ESERCIZI SVOLTI

ALGORITMI E STRUTTURE DATI

LAUREA TRIENNALE IN SCIENZE INFORMATICHE

Perego Luca

Università degli studi di Milano-Bicocca

A.A. 2022/2023

Indice

DIVIDE ET IMPERA

Variante intero

Scrivere un algoritmo che utilizzi la tecnica Divide et Impera per risolvere il seguente problema: dati un vettore V e un intero x, determinare quante coppie consecutive di x sono presenti in V.

Scrivere quindi l'equazione di ricorrenza che esprime il tempo di calcolo di tale algoritmo e risolverla con un metodo a piacere.

Pseudocodice

```
function numeroCoppie(V, x, l, r) {

        if ( V.length < 2) {
                return 0
        }

        mid := (l+r)/2

        left := numeroCoppie(V, x, l, mid)
        right := numeroCoppie(V, x, mid+1, r)

        if (V[mid] = V[mid+1]) {
                return left + right + 1
        }  else {
                return left + right
        }
}
```

Equazione di ricorrenza

$$T(n) = \begin{cases} \theta(1) \\ 2T \cdot \left(\frac{n}{2}\right) + \theta(1) \end{cases}$$

Calcolo dei Tempi [Teorema dell'esperto]

$a = 2,\ b = 2$

$\exists \varepsilon > 0$ t.c. $f(n) = \theta\left(n^{\log_b a - \varepsilon}\right) \rightarrow T(n) = \theta\left(n^{\log_b a}\right)$ quindi $T(n) = \theta\left(n^{\log_2 2}\right) = \theta\left(n^1\right) = \theta(n)$

Variante booleano

Scrivere un algoritmo che utilizzi la tecnica Divide et Impera per risolvere il seguente problema: dati due vettori V e W di uguale lunghezza e contenenti numeri interi, stabilire se esiste una posizione tale che V[i] = W[i+1] = V[i+2]

Scrivere quindi l'equazione di ricorrenza che esprime il tempo di calcolo di tale algoritmo e risolverla con un metodo a piacere.

Pseudocodice

```
function consecutivi(V, W, l, r) {

        if ( W.length < 2) {    // Vettori di uguale lunghezza, quello limitante è W
                return false
        }

        mid := (l+r)/2

        left := consecutivi(V, W, l, mid)
        right := consecutivi(V, W, mid+1, r)

        if (V[mid] = W[mid+1] = V[mid+2]) {
                return true
        } else {
                return left or right
        }
}
```

Equazione di ricorrenza

$$T(n) = \begin{cases} \theta(1) \\ 2T \cdot \left(\frac{n}{2}\right) + \theta(1) \end{cases}$$

Calcolo dei Tempi [Teorema dell'esperto]

$a = 2,\ b = 2$

$\exists \varepsilon > 0$ t.c. $f(n) = \theta\left(n^{\log_b a - \varepsilon}\right) \rightarrow T(n) = \theta\left(n^{\log_b a}\right)$ quindi $T(n) = \theta\left(n^{\log_2 2}\right) = \theta\left(n^1\right) = \theta(n)$

Caso particolare

Scrivere un algoritmo che risolva il seguente problema, utilizzando la tecnica Divide et Impera: dato un vettore V contenente n > 0 valori interi, calcolare e restituire il valore:

$$(V[1] \cdot V[2]) \; + \; (V[2] \cdot V[3]) \; + \; \ldots \; + \; (V[n-1] \cdot V[n])$$

Scrivere poi l'equazione di ricorrenza che esprima il tempo di calcolo dell'algoritmo e risolverla utilizzando un metodo a piacere.

Pseudocodice

```
function equazione (V, l, r) {

        if (V.length < 2) {
                return 0
        }

        mid := (l+r)/2

        left := equazione (V, l, mid)
        right := equazione (V, mid+1, r)

        return left + right + (V[mid] * V[mid+1])
}
```

Equazione di ricorrenza

$$T(n) = \begin{cases} \theta(1) \\ 2T \cdot \left(\frac{n}{2}\right) + \; \theta(1) \end{cases}$$

Calcolo dei Tempi [Teorema dell'esperto]

