

ALGORITMI E STRUTTURE DATI

Preparazione esame

(Ducange, Vaglini, Alfeo)

INDICE

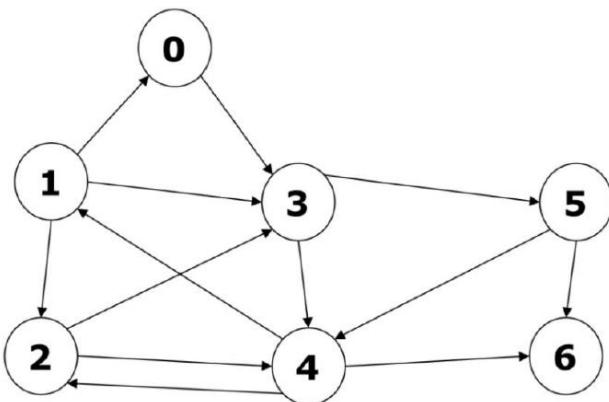
| | |
|----------------------------------|--------|
| Pretest - Quiz Domande..... | pag.1 |
| Pretest - Quiz Risposte..... | pag.5 |
| Ducange (Pratica) Domande..... | pag.7 |
| ABR Alfeo (Domande) - LAB..... | pag.15 |
| ABR Alfeo (Risposte) - LAB..... | pag.17 |
| HEAP Alfeo (Domande) - LAB..... | pag.31 |
| HEAP Alfeo (Risposte) - LAB..... | pag.34 |
| Complessità..... | pag.38 |
| Iterazione - Ricorsione..... | pag.44 |
| Linear Search..... | pag.48 |
| BinSearch..... | pag.52 |
| Exchange..... | pag.54 |
| Selection Sort (Iterativa)..... | pag.55 |
| Selection Sort (Ricorsiva)..... | pag.60 |
| BubbleSort..... | pag.65 |
| QuickSort..... | pag.70 |
| MergeSort (Array)..... | pag.78 |
| MergeSort (Liste)..... | pag.80 |
| Insertion Sort..... | pag.91 |

INDICE

| | |
|-------------------------------------|----------------|
| Tipi di Alberi..... | pag.93 |
| Class BinTree (Albero Binario)..... | pag.96 |
| Albero Generico..... | pag.114 |
| ABR..... | pag.124 |
| Class Heap..... | pag.135 |
| HeapSort..... | pag.146 |
| Counting Sort..... | pag.157 |
| Radix Sort..... | pag.165 |
| Hash..... | pag.168 |
| PLSC..... | pag.173 |
| Algoritmo di Huffman..... | pag.187 |
| Grafi Orientati..... | pag.195 |
| Algoritmo di Dijkstra..... | pag.199 |
| Algoritmi Non Deterministici..... | pag.204 |
| Recap Complessità..... | pag.206 |
| Problemi..... | pag.208 |

PRETEST-QUIZ DOMANDE

-
- 1 - Date le seguenti coppie di stringhe (xyzzy – xxyzx), (xyzzyx - xxyzxy) quanto sono lunghe le PLSC?
- 2 - Mostrare l'output al primo e al secondo step del **radix sort** (a, b, c, d, e) sulla lista [ace ceb bec abc eba bba]
- 3 - Mostrare l'output al secondo step del **radix sort** sulla lista [190, 051, 054, 207, 088, 010]
- 4 - **Visita in profondità** di un grafico salvato in lista di adiacenza



- 5 - Qual è la complessità dell'algoritmo di Dijkstra?
- 6 - A quale categoria di problemi appartiene la soddisfabilità della formula logica nel I ordine (SAT-I)?
RISPOSTE: P, NP, NP-Completi, N.A.
- 7 - Un problema decisionale si definisce appartenente a NP:
- 8 - Un problema decisionale si definisce appartenente a P:
- 9 - Un problema si definisce appartenente a P:
- 10 - Il mergeSort può avere complessità peggiore di $O(n^2)$?
- 11 - Date le seguenti lettere e le relative frequenze {(a, 45) (b, 13) (c, 12) (d, 16) (e, 9) (f, 5)} determinare la loro codifica binaria utilizzando l'**algoritmo di Huffman**
- 12 - Dato le seguenti lettere con la relativa frequenza, indicare la codifica binaria corretta utilizzando l'**algoritmo di Huffman** (A 44, B 53, C 12, D 10, E 15, G 35)
- 13 - Sono più efficienti gli algoritmi ricorsivi o iterativi?
- 14 - Cosa si considera per valutare la complessità di un algoritmo
RISPOSTE: NUMERO ISTRUZIONI, MEDIA TEMPO ESECUZIONI AL VARIARE DI n, SPAZIO DI MEMORIA, N.A.

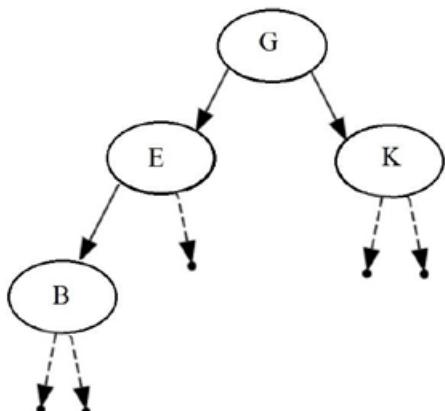
PRETEST-QUIZ DOMANDE

15 - Per la ricerca nell'ABR quale delle seguenti è vera:

- a) Caso peggiore quando l'albero è completamente bilanciato
- b) Caso migliore quando ricerco dalla radice
- c) Caso migliore quando ricerco dalla foglia
- d) Caso migliore quando l'albero è completamente bilanciato

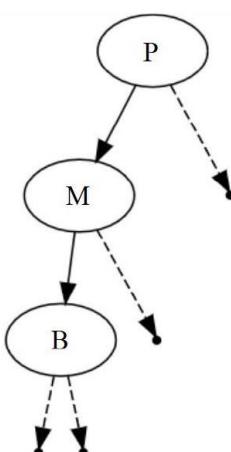
16 - Dato il seguente albero definire la risposta corretta

RISPOSTE: NON È ABR, È COMPLETAMENTE BILANCIATO, HA UNA SOLA FOGLIA, N.A.

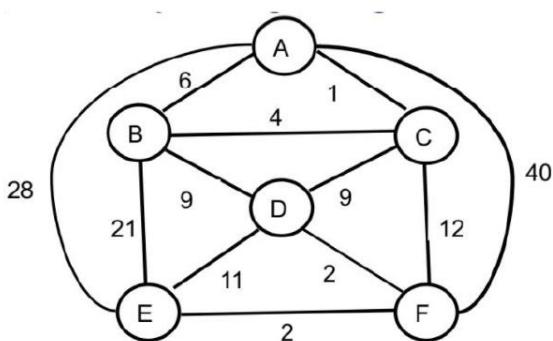


17 - Qual è la definizione corretta per il seguente albero?

(nodi in ordine alfabetico con A<Z, per chiarire la differenza tra figlio sinistro e destro, sono riportati anche i figli vuoti dei nodi)

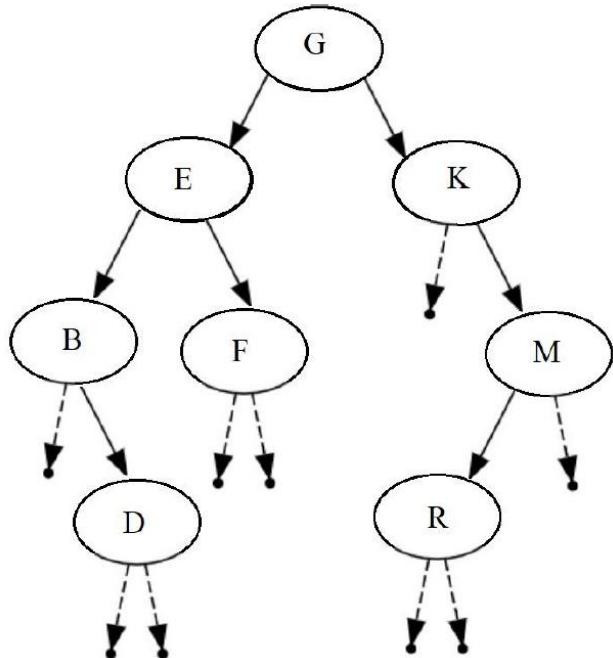


18 - Dato il seguente grafo, indicare la lunghezza del cammino minimo tra A e D

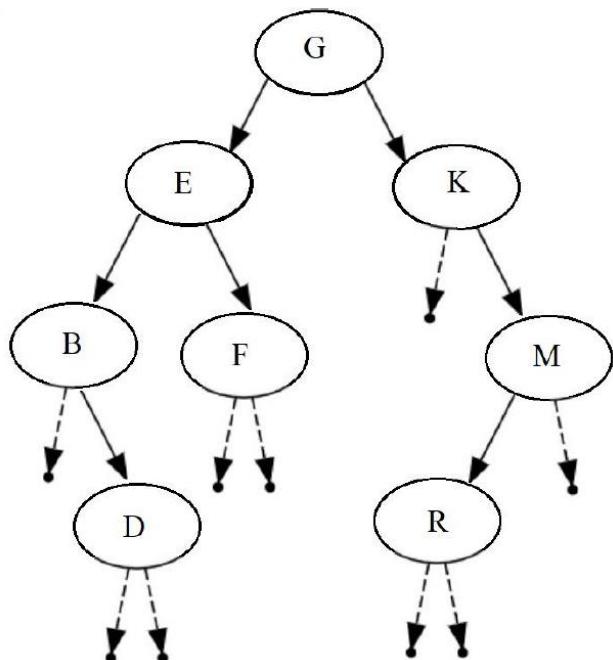


PRETEST-QUIZ DOMANDE

19 - Quale definizione è corretta per il seguente albero? (nodi in ordine alfabetico con A<Z, per chiarire la differenza tra figlio sinistro e destro, sono riportati anche i figli vuoti dei nodi)
[le opzioni: è degenere, è bilanciato, non è un vero albero di ricerca, nessuna delle altre]



20 - Fare la visita anticipata del seguente albero (pre-order)
(nodi in ordine alfabetico con A<Z, per chiarire la differenza tra figlio sinistro e destro, sono riportati anche i figli vuoti dei nodi)



PRETEST-QUIZ DOMANDE

21 - Dato il seguente pezzo di codice indicare l'output

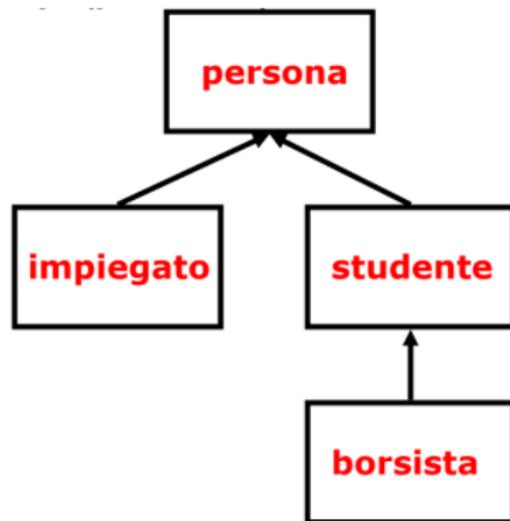
```
template<class tipo>
tipo maxT(tipo x, tipo y) {
    static int a; a++; cout << a << endl;
    return (x>y) ? x : y;
}
```

```
void main(){
    cout << maxT<int>(101,102) << endl;
    cout << maxT<int>(101,102)<< endl;
    cout << maxT<double>(101,102) << endl;
}
```

22- Dato il seguente pezzo di codice di classi derivate quale dei seguenti assegnamenti è errato

[le opzioni: pp=ps, N.A., pp=&b, ps=&p]

```
void main(){
    studente s; persona p; borsista b;
    studente* ps; persona * pp;
```



23 - Da cosa dipende la complessità di un algoritmo?

[le opzioni: dimensione dei dati, media delle iterazioni, spazio di memoria, nessuna delle altre]

PRETEST-QUIZ RISPOSTE

1 → 1° delle coppie 3 (xyz) - 2° delle coppie 4 (xyzx) oppure (xyzx)

2 → Primo step: eba bba cebbec abcace – Secondo step eba bba abcace cebbec

3 → 207, 010, 051, 054, 088, 190

4 → 0,3,4,1,2,6,5

5 → $O(n \log n + m \log n)$

6 → N.A. (SAT nella logica del l'ordine non è risolvibile con un algoritmo)

7 → Se esiste un algoritmo Nondeterministico che verifica una sua soluzione in tempo polinomiale

8 → Se esiste un algoritmo deterministico che verifica una sua soluzione in tempo polinomiale

9 → Quando il problema è decisionale ed è risolvibile in tempo polinomiale con un algoritmo deterministico

10 → No, Il MergeSort è sempre $O(n \log n)$

11 → (a, 0) (b, 101) (c, 100) (d, 111) (e, 1101) (f, 1100)

12 → (A, 10) (B, 11) (C, 0111) (E, 010) (D, 0110) (G, 00)

13 → Dipende dalla situazione, per esempio l'algoritmo per trovare il numero dalla serie di Fibonacci in modo iterativo può essere migliore di uno in modo ricorsivo in termini di complessità $O(n)$ vs $O(2^n)$

14 → N.A. (La complessità in tempo di esecuzione di un programma deve essere analizzata nel suo comportamento asintotico al crescere della dimensione che dipende dai dati presi in ingresso)

15 → d) perché la complessità è $O(\log n)$, quindi caso a) è escluso. Il caso b) non ha senso, perché qualsiasi ricerca parte dalla radice. Il caso c) nemmeno questo ha senso perché non si parte dalla foglia e non si va a ritroso in un albero

PRETEST-QUIZ RISPOSTE

16 → N.A. l'albero è un ABR (a SX della radice c'è sempre il nodo minore e a DX quello maggiore)

17 → Si definisce albero binario di ricerca DEGENERE (con tanti livelli, in questo caso addossati tutti a SX)

18 → 10 (ACD)

19 → N.A. è un albero generico

20 → G E B D F K M R

21 → 1102; 2 102; 1 102

22 → ps=&p assegnamento non ammissibile (studente è tipo derivato)

23 → N.A. la complessità di un algoritmo dipende dalla dimensione dei dati di un problema la cui risoluzione è rappresentata in termini di spazio e tempo

DUCANGE (PRATICA) DOMANDE

PARTE TEORICA

1. Che cosa è una lista (RICORSIVA)?
2. Che cosa è una relazione di ricorrenza?
3. HeapSort quale caratteristica sfrutta?
Spiega il corpo principale dell'HeapSort
4. Qual è la complessità dell'HeapSort?
5. A che cosa serve il MergeSort?
Spiega il corpo principale del MergeSort
6. Quale sarà l'output del MergeSort?
7. Complessità MergeSort?
8. Come si chiama il paradigma che utilizziamo nel MergeSort?
9. Metodo Hash: perché e in quale contesto è stato introdotto questo concetto?
10. Qual è la differenza tra un Hash in cui non posso cancellare gli elementi e un Hash in cui posso farlo?
11. Complessità algoritmo della stampa della PLSC
12. Parla dello Heap
13. A che cosa serve l'algoritmo di Dijkstra?
14. “Cammino minimo di che cosa (DIJKSTRA)”
15. Cosa produce in output l'algoritmo di Dijkstra?
16. Spiega lo pseudocodice dell'algoritmo di Dijkstra
17. Qual è il limite inferiore raggiungibile da un algoritmo di ordinamento e quali algoritmi lo raggiungono?
18. Indicare i vantaggi di complessità di questi algoritmi che raggiungono questo limite inferiore, rispetto ad altri algoritmi basati sui confronti
19. Spiega il RadixSort

DUCANGE (PRATICA) DOMANDE

PARTE TEORICA

20. Spiega il codice del CountingSort
21. A che cosa serve l'algoritmo di Huffman?
22. Parlami dell'algoritmo. Che strategie usa? Cosa è lo Heap passato tra i parametri?
Cosa vuol dire costruire un minHeap?
23. Che cosa è la lista di adiacenza in un grafo?

DUCANGE (PRATICA) DOMANDE

“POSSIBILI RISPOSTE” PARTE TEORICA

1. Una lista è una struttura composta da un campo che contiene l'informazione e da un puntatore ad un elemento con le stesse caratteristiche (che sarebbe il successivo)
→ Quindi la lista stessa è una struttura ricorsiva
2. È una relazione che definisce il tempo di esecuzione di qualsiasi programma ricorsivo, in maniera induttiva studiando la funzione in due casi:
 - Caso base → Caso in cui la funzione non effettua nessuna chiamata e termina la ricorsione
 - Caso ricorsivo → Caso in cui la funzione richiama sé stessa su un numero inferiore di elementi, fino a quando non ricade nel caso base
//Possono esistere più di un caso base
3. L'HeapSort è un algoritmo di ordinamento che prende un array riordinandolo in modo crescente così:
 - Prende l'array di dimensione n e lo trasforma in uno heap tramite la funzione **buildHeap**
 - Per n volte esegue l'estrazione della radice scambiando, ogni volta il primo elemento dell'array con quello puntato da last (l'ultimo) e poi, il primo elemento appena scambiato (che era last) viene fatto scendere al posto giusto tramite la funzione **down**
4. La complessità dell'HeapSort è $O(n \log n)$
5. È un algoritmo di ordinamento che ordina una sequenza di elementi presi in ingresso così: (considero una lista s)
 - Divide la lista s in due sottoliste s1 e s2 (bilanciate) tramite la funzione **split**
 - Richiamo la funzione **mergeSort** su entrambe le liste s1 ed s2 per ordinarle
 - Infine, fonderò con la funzione **merge** entrambe le liste s1 ed s2 nella lista s1 che risulterà ordinata
6. La lista s1 ordinata
7. La complessità dell'MergeSort è $O(n \log n)$
8. Si chiama metodo DIVIDE ET IMPERA:
 - Serve per risolvere un problema per un certo insieme di dati.
Lo si affronta applicando l'algoritmo ricorsivamente su sottoinsiemi di dati del problema (dette partizioni) e ricomponendo poi i risultati ottenuti (lavoro di combinazione)

DUCANGE (PRATICA) DOMANDE

“POSSIBILI RISPOSTE” PARTE TEORICA

9. Il metodo Hash è utilizzato per accedere direttamente ad un elemento di un determinato insieme, tramite un indice contenuto in un array nel seguente modo:
 - Si definisce una funzione Hash h che associa ad ogni elemento dell'insieme una posizione nell'array (indirizzo Hash)
 - Se un determinato elemento che sto cercando è presente nell'insieme, esso deve trovarsi nella posizione corrispondente all'indice $h(x)$
10. Un Hash in cui posso cancellare gli elementi lo ottengo tramite accesso *diretto*, dove la funzione Hash conterrà un indice che associerà agli elementi, senza avere bisogno di memorizzarli → Quindi cancellabili
Invece, un Hash in cui non posso cancellare gli elementi ce l'ho quando utilizzo *l'indirizzamento aperto*:
 - In questo metodo vige la legge di scansione lineare che, in fase di inserimento di un elemento nell'array, una volta inserito nella struttura questo non può essere più cancellato
11. La complessità della print(PLSC) è $O(n+m)$
12. Lo Heap è un albero binario quasi bilanciato con le seguenti proprietà:
 - I nodi dell'ultimo livello sono addossati a sinistra
 - In ogni sottoalbero l'etichetta della radice è maggiore o uguale a quella di tutti i suoi discendenti
 - Si può memorizzare in un array indicizzato
 - Se considero l'indice i di un determinato nodo dello Heap, memorizzato nell'array posso trovarne:
 - Il figlio sinistro con la formula $\rightarrow 2i+1$
 - Il figlio destro con la formula $\rightarrow 2i+2$
 - Il Padre con la formula $\rightarrow (i-1)/2$
13. Trova i cammini minimi da un nodo di partenza di un grafo, detto nodo sorgente a tutti gli altri nodi
14. Di un grafo etichettato
15. Una tabella finale dove ho per ogni nodo, tutta la lista dei predecessori e l'equivalente distanza

DUCANGE (PRATICA) DOMANDE

“POSSIBILI RISPOSTE” PARTE TEORICA

16. Lo pseudocodice di Dijkstra funziona così:

- Considero l’insieme dei nodi non sistemati Q pari a tutti gli altri nodi contenuti in N (Q è un minHeap, una specie di coda con priorità)
- Considero un nodo sorgente p0 e per ogni altro nodo p diverso da p0 faccio un ciclo dove:
 - Pongo la distanza di p da quello sorgente = infinito e, pongo anche il suo predecessore=vuoto
- Alla fine del ciclo imposto la distanza del nodo sorgente da sé stesso come nulla
- Faccio un altro ciclo finchè nel minHeap Q ho più di un nodo dove:
 - Estraggo da questa coda con priorità il nodo p, che ha distanza mimina da quello sorgente
 - Faccio un altro ciclo per ogni nodo q che è successore di p dove:
 - Creo una variabile l_{pq} in cui metto la lunghezza dell’arco che unisce p a q (quindi al suo successore)
 - Verifico se ($l_{pq} + \text{la distanza del nodo estratto}$) sia inferiore alla distanza del successore q dalla sorgente
 - Allora aggiorno la distanza del successore q dalla sorgente mettendoci la somma precedente
 - Aggiorno il precedente di q mettendoci p(nodo estratto)
 - Prendo il nuovo nodo q modificato, lo reinserisco nel minHeap Q che per la politica di coda con priorità che ha, andrà a posizionarsi nel posto giusto

17. Il limite inferiore è < di $O(n \log n)$ e gli algoritmi che lo raggiungono sono il RadixSort e il CountingSort con complessità $\rightarrow O(n+k)$

18. A differenza di altri algoritmi prevedono un dominio di azione

19. Dato un array di n numeri interi

- In una variabile d metto il numero di cifre che compone un numero dell’array
- Controllo su che base sono i miei numeri (per esempio da 0 a 9 la base è 10) e, questa base la chiamerò k
- Creo una lista di vettori di dimensione k con ciascun vettore di dimensione n
- Faccio un ciclo d volte e per ciascuna volta
 - Considero la d-esima cifra di ciascun numero (parto dalla cifra meno significativa e vado a ritroso)
 - Verifico dove si trova d=k, e inserisco il numero nella posizione k della lista di vettori (cioè nel contenitore k)
 - Ogni inserimento nel contenitore aumenterà il valore del contenitore
 - Alla fine di ogni ciclo prendo le componenti di ogni contenitore a partire dalla prima (dal basso verso l’alto e da sinistra verso destra) e le inserisco ordinatamente nell’array di partenza

DUCANGE (PRATICA) DOMANDE

“POSSIBILI RISPOSTE” PARTE TEORICA

20. Si considera un array A di n numeri interi

- Si considera il valore massimo dei numeri e lo chiamo k
(Es. numeri da 0 a 7, il massimo è 7)
- Considero un Array di k+1 elementi che sarà il mio array ausiliario
- Utilizzo due indici: i per scorrere l'array A e j per l'array C
- Faccio un ciclo dove inizializzo l'array C con valori nulli
- Faccio un altro ciclo (con j) dove nell'array C alla posizione A[j] ci inserisco +1
//Gioco sul valore di A che è rappresentato dall'indice di C, cosicchè l'indice di C rappresenta il numero di A e il valore sotto al medesimo indice rappresenta il numero di volte che trovo il numero in A, e che aumenterà ogni volta che faccio +1
- Faccio un altro ciclo per scorrere il vettore ausiliario modificato (nel passaggio precedente)
- Dentro il ciclo ne faccio uno while finché non finisce la frequenza (>0), che rappresenta i valori di A e quindi:
- Reinserisco ordinatamente per ogni frequenza >0 il valore dell'indice nell'array A e quindi ho l'array A ordinato

21. A produrre una codifica binaria del cammino partendo dalla radice fino ad arrivare ad un determinato carattere e, questo percorso si chiama → Codice prefisso

22. L'algoritmo prende in ingresso un minHeap di n elementi, che è un albero costruito inversamente dal solito, la cui radice è il valore più piccolo (una specie di coda con priorità)

- Nell'algoritmo faccio un ciclo fino a n-1 in cui ad ogni passo:
 - Creo un nuovo nodo inserendoci i valori della frequenza dei primi due nodi che estraggo dal minHeap; questi primi due nodi che estraggo sono i più piccoli di questa coda con priorità
 - Inserisco il nuovo nodo creato dentro il minHeap che andrà a posizionarsi al posto giusto, per l'ordinamento
- Alla fine del ciclo avrò un solo elemento nel minHeap che rappresenterà l'albero con gli elementi ordinati, e non devo fare altro che estrarlo

23. È un vettore di liste dove ogni elemento corrisponde ad un nodo del grafo e tutti i successori sono i nodi a cui è collegato

DUCANGE (PRATICA) DOMANDE

PARTE IMPLEMENTATIVA (SOLUZIONI NELLE PAGINE DI TEORIA DA “SelectionSort in poi”)

- Implementare il selectionSort
-

- Implementare il bubbleSort
-

- Implementare il quickSort
-

- Implementare il mergeSort
(corpo principale - a scelta tra quello per vettori e per liste)
-

- Creare una struttura dati albero (binario / generico)
 - Dato un albero binario, scrivi la funzione che calcola il numero di foglie
 - Dato un albero binario, scrivi la funzione che calcola il numero di nodi
 - Dato un albero binario, scrivi la funzione che cancella l'intero albero
 - Dato un albero binario scrivi la funzione che cerca all'interno una chiave data e lo restituiscia
-

- Scrivi la classe heap e i principali metodi pubblici
-

- Implementare lo HeapSort (corpo principale)
 - Implementare la buildHeap
-

- Scrivi il codice del countingSort
 - Scrivi lo pseudocodice del radixSort
-

- Scrivi la funzione di inserimento nell'Hash
-

- Algoritmo della PLSC
 - PLSC in programmazione dinamica
 - Scrivi funzione di stampa della PLSC
-

- Implementare l'algoritmo di Huffman
-

DUCANGE (PRATICA) DOMANDE

PARTE IMPLEMENTATIVA (SOLUZIONI NELLE PAGINE DI TEORIA DA “SelectionSort in poi”)

- Dichiarazione della Classe Grafo gestito con liste di adiacenza
 - Implementazione dei campi
 - Implementazione dei metodi più importanti
-
- Pseudocodice algoritmo di Dijkstra
-
- Scrivi una funzione che elimina l’ultimo elemento di una lista (RICORSIVAMENTE)

SOLUZIONE (NON PRESENTE NELLE PAGINE DI TEORIA):

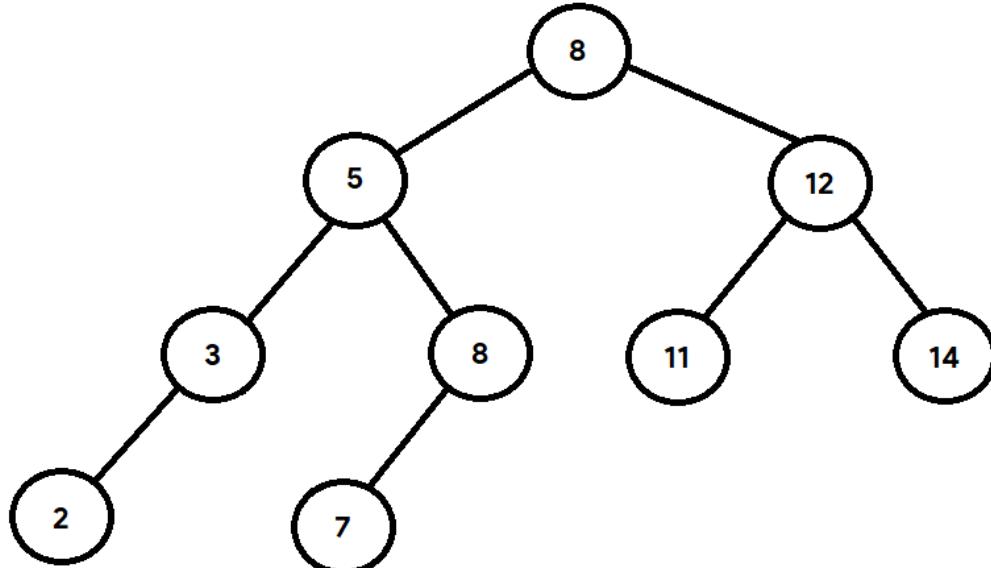
```
void eliminaUltimo(elem* &l){                                //0
    if(!l) return;                                         //1
    if(l->next == NULL){                                 //2
        delete l;                                         //3
        l=NULL;                                         //4
        return;                                         //5
    }
    eliminaUltimo(l->next);                            //6
}
```

0. &l perché cambia la lista
1. Lista vuota → Esco
2. Se non ho elementi dopo la lista, vuol dire che sono sull’ultimo elemento
3. Elimino il puntatore alla lista
4. Pongo a NULL questo elemento
5. Chiudo la funzione, visto che l’ho eliminato
6. Vuol dire che non sono entrato nei casi precedenti, dato che non ho trovato l’ultimo elemento → Vado a vedere se lo trovo tra i successivi nel resto della lista

ABR ALFEO (DOMANDE) - LAB

ABR - INPUT DI ESEMPIO PER GLI ESERCIZI D'ESAME : 8 5 3 8 2 7 12 11 14

Dopo la funzione di inserimento di questi elementi nell'albero, otterrò l'ABR fatto così:



OSS

- Se si considera l'altezza (livello) dell'albero si parte dalla radice con altezza (livello) = 0
- Se si considera l'altezza (livello) di un nodo si parte dalla radice con altezza (livello) = 1

//Queste due regole valgono a meno di istruzioni diversamente fornite da un esercizio

ESERCIZI ESAME

1 - SOMMA CONCORDI PARI - RADICE CONCORDE

- Somma dei nodi concordi pari, dove per concordi pari si definisce il nodo il cui valore è pari come anche il padre, la radice è concorde (pari)
Output atteso: 34

2 - SOMMA CONCORDI - RADICE DISCORDE

- Definisci una funzione che produca la stampa della sommatoria dei label dei nodi concordi. Un nodo si dice concorde se ha label pari e anche il padre ha label pari, oppure se ha label dispari e anche il padre ha label dispari. La radice per convenzione non è concorde
Output atteso: 29

3 - SOMMA NODI PARI - ALTEZZA PARI (NO RADICE, NO FOGLIE)

- Sommatoria dei soli nodi pari dove si definisce pari il nodo la cui altezza è pari, solo nodi, niente foglie (la radice per convenzione non si considera)
Output atteso: 12

4 - PRODOTTO ALTEZZA SOTTOALBERI

- Prodotto dell'altezza del sottoalbero destro per l'altezza del sottoalbero sinistro
Output atteso: 6

ABR ALFEO (DOMANDE) - LAB

... CONTINUO ESERCIZI ESAME

5 - SOMMA CONCORDI NODI PARI/DISPARI - ALTEZZA PARI/DISPARI

- Sommatoria dei nodi concordi dove concorde è il nodo con etichetta pari (o dispari) ed altezza pari (o dispari)
Output atteso: 28

6 - SOMMA NODI INCOMPLETI - ALTEZZA DISPARI

- Riferiamo come incompleti i nodi con meno di due figli. Crea una funzione che stampi la sommatoria dei label dei nodi incompleti di altezza dispari. La radice ha altezza 1
Output atteso: 36

7 - SOMMA NODI COMPLETI - ALTEZZA DISPARI

- Sommatori di nodi completi di altezza dispari, per completo si definisce il nodo che ha due figli (radice altezza 1)
Output atteso: 8

8 - DIFFERENZA (NODI ALTEZZA PARI - NODI ALTEZZA DISPARI)

- Una funzione che stampa la differenza tra la sommatoria dei nodi di altezza dispari e nodi di altezza pari. La radice per convenzione ha altezza 0
Output atteso: 18

9 - DIFFERENZA (SOMMATORIA FOGLIE ALTEZZA DISPARI) - VALORE RADICE

- Scrivi una funzione che stampa la differenza tra la sommatoria delle foglie ad altezza dispari e il valore della radice dell'albero. La radice per convenzione ha altezza zero
Output atteso: 1

10 - SOMMATORIA FOGLIE - ALTEZZA DISPARI

- Scrivi la funzione che produce la stampa della sommatoria delle foglie di altezza dispari. La radice dell'albero ha altezza 0.
Output atteso: 9

11- STAMPA FOGLIE - ETICHETTA DISPARI

- Definisci una funzione che stampa dell'albero binario solo le foglie con etichette dispari.
Output atteso: 7 11

12 - STAMPA ALTEZZA ELEMENTO PRESO IN INGRESSO

- Una funzione che ritorna l'altezza di un elemento passato in input.
La radice ha altezza 0. Si passi "7" come elemento in input alla funzione.
Output atteso: 3

ABR ALFEO (RISPOSTE) - LAB

ABR

1 → SOLUZIONE

```
int sommaConcordiPari(Node *tree, int father){  
    int valConcorde=0; //1  
    if(!tree) return 0;  
    if ((father%2==0) && (tree->value%2==0)) //2  
        valConcorde=tree->value; //3  
    return sommaConcordiPari(tree->left, tree->value)  
        + sommaConcordiPari(tree->right, tree->value)  
        + valConcorde; //4  
}
```

//Per ragioni di spazio sono andato a capo nelle somme, su C++ è tutto unito

1. Variabile che conterrà il valore concorde pari che, verrà trovato in questa chiamata e che, andrà poi sommato
2. Controllo se padre e valore attuale del nodo hanno lo stesso resto della divisione (=0), quindi sono concordi pari tra loro
3. Aggiorno il valore concorde dandogli il valore attuale del nodo
4. Richiamo la funzione ricorsivamente a DX e a SX per la sommatoria totale (dando ad entrambi come father il valore attuale del nodo) + sommo il valore concorde attualmente trovato

→ MAIN

```
cout<<ABR.sommaConcordiPari(ABR.getRoot(),0);
```

- Nel main metto 0 a father per far sì, che sia concorde pari alla radice 8 (che sarebbe il figlio di 0 in questo caso) → Da come dice il testo dell'esercizio

ABR ALFEO (RISPOSTE) - LAB

ABR

2 → SOLUZIONE

```
int sommaConcordi(Node *tree, int father){  
    int valConcorde=0; //1  
    if(!tree) return 0;  
    if (((father%2==0) && (tree->value%2==0))  
        || (!(father%2==0) && !(tree->value%2==0))) //2  
        valConcorde=tree->value; //3  
    return sommaConcordi(tree->left, tree->value)  
        + sommaConcordi(tree->right, tree->value)  
        + valConcorde; //4  
}
```

//Per ragioni di spazio sono andato a capo nelle somme e nella if, su C++ è tutto unito

