

## Riassunto ASD: 1° Semestre

### Algoritmi Ordinamento -1

#### **insertionSort(int[] A)**

ordinamento: for e while annidati con scambi e confronti

stabile: sì            in place: sì

$T(n) = O(n^2)$              $T_o(n) = \theta(n)$              $T_p(n) = \theta(n^2)$

#### **mergeSort(int[] A, int p, int q)**

ordinamento: spezza array in due ricorsivamente fino a che  $p == q$ , poi ordina e unisce parti con Merge

stabile: sì            in place: no

$T(n) = \theta(n \log n)$  costante

#### **merge(int[] A, int p, int q, int r)** (supporto per MergeSort)

dato un array A e indici  $p < r < q$ , array ordinato intervalli  $[p, r]$  e  $[r+1, q]$

ordina A da  $[p, q]$ : salva il minore tra p e  $r+1$  e ne incrementa indice fino che  $p == r$  o  $r+1 == q$

stabile: sì            in place: no

$T(n) = \theta(q - p)$  costante

#### **selectionSort(int[] A)**

ordinamento: ricorsione coda (trasformabile ciclo for) cerca il minimo array e lo mette davanti

stabile: no            in place: sì

$T(n) = \theta(n^2)$

### Strutture Dati -1

#### **Linked List (lista concatenata)** (implementa una coda generica)

valori:

lista L: L.first(), L.last() → puntatori a primo e ultimo elemento lista

elemento x: x.key() → valore, x.next() → puntatore elemento successivo

metodi:

**creaCodaVuota()** → crea nuova coda vuota →  $T(n) = \theta(1)$

**inserimento(List L, Element x)** → aggiunge elemento coda →  $T(n) = \theta(1)$

**cancellazione(List L)** → rimuove elemento in testa coda →  $T(n) = \theta(1)$

#### **Vettore Sovradimensionato** (implementa una coda generica)

Salvati due valori: first, last → indici di primo e ultimo elemento della coda (si lavora modulo n)

#### **Max Heap** (implementa una coda con priorità)

Basata su vettore sovradimensionato.

Albero binario quasi completo → altezza albero  $\theta(\log n)$  dove n numero dei nodi

Ogni nodo: un genitore; due figli, sx e dx; priorità minore o uguale a quella del padre

Heap H: H.length → lunghezza vettore (max)  $\geq$  H.heapSize = n → porzione vettore usata heap (n° nodi)

metodi:

**maxKey(Heap H)** → valore massimo heap (primo elemento) →  $T(n) = \theta(1)$

**left(i), right(i)** → figlio sinistro, destro del nodo →  $T(n) = \theta(1)$

**parent(i)** → genitore del nodo →  $T(n) = \theta(1)$

**extractMaxHeap(Heap h)** → estrae max (scambia ultimo elemento) risistema heap chiamando heapify

$T(n) = O(\log n)$

**heapify(Heap h, int i)** → risistema la max heap (heap ricevuta è una max heap con massimo sbagliato)

$T(n) = O(\log n)$

**insertMaxHeap(Heap h, node k)**

inserisce nodo fine heap, risistema heap portando nodo in alto fino a che è minore del genitore

in place: sì  $T(n) = O(\log n)$

**buildMaxHeap(int[] A)**

dato array A di interi genera una MaxHeap. Si chiama heapify  $n/2$  volte

in place: sì  $T(n) = \theta(n)$

**Algoritmi Ordinamento -2****heapSort(int[] A)**

Trasforma array in maxHeap. Poi ricorsivamente scambia 1° e ultimo elemento, heapSize--, chiama heapify

stabile: no in place: sì

$T(n) = O(n \log n)$

$To(n) = O(n \log n)$

$Tp(n) = O(n \log n)$

**quickSort(int[] A, int p, int q)** (si può migliorare  $Tp$  cercando mediano di A col Select e usarlo su partition)

sfrutta partition: scelto pivot x che pone a destra tutti valori  $> x$  e a sinistra i valori  $\leq x$ .