$a \; = 2, \; b \; = 2$

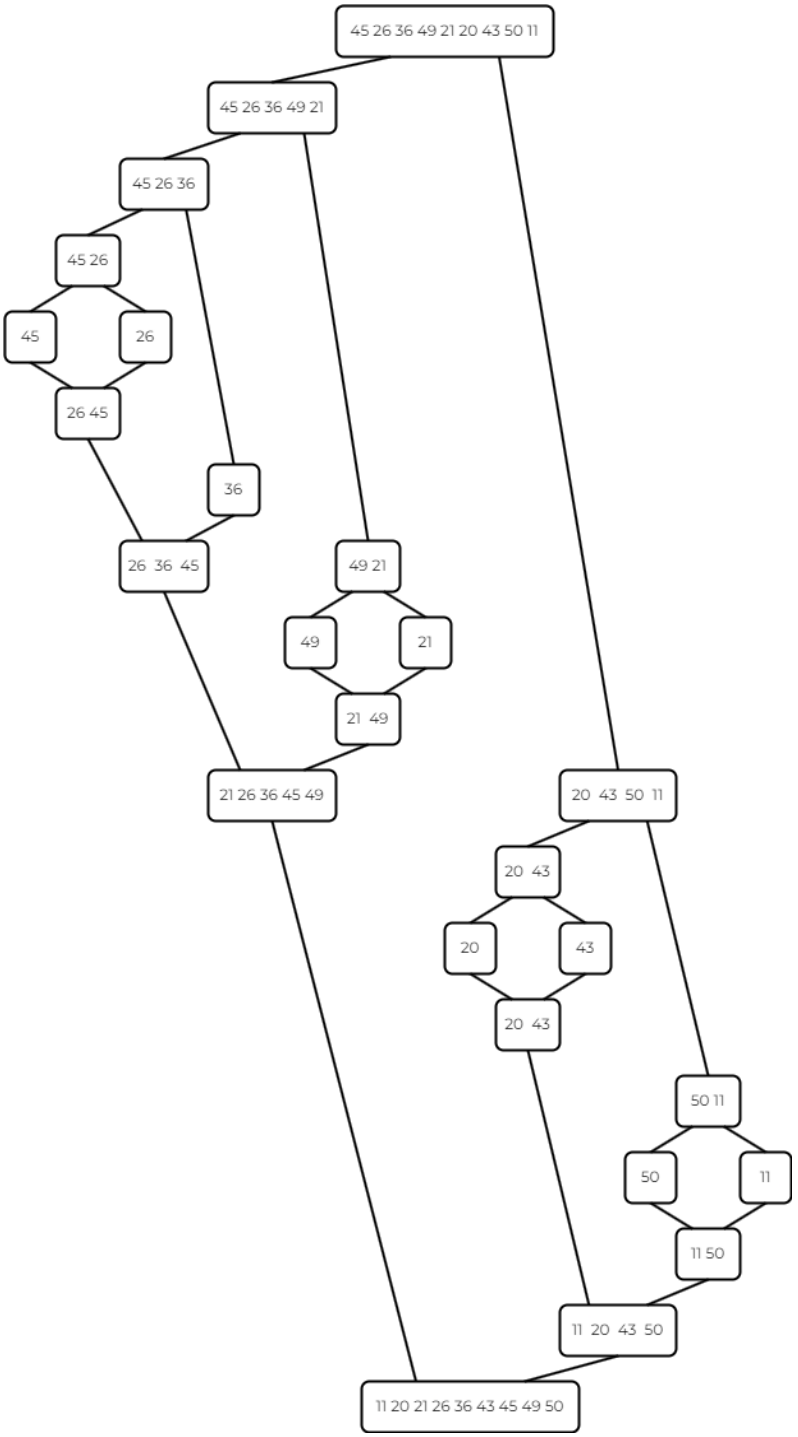
$\exists \varepsilon > 0$ t.c. $f(n) \; = \; \theta\left(n^{\log_b a - \varepsilon}\right) \; \rightarrow \; T(n) \; = \; \theta\left(n^{\log_b a}\right)$ quindi $T(n) \; = \; \theta\left(n^{\log_2 2}\right) = \; \theta\left(n^1\right) = \; \theta(n)$

MERGESORT

Simulare l’esecuzione dell’algoritmo MergeSort sul seguente array illustrando in modo chiaro tutti i passaggi di computazione eseguiti.

[45 26 36 49 21 20 43 50 11]

Simulazione



QUICKSORT

Simulare l'esecuzione del QuickSort sul seguente array indicando chiaramente la procedura Partition, gli scambi che avvengono e il pivot. La scelta su quale tipo di procedura partition usare è libera [Hoare o Lomuto].

[5 2 7 1 3 8 4 6]

Simulazione

🟢= elemento puntato **B** = pivot

Solitamente si sceglie il primo elemento del sottoarray come pivot.
puntatore Sx ≥ Pivot | puntatore Dx < Pivot.

Partizione 0

[**5** 2 7 1 3 8 4 6]
[**5** 2 7 1 3 8 4 6]

[4 2 7 1 3 8 **5** 6]
[4 2 7 1 3 8 **5** 6]
[4 2 7 1 3 8 **5** 6]

[4 2 **5** 1 3 8 7 6]
[4 2 **5** 1 3 8 7 6]
[4 2 **5** 1 3 8 7 6]

[4 2 3 1 **5** 8 7 6]
[4 2 3 1 **5** 8 7 6]
[4 2 3 1 **5** 8 7 6]

Partizione 1 [L]

[**4** 2 3 1]

[1 2 3 **4**]
[1 2 3 **4**]
[1 2 3 **4**]
[1 2 3 **4**]

Partizione 2 [L]

[**1**, 2, 3]
[**1**, 2, 3]
[**1**, 2, 3]

Partizione 3 [L]

[**1**, 2]
[**1**, 2]

Partizione 4 [L]

[1]

Partizione 4 [R]

[2]

Partizione 4

[1 2]

Partizione 3 [R]

[3]

Partizione 3

[1 2 3]

Partizione 2 [R]

[4]

Partizione 2

[1 2 3 4]

Partizione 1 [R]

[**5** 8 7 6]
[**5** 8 7 6]
[**5** 8 7 6]
[**5** 8 7 6]

Partizione 5 [L]

[5]

Partizione 5 [R]

[**8** 7 6]

[6 7 **8**]
[6 7 **8**]
[6 7 **8**]

Partizione 6 [L]

[**6** 7]
[**6** 7]

Partizione 7 [L]

[6]

Partizione 7 [R]

[7]

Partizione 7

[6 7]

Partizione 6 [R]

[8]

Partizione 6

[6 7 8]

Partizione 5

[5 6 7 8]

Partizione 1

[1 2 3 4 5 6 7 8]

HEAP

Build Heap [Max]

Simulare l'esecuzione della funzione BuildHeap sul seguente array:

[63, 41, 51, 67, 45, 46, 11, 70, 62]

Riporta lo stato dell'array dopo ogni operazione di scambio tra gli elementi e illustrando i passaggi. L'array può essere rappresentato in forma di albero se lo si ritiene opportuno, indicando gli indici degli elementi.