1. Variabile che conterrà il valore concorde che, verrà trovato in questa chiamata e che, andrà poi sommato
2. Controllo se padre e valore attuale del nodo hanno lo stesso resto della divisione (=0), quindi sono concordi pari tra loro oppure se padre e valore attuale del nodo non hanno resto della divisione (=0), quindi sono entrambi dispari e concordi dispari fra loro → Per concordi in questo caso prendo sia i concordi pari che dispari
3. Aggiorno il valore concorde dandogli il valore attuale del nodo
4. Richiamo la funzione ricorsivamente a DX e a SX per la sommatoria totale (dando ad entrambi come father il valore attuale del nodo) + sommo il valore concorde attualmente trovato

→ MAIN

```
cout<<ABR.sommaConcordi(ABR.getRoot(),1);
```

- Nel main metto 1 a father per far sì, che sia concorde alla radice 8 (che sarebbe il figlio di 1 in questo caso) → Da come dice il testo dell'esercizio

ABR ALFEO (RISPOSTE) - LAB

ABR

3 → SOLUZIONE

```
int sommaPari(Node *tree, int height){  
    int valPari=0; //1  
    if(!tree) return 0;  
    if(!tree->right && !tree->left) return 0; //2  
    if ( (tree->value%2==0) && (height%2==0) ) //3  
        valPari=tree->value; //4  
    height++; //5  
    return sommaPari(tree->left, height)  
        + sommaPari(tree->right, height)  
        + valPari; //6  
}
```

//Per ragioni di spazio sono andato a capo nelle somme, su C++ è tutto unito

1. Variabile che conterrà il valore pari con l'altezza pari che, verrà trovato in questa chiamata e che, andrà poi sommato
2. Caso in cui si incombe nella foglia, non va considerata (dal testo dell'esercizio)
3. Se l'altezza attuale e valore attuale hanno lo stesso resto della divisione (=0), sono entrambi pari
4. Aggiorno il valore pari dandogli il valore attuale del nodo
5. Aggiorno il valore dell'altezza che aumenta visto, che farò le ricorsioni sui sottoalberi DX e SX
6. Richiamo la funzione ricorsivamente a DX e a SX per la sommatoria totale + sommo il valore pari attualmente trovato

→ MAIN

```
cout<<ABR.sommaPari(ABR.getRoot(),1);
```

- “La radice non si considera” → Pongo 1 nel main visto che 1 non è pari e non viene verificato l'AND della 3° if, poiché non è soddisfatto su height, visto che risulterà dispari → Quindi la radice non influirà sulla somma, da come dice il testo dell'esercizio

ABR ALFEO (RISPOSTE) - LAB

ABR

4 → SOLUZIONE

```
int max(int x, int y){  
    if (x>=y) return x;  
    else return y;  
}
```

- Utilizzo questa funzione che calcola il massimo tra due elementi

```
int altezzaSottoAlbero(Node *tree, int height){  
    height++; //1  
    if(!tree) return (height-1); //2  
    return max( altezzaSottoAlbero(tree->left, height),  
                altezzaSottoAlbero(tree->right, height) ); //3  
}
```

//Per ragioni di spazio sono andato a capo nei parametri della funzione **max**, su C++ è tutto unito

1. Aumento di 1 l'altezza visto che sto ispezionando un sottoalbero (SX o DX, dipende da quale viene passato tra i parametri)
Nel caso in cui non abbia sottoalberi, questa altezza andrà a 0 nel prossimo passaggio
2. Caso in cui ho l'albero vuoto → Allora ritorno l'altezza precedente della foglia o della radice che sarà l'ultima per questo sottoalbero
3. Caso in cui non ho questo sottoalbero che sto analizzando come vuoto
→ Allora ritorno il massimo dell'altezza (con la funzione **max**), tra il sottoalbero SX e DX (del sottoalbero che sto analizzando) e l'altezza aumentata al passaggio //1 per via delle ricorsioni

ABR ALFEO (RISPOSTE) - LAB

ABR

...CONTINUO 4 → SOLUZIONE

```
int prodotto(Node *tree){  
    return altezzaSottoAlbero(tree->left,0)  
        * altezzaSottoAlbero(tree->right,0);  
}
```

//Per ragioni di spazio sono andato a capo nel prodotto, su C++ è tutto unito

- Utilizzo questa funzione extra per tirare fuori il prodotto delle altezze, dei due sottoalberi SX e DX
L'altezza passata tra i parametri è =0 e sarà subito aumentata dentro la funzione, al momento di ciascuna delle due chiamate

→ MAIN

```
cout<<ABR.prodotto(ABR.getRoot());
```

- Gli passo solo la radice

ABR ALFEO (RISPOSTE) - LAB

ABR

5 → SOLUZIONE

```
int sommaConcordiAltezza(Node *tree, int height){  
    int valConcorde=0; //1  
    if(!tree) return 0;  
    if (((height%2==0) && (tree->value%2==0))  
        || (!(height%2==0) && !(tree->value%2==0))) //2  
        valConcorde=tree->value; //3  
    height++; //4  
    return sommaConcordiAltezza(tree->left, height)  
        + sommaConcordiAltezza(tree->right, height)  
        + valConcorde; //5  
}
```

//Per ragioni di spazio sono andato a capo nelle somme e nella if, su C++ è tutto unito

1. Variabile che conterrà il valore concorde (pari/dispari) all'altezza (pari/dispari), che verrà trovato in questa chiamata e che, andrà poi sommato
2. Controllo se altezza e valore attuale del nodo hanno lo stesso resto della divisione (=0), quindi sono concordi pari tra loro oppure se altezza e valore attuale del nodo non hanno resto della divisione (=0), di conseguenza sono entrambi dispari e concordi dispari fra loro → Per concordi prendo sia i concordi pari che dispari
3. Aggiorno il valore concorde dandogli il valore attuale del nodo
4. Aggiorno il valore dell'altezza che aumenta visto, che farò le ricorsioni sui sottoalberi DX e SX
5. Richiamo la funzione ricorsivamente a DX e a SX per la sommatoria totale (con l'altezza aggiornata), + sommo il valore concorde attualmente trovato

→ MAIN

```
cout<<ABR.sommaConcordiAltezza(ABR.getRoot(),1);
```

- Nel main metto 1 a height, perché l'altezza di un nodo parte da 1 (partendo dalla radice)

ABR ALFEO (RISPOSTE) - LAB

ABR

6 → SOLUZIONE

```
int incompleti(Node *tree, int height){  
    if(!tree) return 0;  
    if( (!tree->left || !tree->right) && !(height%2==0) ){ //1  
        height++; //2  
        return tree->value  
            + incompleti(tree->left, height)  
            + incompleti(tree->right, height); //3  
    }  
    height++; //4  
    return incompleti(tree->left, height)  
        + incompleti(tree->right, height); //5  
}
```

//Per ragioni di spazio sono andato a capo nei due blocchi di somme, su C++ è tutto unito

1. Per il nodo incompleto controllo se ho nodi a SX e a DX no, oppure se ho nodi a DX e a SX no, oppure se non ho nodi nè a SX nè a DX (quindi 1 o 0) e, contemporaneamente verifico che l'altezza sia dispari
2. Aggiorno il valore dell'altezza che aumenta visto, che farò le ricorsioni sui sottoalberi DX e SX
3. Ritorno il valore del nodo (che mi rappresenta il nodo incompleto) + le chiamate a SX e a DX, per vedere se avrò altri nodi incompleti nel resto dell'ABR
4. Altrimenti, aggiorno il valore dell'altezza che aumenta, visto che farò le ricorsioni sui sottoalberi DX e SX
5. Controllo il resto dell'ABR, se posso trovare altri nodi incompleti sui sottoalberi SX e DX

→ MAIN

```
cout<<ABR.incompleti(ABR.getRoot(),1);
```

- Nel main metto 1 a height → Altezza della radice data dal testo dell'esercizio

ABR ALFEO (RISPOSTE) - LAB

ABR

7 → SOLUZIONE

```
int completi(Node *tree, int height){  
    if(!tree) return 0;  
    if( (tree->left && tree->right) && !(height%2==0) ){ //1  
        height++; //2  
        return tree->value  
            + completi(tree->left, height)  
            + completi(tree->right, height); //3  
    }  
    height++; //4  
    return completi(tree->left, height)  
        + completi(tree->right, height); //5  
}
```

//Per ragioni di spazio sono andato a capo nei due blocchi di somme, su C++ è tutto unito

1. Per il nodo completo controllo se ho nodi a SX e a DX e, contemporaneamente verifico che l'altezza sia dispari
2. Aggiorno il valore dell'altezza che aumenta visto, che farò le ricorsioni sui sottoalberi DX e SX
3. Ritorno il valore del nodo (che mi rappresenta il nodo completo) + le chiamate a SX e a DX, per vedere se avrò altri nodi completi nel resto dell'ABR
4. Altrimenti aggiorno il valore dell'altezza che aumenta, visto che farò le ricorsioni sui sottoalberi DX e SX
5. Controllo il resto dell'ABR se posso trovare altri nodi completi sui sottoalberi SX e DX

→ MAIN

```
cout<<ABR.completi(ABR.getRoot(),1);
```

- Nel main metto 1 a height → Altezza della radice data dal testo dell'esercizio

ABR ALFEO (RISPOSTE) - LAB

ABR

8 → SOLUZIONE

```
int diffAltezzaPariDispari(Node *tree, int height){

    if(!tree) return 0;

    int diff=tree->value; //1

    if(height%2==0){ //2

        height++;

        return diffAltezzaPariDispari(tree->left,height)
            + diffAltezzaPariDispari(tree->right,height)
            + diff; //4

    }

    else{ //5

        height++;

        return diffAltezzaPariDispari(tree->left,height)
            + diffAltezzaPariDispari(tree->right,height)
            - diff; //7

    }

}
```

//Per ragioni di spazio sono andato a capo nei due blocchi di somme, su C++ è tutto unito

1. Variabile in cui inserirò il nodo attuale che sto verificando, a sommare se ho altezza pari, a sottrarre se ho altezza dispari
2. Se ho altezza pari
3. Aggiorno il valore dell'altezza che aumenta, visto che farò le ricorsioni sui sottoalberi DX e SX
4. Faccio una somma con le chiamate ricorsive sui sottoalberi SX e DX e ci sommo il nodo attuale, che ha altezza pari
5. Altrimenti ho altezza dispari
6. Aggiorno il valore dell'altezza che aumenta, visto che farò le ricorsioni sui sottoalberi DX e SX
7. Faccio una somma con le chiamate ricorsive sui sottoalberi SX e DX e ci sottraggo il nodo attuale, che ha altezza dispari

→ MAIN

```
cout<<ABR.diffAltezzaPariDispari(ABR.getRoot(),0);
```

- Nel main metto 0 a height per convenzione scelta nel testo sulla radice

ABR ALFEO (RISPOSTE) - LAB

ABR

9,10 → SOLUZIONE

```
int sommaFoglieAltezzaDispari(Node *tree, int height){  
    if(!tree) return 0;  
    if( (!tree->left && !tree->right) && !(height%2==0) ){ //1  
        return tree->value  
    height++; //3  
    return sommaFoglieAltezzaDispari(tree->left,height)  
        + sommaFoglieAltezzaDispari(tree->right,height); //4  
}
```

//Per ragioni di spazio sono andato a capo nelle somme, su C++ è tutto unito

1. Verifico la condizione in cui sono su una foglia e ad altezza dispari
2. Ritorno questa foglia
3. Altrimenti aggiorno il valore dell'altezza che aumenta, visto che farò le ricorsioni sui sottoalberi DX e SX
4. Ricorsione sui sottoalberi SX e DX per vedere se ci sono altre foglie ad altezza dispari

→ MAIN SOLUZIONE PER 9

```
cout<<(ABR.sommaFoglieAltezzaDispari(ABR.getRoot(),0))  
    - ABR.getRoot()->value ;
```

- Nel main metto 0 a height per convenzione scelta sulla radice
La funzione mi produce solamente la sommatoria delle foglie ad altezza dispari
→ Quindi a questa sommatoria ci sottraggo il valore della radice

→ MAIN SOLUZIONE PER 10

```
cout<<ABR.sommaFoglieAltezzaDispari(ABR.getRoot(),0);
```

- Nel main metto 0 a height per convenzione scelta sulla radice

ABR ALFEO (RISPOSTE) - LAB

ABR

11 → SOLUZIONE

```
void foglieDispari(Node *tree){  
    if (!tree) return;  
    if( !(tree->value%2==0)  
        && ((!tree->left) && (!tree->right)) ){           //1  
        cout<<tree->value<<" ";                         //2  
        return;                                              //3  
    }  
    foglieDispari(tree->left);                            //4  
    foglieDispari(tree->right);                           //5  
}
```

//Per ragioni di spazio sono andato a capo nella condizione della if, su C++ è tutto unito

1. Controllo se il valore dell'etichetta è dispari e se sono su una foglia (contemporaneamente)
2. Stampo il valore che è una foglia di etichetta dispari
3. Chiudo la funzione
4. Richiamo la funzione sul sottoalbero SX per vedere se ho altre etichette dispari
5. Richiamo la funzione sul sottoalbero DX per vedere se ho altre etichette dispari

→ MAIN

```
ABR.foglieDispari(ABR.getRoot());
```

- Metto solo la funzione essendo void

ABR ALFEO (RISPOSTE) - LAB

ABR

12 → SOLUZIONE

```
int altezzaElemento(Node *tree, int height, int x){

    if(!tree)  return 0;

    int heightElem;                                //1

    if(x==tree->value){                           //2

        heightElem=height;                         //3

        return heightElem;                         //4

    }

    if(x<tree->value){                           //5

        height++;                                 //6

        heightElem=altezzaElemento(tree->left,height,x); //7

    }

    else{                                         //8

        height++;                                 //9

        heightElem=altezzaElemento(tree->right,height,x); //10

    }

    return heightElem;                            //11

}
```

ABR ALFEO (RISPOSTE) - LAB

ABR

...CONTINUO 12 → SOLUZIONE

1. Dichiaro una variabile in cui metterò l'altezza dell'elemento preso in ingresso
2. Sono sulla condizione che mi verifica, se ho trovato l'elemento nella radice che sto analizzando attualmente
3. Aggiorno la variabile che conterrà l'elemento
4. La restituisco chiudendo la funzione
5. Controllo se l'elemento preso in ingresso è più piccolo della radice attuale (per la politica adottata dall'ABR)
6. Aggiorno il valore dell'altezza, aumentandola per fare la ricorsione sul sottoalbero SX
7. Andrò a vedere se lo trovo nel sottoalbero di SX, e la sua altezza la assegno alla variabile presa che verrà poi restituita
8. Altrimenti l'elemento preso in ingresso è più grande della radice attuale (per la politica adottata dall'ABR)
9. Aggiorno il valore dell'altezza aumentandola per fare la ricorsione sul sottoalbero DX
10. Andrò a vedere se lo trovo nel sottoalbero di DX, e la sua altezza la assegno alla variabile presa che verrà poi restituita
11. Restituirò la variabile che conterrà l'altezza dell'elemento preso in ingresso
(dopo aver fatto la scansione in uno dei due sottoalberi di partenza, partendo dalla radice)

→ MAIN

```
cout<<ABR.altezzaElemento(ABR.getRoot(), 0, 7);
```

- Nel main metto 0 a height per convenzione scelta sulla radice

ABR ALFEO (RISPOSTE) - LAB

EXTRA

Per l'inserimento nell'ABR ho utilizzato questa funzione:

```
-----  
void insert(int val){  
    Node *node = new Node(val);  
    Node *pre=NULL;  
    Node *post=root;  
    while(post != NULL){  
        pre=post;  
        if(val<=post->value)  
            post=post->left;  
        else  
            post=post->right;  
    }  
    if(pre==NULL)  
        root=node;  
    else if(val<pre->value)  
        pre->left=node;  
    else  
        pre->right=node;  
    return;  
}  
-----
```

Funzione che inserisce gli elementi all'interno dell'ABR, scansionando per ogni elemento dell'albero dove devo andare a posizionarmi.

Se, durante la scansione dell'ABR l'elemento da inserire è \leq di un nodo nei vari sottoalberi, lo vado ad inserire alla sua sinistra, altrimenti alla sua destra.

HEAP ALFEO (DOMANDE) - LAB

HEAP

1 - Stampa uno heap dove l'elemento il cui figlio destro di un determinato nodo K è tra parentesi quadre

13

9 8

4 3 1

OUTPUT ATTESO:

13 (k=3)

[9] 8

4 3 1

2 - Scrivi la funzione che stampa lo heap e in particolare stampa tra parentesi quadre gli elementi il cui figlio destro ha un determinato valore k passato dal main

10

6

8 4 3 2

OUTPUT ATTESO:

[10] (K=6)

9 6

8 4 3 2

HEAP ALFEO (DOMANDE) - LAB

HEAP

3 - Stampa dello heap, in modo che sia tra parentesi quadre il valore delle etichette dei nodi, il cui figlio sinistro ha un determinato valore k

```
10
 9   6
 8   4   3   2
```

OUTPUT ATTESO:

```
10          (K=8)
[9]   6
 8   4   3   2
```

4 - Stampa dello heap, in modo che sia fra parentesi quadre il valore delle etichette dei nodi il cui padre ha un determinato valore k

```
10
 7   6
 | 4   3   2   5
```

OUTPUT ATTESO:

```
10          (K=7)
 7   6
 [4]  [3]   2   5
```

HEAP ALFEO (DOMANDE) - LAB

HEAP

5 - Implementare la stampa dello heap (dove i nodi sono disposti graficamente in modo da rappresentare lo heap come un albero), come da esempio seguente e dove gli elementi appartenenti al sottoalbero di una etichetta K sono tra le parentesi quadre

10

9 6

8 4 3 2

OUTPUT ATTESO:

10 (K=6)

9 6

8 4 [3] [2]

HEAP ALFEO (RISPOSTE) - LAB

HEAP

//L'esercizio 1 "tradotto": Stampa uno Heap con un elemento tra parentesi quadre il cui figlio destro è un determinato nodo k

1,2 → SOLUZIONE (ES.1 - n=6), (ES.2 - n=7)

```
void stampaPadreFiglioDX(int k){  
    for(int i=0; i<n; i++){  
        if(isFirstChild(i+1)) cout<<endl; //1  
        if(k==h[(2*i+2)])  
            cout<<"["<<h[i]<<"]" " ; //2  
        else  
            cout<<h[i]<<" " ; //3  
    } //4  
}
```

1. Faccio un ciclo per scorrere tutti gli elementi dello Heap
 2. Condizione per andare a capo nella stampa, gli passo $i+1$ perché parto da 0 nell'array dove è memorizzata la radice dello Heap, invece per la politica di posizionamento nell'albero degli elementi dello Heap la radice sta alla posizione 1
 3. Controllo se il nodo k preso in ingresso, è uguale al figlio destro ($2*i+2$) dell'elemento dello Heap che sto analizzando
 4. Stampo questo elemento dello Heap con le parentesi quadre
Occchio agli spazi! Qui ha due spazi in meno della stampa normale al punto 6, perché ho utilizzato le parentesi quadre
 5. Altrimenti non è il figlio destro, quindi
 6. Lo stampo normalmente

→ MAIN SOLUZIONE 1 (k=3)

stampaPadreFiglioDX(3);

→ MAIN SOLUZIONE 2 (k=6)

```
stampaPadreFiglioDX(6);
```

HEAP ALFEO (RISPOSTE) - LAB

HEAP

3 → SOLUZIONE (n=7)

```
void stampaPadreFiglioSX(int k){  
    for(int i=0; i<n; i++){  
        if(isFirstChild(i+1)) cout<<endl; //1  
        if(k==h[(2*i+1)])  
            cout<<"["<<h[i]<<"]" " ; //2  
        else  
            cout<<h[i]<<" " ; //3  
    } //4  
}
```

1. Faccio un ciclo per scorrere tutti gli elementi dello Heap
2. Condizione per andare a capo nella stampa, gli passo $i+1$ perché parto da 0 nell'array dove è memorizzata la radice dello Heap, invece per la politica di posizionamento nell'albero degli elementi dello Heap la radice sta alla posizione 1
3. Controllo se il nodo k preso in ingresso, è uguale al figlio sinistro $(2*i+1)$ dell'elemento dello Heap che sto analizzando
4. Stampo questo elemento dello Heap con le parentesi quadre
Occhio agli spazi! Qui ha due spazi in meno della stampa normale al punto 6, perché ho utilizzato le parentesi quadre
5. Altrimenti non è il figlio sinistro, quindi
6. Lo stampo normalmente

→ MAIN (k=8)

```
stampaPadreFiglioSX(8);
```

HEAP ALFEO (RISPOSTE) - LAB

HEAP

4,5 → SOLUZIONE (ES.4 - n=7), (ES.5 - n=7)

```
void stampaFigliDaPadre(int k){  
    for(int i=0; i<n; i++){  
        if(isFirstChild(i+1)) cout<<endl; //1  
        if(k==h[(i-1)/2])  
            cout<<"["<<h[i]<<"]" " ; //2  
        else  
            cout<<h[i]<<" " ; //3  
    } //4  
}
```

1. Faccio un ciclo per scorrere tutti gli elementi dello Heap
2. Condizione per andare a capo nella stampa, gli passo $i+1$ perché parto da 0 nell'array dove è memorizzata la radice dello Heap, invece per la politica di posizionamento nell'albero degli elementi dello Heap la radice sta alla posizione 1
3. Controllo se il nodo k preso in ingresso, è uguale al padre $((i-1)/2)$ del figlio che sto analizzando in questo momento (scorrendo l'array trovo sia il figlio SX che quello DX)
4. Stampo questo elemento dello Heap con le parentesi quadre (che è uno dei due figli)
Occhio agli spazi ha due spazi! Qui ha due spazi in meno della stampa normale al punto 6, perché ho utilizzato le parentesi quadre
5. Altrimenti non è il padre che sto cercando
6. Stampo normalmente l'elemento

→ MAIN SOLUZIONE 4 ($k=7$)

```
stampaFigliDaPadre(7);
```

→ MAIN SOLUZIONE 5 ($k=6$)

```
stampaFigliDaPadre(6);
```

HEAP ALFEO (RISPOSTE) - LAB

EXTRA

Funzione di verifica primo figlio elemento dello HEAP per poi andare a capo

```
-----  
bool isFirstChild(int i){  
    if( (!i=0) && ( (i&(i-1))==0 )  
        return true;  
    else  
        return false;  
}  
-----
```

Per verificare se ho un primo figlio, in una determinata posizione i passata tra i parametri, devo verificare che i sia diverso da 0 e, contemporaneamente devo fare l'AND bit a bit con il valore precedente, e verificare che faccia 0 → Ciò ci rappresenta che sono in una posizione potenza di 2, e quindi restituisco true che mi rappresenterà poi nella stampa, il fatto che io possa andare a capo

COMPLESSITÀ

RELAZIONE DI RICORRENZA: È una forma ricorsiva che permette di arrivare a capire la complessità di un programma ricorsivo

RELAZIONE DI RICORRENZA

COMPLESSITÀ $T(n)$

$$\begin{aligned} T(0) &= a \\ T(n) &= b + T(n/2) \end{aligned}$$

$O(\log n)$

$$\begin{aligned} T(0) &= a \\ T(1) &= b + a \\ T(2) &= b + b + a = 2b + a \\ T(4) &= b + 2b + a = 3b + a \\ T(8) &= b + 3b + a = 4b + a \\ &\dots \\ &\dots \\ T(n) &= (\log n + 1)b + a \end{aligned}$$

$$//((\log_2 8 = 3) + 1)b + a$$

RELAZIONE DI RICORRENZA

COMPLESSITÀ $T(n)$

$$\begin{aligned} T(1) &= a \\ T(n) &= b + T(n/2) \end{aligned}$$

$O(\log n)$

Simile al precedente

RELAZIONE DI RICORRENZA

COMPLESSITÀ $T(n)$

$$\begin{aligned} T(0) &= T(1) = T(2) = a \\ T(n) &= b + T(n/3) \end{aligned}$$

$O(\log n)$

RELAZIONE DI RICORRENZA

COMPLESSITÀ $T(n)$

$$\begin{aligned} T(0) &= a \\ T(n) &= b + T(n-1) \end{aligned}$$

$O(n)$

$$\begin{aligned} T(0) &= a \\ T(1) &= b + a \\ T(2) &= b + b + a = 2b + a \\ T(3) &= b + 2b + a = 3b + a \\ &\dots \\ &\dots \\ T(n) &= nb + a \end{aligned}$$

RELAZIONE DI RICORRENZA

COMPLESSITÀ $T(n)$

$$\begin{aligned} T(1) &= a \\ T(n) &= b + T(n-1) \end{aligned}$$

$O(n)$

Simile al precedente

COMPLESSITÀ

RELAZIONE DI RICORRENZA

COMPLESSITÀ T(n)

$$\begin{aligned}T(1) &= a \\T(n) &= b + T(n-2)\end{aligned}$$

O(n)

Simile al precedente

RELAZIONE DI RICORRENZA

COMPLESSITÀ T(n)

$$\begin{aligned}T(0) &= a \\T(n) &= b + 2T(n/2)\end{aligned}$$

O(n)

$$T(0) = a$$

$$T(1) = b + 2a$$

$$T(2) = b + 2b + 4a = 3b + 4a$$

$$T(4) = b + 6b + 8a = 7b + 8a$$

.

.

$$T(n) = (2n - 1)b + 2na$$

RELAZIONE DI RICORRENZA

COMPLESSITÀ T(n)

$$\begin{aligned}T(0) &= a \\T(n) &= b + 2T((n-1)/2)\end{aligned}$$

O(n)

Simile al precedente

RELAZIONE DI RICORRENZA

COMPLESSITÀ T(n)

$$\begin{aligned}T(0) &= T(1) = T(2) = 1 \\T(n) &= 3 + 3T(n/3)\end{aligned}$$

O(n)

RELAZIONE DI RICORRENZA

COMPLESSITÀ T(n)

$$\begin{aligned}T(1) &= a \\T(n) &= bn + 2T(n/2)\end{aligned}$$

O(n log n)

$$T(1) = a$$

$$T(2) = 2b + 2a$$

$$T(4) = 4b + 4b + 4a$$

$$T(8) = 8b + 8b + 8b + 8a = 3(8b) + 8a$$

$$T(16) = 16b + 16b + 16b + 16b + 16a = 4(16b) + 16a$$

.

.

$$T(n) = (n \log n) + na$$

COMPLESSITÀ

RELAZIONE DI RICORRENZA

$$\begin{aligned} T(0) &= a \\ T(n) &= bn + T(n-1) \end{aligned}$$

COMPLESSITÀ $T(n)$

$$O(n^2)$$

$$T(1) = a$$

$$T(2) = 2b + a$$

$$T(3) = 3b + 2b + a$$

.

$$T(n) = (n + n-1 + n-2 + \dots + 2)b + a = (n(n+1)/2 - 1)b + a \quad // \text{somma dei primi } n \text{ numeri positivi}$$

RELAZIONE DI RICORRENZA

$$\begin{aligned} T(1) &= 1 \\ T(n) &= bn + T(n-1) \end{aligned}$$

COMPLESSITÀ $T(n)$

$$O(n^2)$$

Simile al precedente

RELAZIONE DI RICORRENZA

$$\begin{aligned} T(1) &= 1 \\ T(n) &= n + T(n-1) \end{aligned}$$

COMPLESSITÀ $T(n)$

$$O(n^2)$$

Simile al precedente

RELAZIONE DI RICORRENZA

$$\begin{aligned} T(1) &= 1 \\ T(n) &= b + 2T(n-1) \end{aligned}$$

COMPLESSITÀ $T(n)$

$$O(2^n)$$

$$T(1) = a$$

$$T(2) = b + 2a$$

$$T(3) = b + 2b + 4a = 3b + 4a$$

$$T(4) = 7b + 8a$$

.

$$T(n) = (2^{(n-1)} - 1)b + 2^{(n-1)}a$$

RELAZIONE DI RICORRENZA

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 1 + 2T(n-1) \end{aligned}$$

COMPLESSITÀ $T(n)$

$$O(2^n)$$

Simile al precedente

RELAZIONE DI RICORRENZA

$$\begin{aligned} T(0) &= a \\ T(n) &= b + 2T(n-1) \end{aligned}$$

COMPLESSITÀ $T(n)$

$$O(2^n)$$

Simile al precedente

COMPLESSITÀ

RELAZIONE DI RICORRENZA

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2 + 2T(n-1) \end{aligned}$$

COMPLESSITÀ T(n)

$$O(2^n)$$

Simile al precedente

RELAZIONE DI RICORRENZA

$$\begin{aligned} T(0) &= T(1) = d \\ T(n) &= b + T(n-1) + T(n-2) \end{aligned}$$

COMPLESSITÀ T(n)

$$O(2^n)$$

RELAZIONE DI RICORRENZA

$$\begin{aligned} T(1) &= d \\ T(n) &= bn + 3T(n/2) \end{aligned}$$

$$\begin{array}{ll} n \leq 1 \\ n > 1 \end{array}$$

COMPLESSITÀ T(n)

$$O(n^{\log_2 3})$$

Moltiplicazione veloce
 $O(n^{1.59})$

METODO DIVIDE ET IMPERA

RELAZIONE DI RICORRENZA GENERICA

$$\begin{aligned} T(n) &= d & n \leq 1 \\ T(n) &= aT(n/b) + c & n > 1 \end{aligned}$$

- **c** = combinazione dei risultati ottenuti dai sottoinsiemi
- **aT(n/b)** = partizionamento in sottoinsiemi di **b** elementi **a** volte

RELAZIONE DI RICORRENZA

$$\begin{aligned} T(0) &= d \\ T(n) &= c + T(n/2) \end{aligned}$$

COMPLESSITÀ T(n)

$$O(\log n)$$

La chiamata è 1 → **a=1**

Due sottoinsiemi → **b=2**

Combinazione **c costante**

RELAZIONE DI RICORRENZA

$$\begin{aligned} T(0) &= d \\ T(n) &= c + 2T(n/2) \end{aligned}$$

COMPLESSITÀ T(n)

$$O(n)$$

Le chiamate sono 2 → **a=2**

Due sottoinsiemi → **b=2**

Combinazione **c costante**

COMPLESSITÀ

RELAZIONE DI RICORRENZA

COMPLESSITÀ $T(n)$

$$\begin{aligned} T(0) &= d \\ T(n) &= cn + 2T(n/2) \end{aligned}$$

$$\begin{array}{ll} n \leq 1 \\ n > 1 \end{array}$$

$\Rightarrow O(n \log n)$

Le chiamate sono 2 $\rightarrow a=2$

Due sottoinsiemi $\rightarrow b=2$

Combinazione c dipende da n

METODO DIVIDE ET IMPERA - FORMA GENERICA

RELAZIONE DI RICORRENZA GENERICA

$$\begin{aligned} T(n) &= d \\ T(n) &= cn^k + aT(n/b) \end{aligned}$$

$n \leq m$ // se l'insieme su cui lavoro è $\leq m$ so fare il risultato
 $n > m$ // altrimenti

- $c > 0$ = ho sempre questo lavoro di combinazione
- m = numero di situazioni in cui so calcolare il risultato direttamente
- a = numero di chiamate
- b^k = numero di sottoinsiemi che faccio

SOLUZIONI CHE OTTENGO A SECONDA DELLE RELAZIONI TRA a e b^k :

$$T(n) \in O(n^k)$$

$$a < b^k$$

// polinomiale

$$T(n) \in O(n^k \log n)$$

$$a = b^k$$

$$T(n) \in O(n^{\log_b a})$$

$$a > b^k$$

// esponenziale

COMPLESSITÀ

RELAZIONI DI RICORRENZA LINEARI

FORMA GENERALE

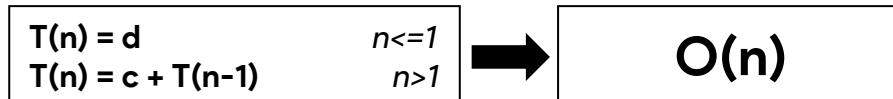
$$\begin{aligned} T(n) &= d \\ T(n) &= cn^k + a_1T(n-1) + a_2T(n-2) + \dots + a_rT(n-r) \end{aligned}$$

- $a_i=1 \rightarrow$ Soluzione polinomiale e gli altri a_i sono tutti 0 altrimenti soluzione esponenziale

CLASSI DI RICORRENZA DI RELAZIONI LINEARI

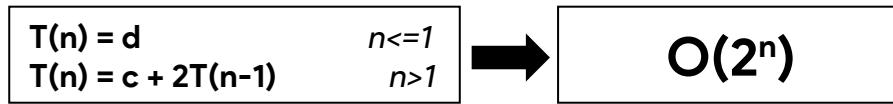
RELAZIONE DI RICORRENZA

COMPLESSITÀ $T(n)$



RELAZIONE DI RICORRENZA

COMPLESSITÀ $T(n)$



SOLUZIONE POLINOMIALE



- $c > 0$
- Se $k=1$ ottengo $O(n^2)$
- Se n non c'è alle combinazioni ottengo $O(n)$

ITERAZIONE-RICORSIONE

- Considero la serie di Fibonacci così definita:

- $f_0=0$
- $f_1=1$
- $f_n = f_{n-1} + f_{n-2}$

- Dato un numero n, al passo n dovrò tirare fuori il numero che mi produce la serie

SERIE DI FIBONACCI RICORSIVA

```
int fibonacci(int n){  
    if(n==0) return 0;  
    if(n==1) return 1;  
    return fibonacci(n-1) + fibonacci(n-2);  
}
```

RELAZIONE DI RICORRENZA

$$\begin{aligned} T(0) &= T(1) = d \\ T(n) &= b + T(n-1) + T(n-2) \end{aligned}$$

COMPLESSITÀ T(n)

$\mathcal{O}(2^n)$

- Ho un lavoro di combinazione costante = b
- Faccio due chiamate su due insiemi più piccoli
- $T(n-1) + T(n-2)$ è come avere $2T(n-1)$
- Tutte le chiamate di $\text{fibonacci}(n-2)$ sono anche dentro $\text{fibonacci}(n-1)$

ITERAZIONE-RICORSIONE

ESEMPIO APPLICAZIONE ALGORITMO

1° Passo: //calcolo su Fibonacci con $n=3$

- $n=0 \rightarrow$ No
 - $n=1 \rightarrow$ No
 - $\text{primaRicorsione}(3-1)+\text{secondaRicorsione}(3-2)$
-

2° Passo // $n=2$

- Sono nella `primaRicorsione`

- $n=0 \rightarrow$ No
 - $n=1 \rightarrow$ No
 - $\text{prima.primaRicorsione}(2-1)+\text{prima.secondaRicorsione}(2-2)$
-

2.1° Passo // $n=1$

- Sono nella `prima.primaRicorsione`

- $n=0 \rightarrow$ No
 - $n=1 \rightarrow$ Si \rightarrow Ritorno 1 al passo 2° (chiudendo la ricorsione)
-

2.2° Passo // $n=0$

- Sono nella `prima.secondaRicorsione`

- $n=0 \rightarrow$ Si \rightarrow Ritorno 0 al passo 2° (chiudendo la ricorsione)
-

