti i successori sono stati visitati), u diventa nero
 - + il tempo di inizio e fine visita di un nodo u sono registrati in due variabili:
 - $d[u]$: tempo di inizio visita
 - $f[u]$: tempo di fine visita
 - + si ha sempre $d[u] < f[u]$

ESEMPIO: illustrare l'ordinamento topologico a partire dal nodo a



- | | | |
|-----------------------------|----------------------------|---|
| 1. parto da a: $d[a]=1$ | 9. ne scelgo uno: h | 17. Torso indietro, l non ha successori |
| 2. a ha 2 successori: b e c | 10. parto da h: $d[h]=4$ | 18. $f[l]=8$, l nero |
| 3. Ne scelgo uno: b | 11. h ha 2 successi: l e m | 19. h ha successore m ed l, entrambi neri |
| 4. parto da b: $d[b]=2$ | 12. $h \rightarrow d[l]=5$ | 20. $f[h]=9$ |
| 5. b ha 2 successi: c e d | 13. l ha 1 successore: m | 21. c ha successi d e e |
| 6. ne scelgo uno: c | 14. $m \rightarrow d[m]=6$ | 22. $e \rightarrow d[e]=10$ |
| 7. parto da c: $d[c]=3$ | 15. m non ha successi | ... |
| 8. c ha 3 successi: d, e, h | 16. $f[m]=7$, m nero | |