Nodo = i
Figlio sx = 2i
Figlio dx = 2i+1

[63, 41, 51, 67, 45, 46, 11, 70, 62]
[63, 41, 51, 67, 45, 46, 11, 70, 62]
[63, 41, 51, 67, 45, 46, 11, 70, 62]
[63, 41, 51, 67, 45, 46, 11, 70, 62]
[63, 41, 51, 67, 45, 46, 11, 70, 62]
[63, 41, 51, 67, 45, 46, 11, 70, 62]

Scambio

[63, 41, 51, 70, 45, 46, 11, 67, 62]
[63, 41, 51, 70, 45, 46, 11, 67, 62]
[63, 41, 51, 70, 45, 46, 11, 67, 62]
[63, 41, 51, 70, 45, 46, 11, 67, 62]
[63, 41, 51, 70, 45, 46, 11, 67, 62]
[63, 41, 51, 70, 45, 46, 11, 67, 62]
[63, 41, 51, 70, 45, 46, 11, 67, 62]
[63, 41, 51, 70, 45, 46, 11, 67, 62]

Scambio

[63, 70, 51, 41, 45, 46, 11, 67, 62]
[63, 70, 51, 41, 45, 46, 11, 67, 62]

Scambio

[63, 70, 51, 67, 45, 46, 11, 41, 62]
[63, 70, 51, 67, 45, 46, 11, 41, 62]
[63, 70, 51, 67, 45, 46, 11, 41, 62]
[63, 70, 51, 67, 45, 46, 11, 41, 62]
[63, 70, 51, 67, 45, 46, 11, 41, 62]
[63, 70, 51, 67, 45, 46, 11, 41, 62]
[63, 70, 51, 67, 45, 46, 11, 41, 62]

Scambio

[63, 70, 51, 67, 45, 46, 11, 41, 62]
[63, 70, 51, 67, 45, 46, 11, 41, 62]

Scambio

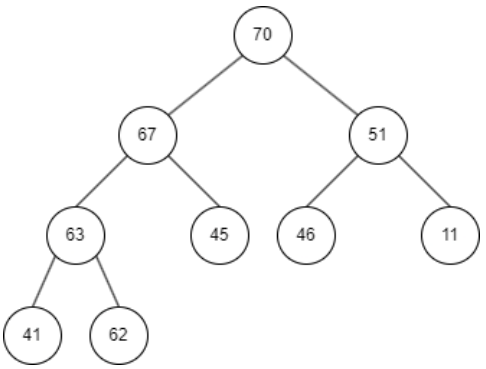
[70, 63, 51, 67, 45, 46, 11, 41, 62]
[70, 63, 51, 67, 45, 46, 11, 41, 62]

Scambio

[70, 67, 51, 63, 45, 46, 11, 41, 62]
[70, 67, 51, 63, 45, 46, 11, 41, 62]
[70, 67, 51, 63, 45, 46, 11, 41, 62]
[70, 67, 51, 63, 45, 46, 11, 41, 62]
[70, 67, 51, 63, 45, 46, 11, 41, 62]

Max Heap Costruito

[70, 67, 51, 63, 45, 46, 11, 41, 62]



HeapSort

Simulare l'esecuzione dell'algoritmo HeapSort sul seguente array:

[7, 1, 3, 2, 9, 8]

Illustra i vari passaggi, mostrando lo stato dell'array dopo ogni operazione di scambio. In particolare, si mostrino gli effetti della funzione BuildHeap utilizzata per formare un heap e successivamente si mostrino gli scambi effettuati dall'algoritmo di ordinamento stesso.
Se lo si ritiene opportuno, l'array può essere rappresentato sotto forma di albero, indicando chiaramente gli indici degli elementi.

A = {}

BuildHeap

[7, 1, 3, 2, 9, 8]
[7, 1, 3, 2, 9, 8]
[7, 1, 3, 2, 9, 8]
[7, 1, 3, 2, 9, 8]

Scambio

[7, 1, 8, 2, 9, 3]
[7, 1, 8, 2, 9, 3]
[7, 1, 8, 2, 9, 3]
[7, 1, 8, 2, 9, 3]
[7, 1, 8, 2, 9, 3]
[7, 1, 8, 2, 9, 3]

Scambio

[7, 9, 8, 2, 1, 3]
[7, 1, 8, 2, 9, 3]
[7, 1, 8, 2, 9, 3]
[7, 1, 8, 2, 9, 3]
[7, 1, 8, 2, 9, 3]

Scambio

[7, 9, 8, 2, 1, 3]
[7, 9, 8, 2, 1, 3]
[7, 9, 8, 2, 1, 3]
[7, 9, 8, 2, 1, 3]
[7, 9, 8, 2, 1, 3]
[7, 9, 8, 2, 1, 3]

Scambio

[9, 7, 8, 2, 1, 3]
[9, 7, 8, 2, 1, 3]
[9, 7, 8, 2, 1, 3]
[9, 7, 8, 2, 1, 3]