Ritorno al 2° Passo

- Ritorno 1 + 0 al 1° passo (chiudendo la `primaRicorsione`)
-

3° Passo // $n=1$

- Sono nella `secondaRicorsione`

- $n=0 \rightarrow$ No
 - $n=1 \rightarrow$ Ritorno 1 al passo 1° (chiudendo la `secondaRicorsione`)
-

Ritorno al 1° Passo

- Ritorno $(1 + 0) + 1 = 2$ chiudendo tutta la funzione e alla posizione 3 della serie di Fibonacci ho il valore 2 per come è definita
-

ITERAZIONE-RICORSIONE

SERIE DI FIBONACCI ITERATIVA

```
int fibonacci(int n){  
    int k;  
    int j=0;  
    int f=1;  
    for(int i=1; i<=n; i++){  
        k=j;  
        j=f;  
        f=k+j;  
    }  
    return j;  
}
```

COMPLESSITÀ T(n)

O(n)

- Faccio una somma ad ogni passo e i passi in totale sono n

CONCLUSIONI:

- La complessità nella funzione iterativa è inferiore a quella ricorsiva ma utilizzo un minimo di memoria in più per la variabile k
- Nella funzione ricorsiva non ho aumento della memoria

ITERAZIONE-RICORSIONE

ESEMPIO APPLICAZIONE ALGORITMO

1° Passo: //calcolo su Fibonacci con $n=3$

- k
 - $j=0$
 - $f=1$
 - Ciclo con $i=1$
-

1.1° Passo: // $n=3, i=1, j=0, f=1$

- $i \leq n \rightarrow 1 \leq 3$ Sì
 - $k=j=0$
 - $j=f=1$
 - $f=k+j=0+1 = 1$
 - $i=2$ ($i++$)
-

1.2° Passo: // $n=3, i=2, j=1, f=1$

- $i \leq n \rightarrow 2 \leq 3$ Sì
 - $k=j=1$
 - $j=f=1$
 - $f=k+j=1+1 = 2$
 - $i=3$ ($i++$)
-

1.3° Passo: // $n=3, i=3, j=1, f=2$

- $i \leq n \rightarrow 3 \leq 3$ Sì
 - $k=j=1$
 - $j=f=2$
 - $f=k+j=1+2 = 3$
 - $i=4$ ($i++$)
-

1.4° Passo: // $n=3, i=4, j=2, f=3$

- $i \leq n \rightarrow 4 \leq 3$ No esco dal ciclo
- Ritorno j che è 2 chiudendo tutta la funzione e alla posizione 3 della serie di fibonacci ho il valore 2 per come è definita

LINEAR SEARCH

RICERCA LINEARE (ITERATIVA)

```
int linearSearch(int A[], int n, int x){  
    int b=0;  
    for(int i=0; !b && (i<n); i++)  
        if(A[i] == x)  
            b=1;  
    return b;  
}
```

FUNZIONAMENTO:

- Si controlla se un elemento x sta nell'array
- Si utilizza la variabile b per indicare che ho trovato l'elemento con 1, 0 altrimenti
- Scorro l'array e controllo:
 - Finchè l'elemento b non mi si aggiorna → !b vuol dire !0, cioè finchè ancora non è diventato 1 che vuol dire che ancora non l'ho trovato
 - Finchè non scorro tutto l'array ($i < n$)
- Queste due condizioni devono essere vere contemporaneamente ($\&\&$) per continuare il ciclo
- Se una delle due condizioni diventa falsa vuol dire che esco dal ciclo perché
 - Se diventa falsa la prima → Ho trovato l'elemento
 - Se diventa falsa la seconda → Non ho trovato l'elemento e ho scorso tutto l'array

Complessità

O(n)

- Nel caso peggiore scorro tutto l'array per trovare l'elemento

LINEAR SEARCH

ESEMPIO FUNZIONAMENTO ALGORITMO

- Considero un vettore A[] di 4 elementi definito così → n=4

- Considero che l'elemento da cercare sia 5 → x=5

| Indice → i | A[0] | A[1] | A[2] | A[3] |
|-----------------|------|------|------|------|
| Elementi → A[i] | 3 | 2 | 5 | 1 |

1° Passo: //n=4, x=5

b=0

Parte il ciclo, i=0

1.1° Passo: //n=4, x=5, b=0, i=0

Ciclo:

- !b && i<n → !0 && 0<4 Sì
 - A[i]=x → A[0]=x → 3=5 No
 - i=1 (i++)
-

1.2° Passo: //n=4, x=5, b=0, i=1

Ciclo:

- !b && i<n → !1 && 1<4 Sì
 - A[i]=x → A[1]=x → 2=5 No
 - i=2 (i++)
-

1.3° Passo: //n=4, x=5, b=0, i=2

Ciclo:

- !b && i<n → !2 && 2<4 Sì
 - A[i]=x → A[2]=x → 5=5 Sì → b=1
 - i=3 (i++)
-

1.4° Passo: //n=4, x=5, b=1, i=3

Ciclo:

- !b && i<n → !3 && 3<4 No → b=1 Esco dal ciclo
 - Ritorno 1 il che vuol dire che ho trovato l'elemento
-

LINEAR SEARCH

RICERCA LINEARE (RICORSIVA)

```
int RlinearSearch(int A[], int x, int n, int i=0){  
    if(i==n)  
        return 0;  
    if(A[i]==x)  
        return 1;  
    return RlinearSearch(A, x, n, i+1);  
}
```

//tra i parametri, si mette solo int i → Lo 0 si mette alla chiamata di questa funzione, tra i parametri nel main

FUNZIONAMENTO:

- Primo caso base della funzione ricorsiva, è che l'array abbia un solo elemento e ci arrivo alla fine di tutte le chiamate ricorsive → E questa cosa mi dice che sono arrivato in fondo all'array senza trovare l'elemento
- Secondo caso base della funzione ricorsiva, è che abbia trovato l'elemento
→ Ritorno 1
- Richiamo la ricorsione su un numero inferiore di elementi ($n-1$) poiché aumento l'indice i di 1
- Faccio la chiamata ricorsiva finché non ricado su uno dei due casi base scritti precedentemente

RELAZIONE DI RICORRENZA

$$\begin{aligned} T(0) &= a \\ T(n) &= b + T(n-1) \end{aligned}$$

COMPLESSITÀ $T(n)$

$$O(n)$$

LINEAR SEARCH

ESEMPIO FUNZIONAMENTO ALGORITMO

- Considero un vettore A[] di 4 elementi definito così → n=4
- Considero che l'elemento da cercare sia 5 → x=5
- Da i parametri gli viene passato i=0

| Indice → i | A[0] | A[1] | A[2] | A[3] |
|-----------------|------|------|------|------|
| Elementi → A[i] | 3 | 2 | 5 | 1 |

1° Passo: //n=4, x=5, i=0

- Primo caso base i=n → 0=4 No
 - Secondo caso base A[i]=x → A[0]=x → 3=5 No
 - Ricorsione con i=1 (i+1)
-

1.1° Passo: //n=4, x=5, i=1

- Primo caso base i=n → 1=4 No
 - Secondo caso base A[i]=x → A[1]=x → 2=5 No
 - Ricorsione con i=2 (i+1)
-

1.1.1° Passo: //n=4, x=5, i=2

- Primo caso base i=n → 2=4 No
 - Secondo caso base A[i]=x → A[2]=x → 5=5 Sì
 - Ho trovato l'elemento
 - Chiudo questa chiamata ricorsiva e restituisco 1 al passo 1.1°
-

1.2° Passo

- Chiudo anche questa chiamata ricorsiva e restituisco 1 al passo 1°
-

2° Passo

- Chiudo l'intera funzione restituendo 1 (elemento trovato)

BINSEARCH

```
int binSearch(InfoType *A, InfoType x, int i=0, int j=n-1){           //0
    if(i>j)      return 0;                                         //1
    int k=(i+j)/2;                                              //2
    if(x == A[k])      return 1;                                         //3
    if(x < A[k])      return binSearch(A, x, i, k-1);                //4
    else return binSearch(A, x, k+1, j);                           //5
}
```

//tra i parametri, si mette solo int i e int j → Lo 0 e (n-1) si mettono alla chiamata di questa funzione, tra i parametri nel main

InfoType è un tipo generico che può rappresentare qualsiasi tipo che voglio utilizzare

Complessità

O(log n)

//o vado a SX o vado a DX dell'array

PROCEDIMENTO:

- 0- Gli indici passati sono i per il primo vettore e j per l'ultimo vettore
(visto che suddividerò l'array in 2 sottoarray, divisi dall'elemento centrale ottenuto al passaggio 2)
- 1- Caso di uscita poiché sono già andato oltre la lunghezza del vettore
→ Si scambiano l'inizio e fine del vettore
- 2- Elemento al centro
- 3- Caso in cui ho già trovato il mio elemento esattamente nel centro
- 4- Caso in cui vado a scansionare la parte a SX dell'array
- 5- Caso in cui vado a scansionare la parte a DX dell'array

OSS

Per fare questa ricerca si presuppone che l'array sia già dato in partenza ordinato in modo crescente

BINSEARCH

ESEMPIO FUNZIONAMENTO ALGORITMO

- Considero un vettore ordinato A[] di 4 elementi definito così → n=4
- Considero che l'elemento da cercare sia 5 → x=5
- La partenza è a i=0 e j=n-1=3

| Indice → i | A[0] | A[1] | A[2] | A[3] |
|-----------------|------|------|------|------|
| Elementi → A[i] | 1 | 2 | 3 | 5 |

1° Passo: //j=3, i=0, x=5

- $i > j \rightarrow 0 > 3$ No
 - $k = (i+j)/2 \rightarrow (0+3)/2 \rightarrow 3/2 = 1$ (divisione intera)
 - $x = A[k] \rightarrow 5 = A[1] \rightarrow 5 = 2$ No
 - $x < A[k] \rightarrow 5 < A[1] \rightarrow 5 = 2$ No
 - Ricorsione su metà dell'array DX passando come $i = k+1 = 2$
-

1.1° Passo: //j=3, i=2, x=5

- $i > j \rightarrow 2 > 3$ No
 - $k = (i+j)/2 \rightarrow (2+3)/2 \rightarrow 5/2 = 2$ (divisione intera)
 - $x = A[k] \rightarrow 5 = A[2] \rightarrow 5 = 3$ No
 - $x < A[k] \rightarrow 5 < A[2] \rightarrow 5 < 3$ No
 - Ricorsione su metà dell'array DX passando come $i = k+1 = 3$
-

1.1.1° Passo: //j=3, i=3, x=5

- $i > j \rightarrow 3 > 3$ No
- $k = (i+j)/2 \rightarrow (3+3)/2 \rightarrow 6/2 = 3$ (divisione intera)
- $x = A[k] \rightarrow 5 = A[3] \rightarrow 5 = 5$ Sì
- Ritorno 1 (elemento trovato) chiudendo le chiamate ricorsive e tutta la funzione

EXCHANGE

```
void exchange(int &x, int &y){  
    int temp=x;  
    x=y;  
    y=temp;  
}
```

Complessità

O(1)

- Semplice funzione che prende dai parametri due valori e attraverso una variabile di appoggio li restituisce scambiati
- Il fatto che tornino cambiati è dovuto grazie al riferimento → &

SELECTION SORT (Iterativa)

```
void selectionSort(int A[], int n){  
    for(int i=0; i<n-1; i++){  
        //O(n)  
        int min=i;  
        for(int j=i+1; j<n; j++)  
            if(A[j] < A[min])  
                min=j;  
        exchange(A[i], A[min]);  
        //fuori dal for interno  
    }  
}
```

Complessità

$O(n^2)$

//Numero di scambi $O(n)$

“L’algoritmo ordina in modo crescente procedendo così:

1. Considera il primo indice come indice dell’elemento minimo
2. Confronta l’elemento dell’indice considerato con gli elementi di indice successivo
3. Scorrendo il restante array se trova un elemento più piccolo di esso, si scambia di posizione (così lo mette nella posizione più piccola)
4. Se non ha scambiato di posizione, vuol dire che l’elemento è già il più piccolo, e di conseguenza è nella posizione giusta
5. Aumento l’indice di 1 e lo considero come indice del secondo elemento più piccolo ripartendo dal punto 2... e così via per il terzo ecc...”

SELECTION SORT (Iterativa)

ESEMPIO FUNZIONAMENTO ALGORITMO

- Considero un vettore A[] di 4 elementi definito così → n=4

| Indice → i | A[0] | A[1] | A[2] | A[3] |
|-----------------|------|------|------|------|
| Elementi → A[i] | 3 | 2 | 5 | 1 |

1° Passo: //n=4

Ciclo esterno:

- i=0
 - i<3 → 0<3 Sì
// Non arrivo all'ultimo elemento perché l'algoritmo alla fine l'ultimo elemento lo avrà posizionato al punto giusto, quindi salto un passaggio
 - min=i=0
-

1.1° Passo:

Ciclo interno: //min=0, i=0

- j=i+1=1
 - j<4 → 1<4 Sì
 - if (A[j]<A[min]) → A[1]<A[0] → 2<3 Sì
 - min=j=1
//in questo momento il minimo è il valore 2
 - j++ → 2
-

1.2° Passo:

Ciclo interno: //min=1, i=0, j=2

- j<4 → 2<4 Sì
 - if (A[j]<A[min]) → A[2]<A[1] → 5<2 No
 - j++ → 3
-

1.3° Passo:

Ciclo interno: //min=1, i=0, j=3

- j<4 → 3<4 Sì
 - if (A[j]<A[min]) → A[3]<A[1] → 1<2 Sì
 - min=j=3
//in questo momento il minimo è il valore 1
 - j++ → 4
-

SELECTION SORT (Iterativa)

... CONTINUO ESEMPIO FUNZIONAMENTO ALGORITMO

1.4° Passo:

Ciclo interno: // $min=3, i=0, j=4$

- $j < 4 \rightarrow 4 < 4$ No → Esci dal ciclo interno
- Scambio $A[i]$ con $A[min]$ → $A[0]$ con $A[3]$ → 1 con 3
(VEDI SCHEMA PRECEDENTE COME ERANO PRIMA DELLO SCAMBIO)

OTTENGO:

| Indice → i | $A[0]$ | $A[1]$ | $A[2]$ | $A[3]$ |
|-------------------|--------|--------|--------|--------|
| Elementi → $A[i]$ | 1 | 2 | 5 | 3 |

// L'elemento 1 è posizionato correttamente in maniera ordinata

- $i=1 (i++)$
-

2° Passo:

// $n=4, i=1$

Ciclo esterno:

- $i < 3 \rightarrow 1 < 3$ Sì
 - $min=i=1$
-

2.1° Passo:

Ciclo interno: // $min=1, i=1$

- $j=i+1=2$
 - $j < 4 \rightarrow 2 < 4$ Sì
 - if ($A[j] < A[min]$) → $A[2] < A[1]$ → 5 < 2 No
 - $j++ \rightarrow 3$
-

2.2° Passo:

Ciclo interno: // $min=1, i=1, j=3$

- $j < 4 \rightarrow 3 < 4$ Sì
- if ($A[j] < A[min]$) → $A[3] < A[1]$ → 3 < 2 No
- $j++ \rightarrow 4$

SELECTION SORT (Iterativa)

... CONTINUO ESEMPIO FUNZIONAMENTO ALGORITMO

2.3° Passo:

Ciclo interno: // $min=1, i=1, j=4$

- $j < 4 \rightarrow 4 < 4$ No \rightarrow Esco dal ciclo interno
- Scambio $A[i]$ con $A[min] \rightarrow A[1]$ con $A[1] \rightarrow 2$ con 2, l'array è invariato
(VEDI SCHEMA PRECEDENTE COME ERANO PRIMA DELLO SCAMBIO)

OTTENGO:

| Indice $\rightarrow i$ | $A[0]$ | $A[1]$ | $A[2]$ | $A[3]$ |
|-----------------------------|--------|--------|--------|--------|
| Elementi $\rightarrow A[i]$ | 1 | 2 | 5 | 3 |

// In poche parole ho scambiato 2 con sé stesso, è come se non avessi fatto scambi

// L'elemento 1 e 2 sono posizionati correttamente in maniera ordinata

- $i=2 (i++)$
-

3° Passo:

// $n=4, i=2$

Ciclo esterno:

- $i < 3 \rightarrow 2 < 3$ Sì
 - $min=i=2$
-

3.1° Passo:

Ciclo interno: // $min=2, i=2$

- $j=i+1=3$
 - $j < 4 \rightarrow 3 < 4$ Sì
 - if ($A[j] < A[min]$) $\rightarrow A[3] < A[2] \rightarrow 3 < 5$ Sì
 - $min=j=3$
 - $j++ \rightarrow 4$
- //in questo momento il minimo è il valore 3

SELECTION SORT (Iterativa)

... CONTINUO ESEMPIO FUNZIONAMENTO ALGORITMO

3.2° Passo:

Ciclo interno: // $min=3, i=2, j=4$

- $j < 4 \rightarrow 4 < 4$ No \rightarrow Esco dal ciclo interno
- Scambio $A[i]$ con $A[min] \rightarrow A[2]$ con $A[3] \rightarrow 5$ con 3
(VEDI SCHEMA PRECEDENTE COME ERANO PRIMA DELLO SCAMBIO)

OTTENGO:

| Indice $\rightarrow i$ | $A[0]$ | $A[1]$ | $A[2]$ | $A[3]$ |
|-----------------------------|--------|--------|--------|--------|
| Elementi $\rightarrow A[i]$ | 1 | 2 | 3 | 5 |

// L'elemento 1,2 e 3 sono posizionati correttamente in maniera ordinata, anche il 5 lo sarebbe ma vanno considerati tutti i passaggi dei cicli

- $i=3$ ($i++$)
-

4° Passo:

// $n=4, i=3$

Ciclo esterno:

- $i < 3 \rightarrow 3 < 3$ No \rightarrow Esco dal ciclo esterno e dall'algoritmo con l'array ordinato

| Indice $\rightarrow i$ | $A[0]$ | $A[1]$ | $A[2]$ | $A[3]$ |
|-----------------------------|--------|--------|--------|--------|
| Elementi $\rightarrow A[i]$ | 1 | 2 | 3 | 5 |

SELECTION SORT (Ricorsiva)

```
void r_selectionSort(int *A, int n, int i=0){ //0
    if(i==n-1) return; //1
    int min=i;
    for(int j=i+1; j<n; j++)
        if(A[j]<A[min])
            min=j;
    exchange(A[i], A[min]);
    r_selectionSort(A, n, i+1) //2
}
```

//tra i parametri, si mette solo int i → Lo 0 si mette alla chiamata di questa funzione, tra i parametri nel main

RELAZIONE DI RICORRENZA

COMPLESSITÀ T(n)

$$\begin{aligned}T(1) &= a \\T(n) &= bn + T(n-1)\end{aligned}$$

$$O(n^2)$$

- Considerando il ciclo esterno della funzione iterativa

→ `for(int i=0; i<n-1; i++)`

- int i=0 è sostituita nella ricorsione dal codice in //0
- i<n-1 è sostituita nella ricorsione dal codice in //1
- i++ è sostituita nella ricorsione dal codice in //2

SELECTION SORT (Ricorsiva)

ESEMPIO FUNZIONAMENTO ALGORITMO

- Considero un vettore A[] di 4 elementi definito così → n=4

| Indice → i | A[0] | A[1] | A[2] | A[3] |
|-----------------|------|------|------|------|
| Elementi → A[i] | 3 | 2 | 5 | 1 |

1° Passo: //i=0 passato tra i parametri, n=4

- $i = n-1 \rightarrow 0 = 3$ No
 - $\min = i = 0$
-

2° Passo:

Ciclo: // $\min = 0, i = 0$

- $j = i + 1 = 1$
 - $j < 4 \rightarrow 1 < 4$ Sì
 - if ($A[j] < A[\min]$) → $A[1] < A[0] \rightarrow 2 < 3$ Sì
 - $\min = j = 1$ //in questo momento il minimo è il valore 2
 - $j++ \rightarrow 2$
-

2.1° Passo:

Ciclo: // $\min = 1, i = 0, j = 2$

- $j < 4 \rightarrow 2 < 4$ Sì
 - if ($A[j] < A[\min]$) → $A[2] < A[1] \rightarrow 5 < 2$ No
 - $j++ \rightarrow 3$
-

2.2° Passo:

Ciclo: // $\min = 1, i = 0, j = 3$

- $j < 4 \rightarrow 3 < 4$ Sì
- if ($A[j] < A[\min]$) → $A[3] < A[1] \rightarrow 1 < 2$ Sì
- $\min = j = 3$ //in questo momento il minimo è il valore 1
- $j++ \rightarrow 4$

SELECTION SORT (Ricorsiva)

... CONTINUO ESEMPIO FUNZIONAMENTO ALGORITMO

2.3° Passo:

Ciclo: // $min=3, i=0, j=4$

- $j < 4 \rightarrow 4 < 4$ No \rightarrow Esco dal ciclo
- Scambio $A[i]$ con $A[min] \rightarrow A[0]$ con $A[3] \rightarrow 1$ con 3
(VEDI SCHEMA PRECEDENTE COME ERANO PRIMA DELLO SCAMBIO)

OTTENGO:

| Indice $\rightarrow i$ | $A[0]$ | $A[1]$ | $A[2]$ | $A[3]$ |
|-----------------------------|--------|--------|--------|--------|
| Elementi $\rightarrow A[i]$ | 1 | 2 | 5 | 3 |

// L'elemento 1 è posizionato correttamente in maniera ordinata

- Ricorsione con i aumentato di 1 $\rightarrow i=1$
-

3° Passo:

// $n=4, i=1$

Prima chiamata Ricorsiva

- $i=n-1 \rightarrow 1=3$ No
 - $min=i=1$
-

4° Passo:

Ciclo: // $min=1, i=1$

- $j=i+1=2$
 - $j < 4 \rightarrow 2 < 4$ Si
 - if ($A[j] < A[min]$) $\rightarrow A[2] < A[1] \rightarrow 5 < 2$ No
 - $j++ \rightarrow 3$
-

4.1° Passo:

Ciclo: // $min=1, i=1, j=3$

- $j < 4 \rightarrow 3 < 4$ Si
- if ($A[j] < A[min]$) $\rightarrow A[3] < A[1] \rightarrow 3 < 2$ No
- $j++ \rightarrow 4$

SELECTION SORT (Ricorsiva)

... CONTINUO ESEMPIO FUNZIONAMENTO ALGORITMO

4.2° Passo:

Ciclo: //min=1, i=1, j=4

- $j < 4 \rightarrow 4 < 4$ No \rightarrow Esco dal ciclo
- Scambio $A[i]$ con $A[min]$ $\rightarrow A[1]$ con $A[1]$ $\rightarrow 2$ con 2
(VEDI SCHEMA PRECEDENTE COME ERANO PRIMA DELLO SCAMBIO)

OTTENGO:

| Indice $\rightarrow i$ | $A[0]$ | $A[1]$ | $A[2]$ | $A[3]$ |
|-----------------------------|--------|--------|--------|--------|
| Elementi $\rightarrow A[i]$ | 1 | 2 | 5 | 3 |

// In poche parole ho scambiato 2 con sé stesso, è come se non avessi fatto scambi

// L'elemento 1 e 2 sono posizionati correttamente in maniera ordinata

- Ricorsione con i aumentato di 1 $\rightarrow i=2$
-

5° Passo:

// $n=4$, $i=2$

Seconda chiamata Ricorsiva

- $i=n-1 \rightarrow 2=3$ No
 - $min=i=2$
-

6° Passo:

Ciclo: // $min=2$, $i=2$

- $j=i+1=3$
 - $j < 4 \rightarrow 3 < 4$ Si
 - if ($A[j] < A[min]$) $\rightarrow A[3] < A[2] \rightarrow 3 < 5$ Si
 - $min=j=3$
 - $j++ \rightarrow 4$
- //in questo momento il minimo è il valore 3

SELECTION SORT (Ricorsiva)

... CONTINUO ESEMPIO FUNZIONAMENTO ALGORITMO

6.1° Passo:

Ciclo: //min=3, i=2, j=4

- $j < 4 \rightarrow 4 < 4$ No \rightarrow Esco dal ciclo
- Scambio $A[i]$ con $A[min] \rightarrow A[2]$ con $A[3] \rightarrow 5$ con 3
(VEDI SCHEMA PRECEDENTE COME ERANO PRIMA DELLO SCAMBIO)

OTTENGO:

| Indice $\rightarrow i$ | $A[0]$ | $A[1]$ | $A[2]$ | $A[3]$ |
|-----------------------------|--------|--------|--------|--------|
| Elementi $\rightarrow A[i]$ | 1 | 2 | 3 | 5 |

// L'elemento 1,2 e 3 sono posizionati correttamente in maniera ordinata, anche il 5 lo sarebbe ma vanno considerato anche il resto del codice (chiamata ricorsiva)

- Ricorsione con i aumentato di 1 $\rightarrow i=3$
-

7° Passo:

// $n=4$, $i=3$

Terza chiamata Ricorsiva

- $i=n-1 \rightarrow 3=3$ Sì \rightarrow Faccio return e chiudo tutte le chiamate ricorsive, ottenendo l'array ordinato

| Indice $\rightarrow i$ | $A[0]$ | $A[1]$ | $A[2]$ | $A[3]$ |
|-----------------------------|--------|--------|--------|--------|
| Elementi $\rightarrow A[i]$ | 1 | 2 | 3 | 5 |

BUBBLESORT

```
void bubbleSort(int A[], int n){  
    for(int i=0; i<n-1; i++)  
        for(int j=n-1; j>=i+1; j--)  
            if(A[j]<A[j-1])  
                exchange(A[j], A[j-1]);  
}
```

Numero di scambi $\rightarrow O(n^2)$

Complessità

O(n^2)

“L’algoritmo ordina in modo crescente procedendo così:

1. Prende l’ultimo elemento dell’array
2. Lo confronta con i precedenti (penultimo, terzultimo ecc..), quindi scorre l’array dalla fine verso l’inizio
3. Se trova un elemento più grande di esso, si scambia di posizione
4. Se invece, si ferma in quella posizione questo accade perché:
 - Ha trovato un elemento più piccolo di esso
 - È arrivato alla prima posizione dell’array (più piccola) e di conseguenza è l’elemento più piccolo dell’array.
5. Si riparte dal punto due considerando il penultimo elemento dell’array e così via, per il terzultimo ecc...”

BUBBLE SORT

ESEMPIO FUNZIONAMENTO ALGORITMO

- Considero un vettore A[] di 4 elementi definito così → n=4

| Indice → i | A[0] | A[1] | A[2] | A[3] |
|-----------------|------|------|------|------|
| Elementi → A[i] | 3 | 2 | 5 | 1 |

1° Passo: //n=4

Ciclo esterno: i=0

- $i < n-1 \rightarrow 0 < 3$ Sì
// Non arrivo all'ultimo elemento perché l'algoritmo fa sì che ci arrivi tramite l'indice del ciclo interno
-

1.1° Passo:

Ciclo interno: //i=0, n=4

- $j = n-1 = 3$
- $j >= i+1 \rightarrow 3 >= 1$ Sì
- $\text{if}(A[j] < A[j-1]) \rightarrow A[3] < A[2] \rightarrow 1 < 5$ Sì
- Li scambio tra loro, A[3] con A[2], cioè 1 con il 5
(VEDI SCHEMA PRECEDENTE COME ERANO PRIMA DELLO SCAMBIO)

OTTENGO:

| Indice → i | A[0] | A[1] | A[2] | A[3] |
|-----------------|------|------|------|------|
| Elementi → A[i] | 3 | 2 | 1 | 5 |

- $j = 2$ ($j--$)
-

1.2° Passo:

Ciclo interno: //i=0, n=4, j=2

- $j >= i+1 \rightarrow 2 >= 1$ Sì
- $\text{if}(A[j] < A[j-1]) \rightarrow A[2] < A[1] \rightarrow 1 < 2$ Sì
- Li scambio tra loro, A[2] con A[1], cioè 1 con il 2
(VEDI SCHEMA PRECEDENTE COME ERANO PRIMA DELLO SCAMBIO)

OTTENGO:

| Indice → i | A[0] | A[1] | A[2] | A[3] |
|-----------------|------|------|------|------|
| Elementi → A[i] | 3 | 1 | 2 | 5 |

- $j = 1$ ($j--$)
-

BUBBLE SORT

... CONTINUO ESEMPIO FUNZIONAMENTO ALGORITMO

1.3° Passo:

Ciclo interno: // $i=0, n=4, j=1$

- $j \geq i+1 \rightarrow 1 \geq 1$ Sì
- if($A[j] < A[j-1]$) $\rightarrow A[1] < A[0] \rightarrow 1 < 3$ Sì
- Li scambio tra loro, $A[1]$ con $A[0]$, cioè 1 con il 3
(VEDI SCHEMA PRECEDENTE COME ERANO PRIMA DELLO SCAMBIO)

OTTENGO:

| Indice $\rightarrow i$ | $A[0]$ | $A[1]$ | $A[2]$ | $A[3]$ |
|-----------------------------|--------|--------|--------|--------|
| Elementi $\rightarrow A[i]$ | 1 | 3 | 2 | 5 |

//Ho il valore 1 sistemato al suo posto in maniera corretta in ordine crescente

-
- $j=0$ ($j--$)
-

1.4° Passo:

Ciclo interno: // $i=0, n=4, j=0$

- $j \geq i+1 \rightarrow 0 \geq 1$ No \rightarrow Esco dal ciclo interno
 - $i=1$ ($i++$)
-

2° Passo: // $n=4, i=1$

Ciclo esterno:

- $i < n-1 \rightarrow 1 < 3$ Sì
-

2.1° Passo:

Ciclo interno: // $i=1, n=4$

- $j=n-1=3$
- $j \geq i+1 \rightarrow 3 \geq 2$ Sì
- if($A[j] < A[j-1]$) $\rightarrow A[3] < A[2] \rightarrow 5 < 2$ No
- $j=2$ ($j--$)

BUBBLE SORT

... CONTINUO ESEMPIO FUNZIONAMENTO ALGORITMO

2.2° Passo:

Ciclo interno: // $i=1, n=4, j=2$

- $j \geq i+1 \rightarrow 2 \geq 2$ Sì
- if($A[j] < A[j-1]$) $\rightarrow A[2] < A[1] \rightarrow 2 < 3$ Sì
- Li scambio tra loro, $A[2]$ con $A[1]$, cioè 2 con il 3
(VEDI SCHEMA PRECEDENTE COME ERANO PRIMA DELLO SCAMBIO)

OTTENGO:

| Indice $\rightarrow i$ | $A[0]$ | $A[1]$ | $A[2]$ | $A[3]$ |
|-----------------------------|--------|--------|--------|--------|
| Elementi $\rightarrow A[i]$ | 1 | 2 | 3 | 5 |

//Ho i valori 1 e 2 sistemati al loro posto in maniera corretta in ordine crescente
(anche il 5 lo è ma devo terminare i cicli)

- $j=1 (j--)$
-

2.3° Passo:

Ciclo interno: // $i=1, n=4, j=1$

- $j \geq i+1 \rightarrow 1 \geq 2$ No \rightarrow Esco dal ciclo interno
 - $i=2 (i++)$
-

3° Passo: // $n=4, i=2$

Ciclo esterno:

- $i < n-1 \rightarrow 2 < 3$ Sì
-

3.1° Passo:

Ciclo interno: // $i=2, n=4$

- $j=n-1=3$
- $j \geq i+1 \rightarrow 3 \geq 3$ Sì
- if($A[j] < A[j-1]$) $\rightarrow A[3] < A[2] \rightarrow 5 < 2$ No
- $j=2 (j--)$

BUBBLE SORT

... CONTINUO ESEMPIO FUNZIONAMENTO ALGORITMO

3.2° Passo:

Ciclo interno: // $i=2, n=4, j=2$

- $j >= i+1 \rightarrow 2 >= 3$ No \rightarrow Esco dal ciclo interno
 - $i = 3$ ($i++$)
-

4° Passo: // $n=4, i=3$

Ciclo esterno:

- $i < n-1 \rightarrow 3 < 3$ No \rightarrow Esco anche dal ciclo interno ed ho l'array ordinato così:

| Indice $\rightarrow i$ | $A[0]$ | $A[1]$ | $A[2]$ | $A[3]$ |
|-----------------------------|--------|--------|--------|--------|
| Elementi $\rightarrow A[i]$ | 1 | 2 | 3 | 5 |

QUICKSORT

```
void quickSort(int A[], int inf=0, int sup=n-1){  
    int perno=A[(inf+sup)/2];  
    int s=inf, d=sup;  
    while(s<=d){  
        while(A[s]<perno)  
            s++;  
        while(A[d]>perno)  
            d--;  
        if(s>d)  
            break;  
        exchange(A[s], A[d]);  
        s++;  
        d--;  
    }  
    if(inf<d)  
        quickSort(A, inf, d);  
    if(s<sup)  
        quickSort(A, s, sup);  
}
```

//tra i parametri, si mette solo int inf e int sup → Lo 0 e n-1 si mettono alla chiamata di questa funzione, tra i parametri nel main

OSS:

- Inizialmente gli passo n-1 come sup poiché gli elementi sono n e vanno da 0 a n-1, di conseguenza il sup deve rappresentare proprio l'ultimo elemento

QUICKSORT

“L’algoritmo ordina in modo crescente procedendo così:

1. Inizialmente scelgo un perno (come elemento centrale)
2. Scelgo 2 indici s=estremo inferiore e d=estremo superiore, dove l'estremo inferiore è inf passato tra i parametri e l'estremo superiore è sup passato anch'esso tra i parametri
3. Finchè i due indici non si scambiano tra loro eseguo i seguenti passaggi:
 - Controllo verso DX (\rightarrow) aumentando s di 1, se l'elemento dell'array in posizione s rimane più piccolo del perno. Se non lo è, esco dal primo while (interno) e vado al while (interno) successivo, tenendomi “da parte” l'elemento che ho appena trovato più grande del perno in posizione s
 - Entro nel secondo while interno e: Controllo verso SX(\leftarrow) diminuendo d di 1, se l'elemento dell'array in posizione d rimane più grande del perno. Se non lo è esco dal secondo while (interno) e vado alle istruzioni successive, tenendomi anche qua “da parte” l'elemento che ho appena trovato più piccolo del perno in posizione d
4. Se, controllando verso DX aumentando s, e verso SX diminuendo d, non trovo un elemento più grande del perno o più piccolo del perno, s e d si scambiando tra loro \rightarrow Quindi $s > d \rightarrow$ Esco con l'istruzione break (il che vuol dire che sono già ordinati)
5. Se il punto 4 non viene eseguito, i valori di indice s e d dell'array tenuti “da parte” si scambiano (visto che a SX del perno devo avere valori $<$ ad esso e a DX invece, maggiori ad esso)
6. Diminuendo d ed aumentando s, si ripete il punto 5, fino a quando non si ricade nel punto 4
7. Ora tutto a SX del perno è più piccolo del perno e tutto a DX del perno è più grande del perno
8. All’uscita del primo while controllo se $inf < d$ (questo mi dice che l’array a SX del perno è da ordinare), e se così fosse richiamo *ricorsivamente* la funzione nella sottoporzione di array di SX con un nuovo perno ecc.. (dal punto 1)
9. Appena finisco la chiamata ricorsiva controllo se $s < sup$ (questo mi dice che l’array a DX del perno è da ordinare), e se così fosse anche qui richiamo *ricorsivamente* la funzione nella sottoporzione di array di DX con un nuovo perno ecc..(dal punto 1)
10. Alla fine di entrambe le chiamate ricorsive ho che inf e sup si sono scambiati (tramite s e d), allora non ho più elementi da ordinare e quindi l’array è ordinato (in ordine crescente)”