riordina tramite una ricorsione ad albero sui due rami sinistro e destro

stabile: no in place: sì

$T(n) = O(n^2)$

$To(n) = Tm(n) = \theta(n \log n)$

$Tp(n) = O(n^2)$

**Algoritmo utili**

**partition(int[] A, int p, int q)** (supporto per QuickSort e QuickSelect)

q è il pivot: termina con valori  $\leq$  a sx di q,  $>$  a dx di q. restituisce la posizione in cui finisce q

stabile: no in place: sì

$T(n) = \theta(n)$

**MOMSelect(int[] A, int p, int q, int i)**

*obiettivo*: trovare valore x dell'i-esimo elemento array se fosse ordinato (i-esimo valore più piccolo)

p e q: limiti sx e dx, i: i-esimo elemento da cercare

*procedimento*:

- 1) si spezza array in blocchi da 5 elementi (in caso ultimo blocco con meno elementi)
- 2) si ordina ogni blocco
- 3) si prende valore centrale di ogni blocco (mediano)
- 4) si salvano i mediani in un array di supporto B di dimensione  $n/5$
- 5) si procede ricorsivamente al punto 1 fino a che non resta un solo blocco e se ne prende il mediano (ho trovato  $y :=$  mediano dei mediani)
- 6) chiama partition su A usando perno  $x \rightarrow$  scopro  $j :=$  posizione di x, array ora è diviso tra  $\leq$  e  $>$  di y
- 7)  $i == j$ ?
  - a. ho trovato x: è y stesso
  - b. ricorsivamente al punto 1 ma solo sulla metà array di interesse fino a che  $i == j$

*proprietà di y, mediano dei mediani*:

minori =  $\{x \in A : x \leq y\} \Rightarrow \frac{1}{4}|A| \leq |\text{minori}| \leq \frac{3}{4}|A|$ ; maggiori =  $\{x \in A : x > y\} \Rightarrow \frac{1}{4}|A| \leq |\text{maggiori}| \leq \frac{3}{4}|A|$

ovvero: y è un valore abbastanza centrale array

$T(n) = \{ \theta(1) \text{ se } n=1 \quad T(1/5*n) + T(3/4*n) + \theta(n) \text{ se } n>1 \}$

dato che  $1/5 + 3/4 < 1 \Rightarrow$  dal lemma segue che:

$T(n) = \theta(n)$

**binarySearch(int[] A, int p, int q, int x)**

trova indice i del valore x da un array ordinato in  $O(\log n)$

calcola  $r = p+q / 2$ . Ricorsivamente con  $[p, r-1]$  (se  $x < r$ ) o  $[r+1, q]$  (se  $x > r$ ) fino a che  $x == A[r]$  oppure  $p == q$

(ottimizzazione: ricerca esponenziale ( $j = 2*j$  mentre  $a[j] < r$ ), poi  $\text{binarySearch}(A, j/2, j, x)$ , ma utile se  $i < n/2$ )

$T(n) = O(\log n)$

$To(n) = \theta(1)$

$Tp(n) = \theta(\log n)$

## Algoritmi Ordinamento -3 con $T(n) = \theta(n)$

Basati su assunzioni input

### **countingSort(int[] A, int[] B, int k)**

*assunzioni:*

$0 \leq A[i] \leq k, k \in O(n)$

*procedimento:*

genera array supporto C con  $C[i] = n^\circ$  volte che i compare in A. (Per stabilità poi:  $C[i] += C[i-1]$ )

ciclo for da n a i:  $B[ C[ A[i] ] ] = A[i]$  e  $C[ A[i] ] -= 1$

(guardo valore di x in a[i], da cui guardo posizione in cui va salvato da C (per la stabilità), su cui scrivo in B)

stabile: si in place: no

$T(n) = O(n + k)$        $To(n) = O(n + k)$        $Tp(n) = O(n + k)$

### **radixSort(int[] A, int d)**

*assunzioni:*

ogni valore array ha al massimo d cifre.  $d \in [0, B-1]$  dove B è la base.  $d \in O(n)$

*procedimento:*

si ordinano i numeri d volte dalla cifra meno significativa alla più significativa.

i numeri vengono ordinati al massimo d volte (se hanno esattamente d cifre).

stabile: si in place: no

$T(n) = O(n * d)$

### **bucketSort(int[] A)**

*assunzioni:*

$0 \leq A[i] \leq 1$ . I valori devono essere distribuiti abbastanza uniformemente