A = {9}

Nuovo Heap

[3, 7, 8, 2, 1]

BuildHeap

[3, 7, 8, 2, 1]
[3, 7, 8, 2, 1]
[3, 7, 8, 2, 1]
[3, 7, 8, 2, 1]
[3, 7, 8, 2, 1]

Scambio

[8, 7, 3, 2, 1]
[8, 7, 3, 2, 1]
[8, 7, 3, 2, 1]
[8, 7, 3, 2, 1]
[8, 7, 3, 2, 1]

A = {9, 8}

Nuovo Heap

[1, 7, 3, 2]

BuildHeap

[1, 7, 3, 2]
[1, 7, 3, 2]
[1, 7, 3, 2]
[1, 7, 3, 2]

Scambio

[7, 1, 3, 2]
[7, 1, 3, 2]

Scambio

[7, 2, 3, 1]
[7, 2, 3, 1]
[7, 2, 3, 1]
[7, 2, 3, 1]
[7, 2, 3, 1]
[7, 2, 3, 1]

A = {9, 8, 7}

Nuovo Heap

[1, 2, 3]

BuildHeap

[1, 2, 3]
[1, 2, 3]
[1, 2, 3]

Scambio

[3, 2, 1]
[3, 2, 1]
[3, 2, 1]
[3, 2, 1]
[3, 2, 1]

A = {9, 8, 7, 3}

Nuovo Heap

[1, 2]

BuildHeap

[1, 2]
[1, 2]

Scambio

[2, 1]
[2, 1]
[2, 1]
[2, 1]

A = {9, 8, 7, 3, 2}

Heap.size < 2

A = {9, 8, 7, 3, 2, 1}

PILE

Definizione

Struttura dati che implementa la politica di gestione LIFO (Last In First Out)

Procedure & Attributi

	Array	Lista
Push	$\theta(1)$	$\theta(1)$
Pop	$\theta(1)$	$\theta(1)$
StackEmpty	$\theta(1)$	$\theta(1)$
Top	$\theta(1)$	$\theta(1)$

Esercizio

Scrivere un algoritmo iterativo che risolva il seguente problema, valutandone poi i tempi di esecuzione nel caso migliore e peggiore, specificando quando questi si verificano.

Siano date 2 pile P e Q contenenti numeri naturali ordinati in ordine non crescente (l'elemento in cima è il più grande) ed una pila vuota R.
Inserire nella pila R tutti i numeri presenti in P e Q in modo che appaiano in ordine non crescente.

```
Function transferPila (P, Q, R) {

        while (!stackEmpty(P) and !stackEmpty(Q)) {

                if (P.top != NIL and P.top ≥ Q.top) {
                        push(T, P.pop)
                }
                if (Q.top != NIL and Q.top ≥ P.top) {
                        push(T, Q.pop)
                }
        }

        while (!T.stackempty) {
                push(R, T.pop)
        }
}
```

CODE

Definizione

Struttura dati che implementa la politica di gestione FIFO (First In First Out)

Procedure & Attributi

	Array	Lista
Enqueue	$\theta(1)$	$\theta(1)$
Dequeue	$\theta(1)$	$\theta(1)$
QueueEmpty	$\theta(1)$	$\theta(1)$
Top	$\theta(1)$	$\theta(1)$

Esercizio

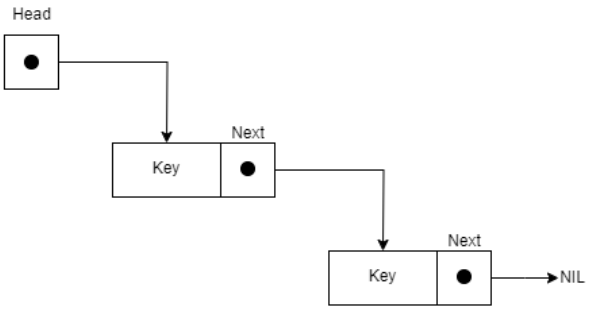
Si considerino due code C1 e C2 contenenti lo stesso numero di valori interi. Fornire un algoritmo iterativo che, dati C1, C2 e un valore x in input, restituisce una coda C3 contenente tutti i valori diversi da x alternando tra elementi di C1 e di C2. Valutare poi i tempi di calcolo nel caso migliore e peggiore.

```
function listaAlternata (C1, C2, x) {
        isC1Last := false

        While (!queueEmpty(C1) and !queueEmpty(C2)) {

                if (C1.top != x) {
                        if (!isC1Last) {
                                enqueue(C3, dequeue(C1))
                                isC1Last := true
                        }
                } else dequeue(C1)

                if (C2.top != x) {
                        if (isC1Last) {
                                enqueue(C3, dequeue(C2))
                                isC1Last := false
                        }
                } else dequeue(C2)

                isC1Last := isC1Last and queueEmpty(C1)
        }
}
```

LISTE SEMPLICI

Definizione

Struttura dati dinamica in cui ogni elemento punta al successivo



Esercizio

Scrivere un algoritmo che risolva il seguente problema e valutare i tempi di esecuzione nel caso migliore e peggiore.

Siano L1 e L2 due liste semplici contenenti numeri interi ordinati in ordine crescente, con campi key e next per indicarne il valore e il successivo.
Creare una lista L3 contenente tutti gli elementi di L1 e L2 in ordine decrescente.