QUICKSORT

Il numero di chiamate che faccio ricorsivamente dipende da dove viene posizionato il perno (chiamo K):

K=1 (CASO PEGGIORE)

RELAZIONE DI RICORRENZA COMPLESSITÀ T(n)

$$\begin{aligned} T(0) &= a \\ T(n) &= bn + T(n-1) \end{aligned} \longrightarrow O(n^2)$$

K=n/2 (CASO MIGLIORE)

RELAZIONE DI RICORRENZA COMPLESSITÀ T(n)

$$\begin{aligned} T(1) &= a \\ T(n) &= bn + 2T(n/2) \end{aligned} \longrightarrow O(n \log n)$$

(CASO MEDIO)

RELAZIONE DI RICORRENZA COMPLESSITÀ T(n)

$$\begin{aligned} T(1) &= a \\ T(n) &= bn + 2T(n/2) \end{aligned} \longrightarrow O(n \log n)$$

// La stessa nel caso medio e nel caso migliore

QUICKSORT

FUNZIONAMENTO ALGORITMO

Dato il seguente array di 5 elementi (n=5)

| | | | | | |
|----------------|----------|----------|----------|----------|----------|
| | 0 | 1 | 2 | 3 | 4 |
| Array A | 2 | 5 | 3 | 1 | 7 |

→ Chiamo `quickSort(A,0,4)` //4 viene da $n-1$

1° Passo (sistemo le posizioni) ponendo il perno = $A[(inf+sup)/2] = A[(0+4)/2] = A[2]=3$

- inf=0=s
- sup=4=d
- perno=3

| | | | | |
|----------|----------|----------|----------|----------|
| 0 | 1 | 2 | 3 | 4 |
| 2 | 5 | 3 | 1 | 7 |
| s | | | d | |

→ faccio il ciclo finchè $s \leq d$ ($0 \leq 4$) e faccio due cicli while in parallelo

2° Passo // $s=0, d=4, perno=3$

while $A[s] < perno \rightarrow A[0] < 3 \rightarrow 2 < 3$ sì $\rightarrow s++$

while $A[d] > perno \rightarrow A[4] > 3 \rightarrow 7 > 3$ sì $\rightarrow d--$

- $s=1$
- $d=3$
- $perno=3$

| | | | | |
|----------|----------|----------|----------|----------|
| 0 | 1 | 2 | 3 | 4 |
| 2 | 5 | 3 | 1 | 7 |
| s | | | d | |

QUICKSORT

... CONTINUO FUNZIONAMENTO ALGORITMO

3° Passo //s=1, d=3, perno=3

while A[s]<perno → A[1]<3 → 5<3 NO

while A[d]>perno → A[3]>3 → 1>3 NO

s>d → 1>3 NO

scambio A[s] con A[d] → A[1] con A[3] → 5 con 1

s++ d--

- s=2
- d=2
- perno=3

0 1 2 3 4
2 **1** **3** **5** 7
 s, d

4° Passo //s=2, d=2, perno=3

while A[s]<perno → A[2]<3 → 3<3 NO

while A[d]>perno → A[2]>3 → 3>3 NO

s>d → 3>3 NO

scambio A[s] con A[d] → A[2] con A[2] → 3 con 3

//scambio il perno con sé stesso

s++ d--

- s=3
- d=1
- perno=3

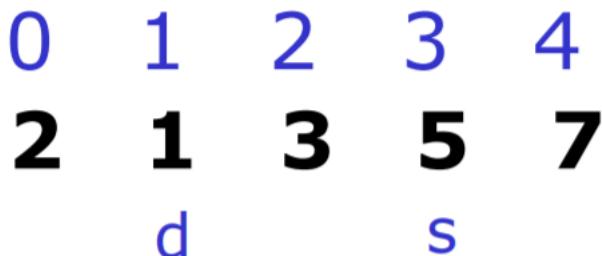
0 1 2 3 4
2 1 **3** **5** 7
 s, d

QUICKSORT

... CONTINUO FUNZIONAMENTO ALGORITMO

5° Passo //s=3, d=1, perno=3

Sono nel caso in cui $s > d$ ($3 > 1$ Sì) entro nella condizione dell'if e faccio break terminando il while



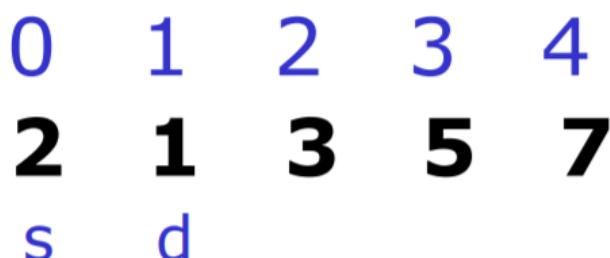
→ Tutto ciò che sta a SX del perno è più piccolo e tutto ciò che sta a DX del perno è più grande → Il perno è nella posizione giusta dell'ordinamento ma, devo vedere di ordinare il restante dei due sottoarray

6° Passo //s=3, d=1, inf=0, sup=4

if($inf < d$) → 0<1 Sì → quickSort(A,inf,d) → quickSort(A,0,1)

//inizio una serie di chiamate ricorsive

- $inf = 0 = s$
- $sup = 1 = d$
- $perno = A[(inf+sup)/2] = A[(0+1)/2] = A[0] = 2$ // $1+0/2 = 0$ divisione intera



→ faccio il ciclo finchè $s \leq d$ ($0 \leq 1$) e faccio due cicli while in parallelo

QUICKSORT

... CONTINUO FUNZIONAMENTO ALGORITMO

6.1° Passo // $s=0, d=1, perno=2, inf=0, sup=1$

while $A[s] < perno \rightarrow A[0] < 2 \rightarrow 2 < 2$ NO

while $A[d] > perno \rightarrow A[1] > 2 \rightarrow 1 > 2$ NO

$s > d \rightarrow 0 > 1$ NO

scambio $A[s]$ con $A[d]$ $\rightarrow A[0]$ con $A[1] \rightarrow 2$ con 1

$s++ d--$

- $s=1$
- $d=0$
- $perno=2$

| | | | | |
|----------|----------|----------|----------|----------|
| 0 | 1 | 2 | 3 | 4 |
| 1 | 2 | 3 | 5 | 7 |
| d | s | | | |

Sono nel caso in cui $s > d$ ($1 > 0$) entro nella condizione dell'if e faccio break terminando il while

if($inf < d$) $\rightarrow 0 < 0$ NO!

if($s < sup$) $\rightarrow 1 < 1$ NO!

\rightarrow Termino il punto 6°, con la parte SX del sottoarray iniziale ordinata

7° Passo // $s=3, d=1, inf=0, sup=4$

if($s < sup$) $\rightarrow 3 < 4$ Sì $\rightarrow quickSort(A, s, sup) \rightarrow quickSort(A, 3, 4)$

// inizio una serie di chiamate ricorsive

- $inf=3=s$
- $sup=4=d$
- $perno = A[(inf+sup)/2] = A[(3+4)/2] = A[3]=5$ // $3+4/2 = 3$ divisione intera

| | | | | |
|----------|----------|----------|----------|----------|
| 0 | 1 | 2 | 3 | 4 |
| 1 | 2 | 3 | 5 | 7 |
| | s | d | | |

\rightarrow faccio il ciclo finchè $s \leq d$ ($3 \leq 4$) e faccio due cicli while in parallelo

QUICKSORT

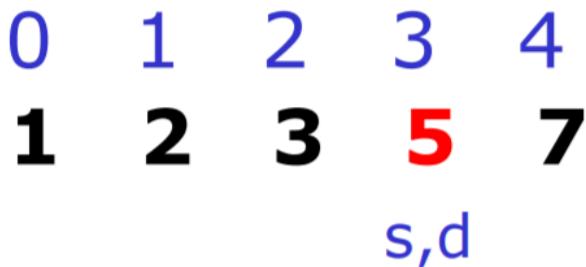
... CONTINUO FUNZIONAMENTO ALGORITMO

7.1° Passo //s=3, d=4, perno=5

while A[s]<perno → A[3]<5 → 5<5 NO

while A[d]>perno → A[4]>5 → 7>5 sì → d--

- s=3
- d=3
- perno=5



7.2° Passo //s=3, d=3, perno=5, inf=3, sup=4

while A[s]<perno → A[3]<5 → 5<5 NO

while A[d]>perno → A[3]>5 → 5>5 NO

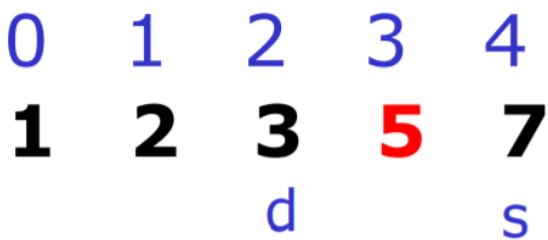
s>d → 3>3 NO

scambio A[s] con A[d] → A[3] con A[3] → 5 con 5

//scambio il perno con sé stesso

s++ d--

- s=4
- d=2
- perno=5



Sono nel caso in cui s>d (4>2) entro nella condizione dell'if e faccio break terminando il while
if(inf<d) → 3<2 NO!

if(s<sup) → 4<4 NO!

→ Nessuna chiamata ricorsiva e termine il punto 7° e termine l'algoritmo con l'array ordinato così → 1 2 3 5 7

MERGESORT (ARRAY)

```
void merge(int A[], int start, int mid, int end){  
    int iSx=start; //1  
    int iDx=mid; //2  
    int cont=0; //3  
    int Appo[end+1]; //4  
    while(iSx<mid || iDx<end){ //5  
        if(A[iSx]<A[iDx]){ //6  
            Appo[cont]=A[iSx]; //7  
            iSx++, cont++; //8  
        }  
        else{ //9  
            Appo[cont]=A[iDx]; //10  
            iDx++, cont++; //11  
        }  
    } //12  
}
```

Funzionamento MERGE:

1. Indice di partenza per sottoarray di SX
2. Indice di partenza per sottoarray di DX
3. Variabile che farà da indice per posizionare gli elementi dell'array di appoggio al posto giusto
4. Array di appoggio di dimensione (+1) più grande di quello di partenza
5. Finchè non mi finisce uno dei due sottoarray
6. Controllo se il valore più piccolo sia nel sottoarray di SX
7. Nel caso lo sia, quel valore lo metto nell'array di appoggio nella posizione cont attuale
8. Vado avanti con il sottoarray di SX (iSx++) e aumento il cont per il successivo elemento da inserire
9. Sono nel caso in cui il valore più piccolo è nel sottoarray di DX
10. Quel valore lo metto nell'array di appoggio nella posizione cont attuale
11. Vado avanti con il sottoarray di DX (iDx++) e aumento il cont per il successivo elemento da inserire
12. *Da questo punto in poi non mi rimane altro che gestire gli ultimi elementi di uno dei due sottoarray rimasti (DX o SX) inserendoli nell'array di appoggio e poi ricopio tutto l'array di appoggio nell'array di partenza (PARTE MANCANTE NEL CODICE)*

MERGESORT (ARRAY)

```
void mergeSort(int A[], int start, int end){  
    int mid;  
    if(start<end){  
        //Se ci sono almeno due elementi  
        mid=(start+end)/2;  
        //DIVIDE  
        mergeSort(A, start, mid);  
        //CONQUER  
        mergeSort(A, mid+1, end);  
        //CONQUER  
        merge(A, start, mid+1, end);  
        //COMBINE  
    }  
}
```

Complessità (CASO PEGGIORE, CASO MEDIO, CASO MIGLIORE)

O($n \log n$)

“L’algoritmo ordina in modo crescente procedendo così:

- Divido in due l’array principale → Sottoarray di SX e sottoarray di DX
- Scelgo come elemento centrale *mid* il quale va prendere il valore a metà strada tra l’inizio e la fine dell’array.
- Chiamo prima sul sottoarray di SX la funzione MERGESORT che ridivide ulteriormente il sottoarray di SX in maniera ricorsiva fino a quando non ho un solo elemento che chiude la ricorsione.
- Stessa cosa del passaggio precedente sul sottoarray di DX
- Utilizzo poi la funzione MERGE che fonde i due sottoarray (SX e DX) in un sottoarray di appoggio, grazie al quale avviene la fase di ordinamento

FASI ORDINAMENTO MERGESORT

- DIVIDE (Punto a metà strada dell’array)
- CONQUER (MergeSort)
- COMBINE (Merge)

MERGESORT (LISTE)

```
void mergeSort(Elem *&s1){  
    if(s1==NULL || s1->next==NULL) //1  
        return;  
    Elec *s2=NULL;  
    split(s1, s2);  
    mergeSort(s1);  
    mergeSort(s2);  
    merge(s1, s2);  
}  
-----
```

1- Fase di controllo se ho almeno due elementi nella lista

COMPLESSITÁ TOTALE ALGORITMO

O(n logn)

$bn + 2T(n/2)$:

- **bn** = Dato dal lavoro di combinazione (merge) dopo aver fatto la split
- **log n** = Dato da 2 chiamate su 2 sottoinsiemi bilanciati, metà degli elementi della lista di partenza (mergesort(s1) e mergesort (s2))

```
void split(Elem *&s1, Elec *&s2){  
    if(s1==NULL || s1->next==NULL)  
        return;  
    Elec *p=s1->next;  
    s1->next=p->next;  
    p->next=s2;  
    s2=p;  
    split(s1->next, s2);  
}
```

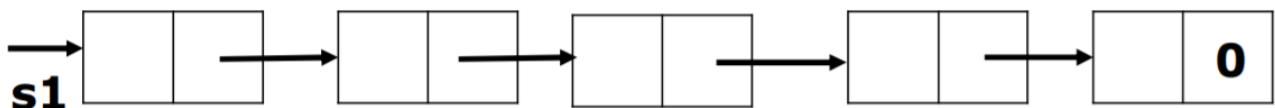
O(n)

Divido la lista s1 in 2 parti con la lista s2, guardando tutti gli elementi

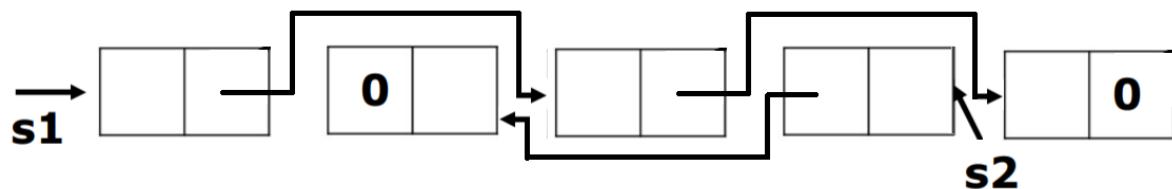
MERGESORT (LISTE)

FUNZIONAMENTO SPLIT

Divide la lista in due sottoliste mettendo gli elementi in posizione pari in s2 e quelli in posizione dispari in s1



→ Diventa



```
void merge(Elem *&s1, Elek *s2){  
    if(s2==NULL) //0  
        return;  
    if(s1==NULL){ //1  
        s1=s2;  
        return;  
    }  
    if(s1->inf <= s2->inf) //2  
        merge(s1->next, s2); //3  
    else{  
        merge(s2->next, s1); //4  
        s1=s2;  
    }  
}
```

O(n)

Confronto tutti gli elementi fra loro, delle due liste separate

MERGESORT (LISTE)

1. Se non ho nulla da fondere dalla seconda lista
2. Se non ho nulla da fondere dalla prima lista
3. Se l'elemento della prima lista è minore di quello della seconda lista
4. Vedi se puoi fondere l'elemento della seconda lista, sull'elemento successivo della prima
5. Altrimenti, vedi se puoi fondere l'elemento della prima lista, sull'elemento successivo della seconda

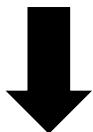
FUNZIONAMENTO COMPLESSIVO ALGORITMO

“L'algoritmo ordina in modo crescente procedendo così:

1. Data una qualunque sequenza di elementi divido la lista in due parti uguali tramite la funzione split la quale:
 - Controlla se ci sono elementi nella lista da dividere (0 elementi o 1 elemento non divide nulla)
 - Utilizzando una lista di appoggio sulla prima sottolista inserisco gli elementi di posizione dispari, mentre sulla seconda sottolista gli elementi di posizione pari
2. Utilizzo il mergeSort per ordinare le due sottoliste. Lo uso sia sulla prima sottolista , sia sulla seconda sottolista, e queste verranno dette ordinate quando avranno un elemento soltanto.
3. Fondo le due parti insieme facendo i confronti tra gli elementi della prima e gli elementi della seconda attraverso la funzione merge la quale:
 - Scorre in parallelo le due liste confrontando i loro primi elementi e scegliendo ogni volta il minore fra i due”

RELAZIONE DI RICORRENZA TOTALE ALGORITMO

$$\begin{aligned}T(0) &= T(1) = a \\T(n) &= bn + 2T(n/2)\end{aligned}$$



COMPLESSITÀ $T(n)$

$O(n \log n)$

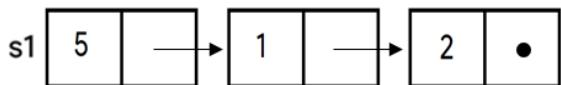
MERGESORT (LISTE)

FUNZIONAMENTO MERGESORT

1° Passaggio

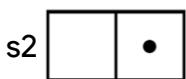
Applico la funzione **mergesort** su questa lista di elementi

- **mergeSort(s1);**



1.1° Passaggio

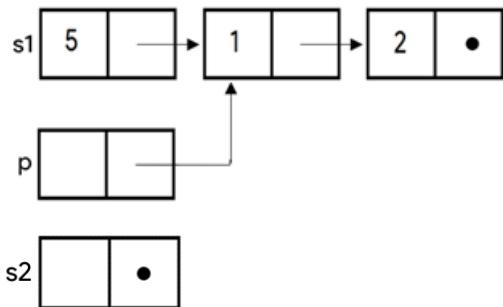
- **s1==NULL || s1->next==NULL** No
- **Elem *s2=NULL;**



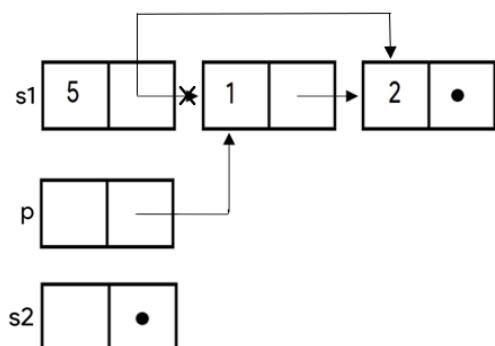
- **split (s1,s2)**

1.2° Passaggio (**split**)

- **s1==NULL || s1->next==NULL** No
- **Elem *p=s1->next;**



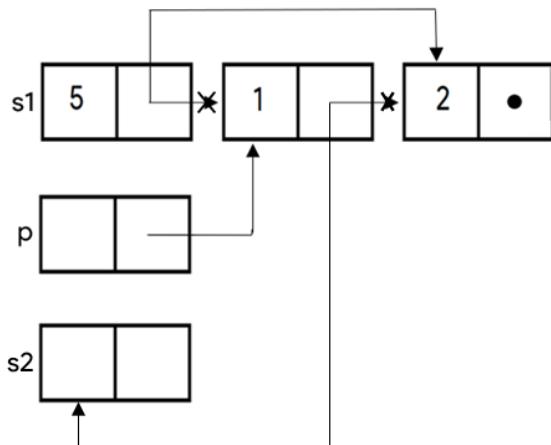
- **s1->next=p->next**



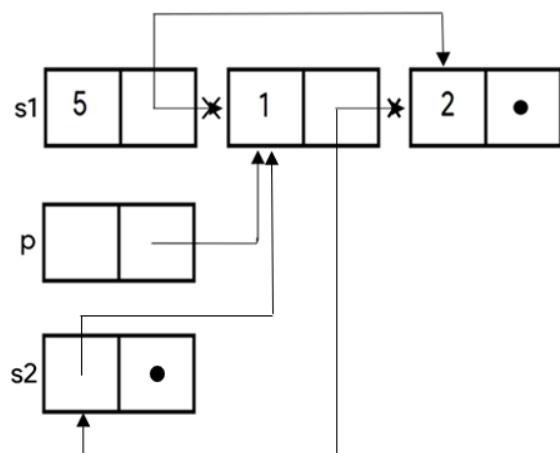
MERGESORT (LISTE)

... CONTINUO 1.2° Passaggio

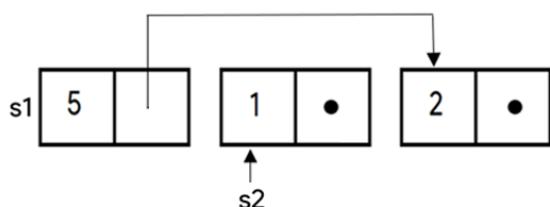
- $p \rightarrow \text{next} = s_2$
//entra in gioco s₂



- $s_2 = p$



OTTENGO QUINDI:



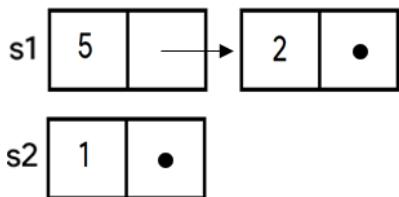
- **split** ($s_1 \rightarrow \text{next}$, s_2)
//ricorsione

MERGESORT (LISTE)

1.2.1° Passaggio (**split-1°ricorsione**)

//Sono su s1->next quindi parto da 2

- s1==NULL || s1->next==NULL Sì, chiudo la ricorsione e chiudo la funzione split ed avrò le due liste s1 e s2 così separate:

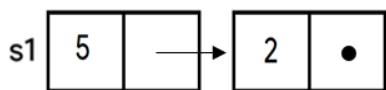


- Torno a 1.2° e chiamo **mergeSort(s1)**;

1.3° Passaggio

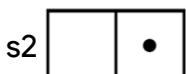
Applico la funzione **mergesort** su questa lista di elementi

- **mergeSort(s1);**



1.3.1° Passaggio **mergeSort(s1)**

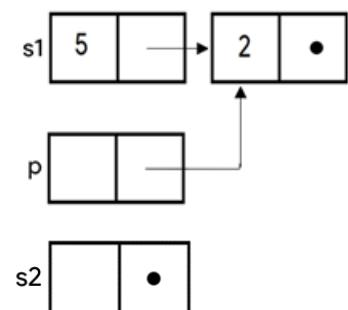
- s1==NULL || s1->next==NULL No
- Elemt *s2=NULL;



- **split (s1,s2)**

1.3.2° Passaggio (**split**)

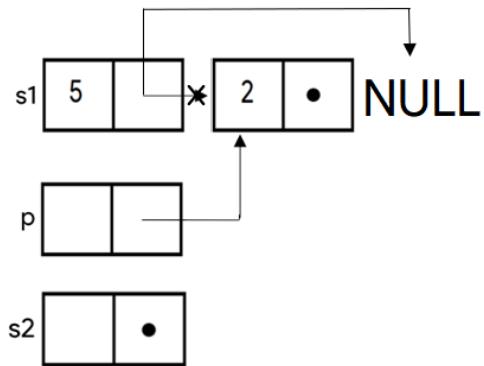
- s1==NULL || s1->next==NULL No
- Elemt *p=s1->next;



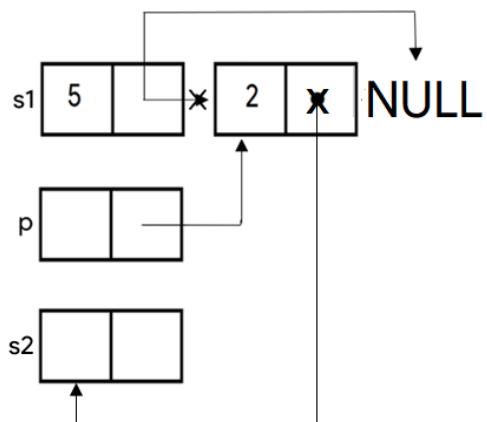
MERGESORT (LISTE)

... CONTINUO 1.3.2° Passaggio

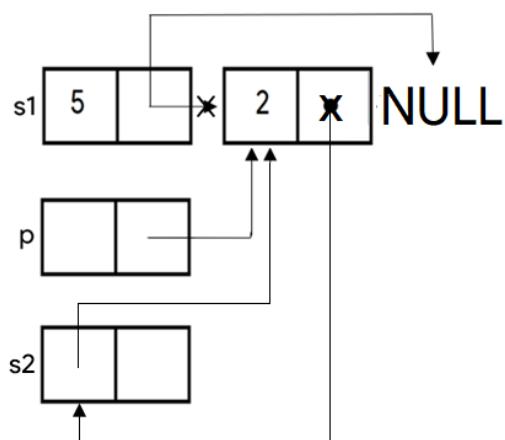
- $s1 \rightarrow next = p \rightarrow next$



- $p \rightarrow next = s2$
//entra in gioco s2



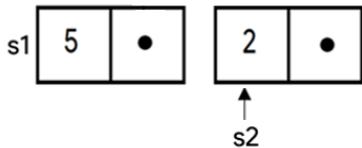
- $s2=p$



MERGESORT (LISTE)

... CONTINUO 1.3.2° Passaggio

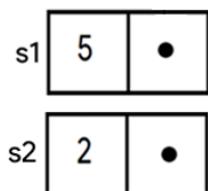
- **split** ($s1 \rightarrow next, s2$)
//ricorsione



1.3.2.1° Passaggio (**split-1° ricorsione**)

//Sono su $s1 \rightarrow next$ che è NULL

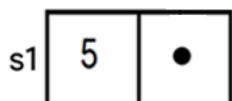
- $s1 == \text{NULL} \ || \ s1 \rightarrow next == \text{NULL}$ Sì, chiudo la ricorsione e chiudo la funzione split ed avrò le due liste s1 e s2 così separate:



- Torno a 1.3.2° e chiamo **mergeSort(s1)**;

1.3.3° Passaggio (**mergeSort(s1)**)

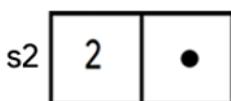
- $s1 == \text{NULL} \ || \ s1 \rightarrow next == \text{NULL}$ Sì
- Chiudo la chiamata ritornando s1



- chiamo **mergeSort(s2)**;

1.3.4° Passaggio(**mergeSort(s2)**) // la lista è s2 data dai parametri anche se si lavora su s1

- $s1 == \text{NULL} \ || \ s1 \rightarrow next == \text{NULL}$ Sì
- Chiudo la chiamata ritornando s1 (che in questo caso sarebbe s2)



- Chiamo la funzione **merge(s1,s2)**

MERGESORT (LISTE)

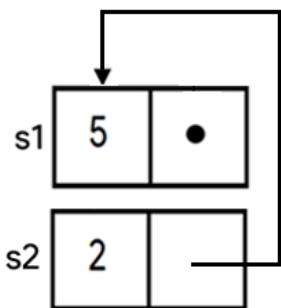
1.3.5° Passaggio (**merge**(s1,s2))

- s2=NULL, No
 - s1=NULL, No
 - s1->inf <= s2->inf → 5<=2 No
 - else(s1->inf > s2->inf) → 5>2 Sì
 - ricorsione **merge**(s2->next, s1)
-

1.3.5.1° Passaggio (**merge**(s2->next, s1))

//Occhio ai parametri passati! Il parametro della funzione s2 dentro la funzione, si riferisce a s1 dato in ingresso e, s1 dentro la funzione a s2->next dato in ingresso

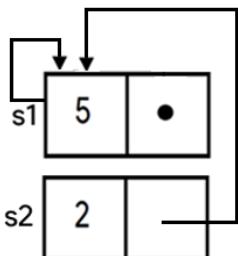
- s2=NULL, No
- s1=NULL Sì
- s1=s2 “s1 deve puntare dove punta s2” → Quindi s2->next deve puntare dove punta s1 (occhio dai parametri)



- Chiudo la ricorsione e torno dopo il passaggio 1.3.5°
-

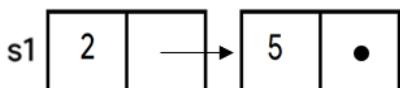
1.3.6° Passaggio

- s1=s2 “s1 deve puntare dove punta s2”



MERGESORT (LISTE)

Chiudo la funzione **merge** e di conseguenza la **mergeSort(s1)** partita al passaggio 1.3° e ritorno la lista s1 così ordinata

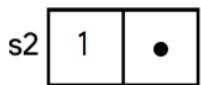


- chiamo **mergeSort(s2)**;

1.4° Passaggio

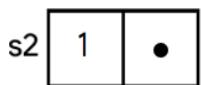
Applico la funzione **mergesort** su questa lista di elementi

- **mergeSort(s2)**;



1.4.1° Passaggio **mergeSort(s2)** // *Dentro il corpo della funzione anche se si tratta di s1 ci si riferisce alla lista s2*

- `s1==NULL || s1->next==NULL` Sì
- Chiudo la chiamata ritornando s1 (che in questo caso sarebbe s2)



- Torno al passaggio 1.4° e chiamo la funzione **merge(s1,s2)**

1.5° Passaggio (**merge(s1,s2)**)

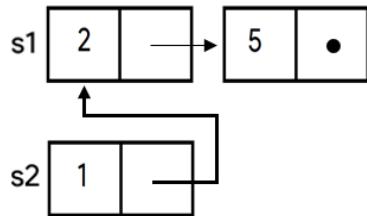
- `s2=NULL`, No
- `s1=NULL`, No
- `s1->inf <= s2->inf` → $2 \leq 1$ No
- `else(s1->inf > s2->inf)` → $2 > 1$ Sì
- ricorsione **merge(s2->next, s1)**

MERGESORT (LISTE)

1.5.1° Passaggio (**merge**(*s2->next*, *s1*))

//Occhio ai parametri passati! Il parametro della funzione *s2* dentro la funzione, si riferisce a *s1* dato in ingresso e, *s1* dentro la funzione a *s2->next* dato in ingresso

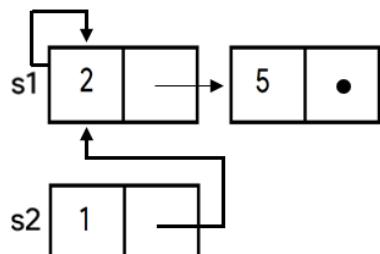
- *s2=NULL*, No
- *s1=NULL* Sì
- *s1=s2* “*s1* deve puntare dove punta *s2*” → Quindi *s2->next* deve puntare dove punta *s1* (occhio dai parametri)



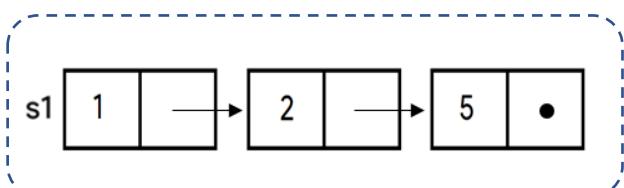
- Chiudo la ricorsione e torno dopo il passaggio 1.5°

1.6° Passaggio

- *s1=s2* “*s1* deve puntare dove punta *s2*”



Chiudo la funzione **merge** e di conseguenza tutta la **mergeSort**(*s1*) partita al passaggio 1° con la lista *s1* così ordinata



INSERTION SORT

```
void sortArray(int A[], int n){  
    int mano=0;  
    int occhio=0;  
    for(int i=1; i<n; ++i){  
        mano=A[i];  
        occhio=i-1;  
        while(occhio>=0 && A[occhio]>mano){  
            A[occhio+1]=A[occhio];  
            --occhio;  
        }  
        A[occhio+1]=mano;  
    }  
}
```

Complessità (CASO PEGGIORE E CASO MEDIO)

O(n^2)

// Quadratica perché ci sono due cicli annidati che si possono estendere fino alla lunghezza di tutto l'array

Complessità (CASO MIGLIORE)

O(n)

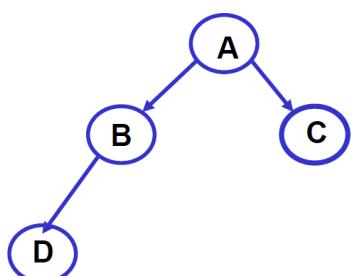
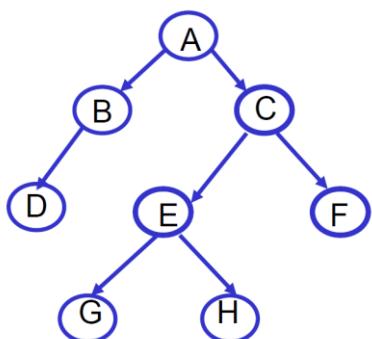
INSERTION SORT

“L’algoritmo ordina in modo crescente procedendo così:

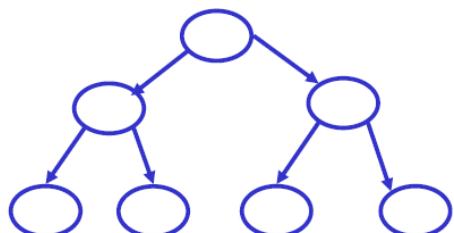
- Si prende un elemento alla volta scorrendo normalmente l’array partendo dalla posizione 1 (non zero) sul ciclo esterno.
- Si confronta questo elemento con i precedenti andando verso sinistra e scambiandolo all’occorrenza nel caso si trovi un elemento più grande di esso, in modo da far andare quello più piccolo a sinistra
- Si fa per ciascun elemento dell’array mantenendo la coerenza dell’ordinamento in modo crescente sulla parte di sinistra
- Lo spostamento viene fatto con una variabile che si chiama MANO
- Invece la lettura sul valore da confrontare viene fatta da OCCHIO che è un indice
- Il ciclo for (esterno) → Inizializza MANO al secondo elemento (2) e OCCHIO al primo elemento (1)
//osserva ++iter
- Il ciclo while (interno) procede spostando l’elemento da valutare verso sinistra di una posizione fino a quando non ne trova uno più piccolo, e appena ne trova uno più piccolo esco dal ciclo”