*procedimento:*

si divide  $[0, 1]$  in n sottointervalli di lunghezza k ( $k = 1/n$ ):  $[i/n, (i+1)/n]$  con  $0 \leq i < n, i \in \mathbb{N}, n \in \mathbb{N}$

si crea B,  $|B| = n$ , array di liste concatenate (ogni lista rappresenta il k-esimo intervallo)

si colloca ogni  $A[i]$  nel relativo  $B[i]$ :  $\text{insert}( B[ \text{floor}(A[i] * n) ], A[i] ) \rightarrow B$  è una tabella di Hash

si ordina ogni k-esimo intervallo,  $B[i]$ , con insertion, merge ecc

$T(n) = \theta(n)$

## Strutture Dati -2

### **Tabelle Hash**

memorizzare insieme elementi  $K \in$  universo valori  $U$  con  $|K| \ll |U|$ . K varia dinamicamente nel tempo.

valori di ogni elemento x: x.key: chiave univoca  $\in [0, |U|-1]$

*obiettivi di efficienza:*

- spazio: uso limitato memoria:  $\theta(|K|) \Rightarrow$  utilizzare array supporto T con  $|T| = m$
- tempo: Inserimento, Ricerca, Cancellazione (IRC) in circa  $\theta(1)$

funzione hash  $h: \{0, \dots, |U|-1\} \rightarrow \{0, \dots, m-1\}$  essendo  $|U| > |T| \Rightarrow h$  non è iniettiva  $\Rightarrow$  collisioni

*gestione delle collisioni:*

- **Hash con Chaining** (soluzione poco usata)

si utilizzano liste concatenate: ogni  $T[i]$  è puntatore alla lista contenente  $x_j \in K : j \in U, h(x_j.\text{key}) = T[i]$

valori:  $n = n^\circ$  elementi presenti tabella,  $m = |T|$  = dimensione tabella  $\Rightarrow n \leq m$

*metodi:*

**insertChaining(int[] T, function h, element x)**  $\rightarrow$  inserimento elemento relativa lista  $\rightarrow T(n) = \theta(1)$

**ricerca, cancellazione:**  $O(|T[ h(x.\text{key}) ]|)$

$Tp(n) = \theta(n)$  (tutti  $x_i$  in un  $T[i]$ )

$Tm(n) = \theta(1+\alpha)$  dove  $\alpha = n/m$  è fattore di carico (se h soddisfa hashing uniforme semplice)

*proprietà di una buona funzione hash:*

- suriettività:  $\forall i \in [0, \dots, m-1], \exists x \in U : h(x.\text{key}) = i$
- uniformità: h deve riempire T modo uniforme (equiprobabile):  $\forall x \in U : P( h(x.\text{key}) == i ) = 1/m$

possibili funzioni di hash:

- **bucketSort** → efficiente se vale hashing uniforme semplice  
 $x.key \in [0, 1), h(x.key) = \text{floor}(x.key * m)$
- **metodo moltiplicazione** → scelgo H, prendo la parte frazionaria di  $x.key * H$  e moltiplico per m  
 $x.key \in \mathbb{N}, \text{ scelgo } H \in (0, 1) \rightarrow h(x.key) = \text{floor}(x.key * H - \text{floor}(x.key * H)) * m$
- **metodo divisione** → prendo modulo della chiave  $\Rightarrow |T| = m$  deve essere un numero primo  
 $x.key \in \mathbb{N}, h(x.key) = (x.key \bmod m)$

- **Open Addressing**

si utilizza una sequenza di scansioni:

$m = |T|$  = dimensione tabella

modificata funzione hash:  $h: \{0, \dots, |U|-1\} * \{0, \dots, m-1\} \rightarrow \{0, \dots, m-1\}$

due input:  $x.key$  e i-esimo tentativo

se  $h(x.key, i)$  è occupata → provo  $h(x.key, i+1)$

$\langle h(x.key, 0), \dots, h(x.key, m-1) \rangle \rightarrow$  data  $x.key$  genero permutazione di  $\langle 0, \dots, m-1 \rangle$  (no ripetizioni)

- **IRC(int[] T, function h, element x):**  $T(n) = O(m)$ 
  - se vale hashing uniforme semplice  $\Rightarrow$  IRC:  $T_m(n) = \theta(1)$

**inserimento** → inserisco x alla prima cella libera trovata dopo i tentativi falliti della funzione  $h(x.key, i)$

**cancellazione** → cerco valore usando h → se lo trovo lo sostituisco con DEL (costante  $\neq$  NIL)

**ricerca** → cerco valore: mi fermo se lo trovo oppure se trovo NIL (se trovo DEL continuo a cercare)

*proprietà* di una buona funzione hash:

- suriettività:  $h(x.key, i)$  deve essere una permutazione di  $\langle 0, \dots, m-1 \rangle$
- uniformità: tutte le  $m!$  possibili permutazioni hanno stessa probabilità  $1/m!$

possibili sequenze di scansioni:

- **lineare** →  $h(k, i) = (h'(k) + a*i) \bmod m \Rightarrow a, m$  devono essere coprimi  
problemi:
  - *Primary Clustering*: in T si formano blocchi celle contigue occupate
  - anche se  $x \neq y, h(x.key, 0) \neq h(y.key, 0)$  da un certo punto sequenze coincidono
    - si utilizzano solo le m permutazioni ordinate
- **quadratica** →  $h(k, i) = (h'(k) + i c_1 + i^2 c_2) \bmod m$   
problemi:
  - verificare che  $c_1, c_2, m$  garantiscano permutazione (es.  $c_1 = c_2 = 1/2, m = 2^x$ )
  - *Secondary Clustering*: se  $h'(x.key) = h'(y.key) \Rightarrow x$  e  $y$  stessa sequenza (dall'inizio)
    - si utilizzano al massimo m sequenze
- **doppio hashing** →  $h(k, i) = (h'(k) + i h''(k)) \bmod m \Rightarrow h''(k), m$  coprimi  $\Rightarrow m$  primo  
problema:
  - se  $h'(x.key) = h'(y.key)$  e  $h''(x.key) = h''(y.key) \Rightarrow x$  e  $y$  stessa sequenza (dall'inizio)
    - si utilizzano al massimo  $m^2$  sequenze

## Alberi Binari

valori nodo x:

x.key, x.left, x.right, x.parent

h altezza albero:  $\Omega(\log n)$ , se bilanciato  $\Rightarrow \Theta(\log n)$

metodi:

**preOrder(node x)**  $\rightarrow$  visito nodo, sx, dx  $\rightarrow T(n) = \Theta(n)$

**inOrder(node x)**  $\rightarrow$  visito sx, nodo, dx  $\rightarrow T(n) = \Theta(n) \Rightarrow$  restituisce array ordinato

**postOrder(node x)**  $\rightarrow$  visito sx, dx, nodo  $\rightarrow T(n) = \Theta(n)$

Per ricostruire un BT dalle sue stampe sono necessarie la inOrder e una tra postOrder e preOrder

## Alberi Binari di Ricerca: BST

Definizione:

- ogni nodo possiede una chiave intera univoca (no ripetizioni)
- se y si trova nel sottoalbero sx (dx) di x  $\Rightarrow y.key < (>) x.key$

metodi:  $\rightarrow T(n) = O(h) \Rightarrow T_p(n) = \Theta(n)$ , se  $n = h$   $T(n) = O(\log n) \Rightarrow h = \log n$  (BST bilanciato)

**BSTSearchMin(T, x)**  $\rightarrow$  ricerca minimo: scendo sempre sx (per max a dx)  $\rightarrow T(n) = O(h)$

**BSTSearch(T, x, k)**  $\rightarrow$  cerca chiave k  $\rightarrow T(n) = O(h)$

**BSTSuccessor(T, x)**  $\rightarrow$  trova minor numero maggiore di x  $\rightarrow T(n) = O(h)$

- se x.right  $\neq$  NIL: BSTSearchMin(x.right)
- else: risalgo albero fino a che un nodo y è un figlio sx di un qualche z  $\Rightarrow z$  è successore

**BSTInsert(T, x)**  $\rightarrow$  aggiungo come foglia (posizione dipende grandezza chiave)  $\rightarrow T(n) = O(h)$

**BSTDelete(T, x)**  $\rightarrow$  rimuove nodo da BST

- 1) se: z ha due figli NIL: cancello z
- 2) se: z ha un solo figlio w: aggancio padre y con figlio w, cancello z
- 3) se: z ha due figli non NIL
  - trovo w successore di x con chiave k'
  - cancello w (successore rientra caso 1 o 2) e scrivo k' al posto di z

Per ricostruire un BST è necessaria solo la postOrder o la preOrder

La in inOrder stampa l'array ordinato  $\Rightarrow$  non offre nessuna informazione  $\Rightarrow$  si possono costruire molti BST