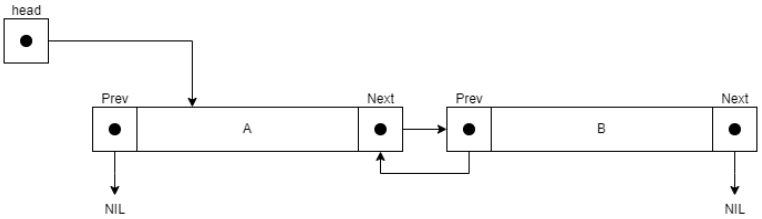
```
function listaDecrescente (L1, L2) {

        while (L1.head != NIL or L2.head != NIL) {

                if (L1.head ≥ L2.head) {
                        listInsert(L1, L3)
                }

                if (L2.head ≥ L1.head) {
                        listInsert(L2, L3)
                }
        }

        while (L1.head != NIL) {
                listInsert(L1, L3)
        }

        while (L2.head != NIL) {
                listInser(L2, L3)
        }
}
```

```
function listInsert (k, L) {

        k.next = L.head
        L.head = k

}
```

LISTE DOPPIAMENTE CONCATENATE

Definizione

Struttura dati dinamica in cui ogni elemento punta al successivo e al predecessore.



Esercizio

???

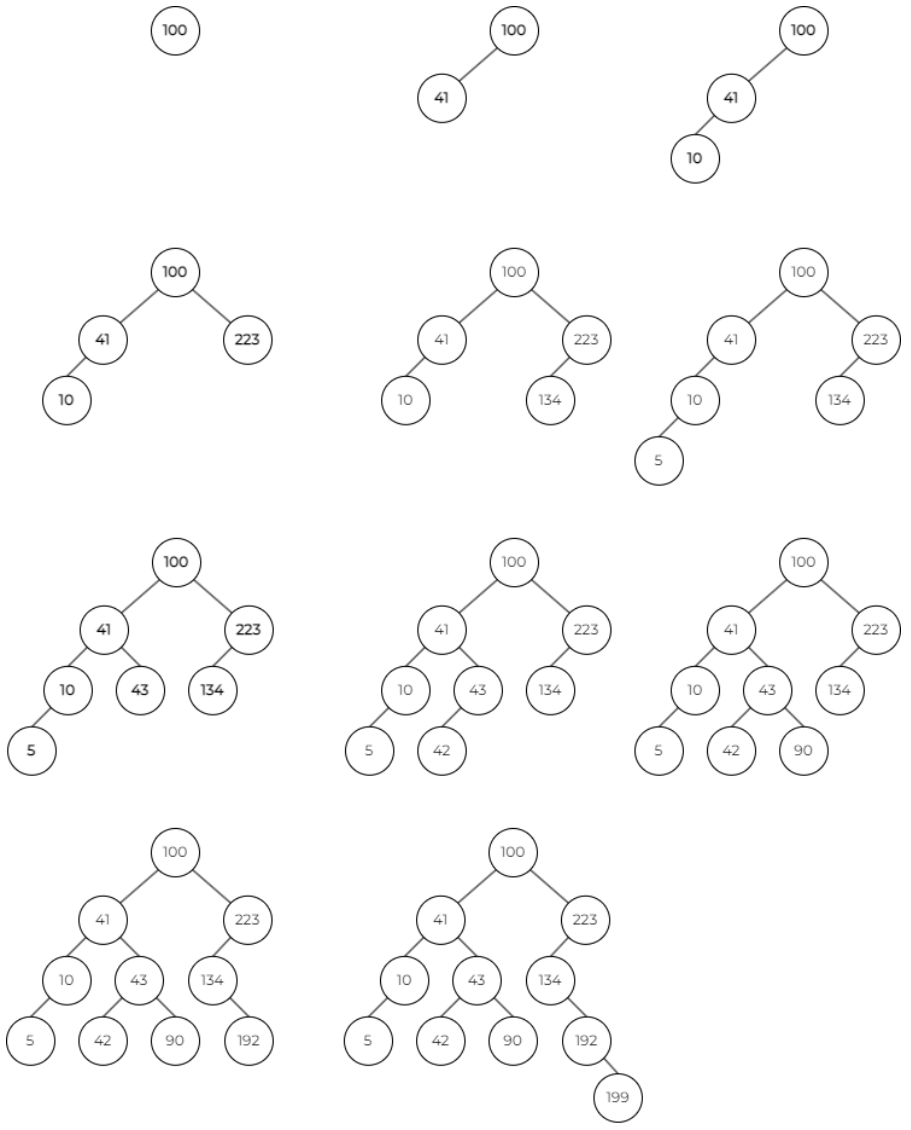
BST

Costruire un albero binario di ricerca inserendo successivamente i seguenti valori:

[100, 41, 10, 223, 134, 5, 43, 42, 90, 192]

Disegnare l'albero ottenuto dopo ogni inserimento.
Successivamente indicare la sequenza di interi effettuando effettuando la visita usando le 3 tecniche: pre-order, in-order, post-order.
Infine eliminare dall'albero i nodi con le chiavi 42, 10 e 100, mostrando i passaggi di eliminazione per ogni chiave, inoltre spiegare brevemente ogni processo di eliminazione.