TIPI DI ALBERI

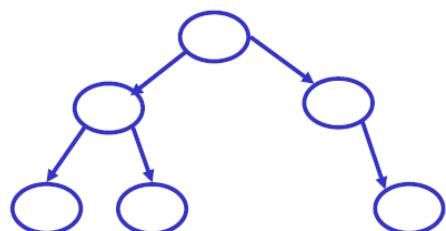
ALBERO BINARIO



ALBERO BINARIO BILANCIATO

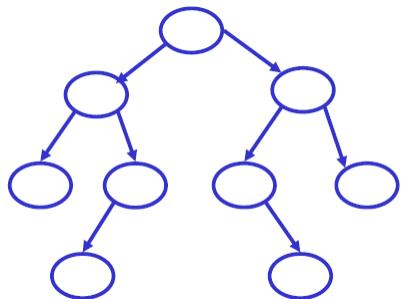


ALBERO BINARIO NON BILANCIATO

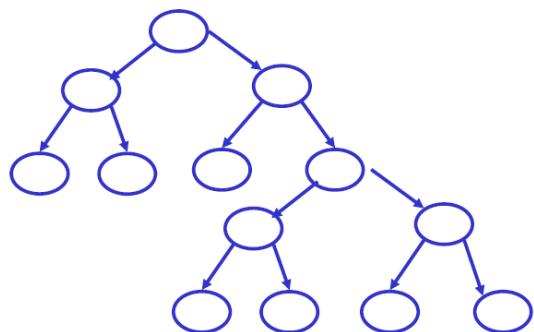


TIPI DI ALBERI

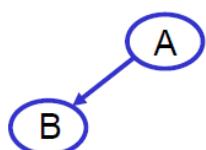
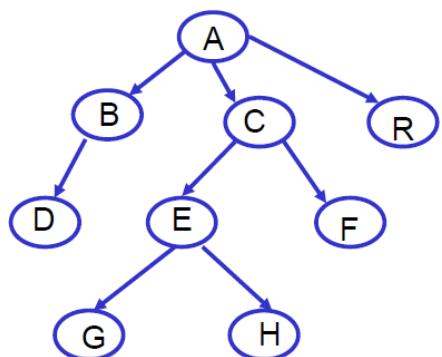
ALBERO BINARIO QUASI BILANCIATO (FINO AL PENULTIMO LIVELLO)



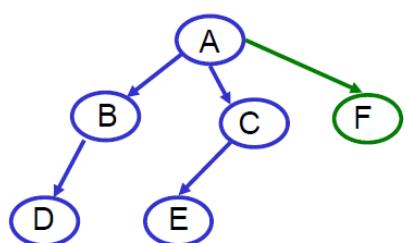
ALBERO PIENAMENTE BINARIO



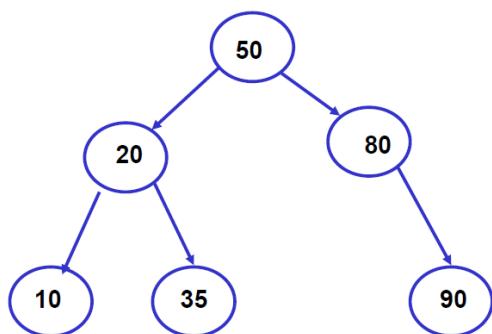
ALBERO GENERICO



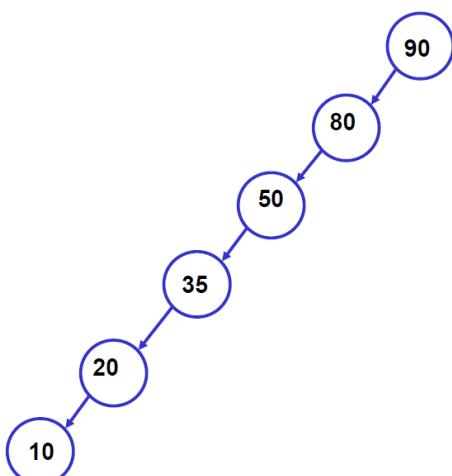
TIPI DI ALBERI



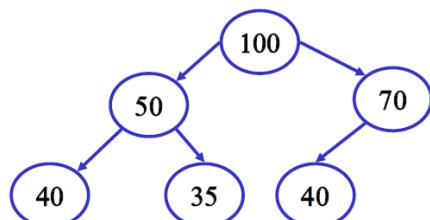
ABR



ABR (DEGENERE)



HEAP



CLASS BinTree (ALBERO BINARIO)

```
class BinTree { //1
    struct Node {
        InfoType label; //2
        Node *left, *right;
    };
    Node *root;

    Node* findNode(InfoType, Node*); //3
    void preOrder(Node*);
    void inOrder(Node*);
    void postOrder(Node*);
    void delTree(Node*&);

    int insertNode(Node*&, InfoType, InfoType, char)

public: //4
    BinTree() { root= NULL; }; //COSTRUTTORE
    ~BinTree(){ delTree(root); }; //DISTRUTTORE
    int find(InfoType x) { return findNode(x, root); };
    void pre() { preOrder(root); };
    void post(){ postOrder(root); };
    void in() { inOrder(root); };
    int insert(InfoType son, InfoType father, char c){
        insertNode(root,son,father,c);
    };
}
```

DICHIARAZIONE DI CLASSE :

- “ 1- Dichiarazione della classe
- 2- Campo privato della struttura che chiamo Nodo
- 3- Da questa funzione in poi sono metodi privati
- 4- Da qui in poi ho la parte pubblica dove utilizzo i metodi precedenti”

CLASS BinTree (ALBERO BINARIO)

```
void preOrder(Node* tree) {                                //Puntatore all'albero
    if (!tree) return;                                     //CASO ALBERO VUOTO
    else {
        cout << tree->label;                            //Esamino La radice
        preOrder(tree->left);                          //Sottoalbero sinistro
        preOrder(tree->right);                         //Sottoalbero destro
    }
}
```

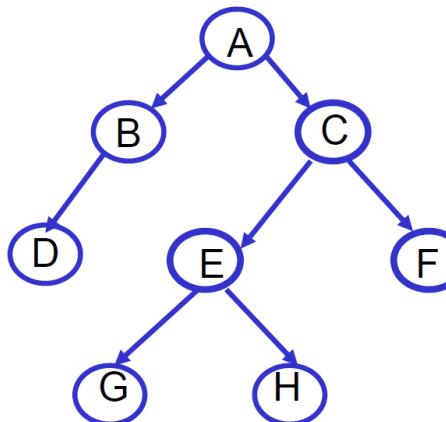
```
void postOrder(Node* tree) {                                //Puntatore all'albero
    if (!tree) return;                                     //CASO ALBERO VUOTO
    else {
        postOrder(tree->left);                         //Sottoalbero sinistro
        postOrder(tree->right);                        //Sottoalbero destro
        cout << tree->label;                            //Esamino La radice
    }
}
```

```
void inOrder(Node* tree) {                                //Puntatore all'albero
    if (!tree) return;                                     //CASO ALBERO VUOTO
    else {
        inOrder(tree->left);                           //Sottoalbero sinistro
        cout << tree->label;                            //Esamino La radice
        inOrder(tree->right);                          //Sottoalbero destro
    }
}
```

CLASS BinTree (ALBERO BINARIO)

ESEMPIO UTILIZZO DELLE TRE VISITE

- Dato il seguente albero



OUTPUT PREORDER
A B D C E G H F

OUTPUT POSTORDER
D B G H E F C A

OUTPUT INORDER
D B A G E H C F

COMPLESSITÀ DELLE VISITE IN FUNZIONE DEL NUMERO DEI NODI

RELAZIONE DI RICORRENZA GENERALE

$$\begin{aligned} T(0) &= a \\ T(n) &= b + T(n_s) + T(n_d) \end{aligned}$$

// Tempo esame label + Complessità a SX + Complessità a DX

RELAZIONE DI RICORRENZA STESSO NUMERO DI NODI NEL SOTTOALBERO DX E SX

$$\begin{aligned} T(0) &= a \\ T(n) &= b + 2T((n-1)/2) \end{aligned}$$

COMPLESSITÀ VISITE (riferita ad entrambe le relazioni di ricorrenza precedenti)

O(n)

COMPLESSITÀ DELLE VISITE IN FUNZIONE DEL NUMERO DEI LIVELLI

RELAZIONE DI RICORRENZA GENERALE

(ALBERO BILANCIATO → STESSO NUMERO DI NODI A SX E A DX)

$$\begin{aligned} T(0) &= a \\ T(n) &= b + 2T(n-1) \end{aligned}$$

// Al livello n = Tempo esame nodo + 2 alberi binari con lo stesso livello pari al livello originale -1 → Quello della foglia che non considero

O(2ⁿ)

CLASS BinTree (ALBERO BINARIO)

```
Node* findNode(Infotype n, Node*tree) { //0  
    if (!tree) return NULL; //1  
    if (tree->label==n) //2  
        return tree;  
    Node* a=findNode(n, tree->left); //3  
    if (a) return a; //4  
    else  
        return findNode(n, tree->right); //5  
}
```

FINDNODE:

“ 0- Restituisco il puntatore al nodo che contiene **n**, cioè l’etichetta da cercare

1- Caso albero vuoto → L’etichetta non c’è ritorno NULL

2- Caso etichetta trovata → Ritorna il puntatore ad essa

3- Cerco a Sinistra dando al nodo **a** il puntatore del nodo di sinistra
(ricorsivamente)

4- Se l’etichetta viene trovata (cioè se il puntatore non è NULL) → Ritorna il
puntatore ad essa tramite **a**

5- Cerco a Destra

OSS:

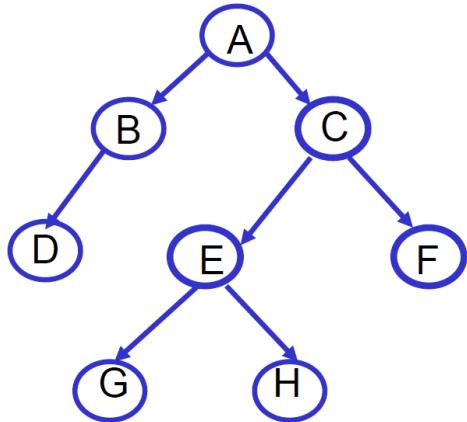
Se più nodi nodi contengono **n**, si ritorna il primo nodo che si incontra facendo la visita anticipata tramite

- //2 (GUARDO LA RADICE)
- //3 (GUARDO A SX)
- //4 (GUARDO A DX)

CLASS BinTree (ALBERO BINARIO)

ESEMPIO FINDNODE

- Dato il seguente albero applico **findNode('C',tree);**



1° Passaggio //SONO SU A

- !tree No
 - Tree->label=n → A=C No
 - Node a=**findNode('C',tree->left)** (ricorsione sottoalbero di SX)
-

1.1° Passaggio //SONO SU B

- !tree No
 - Tree->label=n → B=C No
 - Node a=**findNode('C',tree->left)** (ricorsione sottoalbero di SX)
-

1.1.1° Passaggio //SONO SU C

- !tree No
 - Tree->label=n → D=C No
 - Node a=**findNode('C',tree->left)** (ricorsione sottoalbero di SX)
-

1.1.1.1° Passaggio

- !tree Sì perché la radice andando a sinistra di D è diventata NULL poiché ero alla foglia
- Ritorno NULL e torno al passaggio 1.1.1°

CLASS BinTree (ALBERO BINARIO)

... CONTINUO ESEMPIO FINDNODE

1.1.2° Passaggio // SONO TORNATO SU C

- if a no perché ho ritornato NULL
 - return **findNode**('C', tree->right) (ricorsione nel sottoalbero DX)
-

1.1.2.1° Passaggio

- !tree Sì
 - Chiudo questa ricorsione e la precedente tornando al passaggio 1.1°
-

1.2° Passaggio // SONO TORNATO SU B

- if a no perché ho ritornato NULL
 - return **findNode**('C', tree->right) (ricorsione nel sottoalbero DX)
-

1.2.1° Passaggio

- !tree Sì
 - Chiudo questa ricorsione e la precedente tornando al passaggio 1°
-

2° Passaggio // SONO TORNATO SU A

- if a no perché ho ritornato NULL
 - return **findNode**('C', tree->right) (ricorsione nel sottoalbero DX)
-

2.1° Passaggio // SONO SU C

- !tree No
- Tree->label=n → C=C Sì
- return tree, restituisco questa radice che è il posto in cui si trova il nodo C da cercare e chiudo questa ricorsione insieme a tutta la funzione restituendo C

CLASS BinTree (ALBERO BINARIO)

```
void delTree(Node* &tree) {  
    if (tree) {  
        delTree(tree->left); //1  
        delTree(tree->right); //2  
        delete tree; //3  
        tree=NULL; //4  
    } //5  
}
```

1. Se ho un elemento nell'albero
2. Vado a vedere se ha elementi nel suo sottoalbero SX
3. Vado a vedere se ha elementi nel suo sottoalbero DX
4. Cancello questo elemento
5. Pongo il puntatore finale a NULL

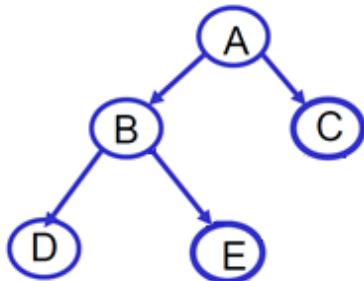
OSS:

Tutte le volte in cui un albero viene modificato si passa un riferimento → & a node!
Se ciò non si dovesse fare agiremmo sulle copie di un albero e non sull'albero stesso

CLASS BinTree (ALBERO BINARIO)

ESEMPIO DELTREE

- Dato il seguente albero applico **delTree(tree)**;



0° Passaggio

- if tree Si (Sono sul nodo A)
-

1° Passaggio

- Chiamata ricorsiva sul sottoalbero SX
-

1.1° Passaggio

- if tree Si (Sono sul nodo B)
 - Chiamata ricorsiva sul sottoalbero SX
-

1.1.1° Passaggio

- if tree Si (Sono sul nodo D)
 - Chiamata ricorsiva sul sottoalbero SX
-

1.1.1.1° Passaggio

- if tree No, D non ha figli a SX
 - Chiudo questa chiamata ricorsiva e torno al passaggio precedente (1.1.1)
-

1.1.2° Passaggio (Sono tornato al nodo D)

- Chiamata ricorsiva sul sottoalbero DX
-

CLASS BinTree (ALBERO BINARIO)

... CONTINUO ESEMPIO DELTREE

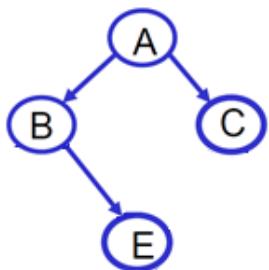
1.1.2.1° Passaggio

- if tree No, D non ha figli a DX → D è una foglia
 - Chiudo anche questa chiamata ricorsiva e torno al passaggio (1.1.2)
-

1.1.3° Passaggio (Sono tornato al nodo D)

- Cancello il puntatore a questa radice
- Pongo tree a NULL
- Chiudo questa ricorsione tornando al punto (1.1)

ALBERO A QUESTO PUNTO



1.2° Passaggio (Sono tornato al nodo B)

- Chiamata ricorsiva sul sottoalbero DX
-

1.2.1° Passaggio

- if tree Si (Sono sul nodo E)
 - Chiamata ricorsiva sul sottoalbero SX
-

1.2.1.1° Passaggio

- if tree No, E non ha figli a SX
 - Chiudo questa chiamata ricorsiva e torno al passaggio precedente (1.2.1)
-

1.2.2° Passaggio (Sono tornato al nodo E)

- Chiamata ricorsiva sul sottoalbero DX
-

CLASS BinTree (ALBERO BINARIO)

... CONTINUO ESEMPIO DELTREE

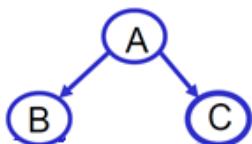
1.2.2.1° Passaggio

- if tree No, E non ha figli a DX → E è una foglia
 - Chiudo anche questa chiamata ricorsiva e torno al passaggio (1.2.2)
-

1.2.3° Passaggio (Sono tornato al nodo E)

- Cancello il puntatore a questa radice
- Pongo tree a NULL
- Chiudo questa ricorsione tornando al punto (1.2)

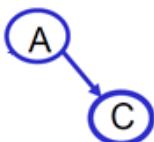
ALBERO A QUESTO PUNTO



1.3° Passaggio (Sono tornato al nodo B)

- Cancello il puntatore a questa radice
- Pongo tree a NULL
- Chiudo questa ricorsione tornando al punto (1)

ALBERO A QUESTO PUNTO



2° Passaggio (Sono tornato al nodo A)

- Chiamata ricorsiva sul sottoalbero DX
-

2.1° Passaggio

- if tree Sì (Sono sul nodo C)
- Chiamata ricorsiva sul sottoalbero SX

CLASS BinTree (ALBERO BINARIO)

... CONTINUO ESEMPIO DELTREE

2.1.1° Passaggio

- if tree No, C non ha figli a SX
 - Chiudo questa chiamata ricorsiva e torno al passaggio precedente (2.1)
-

2.2° Passaggio (Sono tornato al nodo C)

- Chiamata ricorsiva sul sottoalbero DX
-

2.2.1° Passaggio

- if tree No, C non ha figli a DX → E è una foglia
 - Chiudo anche questa chiamata ricorsiva e torno al passaggio (2.2)
-

2.3° Passaggio (Sono tornato al nodo C)

- Cancello il puntatore a questa radice
- Pongo tree a NULL
- Chiudo questa ricorsione tornando al punto (2)

ALBERO A QUESTO PUNTO

A

3° Passaggio (Sono tornato al nodo A)

- Cancello il puntatore a questa radice
- Pongo tree a NULL
- Chiudo questa tutta la funzione è l'albero è eliminato!

ALBERO A QUESTO PUNTO

NULL

CLASS BinTree (ALBERO BINARIO)

INSERIMENTO DI UN NODO PASSI DA FARE:

- Inserisco un nodo (son) come figlio di father (preso in input)
- Lo inserisco a sinistra se mi viene data **c='l'** (che sta per left)
- Lo inserisco a destra invece, se mi viene data **c='r'** (che sta per right)
- Ricorsivamente ritorno **1** se l'operazione ha successo, **0** se non ha successo
- Inoltre, non posso inserire un figlio se:
 - Father non compare nell'albero → Ritorno **0**
 - Ho già un figlio in quella posizione → Ritorno **0**

```
int insertNode (Node* & tree, InfoType son, InfoType father, char c){  
    if (!tree) { //1  
        tree=new Node; //2  
        tree ->label=son;  
        tree ->left = tree ->right = NULL; //3  
        return 1;  
    }  
    Node* a=findNode(father,tree); //4  
    if (!a) return 0; //5  
    if (c=='l' && !a->left) { //6  
        a->left=new Node; //7  
        a->left->label=son;  
        a->left->left =a->left->right=NULL;  
        return 1;  
    }  
    if (c=='r' && !a->right) { //8  
        a->right=new Node; //9  
        a->right->label=son;  
        a->right->left = a->right->right = NULL;  
        return 1;  
    }  
    return 0; //10  
}
```

CLASS BinTree (ALBERO BINARIO)

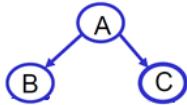
INSERTNODE:

- “ 1- Albero vuoto, posso inserire son direttamente come radice
- 2- Genero la radice
- 3- Sottoalberi SX e DX a NULL in quanto è solo la radice
- 4- Cerco father, e se il padre c’è alla radice dell’albero, ci restituisce il puntatore ad esso
- 5- Se father non c’è torno 0, non posso inserire son
- 6- Inserisco come figlio sinistro verificando che non ci siano figli a sinistra,
altrimenti ritorno 0 alla fine di tutta la funzione (*)
- 7- Imposto son come foglia (SX)
- 8- Inserisco come figlio destro verificando che non ci siano figli a destra,
altrimenti ritorno 0 alla fine di tutta la funzione (*)
- 9- Imposto son come foglia (DX)
- 10- (*) Inserimento impossibile

CLASS BinTree (ALBERO BINARIO)

ESEMPIO INSERTNODE

- Dato il seguente albero Voglio inserire come figlio DX di B il valore E



- Chiamata: **insertNode(tree, 'E', 'B', r)**
-

1° Passaggio (*figlio=E, padre=B, posizione=r=destra*)

- !tree No
 - Node* a=**findNode('B',tree);**
-

1.1° Passaggio //SONO DENTRO LA FINDNODE SU A

- !tree No
 - Tree->label=n → A=B No
 - Node* a=**findNode('B',tree->left)** (ricorsione sottoalbero di SX)
-

1.1.1° Passaggio //SONO DENTRO LA FINDNODE SU B

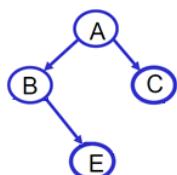
- !tree No
- Tree->label=n → B=B Sì
- return tree, restituisco questa radice che è il posto in cui si trova il nodo B da cercare e chiudo questa ricorsione

Torno al passaggio 1° restituendo B e lo metto nella variabile del nodo a, dichiarata al passaggio 1°

2° Passaggio

- if (!a) No, il padre c'è
- if (c==l && !a->left) No
- if(c==r && !a->right) Sì la variabile c=r e non ho figli a SX del padre prescelto
- Faccio l'inserimento creando un nuovo nodo, dandogli l'etichetta E passata tra i parametri della funzione e ponendo i figli di questo nuovo nodo a NULL in modo da essere una foglia

ALBERO OTTENUTO



CLASS BinTree (ALBERO BINARIO)

EXTRA ALBERI BINARI: CONTA I NODI

```
int nodes(Node* tree) {  
  
    if (!tree) return 0;  
  
    return 1+nodes(tree->left)+nodes(tree->right);  
}
```

“Caso base : albero vuoto

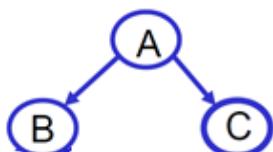
Caso ricorsivo: Radice + Numero di nodi a SX + Numero di nodi a DX”

COMPLESSITÁ

O(n)

ESEMPIO NODES

Dato il seguente albero applico la funzione **nodes(tree)**



1° Passaggio (SONO SU A)

- !tree No
 - return 1+ **nodes(tree->left)** + **nodes(tree->right)**
//ricorsione a SX + ricorsione a DX
-

1.1° Passaggio (FACCIO LA RICORSIONE A SX - SONO SU B)

- !tree No
- return 1+ **nodes(tree->left)** + **nodes(tree->right)**
//ricorsione a SX + ricorsione a DX

CLASS BinTree (ALBERO BINARIO)

... CONTINUO ESEMPIO NODES

1.1.1° Passaggio (RICORSIONE SX – RICORSIONE SX)

- !tree Sì Ritorno 0
-

1.1.2° Passaggio (RICORSIONE SX – RICORSIONE DX)

- !tree Sì Ritorno 0
 - Torno al passaggio 1.1°
-

1.2° Passaggio (SONO TORNATO SU B E VADO SU A)

- Ritorno $1+0+0 = 1$ al passaggio 1° come RICORSIONE A SX
-

1.3° Passaggio (FACCIO LA RICORSIONE A DX - DA A SONO ANDATO IN C)

- !tree No
 - return $1 + \text{nodes}(\text{tree-} \rightarrow \text{left}) + \text{nodes}(\text{tree-} \rightarrow \text{right})$
//ricorsione a SX + ricorsione a DX
-

1.3.1° Passaggio (RICORSIONE DX – RICORSIONE SX)

- !tree Sì Ritorno 0
-

1.3.2° Passaggio (RICORSIONE DX – RICORSIONE DX)

- !tree Sì Ritorno 0
 - Torno al passaggio 1.3°
-

1.4° Passaggio (SONO TORNATO SU C E VADO SU A)

- Ritorno $1+0+0 = 1$ al passaggio 1° come RICORSIONE A DX
-

1.5° Passaggio (SONO TORNATO SU A)

- Ritorno $1+1+1 = 3$ al passaggio 1° come valore totale della funzione che sono il numero dei nodi, e chiudo la funzione completamente
-

CLASS BinTree (ALBERO BINARIO)

EXTRA ALBERI BINARI: CONTA LE FOGLIE

```
int leaves(Node* tree) {  
  
    if (!tree) return 0;                                //1  
  
    if (!tree->left && !tree->right) return 1;          //2  
  
    return leaves(tree->left)+leaves(tree->right);    //3  
  
}
```

Ci sono 2 casi BASE: 1, 2

- “ 1- Albero vuoto
- 2- Non ho figli né a SX né a DX, quindi il nodo che sto analizzando è una foglia
- 3- Ciascuna chiamata ricorsiva si ferma alla foglia

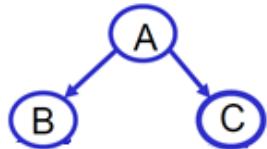
COMPLESSITÁ

O(n)

CLASS BinTree (ALBERO BINARIO)

ESEMPIO LEAVES

Dato il seguente albero applico la funzione **leaves(tree)**



1° Passaggio (SONO SU A)

- !tree No
 - !tree->left && !tree->right No
 - return **leaves(tree->left) + leaves(tree->right);**
//ricorsione a SX + ricorsione a DX
-

1.1° Passaggio (FACCIO LA RICORSIONE A SX - SONO SU B)

- !tree No
 - !tree->left && !tree->right Sì Sono su una foglia
 - Chiudo la ricorsione ritornando 1 al passaggio 1°
-

1.2° Passaggio (FACCIO LA RICORSIONE A DX - SONO SU C)

- !tree No
 - !tree->left && !tree->right Sì Sono su una foglia
 - Chiudo la ricorsione ritornando 1 al passaggio 1°
-

2° Passaggio (SONO TORNATO SU A)

- Chiudo la funzione ritornando la somma di $1+1=2$ che è il numero delle foglie ottenute dai passaggi 1.1° e 1.2°

ALBERO GENERICO

- Per gli alberi generici le seguenti funzioni sono le stesse degli alberi binari:

- Funzione che conta le foglie
- Visita preOrder
- Visita postOrder
- Ricerca findNode

OSS: la visita inOrder non esiste poiché gli alberi generici non sono binari

```
int leaves(Node* tree) {  
    if (!tree) return 0;                                //1  
    if (!tree->left) return 1+leaves(tree->right);    //2  
    return leaves(tree->left)+leaves(tree->right);   //3  
}
```

Ci sono 2 casi BASE: 1, 2

“ 1- Albero vuoto

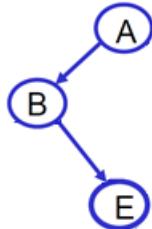
2- Sono su una foglia quando a sinistra non ho figli + vado a scorrere il sottoalbero di destra (da come è definito un albero binario proveniente da un albero generico)

3- Ciascuna chiamata ricorsiva si ferma alla foglia e le somma

ALBERO GENERICO

ESEMPIO LEAVES

Dato il seguente albero applico la funzione **leaves(tree)**



1° Passaggio

//SONO SU A

- !tree No
 - !tree->left No
 - return **leaves(tree->left) + leaves(tree->right);**
//ricorsione a SX + ricorsione a DX
-

1.1° Passaggio (FACCIO LA RICORSIONE A SX)

//SONO SU B

- !tree No
 - !tree->left Si
//vuol dire che a sinistra non ho nulla sommo 1 + vado a vedere con la ricorsione quello che ha alla sua destra
 - return 1+ **leaves(tree->right);**
-

1.1.1° Passaggio RICORSIONE SX – RICORSIONE DX

//SONO SU E

- !tree Si
 - Ritorno 0 al punto precedente 1.1° chiudendo questa ricorsione
-

1.2° Passaggio

//SONO TORNATO SU B

- Sono tornato al passaggio 1.1° e sommo 1+0 e lo ritorno al passaggio 1° (SU A)
-

1.3° Passaggio (FACCIO LA RICORSIONE A DX)

//SONO TORNATO SU A

- !tree Si
 - Ritorno 0 al punto precedente 1.2° chiudendo questa ricorsione
-

1.4° Passaggio

- Sono tornato al passaggio 1° e sommo 1+0=1 che è il numero di foglie e chiudo tutta la funzione
-

ALBERO GENERICO

```
void addSon(Infotype x, Node* &list){ //1
    if(!list){ //2
        list=new Node;
        list->label=x;
        list->left=list->right=NULL;
    }
    else
        addSon(x,list->right); //3
}
```

//list rappresenta la tree che in questo caso deve avere significato come lista di fratelli

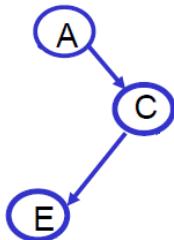
INSERIMENTO DI UN NODO IN FONDO ALLA LISTA DI FRATELLI (TUTTO A DX)

- “ 1- Il riferimento tra i parametri è al puntatore del nodo
- 2- Caso lista vuota inserisco il figlio in quanto è il primo elemento
- 3- Chiamata ricorsiva sul sottoalbero DX poiché a destra ho i fratelli e di conseguenza devo andare ad inserire in fondo alla lista dei fratelli”

ALBERO GENERICO

ESEMPIO ADDSON

Dato il seguente albero applico la funzione **addSon('H',tree)**



1° Passaggio //SONO SU A

- !list No
 - Ricorsione a DX **addSon(x,list->right);** //dove x è 'H'
-

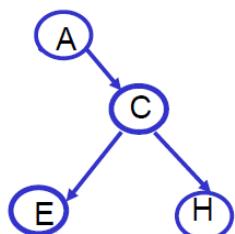
1.1° Passaggio //SONO SU C

- !list No
 - Ricorsione a DX **addSon(x,list->right);** //dove x è 'H'
-

1.1.1° Passaggio

- !list Sì → Vuol dire che non ho figli a DX, e quindi posso inserire il nodo in fondo alla lista dei fratelli
- Creo un nuovo nodo con H come etichetta, inserito come foglia, ponendo i suoi figli entrambi a NULL

OTTENGO



ALBERO GENERICO

```
int insert(InfoType son, InfoType father, Node* &tree){           //0
    Node *a=findNode(father,tree);                                     //1
    if(!a) return 0;                                                 //2
    addSon(son,a->left);                                         //3
    return 1;
}
```

INSERIMENTO DI UN NODO COME ULTIMO FIGLIO DI FATHER (DATO IN INGRESSO)

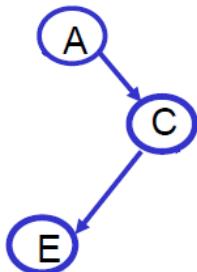
“ O- son è l’etichetta da inserire e &tree perché sto modificando l’albero

- 1- Cerco il padre all’interno dell’albero e se lo trovo restituisco il puntatore al padre mettendolo nel nodo a
- 2- Se non trovo il padre ritorno 0 e non posso fare l’inserimento
- 3- Altrimenti, posso fare l’inserimento e lo faccio con addSon inserendo

l’elemento son come ultimo dei fratelli (ultimo figlio di father); considero come radice a->left (per partire dal primo fratello e se non c’è, addSon inserisce son come primo fratello, di conseguenza è l’unico ed ultimo figlio di father)

ESEMPIO INSERT

Dato il seguente albero applico la funzione **insert**('H', 'E', tree), lo voglio inserire come ultimo figlio di E che scelgo come father



1° Passaggio //SONO SU A

- Node *a=findNode('E',tree) //ricorsione

ALBERO GENERICO

... CONTINUO ESEMPIO INSERT

1.1° Passaggio (DENTRO LA `findNode`)

//SONO SU A

- `!tree` No
 - `tree->label=n` → A=E No
 - `Node* a=findNode('E',tree->left)` (ricorsione sottoalbero di SX)
-

1.1.1° Passaggio

- `!tree` Sì → Non ho nulla a SX di A
 - Ritorno NULL chiudendo questa ricorsione e vado al punto 1.1°
-

1.2° Passaggio

- `if(a)` No
 - `findNode('E',tree->right)` (ricorsione sottoalbero di DX)
-

1.2.1° Passaggio

//SONO SU C

- `!tree` No
 - `tree->label=n` → C=E No
 - `Node* a=findNode('E',tree->left)` (ricorsione sottoalbero di SX)
-

1.2.1.1° Passaggio

//SONO SU E

- `!tree` No
 - `tree->label=n` → E=E Si
 - Ho trovato il padre che stavo cercando e restituisco al punto (1.2.1°) tree che è E cioè il padre che cercavo
-

1.2.2° Passaggio

//SONO TORNATO SU C

- `if(a)` Si → restituisco al punto 1.2° la tree che è E
-

1.3° Passaggio

//SONO TORNATO SU A

- Restituisco al punto 1° la tree che è E chiudendo la `findNode` poiché ho trovato il padre
-

ALBERO GENERICO

... CONTINUO ESEMPIO INSERT

2° Passaggio

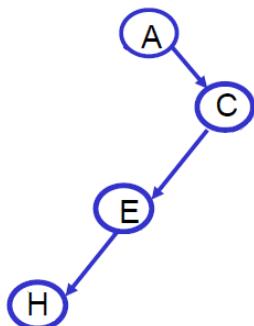
- !a No
 - **addSon('H',a->left)** → Aggiungo il figlio al padre trovato prima E alla sua sinistra (qua è a sinistra l'aggiunta, anche se la addSon la va ad aggiungere alla destra per come è definita)
-

2.1° Passaggio (ricorsione **addSon('H', a->left)**)

//SONO SU LEFT DI E

- !list Sì (dove list è rappresentato dal figlio SX di E, passato tra i parametri)
- Creo il nuovo nodo con l'etichetta H, ed è creato come foglia inserendo NULL ad entrambi i suoi figli → Quindi non è inserito come ultimo figlio DX di father ma, in questo caso come primo figlio di father (a SX)

OTTENGO



- Torno al passaggio 2°
-

3° Passaggio

- Ritorno 1 che rappresenta l'inserimento fatto, e chiudo tutta la funzione

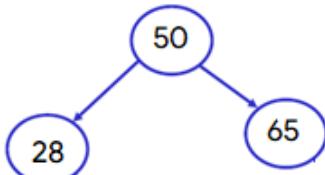
ALBERO GENERICO

EXTRA: SOMMA 1 AD OGNI SUA ETICHETTA

```
void add1Etichetta(Node *tree){  
    if(tree == NULL) return; //Caso base: albero vuoto  
    tree->label++; //Incremento l'etichetta  
    add1Etichetta(tree->left); //Vado ai nodi del Sottoalbero di SX  
    add1Etichetta(tree->right); //Vado ai nodi del Sottoalbero di DX  
}
```

ESEMPIO ADD1ETICHETTA

- Dato il seguente albero, applico la funzione dandogli in ingresso la radice

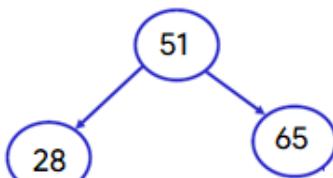


- **add1Etichetta(tree)** //50
-

1° Passaggio (SONO SULLA RADICE – NODO ETICHETTA 50)

