Lista algoritmi

Algoritmi Ordinamento -1

insertionSort(int[] A)
ordinamento: for e while annidati con scambi e confronti
stabile: si in place: si
$T(n) = O(n^2)$ $To(n) = \theta(n)$ $Tp(n) = \theta(n^2)$

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

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) ≥ 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: si T(n) = O(log n)
buildMaxHeap(int[] A)
dato array A di interi genera una MaxHeap. Si chiama heapify n/2 volte
in place: si T(n) = θ(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: si
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 ≤ x.
riordina tramite una ricorsione ad albero sui due rami sinistro e destro
stabile: no in place: si
$T(n) = O(n^2)$ To(n) = Tm(n) = θ(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 ≤ a sx di q, > a dx di q. restituisce la posizione in cui finisce q
stabile: no in place: si
T(n) = θ(n)

MOMSelect(int[] A, int p, int q, int i)
obbiettivo: 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 → scopro j := posizione di x, array ora è diviso tra ≤ 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 ∈ A : x ≤ y} ⇒ ¼|A| ≤ |minori| ≤ ¾|A|; maggiori = {x ∈ A : x > y} ⇒ ¼|A| ≤ |maggiori| ≤ ¾|A|
ovvero: y è un valore abbastanza centrale array
T(n) = { θ(1) se n=1 T(1/5*n) + T(3/4*n) + θ(n) se n>1 }
dato che 1/5 + 3/4 < 1 ⇒ dal lemma segue che:
T(n) = θ(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 binarySearch(A, j/2, j, x), ma utile se i < n/2)
T(n) = O(log n) To(n) = θ(1) Tp(n) = θ(log n)

Algoritmi Ordinamento -3 con T(n) = θ(n)

Basati su assunzioni input

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

assunzioni:

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

procedimento:

genera array supporto C con C[i] = n° 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:

$0 \le A[i] \le 1$. 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 $0 \le i < n$, $i \in \mathbb{N}$, $n \in \mathbb{N}$

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]: insert(B[floor(A[i] * n)], A[i]) → B è una tabella di Hash

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

T(n) = θ(n)

Strutture Dati -2

Tabelle Hash

memorizzare insieme elementi K ∈ universo valori U con |K| << |U|. K varia dinamicamente nel tempo.

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

obbiettivi di efficienza:

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

funzione hash h: {0, … ,|U|-1} ⟶ {0, … , m-1} essendo |U| > |T| ⇒ h non è iniettiva ⇒ collisioni

gestione delle collisioni:

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

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

  valori: n = n° elementi presenti tabella, m = |T| = dimensione tabella ⇒ n ≤ m

  metodi:

  insertChaining(int[] T, function h, element x) → inserimento elemento relativa lista → T(n) = θ(1)

  ricerca, cancellazione: O(|T[h(x.key)]|)

  Tp(n) = θ(n) (tutti x_i in un T[i])

  Tm(n) = θ(1+α) dove α = n/m è fattore di carico (se h soddisfa hashing uniforme semplice)

  proprietà di una buona funzione hash:

  - suriettività: ∀i ∈ [0, … , m-1], ∃x ∈ U : h(x.key) = i
  - uniformità: h deve riempire T modo uniforme (equiprobabile): ∀x ∈ U : P(h(x.key) == i) = 1/m

possibili funzioni di hash:

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

- ***Open Addressing***

si utilizza una sequenza di scansioni:
m = |T| = dimensione tabella
modicata funzione hash: h: {0, … ,|U|-1} * {0, … , m-1} ⟶ {0, … , m-1}
due input: x.key e i-esimo tentativo
se h(x.key, i) è occupata → provo h(x.key, i+1)
< h(x.key, 0), … , h(x.key, m-1) > → data x.key genero permutazione di <0, … , m-1> (no ripetizioni)

- **IRC(int[] T, function h, element x)**: T(n) = O(m)
  - se vale hashing uniforme semplice ⇒ IRC: Tm(n) = θ(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 ≠ 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 <0, … , m-1>
- uniformità: tutte le m! possibili permutazioni hanno stessa probabilità 1/m!

possibili sequenze di scansioni:

- **lineare** → h(k, i) = (h'(k) + a*i) mod m ⇒ a, m devono essere coprimi
  problemi:
  - *Primary Clustering*: in T si formano blocchi celle contigue occupate
  - anche se x ≠ y, h(x.key, 0) ≠ h(y.key, 0) da un certo punto sequenze coincidono
    - si utilizzano solo le m permutazioni ordinate
- **quadratica** → h(k, i) = (h'(k) + $ic_1 + i^2c_2$) mod 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) ⇒ x e y stessa sequenza (dall'inizio)
    - si utilizzano al massimo m sequenze
- **doppio hashing** → h(k, i) = (h'(k) + i h"(k)) mod m ⇒ h"(k), m coprimi ⇒ m primo
  problema:
  - se h'(x.key) = h'(y.key) e h"(x.key) = h"(y.key) ⇒ 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: Ω(log n), se bilanciato ⇒ θ(log n)
metodi:
preOrder(node x) → visito nodo, sx, dx → T(n) = θ(n)
inOrder(node x) → visito sx, nodo, dx → T(n) = θ(n) ⇒ restituisce array ordinato
postOrder(node x) → visito sx, dx, nodo → T(n) = θ(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 ⇒ y.key < (>) x.key

metodi: → T(n) = O(h) ⇒ Tp(n) = θ(n), se n = h T(n) = O(log n) ⇒ h = log n (BST bilanciato)
BSTSearchMin(T, x) → ricerca minimo: scendo sempre sx (per max a dx) → T(n) = O(h)
BSTSearch(T, x, k) → cerca chiave k → T(n) = O(h)
BSTSuccessor(T, x) → trova minor numero maggiore di x → T(n) = O(h)

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

BSTInsert(T, x) → aggiungo come foglia (posizione dipende grandezza chiave) → T(n) = O(h)
BSTDelete(T, x) → 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 ⇒ non offre nessuna informazione ⇒ si possono costruire molti BST