Costruzione del BST



Visite

Pre-order

Ordine: padre → sinistra → destra
BST Pre-order: [100, 41, 10, 5, 43, 42, 90, 223, 134, 192, 199]

In-order

Ordine: sinistra → padre → destra
BST In-order: [5, 10, 41, 42, 43, 90, 100, 134, 192, 199, 223]

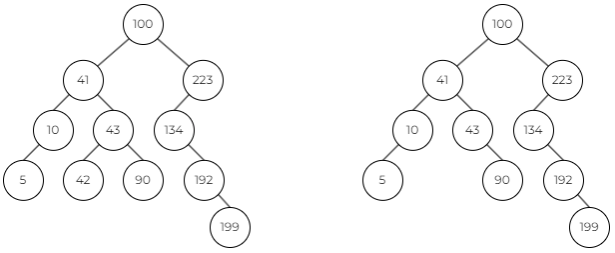
Post-order

Ordine: sinistra → destra → padre
BST Post-order: [5, 10, 42, 90, 43, 41, 199, 192, 134, 223, 100]

Eliminazione di nodi

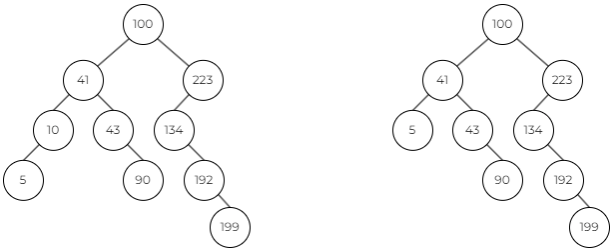
Nodo con 0 figli

Il nodo con key 42 non ha figli, quindi può essere rimosso.



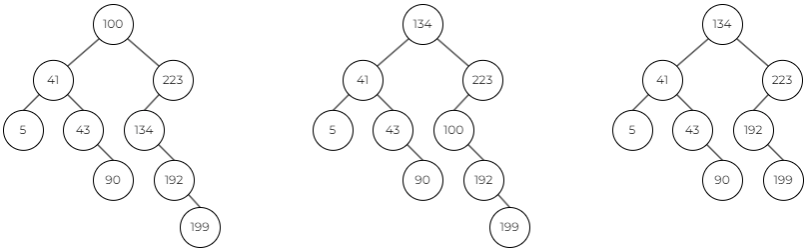
Nodo con 1 figlio

Il nodo con key 10 ha 1 figlio, quindi bisogna eseguire l'operazione di contrazione del nodo.



Nodo con 2 figli

Il nodo con key 100 ha 2 figli, quindi va sostituito con il minimo del sottoalbero destro e poi eliminato effettuando una contrazione se necessaria.

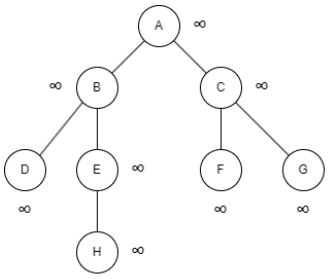


GRAFI

Colori

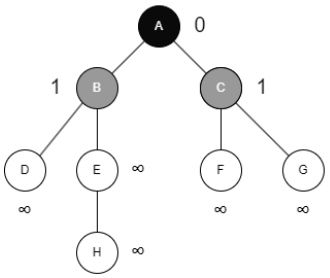
Bianco: vertice non scoperto
Grigio: vertice scoperto
Nero: vertice la cui lista di adiacenza è stata visitata (tutti i vicini almeno grigi)

Visita in Ampiezza [BFS]



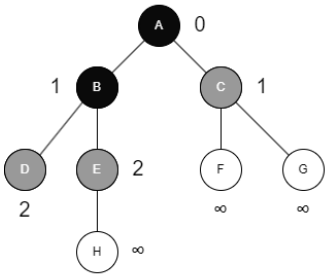
Nodi visitati = [ ]

Coda = [ ]



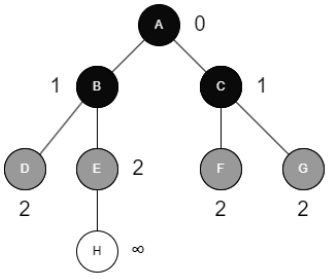
Nodi visitati = [A]

Coda = [B, C]



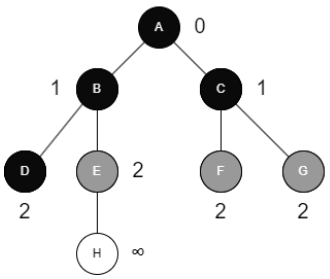
Nodi visitati = [A, B]

Coda = [C, D, E]



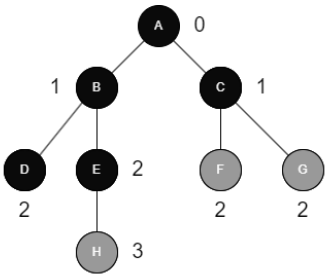
Nodi visitati = [A, B, C]

Coda = [D, E, F, G]



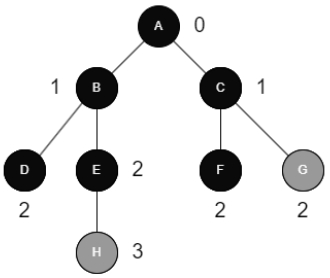
Nodi visitati = [A, B, C, D]

Coda = [E, F, G]



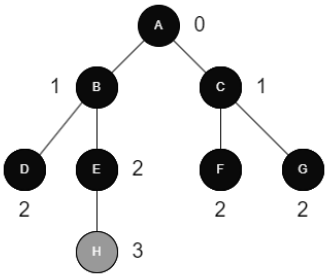
Nodi visitati = [A, B, C, D, E]

Coda = [F, G, H]