- tree=NULL No
- tree->label ++ → 50++ = 51

ALBERO OTTENUTO



2° Passaggio

- Ricorsione sul sottalbero SX

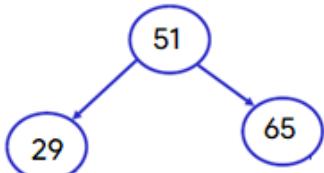
ALBERO GENERICO

... CONTINUO ESEMPIO ADD1ETICHETTA

2.1° Passaggio (SONO SUL NODO CON ETICHETTA 28)

- tree=NULL No
- tree->label ++ → 28++ = 29

ALBERO OTTENUTO



- Ricorsione sul sottalbero SX
-

2.1.1° Passaggio (SONO SU UN NODO VUOTO)

- tree=NULL Si chiude questa ricorsione tornando al passaggio precedente (2.1°)
-

2.2° Passaggio (SONO SUL NODO CON ETICHETTA 29)

//28 aumentato di 1

- Ricorsione sul sottalbero DX
-

2.2.1° Passaggio (SONO SU UN NODO VUOTO)

- tree=NULL Si chiude questa ricorsione tornando al passaggio (2°)
-

3° Passaggio

- Ricorsione sul sottalbero DX

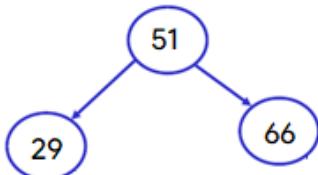
ALBERO GENERICO

... CONTINUO ESEMPIO ADD1ETICHETTA

3.1° Passaggio (SONO SUL NODO CON ETICHETTA 65)

- tree=NULL No
- tree->label ++ → $65++ = 66$

ALBERO OTTENUTO



- Ricorsione sul sottalbero SX
-

3.1.1° Passaggio (SONO SU UN NODO VUOTO)

- tree=NULL Si chiude questa ricorsione tornando al passaggio precedente (3.1°)
-

3.2° Passaggio (SONO SUL NODO CON ETICHETTA 66)

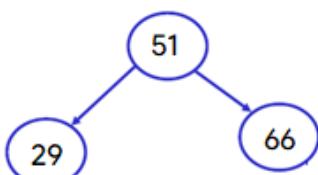
//65 aumentato di 1

- Ricorsione sul sottalbero DX
-

3.2.1° Passaggio (SONO SU UN NODO VUOTO)

- tree=NULL Si chiude questa ricorsione e chiudo tutta la funzione

ALBERO FINALE



ABR

```
Node* findNode(InfoType n, Node *tree){ //0
    if(!tree) return 0; //1
    if(n==tree->label) return tree; //2
    if(n<tree->label) //3
        return findNode(n, tree->left);
    return findNode(n, tree->right); //4
}
```

RICERCA DI UN NODO IN UN ALBERO BINARIO DI RICERCA

- 0- Non metto il riferimento perché non modifco l'albero
 - 1- Caso albero vuoto restituisco 0
 - 2- Caso n=radice cioè ho trovato l'elemento sulla radice ispezionata
 - 3- Caso n<radice mi sposto sul sottoalbero di SX (ricorsivamente) per cercare il nodo
 - 4- Caso n>radice mi sposto sul sottoalbero di DX (ricorsivamente) per cercare il nodo

→ Alla fine delle ricorsioni, o trovo il caso vuoto oppure nella radice trovo il valore

CASO MEDIO E CASO MIGLIORE:

RELAZIONE DI RICORRENZA

COMPLESSITÁ T(n)

$$T(0) = a$$

$$T(n) = b + T(n/2)$$

$O(\log n)$

OSS:

→ Il caso migliore ce l'ho quando l'ABR è bilanciato, che mi fa andare a SX o a DX lavorando su metà dei nodi

CASO PEGGIORE:

RELAZIONE DI RICORRENZA

COMPLESSITÀ T(n)

$$\begin{aligned}T(0) &= a \\T(n) &= b + T(n-1)\end{aligned}$$

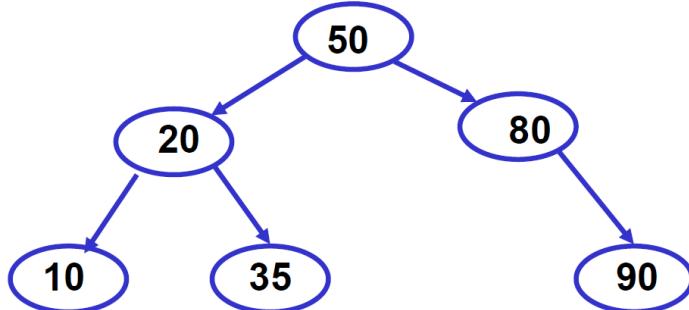
$O(n)$

// I 'ottengo quando l'ABR è degenere cioè con tanti livelli

ABR

ESEMPIO FINDNODE

- Dato il seguente albero binario di ricerca, applico la funzione dandogli in ingresso la radice e il nodo da cercare (considero di cercare come nodo il nodo con etichetta 35)



- **findNode(35, tree)**
-

1° Passaggio (SONO SULLA RADICE - NODO CON ETICHETTA 50)

- !tree No
 - n=tree->label → 35=50 No
 - n<tree->label → 35<50 Sì
 - Ricorsione sul sottoalbero SX poiché essendo binario a SX ho i valori minori
-

1.1° Passaggio (SONO SUL NODO CON ETICHETTA 20)

- !tree No
 - n=tree->label → 35=20 No
 - n<tree->label → 35<20 No
 - Ricorsione sul sottoalbero DX poiché essendo binario a DX ho i valori maggiori
-

1.1.1° Passaggio (SONO SUL NODO CON ETICHETTA 35)

- !tree No
- n=tree->label → 35=35 Sì, ho trovato il nodo in cui ho 35 e lo restituisco chiudendo tutte le chiamate ricorsive

ABR

```
void insertNode(InfoType n, Node* &tree){ //0
    if(!tree){ //1
        tree=new Node;
        tree->label=n;
        tree->left = tree->right = NULL;
        return;
    }
    if(n<tree->label) //2
        insertNode(n,tree->left);
    if(n>tree->label) //3
        insertNode(n,tree->right);
}
```

INSERIMENTO DI UN NODO IN UN ALBERO BINARIO DI RICERCA

- 0- Metto &tree perché sto modificando l'albero
- 1- Caso albero vuoto → creo il nodo e lo inserisco come radice mettendo i figli a NULL sia a DX che a SX
- 2- Se il nodo da inserire è più piccolo della radice che sto analizzando al momento vado nel sottoalbero di SX (ricorsivamente) finché non trovo il posto giusto
- 3- Se il nodo da inserire è più grande della radice che sto analizzando al momento vado nel sottoalbero di DX (ricorsivamente) finché non trovo il posto giusto

OSS

Se l'elemento da inserire è già presente, verranno fatte tante ricorsioni fino all'ultima e, all'ultima non entrerà in nessuno degli if → Si chiuderà la funzione senza alcun inserimento

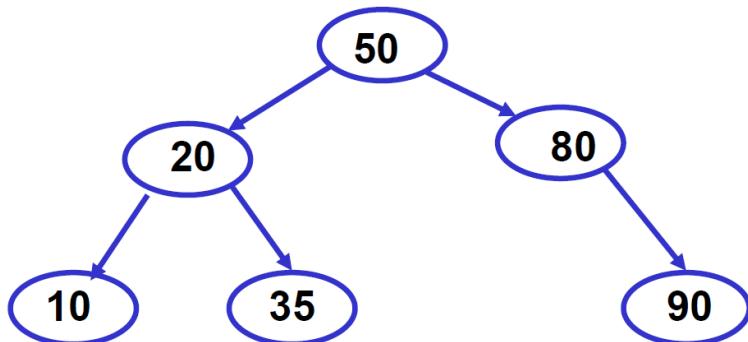
O(log n)

// O vado a SX o a DX per inserire il nodo, quindi scorro n/2 volte l'albero

ABR

ESEMPIO INSERTNODE

- Dato il seguente albero binario di ricerca, applico la funzione dandogli in ingresso la radice e il nodo da inserire (considero di inserire il nodo con etichetta 40)



- `insertNode(40, tree)`
-

1° Passaggio (SONO SULLA RADICE - NODO CON ETICHETTA 50)

- `!tree` No
 - `n<tree->label` → $40 < 50$ Sì
 - Ricorsione sul sottoalbero SX poiché essendo binario a SX ho i valori minori
-

1.1° Passaggio (SONO SUL NODO CON ETICHETTA 20)

- `!tree` No
 - `n<tree->label` → $40 < 20$ No
 - `n>tree->label` → $40 > 20$ Sì
 - Ricorsione sul sottoalbero DX poiché essendo binario a DX ho i valori maggiori
-

1.1.1° Passaggio (SONO SUL NODO CON ETICHETTA 35)

- `!tree` No
- `n<tree->label` → $40 < 35$ No
- `n>tree->label` → $40 > 35$ Sì
- Ricorsione sul sottoalbero DX poiché essendo binario a DX ho i valori maggiori

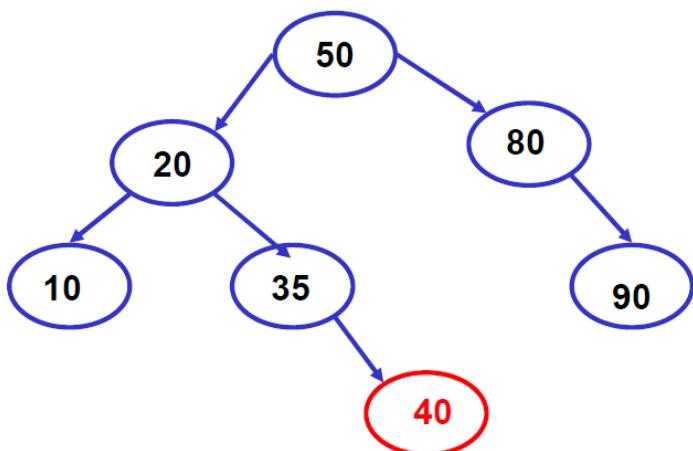
ABR

... CONTINUO ESEMPIO INSERTNODE

1.1.1.1° Passaggio (A DX 35 NON HA FIGLI)

- !tree Sì
 - Posso inserire il nodo rispettando l'ABR e lo inserisco come foglia, ponendo i suoi figli DX e SX a NULL e, chiudo questa chiamata ricorsiva insieme a tutte le altre precedenti, di conseguenza tutta la funzione
-

ALBERO OTTENUTO DOPO L'INSERIMENTO DI 40



ABR

```
void deleteMin(Node* &tree, InfoType &m){ //0
    if(tree->left) //1
        deleteMin(tree->left,m);
    else{ //2
        m=tree->label; //3
        Node *a=tree; //4
        tree=tree->right; //5
        delete a; //6
    }
}
```

TEMPO MIGLIORE E MEDIO

$O(\log n)$

TEMPO PEGGIORE

$O(n)$

// CANCELLANDO DIMINUISCO IL LIVELLO

FASE DI CANCELLAZIONE DEL MINIMO:

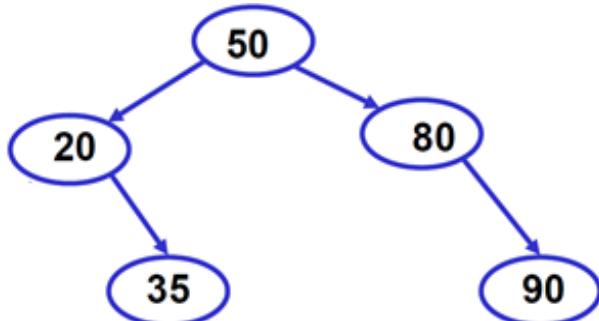
La cancellazione del minimo avviene quando mi sposto sul sottoalbero DX del nodo da cancellare punto **b. i.**

- 0- &tree cambia l'albero, &m etichetta restituita del nodo da cancellare
- 1- Se ci sono nodi a SX vuol dire che devo andare ricorsivamente a SX per trovare quello più piccolo visto che a SX ci stanno quelli più piccoli
- 2- Sono sul nodo più piccolo procedo per la cancellazione
- 3- L'etichetta del nodo più piccolo la restituisco tramite m
- 4- Metto questo nodo in un nuovo nodo
- 5- Connetto il padre dell'etichetta che ho restituito al figlio destro
- 6- Elimino il nodo dell'etichetta che ho restituito

ABR

ESEMPIO DELETEMIN

- Dato il seguente albero binario di ricerca, applico la funzione dandogli in ingresso la radice e un nodo, che prima della chiamata lo posso passare come mi pare, tanto quando viene applicata la funzione mi tornerà cambiato



- **deleteMin(tree,m)**
//per convenienza m=-1, al momento della chiamata, se chiamo questa funzione nel main
-

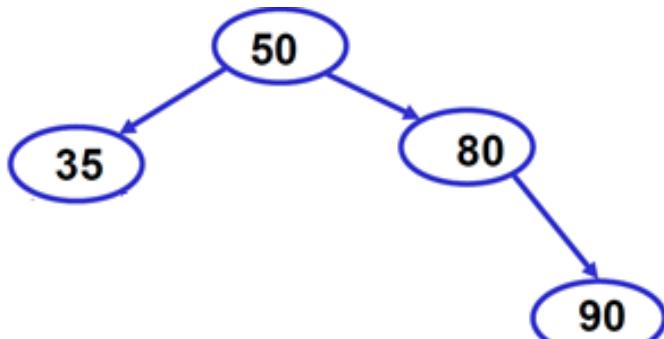
1° Passaggio (SONO SULLA RADICE – NODO ETICHETTA 50)

- tree->left Sì
 - Ricorsione sul sottoalbero SX
-

1.1° Passaggio (SONO SUL NODO CON ETICHETTA 20)

- tree->left No
- Ramo else
- m=tree->label → m=20 che è il valore minimo dell'albero visto che in un ABR il valore minimo sta tutto a SX e, verrà restituito dalla funzione poiché presente &
- Creo un nodo *a e ci metto il puntatore a questa radice, cioè a 20
- tree=tree->right → in 20 ci metto 30, connetto questa radice con il suo sottoalbero DX
- Cancello il nodo *a che conteneva 20

ALBERO OTTENUTO DOPO LA CANCELLAZIONE DEL MINIMO (20)



ABR

CANCELLAZIONE DI UN NODO IN UN ALBERO BINARIO DI RICERCA (FUNZIONAMENTO)

- 1) Cerco il nodo da cancellare effettuando una ricerca
- 2) Se il nodo viene trovato si possono verificare due situazioni diverse:
 - a. Se questo nodo ha un sottoalbero vuoto → Il suo padre viene connesso all'unico suo sottoalbero non vuoto
 - b. Se questo nodo ha entrambi i sottoalberi non vuoti:
 - i. Vado nel sottoalbero DX di esso
 - ii. Cerco in questo sottoalbero il nodo con etichetta più piccola tra tutti i nodi → tramite la **deleteMin**
 - iii. Restituisco l'etichetta più piccola di questo nodo (che poi andrà al posto del nodo da cancellare)
 - iv. "Allaccio" il figlio del nodo con etichetta più piccola con il suo padre
 - v. Cancello il nodo (da cancellare) e metto al posto della sua etichetta quella più piccola trovata prima

```
void deleteNode(InfoType n, Node* &tree){ //0
    if(tree){ //1
        if(n<tree->label){ deleteNode(n,tree->left); return; } //2
        if(n>tree->label){ deleteNode(n,tree->right); return; } //3
        if(!tree->left){ //4
            Node*a=tree;
            tree=tree->right;
            delete a;
            return;
        }
        if(!tree->right){ //5
            Node*a=tree;
            tree=tree->left;
            delete a;
            return;
        }
        deleteMin(tree->right, tree->label); //6
    }
}
```

ABR

- 0- &tree perché l'albero verrà modificato, n=nodo da cancellare
- 1- Verifico che ci sia un albero
- 2- Caso in cui il nodo da cancellare sia minore della radice → Mi sposto a SX nel sottoalbero di SX ricorsivamente, fino a quando non mi trovo nei punti 4- o 5-
- 3- Caso in cui il nodo da cancellare sia maggiore della radice → Mi sposto a DX nel sottoalbero di DX ricorsivamente, fino a quando non mi trovo nei punti 4- o 5-
- 4- Caso in cui n=tree->label e n non ha figli a SX:
 - a. Prendo il nodo in questione lo metto in nuovo nodo
 - b. Collego il padre del nodo in questione al suo figlio DX
 - c. Cancello il nodo a, che è quello da cancellare
- 5- Caso in cui n=tree->label e n non ha figli a X:
 - a. Prendo il nodo in questione lo metto in nuovo nodo
 - b. Collego il padre del nodo in questione al suo figlio SX
 - c. Cancello il nodo a, che è quello da cancellare
- 6- CASO in cui n=tree->label e n ha entrambi i figli
 - a. Cancello il nodo in questione rimpiazzandolo con il minimo del sottoalbero destro tramite la funzione **deleteMin** che me lo restituirà dentro il nodo tramite tree->label (parametro di **deleteMin**)

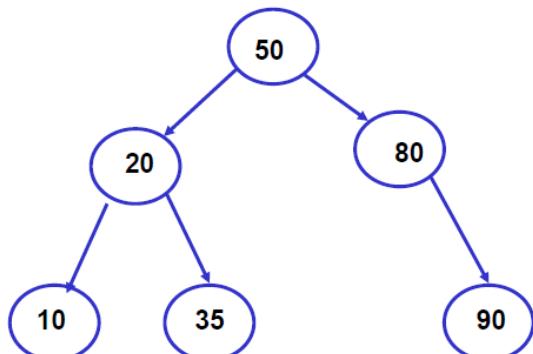
COMPLESSITÀ

O(log n)

ABR

ESEMPIO DELETENODE

- Dato il seguente albero binario di ricerca, applico la funzione dandogli in ingresso la radice e il nodo che voglio cancellare (per esempio voglio cancellare il nodo 20)



- **deleteNode(20, tree)**
-

1° Passaggio (SONO SULLA RADICE – NODO ETICHETTA 50)

- if tree Sì
 - $n < \text{tree-} \rightarrow \text{label} \rightarrow 20 < 50$ Sì
 - Ricorsione sul sottoalbero SX
-

1.1° Passaggio (SONO SUL NODO CON ETICHETTA 20)

- if tree Sì
- $n < \text{tree-} \rightarrow \text{label} \rightarrow 20 < 20$ No
- $n > \text{tree-} \rightarrow \text{label} \rightarrow 20 > 20$ No
- Sono nel caso in cui $20 = 20$ vado a vedere se ha figli a SX e DX
- $\text{!tree-} \rightarrow \text{left}$ No
- $\text{!tree-} \rightarrow \text{right}$ No
- Sono nel caso in cui 20 ha entrambi i figli, chiamo la **deleteMin**(dandogli come radice il sottoalbero DX che andrà a sostituire questo per mantenere l'integrità dell'ABR, e l'altro valore avrà 20 tra i parametri, ma non importa visto che poi ritornerà cambiata dalla funzione stessa)

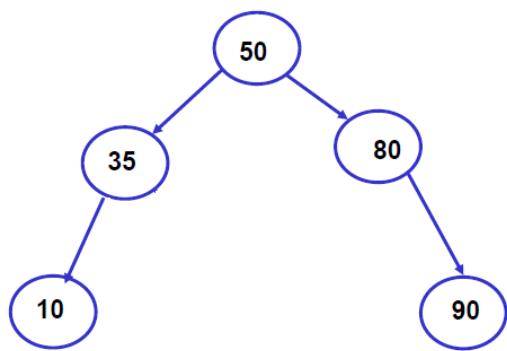
ABR

... CONTINUO ESEMPIO DELETENODE

1.2° Passaggio (SONO SUL NODO 35)

- if tree->left No, 35 non ha nulla a SX
- Ramo else
- m=tree->label → m=35 che è il valore minimo dell'albero, e verrà restituito dalla funzione poiché presente &
- Creo un nodo *a e ci metto il puntatore a questa radice, cioè a 35
- tree=tree->right → in 35 ci metto NULL, connetto questa radice con il suo sottoalbero DX che non ha, di conseguenza va a NULL
- Cancello il nodo *a che conteneva 35
- Restituirò il 35 tramite m nel secondo parametro e chiudendo questa chiamata andrà a posizionarsi al posto di tree->label che era 20 e, che dovevo cancellare all'inizio della funzione

ALBERO OTTENUTO DOPO LA CANCELLAZIONE DI 20



CLASS HEAP

```
class Heap{  
    int *h;  
    int last; //1  
  
    //metodi privati  
    void up(int);  
    void down(int);  
    void exchange(int i, int j){  
        int k=h[i]; h[i]=h[j]; h[j]=k;  
    }  
  
    //metodi pubblici  
    Public:  
    Heap(int);  
    ~Heap();  
    void insert(int); //2  
    int extract(); //3  
};
```

-
- 1- Indice dell'ultimo elemento occupato all'interno del vettore
 - 2- Inserimento di un nodo
 - 3- Estrazione dell'elemento maggiore (radice)

COSTRUTTORE

```
Heap::Heap(int n){  
    h=new int[n]; //Alloco la memoria di dimensione n  
    last=-1; //Poiché è vuoto inizialmente lo heap
```

CLASS HEAP

DISTRUTTORE

```
-----  
Heap::~Heap(){ delete []h; } //cancello il vettore  
-----
```

INSERIMENTO

```
-----  
void Heap::insert(int x){ //0  
    h[++last]=x; //1  
    up(last); //2  
}  
-----
```

- 1- L'elemento x è il valore da inserire
- 2- Prima faccio ++ e poi uso last perché aumento l'ultimo elemento occupato visto che ne sto occupando uno in più ed è lì che ci metto x
- 3- Porto l'elemento appena inserito (di indice last) su alla posizione giusta fino a che non faccio rispettare le condizioni dello heap

```
-----  
void Heap::up(int i){ //0  
    if(i>0) //1  
        if(h[i] > h[(i-1)/2]){ //2  
            exchange(i, (i-1)/2); //3  
            up((i-1)/2); //4  
        } //5  
}  
-----
```

- 0- i è l'indice dell'elemento da far risalire (in questo caso last)
- 1- Se non sono sulla radice
- 2- Controllo se l'elemento che sto analizzando (inserito) è maggiore del suo padre
→ indice del padre **(i-1)/2**
- 3- Scambio figlio col padre poiché il figlio deve essere più piccolo del padre
- 4- Chiamo up sul padre per vedere se l'elemento, cambiato (padre figlio) è ancora maggiore del suo padre (cioè padre del padre del last iniziale)
- 5- Continuo fino a quando viene chiamata la funzione con l'indice 0 (la radice) o quando l'elemento è inferiore al padre

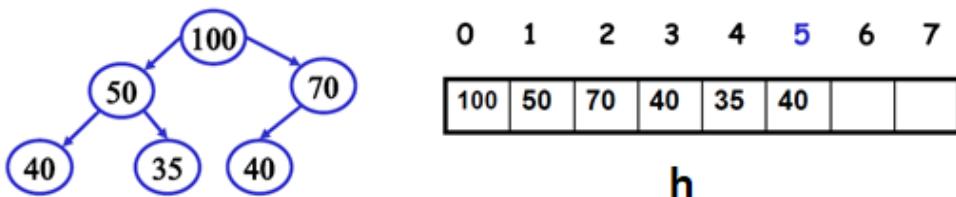
CLASS HEAP

COMPLESSITÀ

O(log n)

- La complessità è logaritmica perché ogni chiamata risale di un livello e i livelli di un albero bilanciato o quasi bilanciato sono **log n**

ESEMPIO DI INSERIMENTO

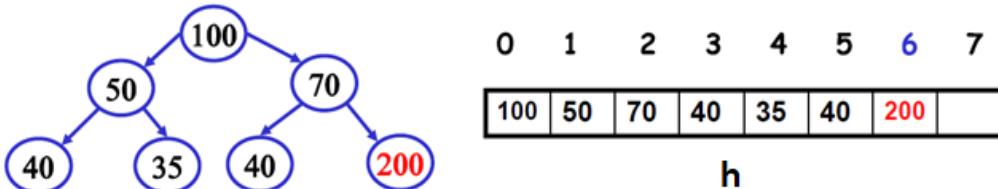


→ considerando questo heap, faccio **insert(200)** //last=5, x=200

1° Passaggio

- $[h++\text{last}] = x \rightarrow h++$ incremento last che da 5 va a 6, poi ci metto x che è 200

OTTENGO:

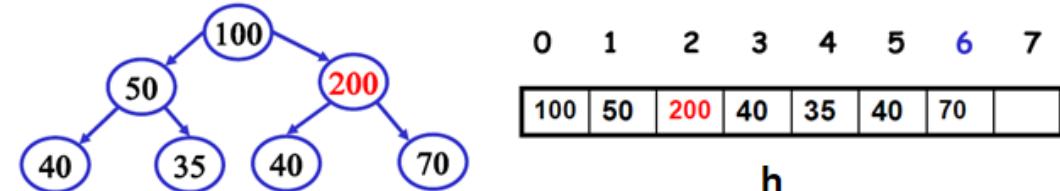


- Chiamo la funzione **up(last)** → up(6)

2° Passaggio ($i=\text{last}$ passato tra i parametri → $i=6$)

- $i>0 \rightarrow 6>0$ Sì
- $h[i]>h[(i-1)/2] \rightarrow h[6] > h[(6-1)/2] \rightarrow$ divisione intera $\rightarrow h[6]>h[2] \rightarrow 200>70$ Sì
- Scambio questi due valori

OTTENGO:



- Ricorsione **up(2);**

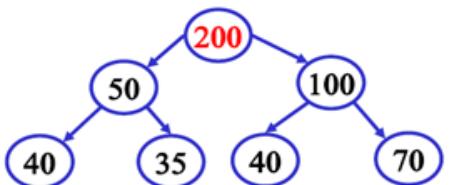
CLASS HEAP

... CONTINUO ESEMPIO DI INSERIMENTO

2.1° Passaggio (i=last passato tra i parametri → i=2)

- $i > 0 \rightarrow 2 > 0$ Sì
- $h[i] > h[(i-1)/2] \rightarrow h[2] > h[(2-1)/2] \rightarrow$ divisione intera $\rightarrow h[2] > h[0] \rightarrow 200 > 100$ Sì
- Scambio questi due valori

OTTENGO:



| | | | | | | | |
|-----|----|-----|----|----|----|----|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 200 | 50 | 100 | 40 | 35 | 40 | 70 | |

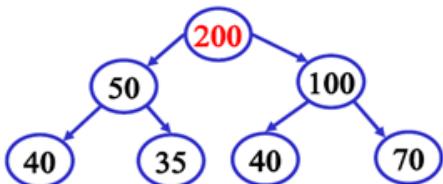
h

- Ricorsione **up(0);**

2.1.1° Passaggio (i=last passato tra i parametri → i=0)

- $i > 0 \rightarrow 0 > 0$ No, chiudo questa ricorsione e la precedente, quindi la *up* finisce e di conseguenza tutta la funzione *insert* poiché ho sistemato 200 a suo posto

HEAP FINALE:



| | | | | | | | |
|-----|----|-----|----|----|----|----|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 200 | 50 | 100 | 40 | 35 | 40 | 70 | |

h

ESTRAZIONE

```
int heap::extract(){
    int r=h[0];                                //1
    h[0]=h[last--];                            //2
    down(0);                                    //3
    return r;                                    //4
}
```

CLASS HEAP

Funzione che restituisce il primo elemento dell'array cioè la radice con il seguente procedimento:

- 1- Dichiaro una variabile e ci metto la radice
- 2- Prendo l'ultimo elemento e lo metto al posto della radice e poi decremento l'ultimo elemento
- 3- Faccio scendere l'elemento appena messo in testa alla radice, tramite scambi padre-figlio, fino a che riesco a mantenere le proprietà dello heap
- 4- Ritorno la radice e di conseguenza ho l'estrazione

OSS:

- Non metto nessun parametro alla funzione, perché non sto andando a passargli nulla ma sto solamente estraendo

```
-----  
void Heap::down(int i){                                //0  
    int son= 2*i+1;                                    //1  
    if(son == last){                                 //2  
        if(h[son]>h[i])  
            exchange(i,last);                         //3  
    }  
    else if(son<last){  
        if(h[son]<h[son+1]) son++;                  //6  
        if(h[son]>h[i]){  
            exchange(i,son);                        //7  
            down(son);                            //8  
        }  
    }  
}                                                       //TERMINO! E VALE ANCHE SE i NON HA FIGLI  
-----
```

CLASS HEAP

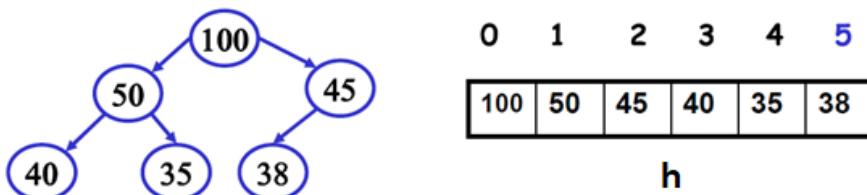
- 0- i è l'indice dell'elemento da far scendere
- 1- Imposto son come indice del figlio sinistro (se esiste)
- 2- Controllo se i ha un solo figlio, cioè se son è l'ultimo dell'array
(mi riferisco al nuovo array, poiché last è cambiato nella funzione extract)
- 3- Controllo se il figlio è maggiore del padre
- 4- Quindi faccio lo scambio, altrimenti termina
- 5- Caso in cui i ha entrambi i figli e questo vuol dire che ho un fratello a DX, poiché a DX di una radice ho elementi > rispetto alla parte SX di una radice
- 6- Confronto fra fratelli: controllo se il figlio di SX è più piccolo di quello di DX dove quello di DX è indicato come [son+1], e se così fosse vado a concentrarmi su quello di DX facendo son++
- 7- Controllo se il figlio SX è maggiore del padre
- 8- Faccio lo scambio
- 9- Chiamo down sulla nuova posizione appena ottenuta

→ COMPLESSITÀ

O(log n)

- La complessità è logaritmica anche qua, perché ogni chiamata risale di un livello e, i livelli di un albero bilanciato o quasi bilanciato sono **log n**

ESEMPIO DI ESTRAZIONE

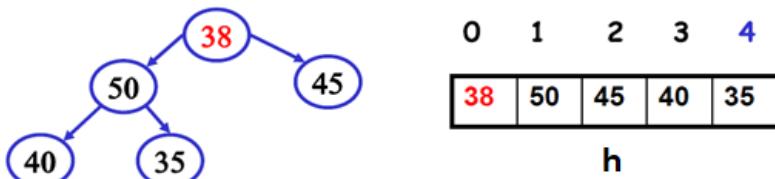


→ Considerando questo heap faccio **extract()** -> 100 (primo elemento dell'array)

1° Passaggio //last=5

- $r=h[0]=100$
- $h[0]=h[last--]$, ci metto 38 e decremento last che diventa 4

OTTENGO:



- Chiamata a **down(0)**

CLASS HEAP

... CONTINUO ESEMPIO DI ESTRAZIONE

2° Passaggio //last=4, i=0 passato tra i parametri

- son=2*i+1=1
- son=last $\rightarrow 1=4$ No
- son<last $\rightarrow 1<4$ Sì
- h[son]<h[son+1] $\rightarrow h[1]<h[2] \rightarrow 50<45$ No
- h[son]>h[i] $\rightarrow h[1]>h[0] \rightarrow 50>38$ Sì
- Li scambio

OTTENGO:

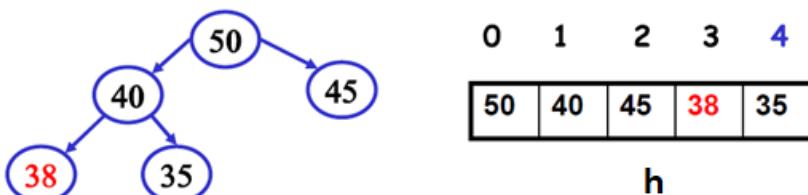


- Ricorsione **down(1) \rightarrow down(son)**
-

2.1° Passaggio //last=4, i=1 passato tra i parametri

- son=2*i+1=3
- son=last $\rightarrow 3=4$ No
- son<last $\rightarrow 3<4$ Sì
- h[son]<h[son+1] $\rightarrow h[3]<h[4] \rightarrow 40<35$ No
- h[son]>h[i] $\rightarrow h[3]>h[1] \rightarrow 40>38$ Sì
- Li scambio

OTTENGO:



- Ricorsione **down(3) \rightarrow down(son)**

CLASS HEAP

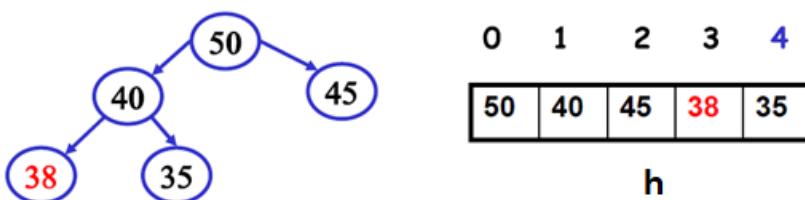
... CONTINUO ESEMPIO DI ESTRAZIONE

2.1.1° Passaggio $//last=4, i=3$ passato tra i parametri

- $son=2*i+1=7$
 - $son=last \rightarrow 7=4$ No
 - $son < last \rightarrow 7 < 4$ No
 - Sono arrivato alla foglia facendo i controlli quindi, chiudo questa ricorsione e le precedenti (cioè tutta la chiamata down) tornando al passaggio 2°
-

3° Passaggio

- Ritorno r che è il valore estratto di 100 con l'heap finale che si presenta così



STAMPA IN ORDINE SIMMETRICO GLI ELEMENTI DI UNO HEAP (INORDER)

```
void inOrderH(int *A, int i, int last){           //i nel main deve essere 0
    if(i>last) return;                            //0
    inOrderH(A, 2*i+1, last);                     //1
    cout<<A[i];                                //2
    inOrderH(A, 2*i+2, last);                     //3
}
```

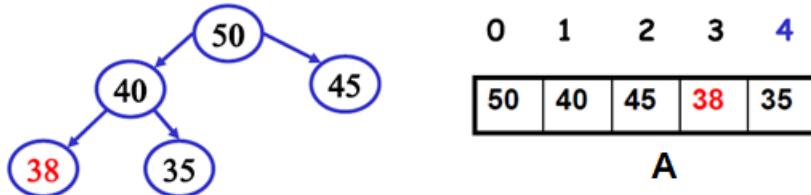
- 0- Condizione che mi fa verificare se sono arrivato alla foglia
- 1- Faccio la chiamata sul sottoalbero SX indicando come radice del sottoalbero SX il figlio SX che è dettato dalla formula $2*i+1$
- 2- Leggo la radice
- 3- Faccio la chiamata sul sottoalbero DX indicando come radice del sottoalbero DX il figlio DX che è dettato dalla formula $2*i+2$

//Classica inOrder con l'albero come array essendo uno HEAP, rispettando le condizioni di uscita da uno heap e considerando i suoi figli con le formule appropriate

CLASS HEAP

ESEMPIO DI INORDERH

- Considerato il seguente Heap e l'array A associato applico la visita **inOrder(A,0,4)**



1° Passaggio // $i=0$, $last=4$ (SONO SUL 50)

- $i > last \rightarrow 0 > 4$ No
 - Chiamata ricorsiva sul sottoalbero SX indicandone il figlio SX dato da $2*i+1=1$
-

1.1° Passaggio // $i=1$, $last=4$ (SONO SUL 40)

- $i > last \rightarrow 1 > 4$ No
 - Chiamata ricorsiva sul sottoalbero SX indicandone il figlio SX dato da $2*i+1=3$
-

1.1.1° Passaggio // $i=3$, $last=4$ (SONO SUL 38)

- $i > last \rightarrow 3 > 4$ No
 - Chiamata ricorsiva sul sottoalbero SX indicandone il figlio SX dato da $2*i+1=7$
-

1.1.1.1° Passaggio // $i=7$, $last=4$

- $i > last \rightarrow 7 > 4$ Si
 - Return e chiudo questa chiamata ricorsiva e vado al passaggio precedente (1.1.1°)
-

1.1.2° Passaggio // $i=3$, $last=4$ (SONO SUL 38)

- Stampo 38
 - Chiamata ricorsiva sul sottoalbero DX indicandone il figlio DX dato da $2*i+2=8$
-

1.1.2.1° Passaggio // $i=8$, $last=4$

- $i > last \rightarrow 8 > 4$ Si
- Return e chiudo questa chiamata ricorsiva chiudendo il passaggio precedente e vado al passaggio (1.1°)

CLASS HEAP

... CONTINUO ESEMPIO DI INORDERH

1.2° Passaggio $//i=1, last=4$ (SONO SUL 40)

- Stampo 40
 - Chiamata ricorsiva sul sottoalbero DX indicandone il figlio DX dato da $2*i+2=4$,
-

1.2.1° Passaggio $//i=4, last=4$ (SONO SUL 35)

- $i > last \rightarrow 4 > 4$ No
 - Chiamata ricorsiva sul sottoalbero SX indicandone il figlio SX dato da $2*i+1=9$
-

1.2.1.1° Passaggio $//i=9, last=4$

- $i > last \rightarrow 9 > 4$ Si
 - Return e chiudo questa chiamata ricorsiva e vado al passaggio precedente (1.2.1°)
-

1.2.2° Passaggio $//i=4, last=4$ (SONO SUL 35)

- Stampo 35
 - Chiamata ricorsiva sul sottoalbero DX indicandone il figlio DX dato da $2*i+2=10$
-

1.2.2.1° Passaggio $//i=10, last=4$

- $i > last \rightarrow 10 > 4$ Si
 - Return e chiudo questa chiamata ricorsiva chiudendo il passaggio precedente e vado al passaggio (1°)
-

2° Passaggio $//i=0, last=4$ (SONO SUL 50)

- Stampo 50
 - Chiamata ricorsiva sul sottoalbero DX indicandone il figlio DX dato da $2*i+2=2$
-

2.1° Passaggio $//i=2, last=4$ (SONO SUL 45)

- $i > last \rightarrow 2 > 4$ No
 - Chiamata ricorsiva sul sottoalbero SX indicandone il figlio SX dato da $2*i+1=5$
-

CLASS HEAP

... CONTINUO ESEMPIO DI INORDERH

2.1.1° Passaggio $//i=5, last=4$

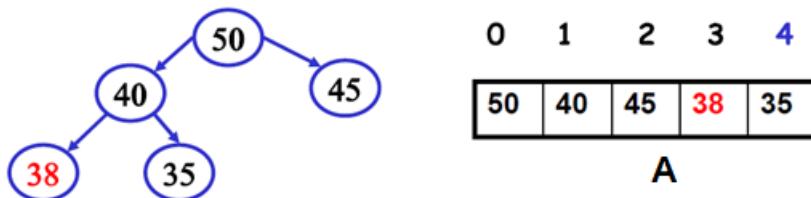
- $i > last \rightarrow 5 > 4$ Sì
 - Return e chiudo questa chiamata ricorsiva e vado al passaggio precedente (2.1°)
-

2.2° Passaggio $//i=2, last=4$ (SONO SUL 45)

- Stampo 45
 - Chiamata ricorsiva sul sottoalbero DX indicandone il figlio DX dato da $2*i+2=6$
-

2.2.1° Passaggio $//i=6, last=4$

- $i > last \rightarrow 6 > 4$ Sì
 - Return e chiudo questa chiamata ricorsiva chiudendo tutta la funzione
-



Inorder sullo HEAP $\rightarrow 38\ 40\ 35\ 50\ 45$

HEAPSORT

```
void heapsort(int *A, int n){ //0
    buildHeap(A, n-1); //1
    int i=n-1;
    while(i>0){ //2
        extract(A,i);
    }
}
```

0- Array da ordinare A e n dimensione array

1- Trasformo l'array A in uno heap

2- Per n volte estraggo la radice con il valore puntato da last (che sarebbe n-1) ed ogni volta i viene decrementato e restituito (che sarebbe il valore di last decrementato)

COMPLESSITÀ

O(n logn)

//La funzione “buildheap” ha complessità $O(n)$, mentre la funzione “extract” ha complessità $O(n \log n)$

FUNZIONAMENTO DELL'ALGORITMO

- Ogni passata dell'algoritmo si estraе la radice (che è il valore maggiore), ad ogni estrazione si scambia il valore con il valore puntato da last (così il valore più grande va in fondo), si decrementa last (così il valore più grande è già nell'ultima posizione e d'ora in avanti si considera la posizione penultima). Infine, il valore che abbiamo scambiato prima si sposta con la funzione “down” verso il basso fino alla posizione corretta.

HEAPSORT

```
void down(int *h, int i, int last){ //0
    int son=(2*i)+1;
    if(son==last){
        if(h[son]>h[i])
            exchange(h,i,last); //1
    }
    else if(son<last){
        if(h[son]<h[son+1]) son++;
        if(h[son]>h[i]){
            exchange(h,i,son); //2
            down(h,son,last);
        }
    }
}
```

- 0- Gli passo sia h, sia last oltre che ad i poiché non siamo in una classe che ha i campi privati per il resto la funzione è identica a quella con la classe tranne che nelle chiamate di “exchange” e “down”
- 1- Anche nella exchange ho necessità di passargli l’array h, altrimenti con la classica exchange, gli potrei passare h[i] e h[last]
- 2- Stessa cosa del discorso precedente con h[i] e h[son] nella exchange classica

Complessità

O(log n)

HEAPSORT

```
void extract(int *h, int &last){ //0  
    exchange(h,0,last--); //1  
    down(h,0,last); //2  
}
```

- 0- &last perché ogni volta che viene richiamata dentro la funzione “heapsort” cambia il suo valore
- 1- Scambio la radice (valore più alto) con l’elemento puntato da last e poi decremento last
- 2- “down” della radice all’interno del nuovo heap, per portare il valore della radice al posto corretto

- Alla fine di n estrazioni ho il vettore ordinato in maniera crescente

Complessità

O(log n)

Perché “exchange” è $O(1)$ e la “down” è $O(\log n)$

```
void buildHeap(int *A, int n){ //0  
    for(int i=n/2; i>=0; i--) //1  
        down(A,i,n); //2  
}
```

- 0- $n+1$ è la dimensione dell’array quindi passandogli $n \rightarrow$ La dimensione dell’array sarà diminuita di 1, perché è $n-1$ dentro l’heapSort
- 1- Ciclo sulla prima metà degli elementi a ritroso
- 2- Eseguo “down” sulla prima metà degli elementi dell’array (la seconda metà sono foglie) partendo dall’elemento centrale tornando indietro fino al primo

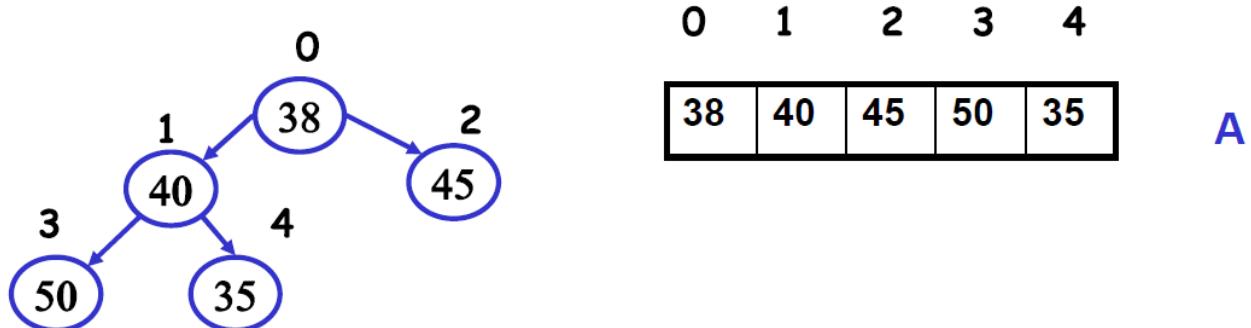
Complessità

O(n)

HEAPSORT

ESEMPIO HEAPSORT

Dato il seguente heap, ci applico l'heapSort → **heapSort(A, int 5);**



1° Passaggio (A, n=5)

- Chiamata a buildHeap(A, n-1) → **buildheap(A,4)**

1.1° Passaggio (A, n=4) //sono dentro la *buildHeap*

- Ciclo $i=n/2=2$
- $i>=0 \rightarrow 2>=0$ Sì
- Chiamata **down(A,2,4)** → A diventa h, 2 è la i, n=4 diventa last

1.1.1° Passaggio (h, i=2, last=4)

- $son=(2*i)+1=5$
- $son=last \rightarrow 5=4$ No
- $son<last \rightarrow 5<4$ No
- Chiudo la chiamata tornando al passaggio precedente (1.1°)

1.2° Passaggio (A, n=4, i=2) //sono tornato dentro la *buildHeap*

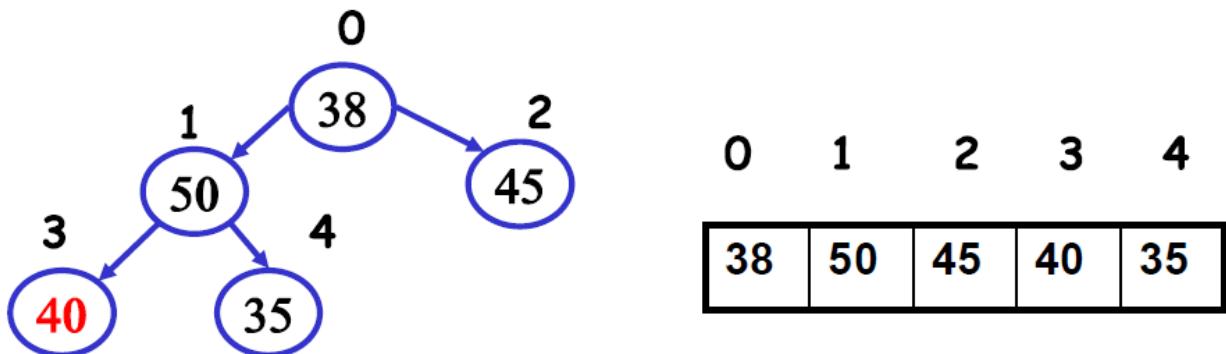
- $i-- \rightarrow 1$
- $i>=0 \rightarrow 1>=0$ Sì
- Chiamata **down(A,1,4)** → A diventa h, 1 è la i, n=4 diventa last

HEAPSORT

... CONTINUO ESEMPIO HEAPSORT

1.2.1° Passaggio (h , $i=1$, $last=4$)

- $son = (2*i) + 1 = 3$
- $son = last \rightarrow 3 = 4$ No
- $son < last \rightarrow 3 < 4$ Sì
- $h[son] < h[son+1] \rightarrow h[3] < h[4] \rightarrow 50 < 35$ No
- $h[son] > h[i] \rightarrow h[3] > h[1] \rightarrow 50 > 40$ Sì
- Scambio $h[i]$ con $h[son]$ $\rightarrow h[1]$ con $h[3]$ $\rightarrow 40$ con 50
//ho uno scambio in quanto ho figlio maggiore del padre



- Ricorsione **down**(h , 3 , 4) // $son=3$

1.2.1.1° Passaggio (h , $i=3$, $last=4$)

- $son = (2*i) + 1 = 7$
- $son = last \rightarrow 7 = 4$ No
- $son < last \rightarrow 7 < 4$ No esco da questa ricorsione e dalla chiamata down precedente tornando al punto (1.2°)

1.3° Passaggio (A , $n=4$, $i=1$) //sono tornato dentro la buildHeap

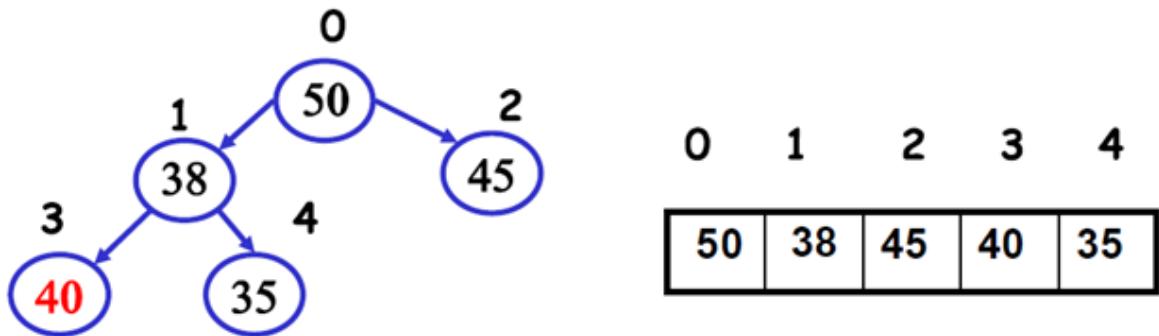
- $i-- \rightarrow 0$
- $i >= 0 \rightarrow 0 >= 0$ Sì
- Chiamata **down**($A, 0, 4$) $\rightarrow A$ diventa h , 0 è la i , $n=4$ diventa $last$

HEAPSORT

... CONTINUO ESEMPIO HEAPSORT

1.3.1° Passaggio (h , $i=0$, $last=4$)

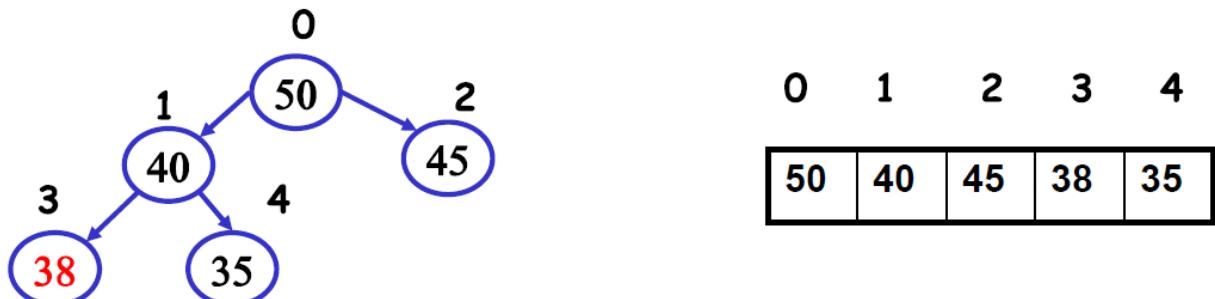
- $son = (2*i) + 1 = 1$
- $son = last \rightarrow 1 = 4$ No
- $son < last \rightarrow 1 < 4$ Sì
- $h[son] < h[son+1] \rightarrow h[1] < h[4] \rightarrow 50 < 35$ No
- $h[son] > h[i] \rightarrow h[1] > h[0] \rightarrow 50 > 38$ Sì
- Scambio $h[i]$ con $h[son]$ $\rightarrow h[1]$ con $h[0]$ $\rightarrow 50$ con 38
//ho uno scambio in quanto ho figlio maggiore del padre



- Ricorsione **down**(h , 1 , 4) // $son=1$

1.3.1.1° Passaggio (h , $i=1$, $last=4$)

- $son = (2*i) + 1 = 3$
- $son = last \rightarrow 3 = 4$ No
- $son < last \rightarrow 3 < 4$ Sì
- $h[son] < h[son+1] \rightarrow h[3] < h[4] \rightarrow 40 < 35$ No
- $h[son] > h[i] \rightarrow h[3] > h[1] \rightarrow 40 > 38$ Sì
- Scambio $h[i]$ con $h[son]$ $\rightarrow h[1]$ con $h[3]$ $\rightarrow 38$ con 40
//ho uno scambio in quanto ho figlio maggiore del padre



- Ricorsione **down**(h , 3 , 4) // $son=3$

HEAPSORT

... CONTINUO ESEMPIO HEAPSORT

1.3.1.1.1° Passaggio (h , $i=3$, $last=4$)

- $son=(2*i)+1=7$
- $son=last \rightarrow 7=4$ No
- $son < last \rightarrow 7 < 4$ No, chiudo la ricorsione e quelle precedenti e torno al punto 1.3°

1.4° Passaggio (A , $n=4$, $i=0$) //sono tornato dentro la buildHeap

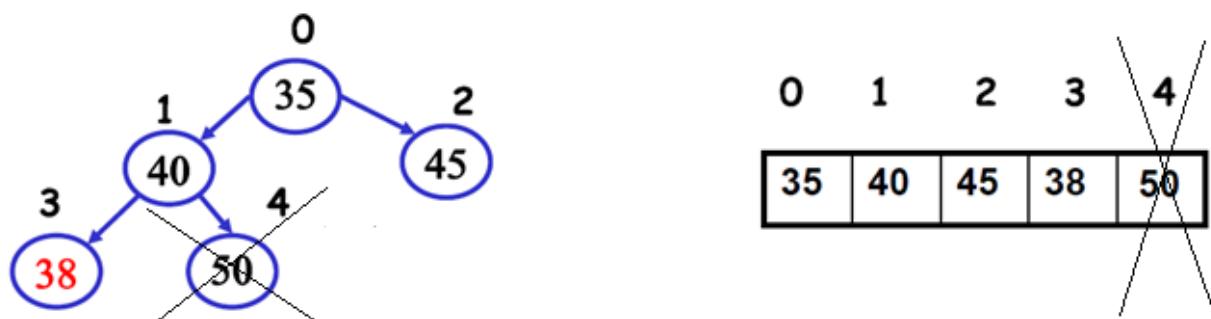
- $i-- \rightarrow -1$
- $i>=0 \rightarrow -1>=0$ No chiudo il ciclo del passaggio 1.1° con n che è tornata 5 poiché al passaggio 1° era 5 \rightarrow A questo punto ho creato lo heap

2° Passaggio ($n=5$)

- $i=n-1=4$
- Ciclo
- $i>0 \rightarrow 4>0$ Sì
- Chiamata **extract**($A, 4$);

2.1° Passaggio ($A=h$, $i=4=last$)

- Scambio $h[0]$ con $h[last]$ \rightarrow 50 con 35 e nel mentre decremento $last--$ che è 3
//Ho estratto il 50



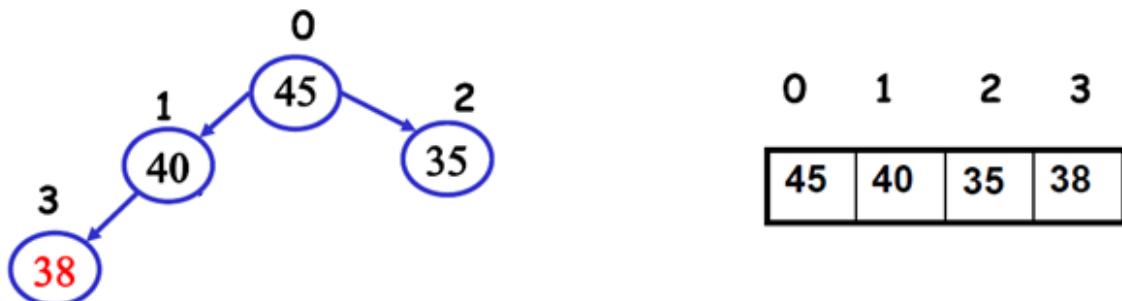
- Chiamata **down**($h, 0, last=3$) \rightarrow last alla chiusura della down tornerà ad i poiché è presente & nella extract, dove viene fatto $last--$

HEAPSORT

... CONTINUO ESEMPIO HEAPSORT

2.1.1 Passaggio ($h, i=0, \text{last}=3$)

- $\text{son} = (2*i) + 1 = 1$
- $\text{son} = \text{last} \rightarrow 1 = 3$ No
- $\text{son} < \text{last} \rightarrow 1 < 3$ Sì
- $h[\text{son}] < h[\text{son}+1] \rightarrow h[1] < h[2] \rightarrow 40 < 45$ Sì
- $\text{son}++ = 2$
- $h[\text{son}] > h[i] \rightarrow h[2] > h[0] \rightarrow 45 > 35$ Sì
- Scambio $h[i]$ con $h[\text{son}] \rightarrow h[0]$ con $h[2] \rightarrow 35$ con 45
//ho uno scambio in quanto ho figlio maggiore del padre



- Ricorsione **down**($h, 1, 3$)
-

2.1.1.1 Passaggio ($h, i=1, \text{last}=3$)

- $\text{son} = (2*i) + 1 = 3$
 - $\text{son} = \text{last} \rightarrow 3 = 3$ Sì
 - $h[\text{son}] > h[i] \rightarrow h[3] > h[1] \rightarrow 38 > 40$ No
 - Chiudo questa ricorsione e tutta la down tornando al passaggio 2° con la i decrementata a 3 al momento della chiamata down
-

3° Passaggio ($i=3$)

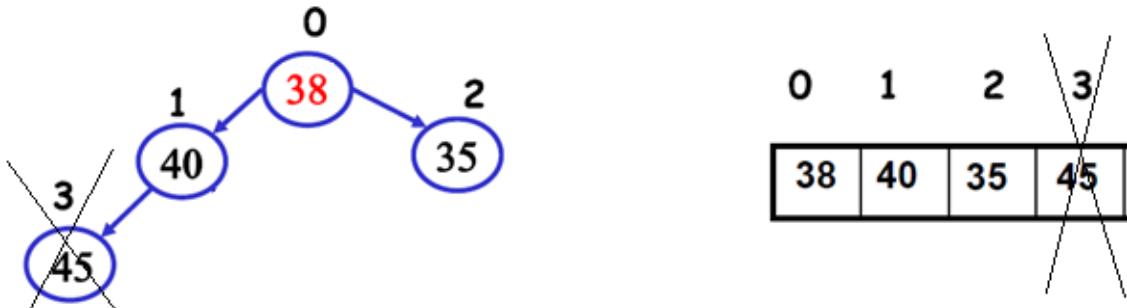
- $i > 0 \rightarrow 3 > 0$ Sì
- Chiamata **extract**($A, 3$);

HEAPSORT

... CONTINUO ESEMPIO HEAPSORT

3.1° Passaggio ($A=h$, $i=3=\text{last}$)

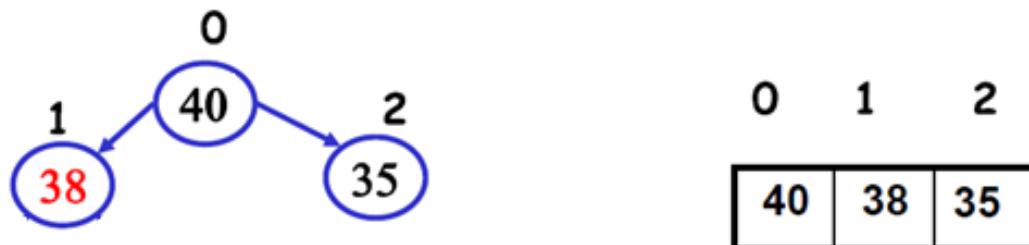
- Scambio $h[0]$ con $h[\text{last}] \rightarrow 45$ con 38 e nel mentre decremento $\text{last}--$ che è 2
//Ho estratto il 45



- Chiamata **down**(h , 0, $\text{last}=2$) \rightarrow last alla chiusura della down tornerà ad i poiché è presente & nella extract

3.1.1° Passaggio (h , $i=0$, $\text{last}=2$)

- $\text{son}=(2*i)+1=1$
- $\text{son}=\text{last} \rightarrow 1=2$ No
- $\text{son}<\text{last} \rightarrow 1<2$ Sì
- $h[\text{son}]<h[\text{son}+1] \rightarrow h[1]<h[2] \rightarrow 40<35$ No
- $h[\text{son}]>h[i] \rightarrow h[1]>h[0] \rightarrow 40>38$ Sì
- Scambio $h[i]$ con $h[\text{son}] \rightarrow h[0]$ con $h[1] \rightarrow 40$ con 38
//ho uno scambio in quanto ho figlio maggiore del padre



- Ricorsione **down**(h , 1, 2)

3.1.1.1 Passaggio (h , $i=1$, $\text{last}=2$)

- $\text{son}=(2*i)+1=3$
- $\text{son}=\text{last} \rightarrow 3=2$ No
- $\text{son}<\text{last} \rightarrow 3<2$ No
- Chiudo questa ricorsione e tutta la down tornando al passaggio 3° con la i decrementata a 2 al momento della chiamata down

HEAPSORT

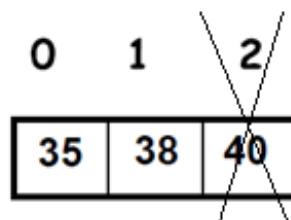
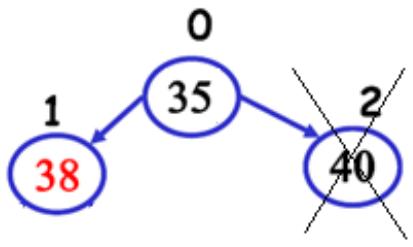
... CONTINUO ESEMPIO HEAPSORT

4° Passaggio ($i=2$)

- $i>0 \rightarrow 2>0$ Sì
- Chiamata **extract(A,2);**

4.1° Passaggio ($A=h$, $i=2=\text{last}$)

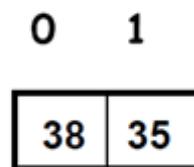
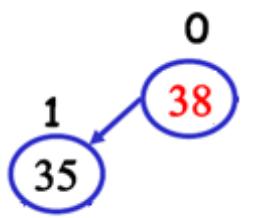
- Scambio $h[0]$ con $h[\text{last}] \rightarrow 40$ con 35 e nel mentre decremento $\text{last}--$ che è 1
//Ho estratto il 40



- Chiamata **down(h, 0, last=1)** → last alla chiusura della down tornerà ad i poiché è presente & nella extract

4.1.1° Passaggio (h , $i=0$, $\text{last}=1$)

- $\text{son}=(2*i)+1=1$
- $\text{son}=\text{last} \rightarrow 1=1$ Sì
- $h[\text{son}]>h[i] \rightarrow h[1]>h[0] \rightarrow 38>35$ Sì
- Scambio $h[i]$ con $h[\text{son}] \rightarrow h[0]$ con $h[1] \rightarrow 35$ con 38
//ho uno scambio in quanto ho figlio maggiore del padre



- Ricorsione **down(h, 1, 1)**

4.1.1.1 Passaggio (h , $i=1$, $\text{last}=1$)

- $\text{son}=(2*i)+1=3$
- $\text{son}=\text{last} \rightarrow 3=1$ No
- $\text{son}<\text{last} \rightarrow 3<1$ No
- Chiudo questa ricorsione e tutta la down tornando al passaggio 4° con la i decrementata a 1 al momento della chiamata down

HEAPSORT

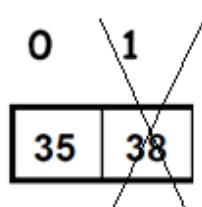
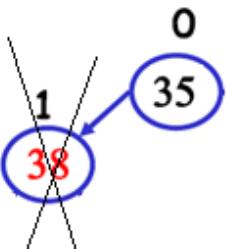
... CONTINUO ESEMPIO HEAPSORT

5° Passaggio ($i=1$)

- $i > 0 \rightarrow 1 > 0$ Sì
 - Chiamata **extract(A,1);**
-

5.1° Passaggio ($A=h$, $i=1=last$)

- Scambio $h[0]$ con $h[last]$ $\rightarrow 38$ con 35 e nel mentre decremento $last--$ che è 0
//Ho estratto il 38



- Chiamata **down(h, 0, last=0)** \rightarrow last alla chiusura della down tornerà ad i poiché è presente & nella extract
-

5.1.1° Passaggio (h , $i=0$, $last=0$)

- $son=(2*i)+1=1$
 - $son=last \rightarrow 1=0$ No
 - $son < last \rightarrow 1 < 0$ No
 - Chiudo questa ricorsione e tutta la down tornando al passaggio 5° con la i decrementata a 1 al momento della chiamata down
-

6° Passaggio ($i=0$)

- $i > 0 \rightarrow 0 > 0$ No
- Fine ciclo, chiudo tutta la funzione non ho più nulla da estrarre e poiché l'elemento più piccolo è già in cima e l'array è ordinato

0 1 2 3 4

| | | | | |
|----|----|----|----|----|
| 35 | 38 | 40 | 45 | 50 |
|----|----|----|----|----|

COUNTING SORT

- Si utilizza quando si conosce il valore massimo e minimo degli elementi da ordinare

FUNZIONAMENTO:

- Per l'algoritmo utilizziamo un array A dove sono inseriti una serie di numeri interi fino a n (0 fino a n-1)
- I numeri interi inseriti possono essere ripetuti lungo l'array
- Per l'ordinamento si utilizza un array ausiliario composto così:
 - Si prendono gli elementi di dell'array di partenza e ne consideriamo il più piccolo e il più grande.
 - Il più piccolo sarà 0 ed il più grande sarà k
 - Sappiamo che tutti i numeri dell'array di partenza saranno compresi tra il minimo 0 e il massimo k
 - Questi numeri nell'array ausiliario sono rappresentati dall'indice stesso dell'array ausiliario e quindi, essendo da 0 a k i numeri l'array ausiliario, deve avere k+1 elementi per farli entrare tutti
 - Gli elementi dell'array rappresentano quante volte sono ripetuti gli elementi (dati dall'indice dell'array ausiliario), nell'array di partenza
- L'inserimento dei valori ordinati utilizzando l'array ausiliario avverrà sull'array di partenza con i valori al posto giusto

ARRAY DI PARTENZA A[n]

- n=14
- Minimo = 0
- Massimo=7
- Possibili valori da 0 a 7 (0 a k) $\rightarrow 8=k+1$

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| 7 | 7 | 4 | 4 | 7 | 5 | 4 | 7 | 4 | 5 | 1 | 1 | 0 | 1 |

← Indice da 0 a n
← Valori A[i]

ARRAY AUSILIARIO C[k+1]

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

← Indice da 0 a k+1
← Valori C[i] che indicano quante volte sono ripetuti i valori rappresentati dall'indice k

ARRAY DI PARTENZA A (ORDINATO)

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| 0 | 1 | 1 | 1 | 4 | 4 | 4 | 4 | 5 | 5 | 7 | 7 | 7 | 7 |

// Per scorrere l'array A utilizzo l'indice j e per scorrere l'array ausiliario C utilizzo l'indice i

COUNTING SORT

```
void counting_sort(int A[], int k, int n){  
    int i,j;  
    int C[k+1];  
O(k) for (int i=0; i<=k; i++) C[i]=0; //1  
O(n) for (int j=0; j<n; j++) C[A[j]]++; //2  
    j=0; //3  
    for (int i=0; i<=k; i++) //4  
        while(C[i]>0){ //5  
            A[j]=i; //6  
            C[i]--; //7  
            j++; //8  
        } //I due cicli annidati → O(n+k)  
}
```

- 1- Scorro il vettore ausiliario fino a “ $<=k$ ”, poiché il vettore stesso arriva fino a $k+1$, e lo inizializzo inserendo valori 0
- 2- Faccio scorrere l’indice j fino ad n (dimensione del vettore di partenza) per far sì che si inserisca nel vettore ausiliario alla posizione di $A[j]$ un +1:
 - a. Il vettore C rimane fermo e non scorre, ma allo scorrere di n il valore $A[n]$ a quel momento, rappresenterà l’indice di C
// è come se si muovesse l’array A utilizzando il fatto che i valori di A sono rappresentati dall’indice dell’array di C
 - b. Ogni volta che trovo il valore di $A[j]$ → Il vettore nell’elemento $C[A[j]]$ aumenterà di +1, che rappresenta il numero di volte in cui si trova quel valore a $A[j]$ lungo l’array A
- 3- Riazzero l’indice j
- 4- Scorro gli elementi fino a “ $<=k$ ”, poiché $k+1$ è la lunghezza dell’array ausiliario C
- 5- Finché l’array C ha valori positivi nella posizione i in cui mi trovo (cioè finché posso rappresentare il numero di volte che ho un determinato valore, dato dalla posizione i)
- 6- Scorrendo con j (punto 8), nella posizione $A[j]$ metto il valore di i che rappresenta il numero
- 7- Diminuisco $C[i]$ fino a quando arrivo a 0, e facendo così faccio il punto 6 tante volte quante volte è presente $C[i]$, cioè finché è positivo → Ciò rappresenta il fatto di mettere in ordine i numeri ripetuti un certo numero di volte rappresentati nella memorizzazione dell’array ausiliario

COMPLESSITÀ TOTALE ALGORITMO → **O(n+k)**

COUNTING SORT

ESEMPIO ALGORITMO

- Considero l'array A seguente di 4 elementi

| | | | | |
|------|---|---|---|---|
| i | 0 | 1 | 2 | 3 |
| A[i] | 1 | 4 | 5 | 4 |

- Chiamo **counting_sort(A,5,4)**
 - 5=k che sono i possibili valori che possono essere assunti nell'array
 - 4=n che sono il numero di valori dell'array
-

1° Passaggio (k=5, n=4)

- Dichiaro i per scorrere l'array originario, j per scorrere quello ausiliario
- Dichiara C[6] → C[6] array ausiliario, faccio così perché i numeri vanno da 0 a 5 e quindi in tutto sono 6 le posizioni
- Ciclo che inizializza l'array ausiliario di valori nulli

| | | | | | | |
|------|---|---|---|---|---|---|
| i | 0 | 1 | 2 | 3 | 4 | 5 |
| C[i] | 0 | 0 | 0 | 0 | 0 | 0 |

- Ciclo: j=0
-

1.1° Passaggio (k=5, n=4, j=0)