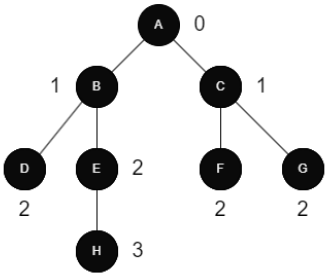
Nodi visitati = [A, B, C, D, E, F]

Coda = [G, H]



Nodi visitati = [A, B, C, D, E, F, G]

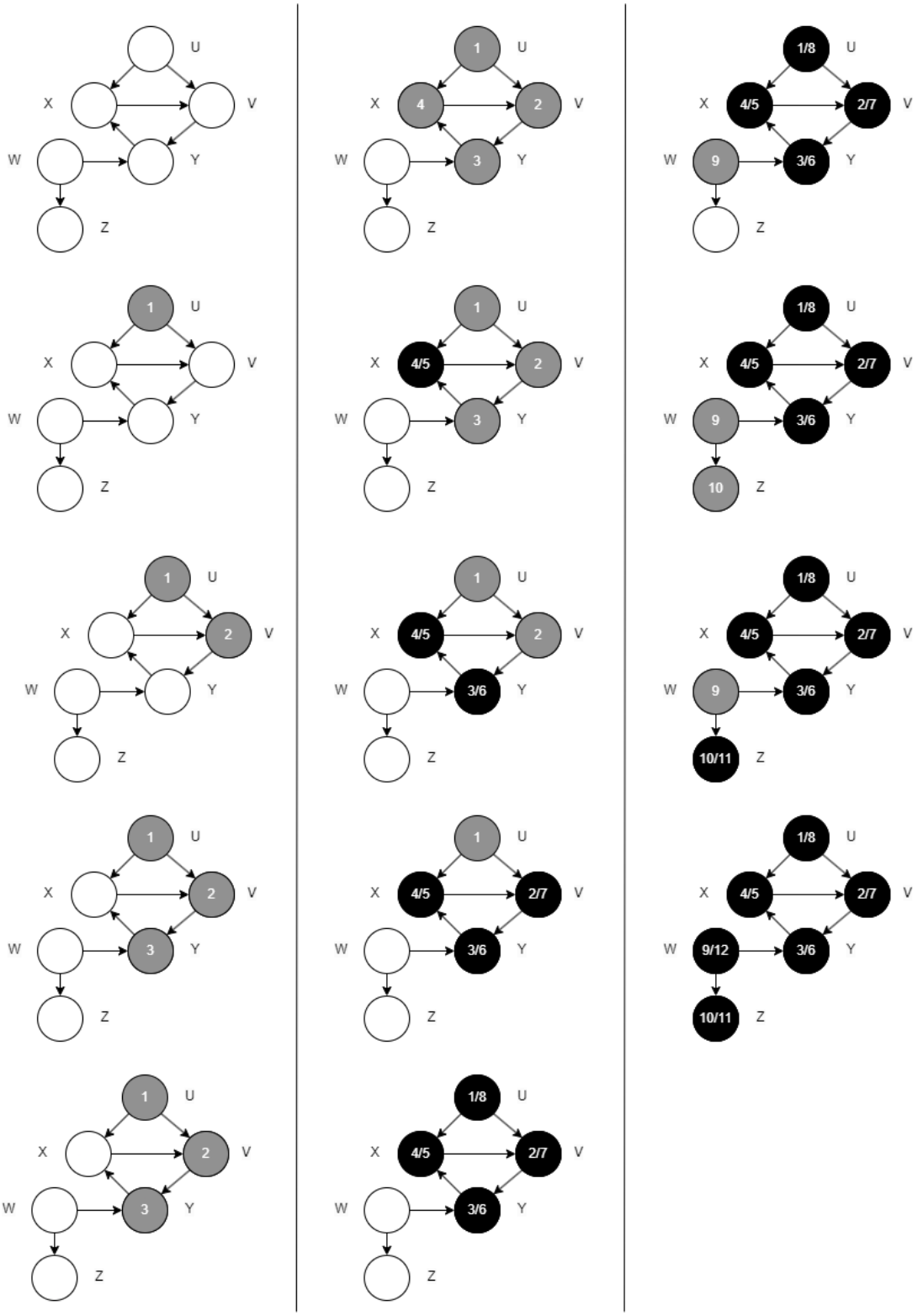
Coda = [H]



Nodi visitati = [A, B, C, D, E, F, G, H]

Coda = [ ]

Visita in Profondità [DFS]



TEORIA FONDAMENTALE

Teorema dell'esperto

Enunciato

Sia $T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$ $\quad a \geq 1,\ b > 1,\ f(n)\ asin.\ pos.$

1. se $\exists\ \varepsilon > 0\ t.c.\ f(n) = 0\left(n^{log_b a - \varepsilon}\right)$ allora $T(n) = \theta\left(n^{log_b a}\right)$
2. se $f(n) = \theta\left(n^{log_b a}\right)$ allora $T(n) = \theta\left(n^{log_b a} \cdot log\ n\right)$
3. se $\exists \varepsilon > 0\ t.c.\ f(n) = \Omega\left(n^{log_b a + \varepsilon}\right)$ e se $\exists\ c < 1\ t.c.\ a \cdot f(\frac{n}{b}) \leq c \cdot f(n)$ $\forall n > n_0$ allora $T(n) = \theta(f(n))$

Caratteristiche e Tempi degli Algoritmi di Ordinamento

SelectionSort

Caso migliore e caso peggiore sono asintoticamente uguali. $T(n) = \theta\left(n^2\right)$

Stable: ❌ In-Place: ✅

InsertionSort

Caso migliore: il vettore è già ordinato $T(n) = \Omega(n)$

Caso peggiore: il vettore è ordinato in senso opposto $T(n) = 0\left(n^2\right)$

Stable: ✅ In-Place: ✅

MergeSort

a = 2	Numero di volte che viene chiamato il metodo ricorsivo
b = 2	Numero di porzioni in cui viene diviso l'array [mid = l+r / 2]
$f(n)$ = $\theta(n)$	Righe di codice che non c'entrano con la ricorsione

quindi applico il secondo caso del teorema dell'esperto:

$\theta(n) = \theta(n^{log_2 2}) \rightarrow \theta(n) = \theta(n) \rightarrow T(n) = \theta(n^{log_2 2} \cdot log\ n) \rightarrow T(n) = \theta(n \cdot log\ n)$

Stable: ✅ In-Place: ❌

QuickSort

Caso migliore: i due sottovettori hanno dimensione $\frac{n}{2}$

$$T(n) = \theta(n \cdot log\, n) \rightarrow T(n \cdot log\, n) = \Omega(n \cdot log\, n)$$

Caso peggiore: il primo sottovettore ha lunghezza n-1 e il secondo ha lunghezza 1

$$T(n) = \theta(n^2) \rightarrow T(n^2) = 0(n^2)$$

Stable: ❌ In-Place: ✅

CountingSort

Caso migliore e caso peggiore sono asintoticamente uguali.

$$T\,(n,\, k) \;=\; \theta(n \,+\, k) \qquad con\;\; k \;=\; O(n) \rightarrow T(n) \;=\; \theta(n)$$

Stable: ✅ In-Place: ❌

HeapSort

Caso migliore e caso peggiore sono asintoticamente uguali.

$$T(n) \;=\; O(n) \,+\, O(n\, log\, n) \;=\; O(n\, log\, n)$$

Stable: ❌ In-Place: ✅

Ricerca Dicotomica

Algoritmo che, dato un vettore ordinato V e un elemento x, restituisce la posizione di quest'ultimo.

Pseudocodice

```
Ricerca_Dicotomica (V, x, l, r) {

        if (r < l) {
                return false
        }
        if (l = r) {
                return r
        }

        mid := floor(\frac{l + r}{2})

        if (x > V[mid) {
                return Ricerca_Dicotomica(V, x, mid+1, r)
        } else {
                return Ricerca_Dicotomica(V, x, l, mid)
        }
}
```

Tempi di Calcolo

Caso migliore e caso peggiore sono asintoticamente uguali.

$$T(n) = \Theta(log_2 n)$$

Pseudocodice delle Strutture Dati

Liste Semplici

List Insert

```
function listInsert (L, k) {

        k.next = L.head
        L.head = k

}
```

List Delete

```
listDelete (L, x) {

        if (L.head = x) {
                L.head = x.next
        } else {

                y := L.head

                while (y.next != x) {
                        y = y.next
                }
                y.next = x.next
        }
}
```

Liste Doppiamente Concatenate

DL List Insert

```
DLListInsert (L, k) {

        x.prev := NIL
        x.next := L.head

        if (L.head != NIL) {
                L.head.prev := x
        }

        L.head := x

}
```

DL List Delete

```
DLListDelete (L, x) {

        if (x.prev != NIL) {
                x.prev.next := x.next
        } else {
                L.head := x.next
        }
}
```

Pile

Stack Search

```
function stackSearch (S, k) {

	found := false
	S2 := stack vuoto

	while (!stackEmpty(S) and !found) {

		x := pop(S)
		push (S2, x)

		if (x = k) {
			found := true
		}
	}

	while (!stackEmpty(S2)) {
		x := pop(S2)
		push(S, x)
	}
	return found
}
```

Stack Delete

```
function stackDelete (S, k) {

	found := false
	S2 := stack vuoto

	while (!stackEmpty(S) and !found) {

		x := pop(S)

		if (x = k) {
			found := true
		} else {
			push(S2, x)
		}
	}
	while (!stackEmpty(S2)) {
		x := pop(S2)
		push(S, x)
	}
	return found
}
```

Stack Sorted Insert

```
function stackSortedInsert (S, k) {

        S2 := stack vuoto

        while (!stackEmpty(S) and top(S) < k) {
                x := pop(S)
                push(S2, x)
        }

        push(S2, k)

        while (!stackEmpty(S2)) {
                x := pop(S2)
                push(S, x)
        }
}
```

Code

Queue Search

```
QueueSearch (Q, k) {

        found := false
        Q2 := coda vuota

        while (!queueEmpty(Q)) {
                x := dequeue(Q)
                Enqueue(Q, x)

                if (x = k) {
                        found := true
                }
        }

        while (!queueEmpty(Q2) {
                enqueue(Q, dequeue(Q2))
        }
}
```

Queue Sorted Insert

```
QueueSortedInsert(Q, k) {

        Q2 := coda vuota

        while (!queueEmpty(Q) and head(Q) < k) {
                enqueue(Q2, dequeue(Q)
        }
        enqueue(Q2, k)

        while (!queueEmpty(Q) {
                enqueue(Q2, dequeue(Q))
        }
        while (!queueEmpty(Q2) {
                enqueue(Q, dequeue(Q2))
        }
}
```