- $j < n \rightarrow 0 < 4$ Si
- $C[A[j]]++ \rightarrow C[A[0]]++ \rightarrow$ Nell'array di C in posizione A[0] dove A[0] contiene l'1, quindi in C[1], metto ++ → Aumento di 1 poiché ho trovato il valore 1 una volta, e questo ne rappresenta la frequenza

| | | | | | | |
|---------|---|---|---|---|---|---|
| A[j] | 0 | 1 | 2 | 3 | 4 | 5 |
| C[A[j]] | 0 | 1 | 0 | 0 | 0 | 0 |

- $j++ (1)$

COUNTING SORT

... CONTINUO ESEMPIO ALGORITMO

1.2° Passaggio ($k=5$, $n=4$, $j=1$)

- $j < n \rightarrow 1 < 4$ Sì
- $C[A[j]]++ \rightarrow C[A[1]]++ \rightarrow$ Nell'array di C in posizione A[1] dove A[1] contiene il 4, quindi in C[4], metto ++ → Aumento di 1 poiché ho trovato il valore 4 una volta e questo ne rappresenta la frequenza

| A[j] | 0 | 1 | 2 | 3 | 4 | 5 |
|---------|---|---|---|---|---|---|
| C[A[j]] | 0 | 1 | 0 | 0 | 1 | 0 |

- $j++ (2)$
-

1.3° Passaggio ($k=5$, $n=4$, $j=2$)

- $j < n \rightarrow 2 < 4$ Sì
- $C[A[j]]++ \rightarrow C[A[2]]++ \rightarrow$ Nell'array di C in posizione A[2] dove A[2] contiene il 5, quindi in C[5], metto ++ → Aumento di 1 poiché ho trovato il valore 5 una volta e questo ne rappresenta la frequenza

| A[j] | 0 | 1 | 2 | 3 | 4 | 5 |
|---------|---|---|---|---|---|---|
| C[A[j]] | 0 | 1 | 0 | 0 | 1 | 1 |

- $j++ (3)$
-

1.4° Passaggio ($k=5$, $n=4$, $j=3$)

- $j < n \rightarrow 3 < 4$ Sì
- $C[A[j]]++ \rightarrow C[A[3]]++ \rightarrow$ Nell'array di C in posizione A[3] dove A[3] contiene il 4, quindi in C[4], metto ++ → Aumento di 1 poiché ho trovato il valore 4 un'altra volta e questo ne rappresenta la frequenza

| A[j] | 0 | 1 | 2 | 3 | 4 | 5 |
|---------|---|---|---|---|---|---|
| C[A[j]] | 0 | 1 | 0 | 0 | 2 | 1 |

- $j++ (4)$

COUNTING SORT

... CONTINUO ESEMPIO ALGORITMO

1.5° Passaggio ($k=5$, $n=4$, $j=4$)

- $j < n \rightarrow 4 < 4$ No esco dal ciclo tornando al punto 1° con il seguente array ausiliario costruito in questo modo

| A[j] | 0 | 1 | 2 | 3 | 4 | 5 |
|---------|---|---|---|---|---|---|
| C[A[j]] | 0 | 1 | 0 | 0 | 2 | 1 |

2° Passaggio ($k=5$, $n=4$)

- $j=0$
 - Ciclo for: $i=0$
-

2.1° Passaggio ($k=5$, $n=4$, $j=0$, $i=0$)

- $i \leq k \rightarrow 0 \leq 5$ Sì
 - Ciclo while
-

2.1.1° Passaggio ($k=5$, $n=4$, $j=0$, $i=0$)

- $C[i] > 0 \rightarrow C[0] > 0 \rightarrow 0 > 0$ No esco da questo ciclo while e aumento i di 1 ($i++$) tornando al passaggio precedente 2.1°
-

2.2° Passaggio ($k=5$, $n=4$, $j=0$, $i=1$)

- $i \leq k \rightarrow 1 \leq 5$ Sì
- Ciclo while

COUNTING SORT

CONTINUO ESEMPIO ALGORITMO

2.2.1° Passaggio ($k=5$, $n=4$, $j=0$, $i=1$)

- $C[i]>0 \rightarrow C[1]>0 \rightarrow 1>0$ Sì
- $A[j]=i \rightarrow A[0]=1$ //RICORDO A array iniziale

| | | | | |
|--------|---|---|---|---|
| i | 0 | 1 | 2 | 3 |
| $A[i]$ | 1 | 4 | 5 | 4 |

- $C[i]-- \rightarrow C[1]-- \rightarrow 1-1=0$

| | | | | | | |
|-----------|---|---|---|---|---|---|
| $A[j]$ | 0 | 1 | 2 | 3 | 4 | 5 |
| $C[A[j]]$ | 0 | 0 | 0 | 0 | 2 | 1 |

- $j++ (1)$
-

2.2.2° Passaggio ($k=5$, $n=4$, $j=1$, $i=1$)

- $C[i]>0 \rightarrow C[1]>0 \rightarrow 0>0$ No esco da questo ciclo while e aumento i di 1 ($i++$) tornando al passaggio 2.2°
-

2.3° Passaggio ($k=5$, $n=4$, $j=1$, $i=2$)

- $i \leq k \rightarrow 2 \leq 5$ Sì
 - Ciclo while
-

2.3.1° Passaggio ($k=5$, $n=4$, $j=1$, $i=2$)

- $C[i]>0 \rightarrow C[2]>0 \rightarrow 0>0$ No esco da questo ciclo while e aumento i di 1 ($i++$) tornando al passaggio precedente 2.3°
-

2.4° Passaggio ($k=5$, $n=4$, $j=1$, $i=3$)

- $i \leq k \rightarrow 3 \leq 5$ Sì
 - Ciclo while
-

2.4.1° Passaggio ($k=5$, $n=4$, $j=1$, $i=3$)

- $C[i]>0 \rightarrow C[3]>0 \rightarrow 0>0$ No esco da questo ciclo while e aumento i di 1 ($i++$) tornando al passaggio precedente 2.4°

COUNTING SORT

... CONTINUO ESEMPIO ALGORITMO

2.5° Passaggio ($k=5$, $n=4$, $j=1$, $i=4$)

- $i \leq k \rightarrow 4 \leq 5$ Sì
 - Ciclo while
-

2.5.1° Passaggio ($k=5$, $n=4$, $j=1$, $i=4$)

- $C[i] > 0 \rightarrow C[4] > 0 \rightarrow 2 > 0$ Sì
- $A[j]=i \rightarrow A[1]=4$ //RICORDO A array iniziale

| i | 0 | 1 | 2 | 3 |
|------|---|---|---|---|
| A[i] | 1 | 4 | 5 | 4 |
| | | | | |

- $C[i]-- \rightarrow C[4]-- \rightarrow 2-1=1$

| A[j] | 0 | 1 | 2 | 3 | 4 | 5 |
|---------|---|---|---|---|---|---|
| C[A[j]] | 0 | 0 | 0 | 0 | 1 | 1 |
| | | | | | | |

- $j++ (2)$
-

2.5.2° Passaggio ($k=5$, $n=4$, $j=2$, $i=4$)

- $C[i] > 0 \rightarrow C[4] > 0 \rightarrow 1 > 0$ Sì
- $A[j]=i \rightarrow A[2]=4$ //RICORDO A array iniziale

| i | 0 | 1 | 2 | 3 |
|------|---|---|---|---|
| A[i] | 1 | 4 | 4 | 4 |
| | | | | |

- $C[i]-- \rightarrow C[4]-- \rightarrow 1-1=0$

| A[j] | 0 | 1 | 2 | 3 | 4 | 5 |
|---------|---|---|---|---|---|---|
| C[A[j]] | 0 | 0 | 0 | 0 | 0 | 1 |
| | | | | | | |

- $j++ (3)$
-

2.5.3° Passaggio ($k=5$, $n=4$, $j=3$, $i=4$)

- $C[i] > 0 \rightarrow C[4] > 0 \rightarrow 0 > 0$ No esco da questo ciclo while e aumento i di 1 ($i++$) tornando al passaggio precedente 2.5°

COUNTING SORT

... CONTINUO ESEMPIO ALGORITMO

2.6° Passaggio ($k=5$, $n=4$, $j=3$, $i=5$)

- $i \leq k \rightarrow 5 \leq 5$ Sì
 - Ciclo while
-

2.6.1° Passaggio ($k=5$, $n=4$, $j=3$, $i=5$)

- $C[i] > 0 \rightarrow C[5] > 0 \rightarrow 1 > 0$ Sì
- $A[j] = i \rightarrow A[3] = 5$ //RICORDO A array iniziale

| i | 0 | 1 | 2 | 3 |
|------|---|---|---|---|
| A[i] | 1 | 4 | 4 | 5 |
| | | | | |

- $C[i]-- \rightarrow C[5]-- \rightarrow 1-1=0$

| A[j] | 0 | 1 | 2 | 3 | 4 | 5 |
|---------|---|---|---|---|---|---|
| C[A[j]] | 0 | 0 | 0 | 0 | 0 | 0 |
| | | | | | | |

- $j++ (4)$
-

2.6.2° Passaggio ($k=5$, $n=4$, $j=4$, $i=5$)

- $C[i] > 0 \rightarrow C[5] > 0 \rightarrow 0 > 0$ No esco da questo ciclo while e aumento i di 1 ($i++$) tornando al passaggio precedente 2.6°
-

2.7° Passaggio ($k=5$, $n=4$, $j=4$, $i=6$)

- $i \leq k \rightarrow 6 \leq 5$ No, esco dal ciclo, chiudendo la funzione con l'array A ordinato e l'array C ausiliario azzerato

| i | 0 | 1 | 2 | 3 |
|------|---|---|---|---|
| A[i] | 1 | 4 | 4 | 5 |
| | | | | |

| A[j] | 0 | 1 | 2 | 3 | 4 | 5 |
|---------|---|---|---|---|---|---|
| C[A[j]] | 0 | 0 | 0 | 0 | 0 | 0 |
| | | | | | | |

RADIX SORT

- Si utilizza quando conosco il numero massimo di cifre su cui posso rappresentare i numeri

FUNZIONAMENTO:

- Considero un array di **n** numeri
- Controllo quante cifre ha ciascun numero e, considero l'insieme delle cifre per ciascun numero come → Numero massimo di cifre e lo chiamo **d**
ES) 0134 → $d=4$
- Considerando che sono numeri interi vanno da 0 a 9, quindi la base per questi numeri è 10 e la chiamerò **k**
- Creo una lista di vettori di dimensione **k** e ciascun vettore ha dimensione **n**
- Faccio un totale di **d** passate e per ciascuna passata:
 - o Considero la d -esima cifra di ciascun numero dell'array, e la inserisco nel vettore dentro il contenitore di posizione k -esima, dove la " d -esima cifra considerata = k -esimo indice del contenitore"
 - o Estraggo i numeri creando un nuovo array sempre di dimensione n
 - o Decremento **d** e riparto dal punto precedente

OSS: Possono rimanere allocazioni vuote nei vari contenitori

ESEMPIO)

- ARRAY[n]
- $n = 6$

| | | | | | |
|-----|-----|-----|-----|-----|-----|
| 190 | 051 | 054 | 207 | 088 | 010 |
|-----|-----|-----|-----|-----|-----|

→ 1° PASSATA

- $d = 3$ (3° cifra del numero da considerare)
- $k = 10$

| | | | | | | | | | |
|-----|-----|---|---|-----|---|---|-----|-----|---|
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| 010 | | | | | | | | | |
| 190 | 051 | | | 054 | | | 207 | 088 | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

→ ESTRAENDO I NUMERI RILEGGENDOLI DA SINISTRA A DESTRA E DAL BASSO VERSO L'ALTO, OTTENGO IL NUOVO ARRAY

| | | | | | |
|-----|-----|-----|-----|-----|-----|
| 190 | 010 | 051 | 054 | 207 | 088 |
|-----|-----|-----|-----|-----|-----|

//a-- =2

RADIX SORT

→ 2° PASSATA

- $d = 2$ (2° cifra del numero da considerare)
- $k = 10$

| | | | | | | | | | | |
|-----|-----|---|---|-----|-----|---|---|-----|-----|--|
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | 054 | | | | | |
| 207 | 010 | | | 051 | | | | 088 | 190 | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | |

→ ESTRAENDO I NUMERI RILEGGENDOLI DA SINISTRA A DESTRA E DAL BASSO VERSO L'ALTO, OTTENGO IL NUOVO ARRAY

| | | | | | |
|-----|-----|-----|-----|-----|-----|
| 207 | 010 | 051 | 054 | 088 | 190 |
|-----|-----|-----|-----|-----|-----|

//a--=1

→ 3° PASSATA

- $d = 1$ (1° cifra del numero da considerare)
- $k = 10$

| | | | | | | | | | | |
|-----|-----|-----|---|---|---|---|---|---|---|--|
| | | | | | | | | | | |
| | | | | | | | | | | |
| | | | | | | | | | | |
| 088 | | | | | | | | | | |
| 054 | | | | | | | | | | |
| 051 | | | | | | | | | | |
| 010 | 190 | 207 | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | |

→ ESTRAENDO I NUMERI RILEGGENDOLI DA SINISTRA A DESTRA E DAL BASSO VERSO L'ALTO, OTTENGO IL NUOVO ARRAY ORDINATO COSÌ

| | | | | | |
|-----|-----|-----|-----|-----|-----|
| 010 | 051 | 054 | 088 | 190 | 207 |
|-----|-----|-----|-----|-----|-----|

RADIX SORT

PSEUDOCODICE

```
procedura bucketSort (array A di n interi, interi k e d)          //0
sia Y un array di dimensione k                                     //1
for i=1 to k do Y[i]← lista vuota                                //2
for i=1 to n do
    c ← d-esima cifra di A[i] nella rappresentazione in base k   //3
    appendi A[i] alla lista Y[c+1]                                    //4
for i=1 to k do
    copia ordinatamente in A gli elementi della lista Y[i]        //5
```

- 0- La procedura implementa il contenitore, k è il numero possibile di cifre cioè 10 (essendo interi), d è la cifra di riferimento per i contenitori
1- Creo un array di array di lunghezza 10 (base)
2- Inizializzo
3- Parto da 0 e ci copio il numero con la cifra meno significativa
4- La base va da 0 a 9 ma l'array mi fa andare da 1 a 10
5- Leggo dal basso verso l'alto, da SX a DX

```
algoritmo radixSort(array A di n interi)
d ← 0                                         //cifra di indice 0 (meno significativa)
while (esiste un numero la cui d-esima cifra è != da 0)
    bucketSort (A,10,d)
    d ← d+1
```

A PAROLE:

“Il radixSort richiama il bucketSort t volte, quante sono le cifre su cui viene rappresentato il numero intero. Il bucketSort prende un intero e quale cifra da considerare, partendo dalla meno significativa alla più significativa. Si crea una lista vuota di dimensione k (pari al numero di cifre). Si va a prendere la d-esima cifra dell’elemento contenuta all’interno del vettore e si appende a questo vettore di liste. Si copiano ordinatamente i vari elementi”

Complessità

O(d(n+k))

- d=lunghezza delle sequenze
- k= numero possibili valori per ogni cifra

HASH

Ricerca ad accesso diretto

```
bool hashSearch(InfoType *A, int n, InfoType x){  
    int i=h(x);  
    if(A[i] == x)    return true;  
    return false;  
}
```

Ricerca con scansione lineare

```
bool hashSearch(int *A, int k, int x){           //0  
    int i=h(x);                                //1  
    for(int j=0; j<k; j++){                     //2  
        int pos=(i+j)%k;                      //3  
        if(A[pos] == -1)      return false;       //4  
        if(A[pos] == x)      return true;          //5  
    }  
    return false;                                //6  
}
```

FUNZIONAMENTO:

- 0- x è l'elemento da cercare, k sono le posizioni
- 1- Calcolo la funzione di hashing
- 2- Scorro il vettore
- 3- Somma in modulo: i è l'indice identificato dall'hashing +j sono le posizioni adiacenti
- 4- Se non trovo nulla (posizione vuota) restituisco falso
- 5- Altrimenti restituisco vero
- 6- Significa che nella posizione in cui guardo c'è un altro elemento, quindi ho una collisione oppure sono entrato in un caso in cui ho un agglomerato

OSS= -1 POSIZIONE VUOTA

HASH

Inserimento

```
int hashInsert(int *A, int k, int x){ //0
    int i=h(x);
    int b=0; //1
    for(int j=0; !b && j<k; j++){ //2
        int pos=(i+j)%k;
        if(A[pos] == -1){ //3
            A[pos]=x; //4
            b=1; //5
        }
    }
    return b; //6
}
```

FUNZIONAMENTO:

- 0- x è l'oggetto da inserire, k sono le posizioni
- 1- Inizializzo la variabile di controllo a 0 che rappresenta l'inserimento:
0-Non inserito, 1-Inserito
- 2- Scorro il vettore finchè non vado alla fine del vettore ($j < k$) oppure, finchè non viene effettuato l'inserimento ($\text{!}b$) → CONTEMPORANEAMENTE
- 3- Indica che c'è un posto libero e quindi posso procedere con l'inserimento
- 4- Inserisco il valore nella posizione
- 5- Inizializzo la variabile b a 1 dicendo che viene fatto l'inserimento
- 6- Restituisco la variabile ad inserimento fatto

OSS: Per l'inserimento si va a scorrere il vettore guardando le posizioni adiacenti a quella non libera

HASH

ESEMPIO DI INSERIMENTO

Dato un array di 101 posizioni che memorizza un insieme al massimo di 80 elementi, indicare il contenuto delle prime 4 celle dell'array, inizialmente vuoto, dopo le operazioni seguenti (indirizzamento con il metodo del resto scansione lineare unitaria):

- a) Inserimento dell'elemento 101
- b) Inserimento dell'elemento 202
- c) Inserimento dell'elemento 204
- d) Inserimento dell'elemento 304

Ho che $n=80$ e $k=101 \rightarrow$ FATTORE DI CARICO $n/k = 80\%$

Calcolo gli indirizzi HASH degli elementi facendo la divisione intera e prendendone il resto

$$101 \rightarrow 101 \% 101 = 0$$

$$202 \rightarrow 202 \% 101 = 0$$

$$204 \rightarrow 204 \% 101 = 2 \quad // 101 ci sta 2 volte in 204 e avanza 2 per arrivare a 204$$

$$304 \rightarrow 304 \% 101 = 1 \quad // 101 ci sta 3 volte in 303 e avanza 1 per arrivare a 304$$

- Faccio la tabella secondo gli indirizzi HASH appena trovati e immetto gli elementi per tutti i passaggi (-1 posizione vuota)

| | INIZIO | Dopo a) | Dopo b) | Dopo c) | Dopo d) |
|---|--------|---------|---------|---------|---------|
| 0 | -1 | 101 | 101 | 101 | 101 |
| 1 | -1 | -1 | 202 | 202 | 202 |
| 2 | -1 | -1 | -1 | 204 | 204 |
| 3 | -1 | -1 | -1 | -1 | 304 |



202
Scorre
sotto alla
prima
posizione
libera
perché la
sua
posizione
era
occupata
da 101



304
Scorre
sotto alla
prima
posizione
libera
perché la
sua
posizione
era
occupata
da 202 e
204

HASH

Inserimento in presenza di cancellazioni

```
int hashInsert(int *A, int k, int x){  
    int i=h(x);  
    int b=0;  
    for(int j=0; !b && j<k; j++){  
        int pos=(i+j)%k;  
        if((A[pos] == -1) || (A[pos] == -2)){ //1  
            A[pos]=x;  
            b=1;  
        }  
    }  
    return b;  
}
```

FUNZIONAMENTO:

- 1- Stesso funzionamento dell'inserimento precedente, solamente cambia il fatto che posso inserire se ho la posizione libera rappresentata da -1 oppure da -2

OSS:

-1 Posizione vuota

-2 Posizione disponibile in seguito ad una cancellazione

La ricerca è la stessa, cioè quella con la scansione lineare

HASH

ESEMPIO DI INSERIMENTO IN PRESENZA DI CANCELLAZIONI

Dato un array di 101 posizioni che memorizza un insieme al massimo di 80 elementi, indicare il contenuto delle prime 4 celle dell'array, inizialmente vuoto, dopo le operazioni seguenti (indirizzamento con il metodo del resto scansione lineare unitaria):

- a) Inserimento dell'elemento 101
- b) Inserimento dell'elemento 202
- c) Inserimento dell'elemento 204
- d) Cancellazione dell'elemento 202
- e) Inserimento dell'elemento 304

Ho che $n=80$ e $k=101 \rightarrow$ FATTORE DI CARICO $n/k = 80\%$

Calcolo gli indirizzi HASH degli elementi facendo la divisione intera e prendendone il resto

$$101 \rightarrow 101 \% 101 = 0$$

$$202 \rightarrow 202 \% 101 = 0$$

$$204 \rightarrow 204 \% 101 = 2 \quad // 101 ci sta 2 volte in 204 e avanza 2 per arrivare a 204$$

$$304 \rightarrow 304 \% 101 = 1 \quad // 101 ci sta 3 volte in 303 e avanza 1 per arrivare a 304$$

- Faccio la tabella secondo gli indirizzi HASH appena trovati e immetto gli elementi per tutti i passaggi (-1 posizione vuota)

| | INIZIO | Dopo a) | Dopo b) | Dopo c) | Dopo d) | Dopo e) |
|---|--------|---------|---------|---------|---------|---------|
| 0 | -1 | 101 | 101 | 101 | 101 | 101 |
| 1 | -1 | -1 | 202 | 202 | -2 | 304 |
| 2 | -1 | -1 | -1 | 204 | 204 | 204 |
| 3 | -1 | -1 | -1 | -1 | -1 | -1 |

202
Scorre
sotto alla
prima
posizione
libera
perché la
sua
posizione
era
occupata
da 101

304 si
posiziona
al posto
giusto
poiché
era libera
dalla
dicitura
-2

PLSC

```
int lenght(char *a, char *b, int i, int j){                //0
    if(i==0 || j==0)      return 0;                          //1
    if(a[i] == b[j])          //2
        return 1+ lenght(a, b, i-1, j-1);
    else
        return max(lenght(a, b, i, j-1), lenght(a, b, i-1,j)); //3
}
```

- 0- Le stringhe passate sono ‘a’ e ‘b’ con lunghezza rispettiva ‘i’ e ‘j’
→ Gli indici partono da 1 poiché quando arrivo a 0 ho finito una stringa
- 1- Nel caso in cui io mi trovi di fronte ad entrambe le stringhe lunghe zero oppure l’una o l’altra lunghe 0, ritorno 0 come lunghezza visto che, la sottosequenza da trovare comune ad entrambe è solamente 0
- 2- Nel caso mi trovassi di fronte all’ultimo carattere delle due sequenze, che è uguale ad entrambe le stringhe → Richiamo la funzione sulle chiamate successive sommandoci 1 poichè sò per certo che il carattere finale ad entrambe le stringhe è comune, essendo uguale dalla condizione
- 3- Nel caso mi trovassi a caratteri finali di entrambe le stringhe diverso, chiamo la funzione che trova il massimo tra:
 - La chiamata ricorsiva sulla prima stringa con tutti i caratteri e la seconda stringa senza l’ultimo carattere
 - La chiamata ricorsiva sulla seconda stringa con tutti i caratteri e la prima stringa senza l’ultimo carattere

- La funzione ha tempo

O(2ⁿ)

OSS:

- La funzione restituisce la lunghezza della PLSC non ci dice quale sia o quali siano !

```
int max(int x, int y){
    if(x>y) return x;
    else return y;           //qui considero anche l’uguaglianza
}
```

PLSC

FUNZIONAMENTO LENGTH

- Considero i due array di caratteri (stringhe) a e b di dimensioni rispettivamente i=2 e j=3

| i | 1 | 2 | j | 1 | 2 | 3 |
|------|---|---|------|---|---|---|
| a[i] | x | y | b[j] | x | z | y |

1° Passaggio (i=2, j=3)

//gli indici partono da 1 perchè a 0 finisco la stringa

- $i=0 \parallel j=0$ No
 - $a[i]=b[j] \rightarrow a[2]=b[3] \rightarrow y=y$ Sì
 - Ritorno 1+ Ricorsione con entrambi gli array di dimensione diminuita di 1
-

1.1° Passaggio (i=1, j=2)

- $i=0 \parallel j=0$ No
 - $a[i]=b[j] \rightarrow a[1]=b[2] \rightarrow x=z$ No
 - Ritorno il massimo fra
 - 1° Ricorsione con l'array a e l'array b di dimensione diminuita di 1
 - 2° Ricorsione con l'array a di dimensione diminuita di 1, e l'array b
-

1.1.1° Passaggio (1° Ricorsione i=1, j=1)

- $i=0 \parallel j=0$ No
 - $a[i]=b[j] \rightarrow a[1]=b[1] \rightarrow x=x$ Sì
 - Ritorno 1+ Ricorsione con entrambi gli array di dimensione diminuita di 1
-

1.1.1.1° Passaggio (1° Ricorsione i=0, j=0)

- $i=0 \parallel j=0$ Sì Chiudo questa ricorsione e ritorno 0 alla ricorsione al passaggio 1.1.1°, la quale si chiude anche essa tornando $1+0=1$ al passaggio 1.1°
-

1.2° Passaggio (1° Ricorsione i=0, j=2)

- $i=0 \parallel j=0$ Sì, chiudo questa ricorsione e ritorno 0 alla ricorsione al passaggio 1.1°, la quale si chiude anche essa tornando $\max(1,0) = 1$ al passaggio 1°
-

2° Passaggio

- Chiudo la funzione tornando la PLSC = 2 data da $\rightarrow 1+1$ (ottenuto completando tutta la ricorsione del passaggio 1.1°)
-

PLSC

PLSC-PROGRAMMAZIONE DINAMICA

```
const int m, n; //0
int L[m+1][n+1]; //1
int quicklenght(char *a, char *b){
    for(int j=0; j<=n; j++) L[0][j]=0; //2
    for(int i=1; i<=m; i++){ //3
        L[i][0]=0;
        for(int j=1; j<=n; j++) //4
            if(a[i] != b[j]) //5
                L[i][j] = max(L[i][j-1], L[i-1][j]);
            else L[i][j] = (L[i-1][j-1]) + 1 ; //6
    }
    return L[m][n]; //7
}
```

- 0- Definisco le variabili che rappresentano la lunghezza delle due stringhe
- 1- Definisco una matrice di lunghezza + 1 su riga e colonna poiché aggiungerò una riga e colonna piena di 0
- 2- Tengo ferma la prima riga e facendo scorrere la colonna la riempio tutta di 0
- 3- Per ogni riga a partire dalla seconda ($i=1$), inizio riempiendo il valore della prima colonna con il valore 0 (che poi sarà tutta la colonna riempita di 0)
- 4- Faccio un ciclo riposizionando il valore di j a 1, così parto dalla seconda colonna
- 5- Controllo se il valore dei caratteri delle due stringhe, ai valori degli indici a quel momento sono diversi tra loro
Se lo sono, il valore della matrice con i e j a quel momento (sotto la diagonale che viene fatta precedentemente), avrà il massimo tra il valore della matrice di riga precedente-colonna attuale e il valore della matrice di riga attuale-colonna precedente
- 6- Il valore che avrà i e j al momento sarà quello della riga e colonna precedenti + 1
- 7- Alla fine, ritorno il valore della matrice di riga e colonna finale della matrice dove ci sarà la lunghezza della PLSC

OSS: In tutti i cicli il \leq è perché ho aggiunto +1 sia ad m sia ad n poiché dovevo aggiungere la riga e la colonna di zeri

COMPLESSITÀ → **$O(n^2)$**

// Meglio dell'algoritmo precedente, che aveva complessità esponenziale

PLSC

ESEMPIO QUICKLENGHT

- Considero i due array di caratteri (stringhe) a e b di dimensioni rispettivamente i=3 e j=3

| | | | |
|------|---|---|---|
| i | 1 | 2 | 3 |
| a[i] | a | b | b |

| | | | |
|------|---|---|---|
| j | 1 | 2 | 3 |
| b[j] | c | b | b |

1° Passaggio

- Dichiaro m=3 e n=3
- Dichiaro una matrice L[4][4] //m+1 e n+1
- Ciclo e riempio tutta la riga 0 di valori 0 con j che va da 0 a 6

| | | j=0 | j=1 | j=2 | j=3 |
|-----|------|------|------|------|------|
| | | b[0] | b[1] | b[2] | b[3] |
| | | c | | | b |
| i=0 | a[0] | | 0 | 0 | 0 |
| i=1 | a[1] | a | | | |
| i=2 | a[2] | b | | | |
| i=3 | a[3] | b | | | |

- Ciclo i=1
-

1.1° Passaggio (i=1, m=3, n=3)

- $i \leq m \rightarrow 1 \leq 3$ Sì
- $L[i][0]=0 \rightarrow L[1][0]=0$

| | | j=0 | j=1 | j=2 | j=3 |
|-----|------|------|------|------|------|
| | | b[0] | b[1] | b[2] | b[3] |
| | | c | | | b |
| i=0 | a[0] | | 0 | 0 | 0 |
| i=1 | a[1] | a | 0 | | |
| i=2 | a[2] | b | | | |
| i=3 | a[3] | b | | | |

- Ciclo interno j=1

PLSC

... CONTINUO ESEMPIO QUICKLENGTH

1.1.1° Passaggio ($i=1, j=1, m=3, n=3$)

- $j \leq n \rightarrow 1 \leq 3$ Sì
- $a[i] \neq b[j] \rightarrow a[1] \neq b[1] \rightarrow 'a' \neq 'c'$? Sì
- $\ln L[i][j] = \max(L[i][j-1], L[i-1][j]) \rightarrow L[1][1] = \max(L[1][0], L[0][1])$
→ Massimo tra 0 e 0 è 0

| | | j=0 | j=1 | j=2 | j=3 |
|-----|------|------|------|------|-----|
| | b[0] | b[1] | b[2] | b[3] | |
| | c | b | b | | |
| i=0 | a[0] | 0 | 0 | 0 | 0 |
| i=1 | a[1] | a | 0 | 0 | |
| i=2 | a[2] | b | | | |
| i=3 | a[3] | b | | | |

- $j=2$ ($j++$)
-

1.1.2° Passaggio ($i=1, j=2, m=3, n=3$)

- $j \leq n \rightarrow 2 \leq 3$ Sì
- $a[i] \neq b[j] \rightarrow a[1] \neq b[2] \rightarrow 'a' \neq 'b'$? Sì
- $\ln L[i][j] = \max(L[i][j-1], L[i-1][j]) \rightarrow L[1][2] = \max(L[1][1], L[0][2])$
→ Massimo tra 0 e 0 è 0

| | | j=0 | j=1 | j=2 | j=3 |
|-----|------|------|------|------|-----|
| | b[0] | b[1] | b[2] | b[3] | |
| | c | b | b | | |
| i=0 | a[0] | 0 | 0 | 0 | 0 |
| i=1 | a[1] | a | 0 | 0 | 0 |
| i=2 | a[2] | b | | | |
| i=3 | a[3] | b | | | |

- $j=3$ ($j++$)

PLSC

... CONTINUO ESEMPIO QUICKLENGTH

1.1.3° Passaggio ($i=1, j=3, m=3, n=3$)

- $j \leq n \rightarrow 3 \leq 3$ Sì
- $a[i] \neq b[j] \rightarrow a[1] \neq b[3] \rightarrow 'a' \neq 'b' ?$ Sì
- $\ln L[i][j] = \max(L[i][j-1], L[i-1][j]) \rightarrow L[1][3] = \max(L[1][2], L[0][3])$
→ Massimo tra 0 e 0 è 0

| | | j=0 | j=1 | j=2 | j=3 |
|-----|------|------|------|------|-----|
| | b[0] | b[1] | b[2] | b[3] | |
| | c | b | b | | |
| i=0 | a[0] | 0 | 0 | 0 | 0 |
| i=1 | a[1] | a | 0 | 0 | 0 |
| i=2 | a[2] | b | | | |
| i=3 | a[3] | b | | | |

- $j=4$ ($j++$)
-

1.1.4° Passaggio ($i=1, j=4, m=3, n=3$)

- $j \leq n \rightarrow 4 \leq 3$ No esco dal ciclo interno e torno al punto 1.1° aumentando i di 1 ($i++$)
-

1.2° Passaggio ($i=2, m=3, n=3$)

- $i \leq m \rightarrow 2 \leq 3$ Sì
- $L[i][0]=0 \rightarrow L[2][0]=0$

| | | j=0 | j=1 | j=2 | j=3 |
|-----|------|------|------|------|-----|
| | b[0] | b[1] | b[2] | b[3] | |
| | c | b | b | | |
| i=0 | a[0] | 0 | 0 | 0 | 0 |
| i=1 | a[1] | a | 0 | 0 | 0 |
| i=2 | a[2] | b | 0 | | |
| i=3 | a[3] | b | | | |

- Ciclo interno $j=1$

PLSC

... CONTINUO ESEMPIO QUICKLENGTH

1.2.1° Passaggio ($i=2, j=1, m=3, n=3$)

- $j \leq n \rightarrow 1 \leq 3$ Sì
- $a[i] \neq b[j] \rightarrow a[2] \neq b[1] \rightarrow 'b' \neq 'c' ?$ Sì
- $\ln L[i][j] = \max(L[i][j-1], L[i-1][j]) \rightarrow L[2][1] = \max(L[2][0], L[1][1])$
→ Massimo tra 0 e 0 è 0

| | | j=0 | j=1 | j=2 | j=3 |
|-----|------|------|------|------|-----|
| | b[0] | b[1] | b[2] | b[3] | |
| | c | b | b | | |
| i=0 | a[0] | 0 | 0 | 0 | 0 |
| i=1 | a[1] | a | 0 | 0 | 0 |
| i=2 | a[2] | b | 0 | 0 | |
| i=3 | a[3] | b | | | |

- $j=2$ ($j++$)
-

1.2.2° Passaggio ($i=2, j=2, m=3, n=3$)

- $j \leq n \rightarrow 2 \leq 3$ Sì
- $a[i] \neq b[j] \rightarrow a[2] \neq b[2] \rightarrow 'b' \neq 'b' ?$ No → Ramo else
- $\ln L[i][j] = (L[i-1][j-1]) + 1 \rightarrow L[2][2] = (L[1][1]) + 1 \rightarrow 0+1=1$

| | | j=0 | j=1 | j=2 | j=3 |
|-----|------|------|------|------|-----|
| | b[0] | b[1] | b[2] | b[3] | |
| | c | b | b | | |
| i=0 | a[0] | 0 | 0 | 0 | 0 |
| i=1 | a[1] | a | 0 | 0 | 0 |
| i=2 | a[2] | b | 0 | 0 | 1 |
| i=3 | a[3] | b | | | |

- $j=3$ ($j++$)

PLSC

... CONTINUO ESEMPIO QUICKLENGTH

1.2.3° Passaggio ($i=2, j=3, m=3, n=3$)

- $j \leq n \rightarrow 3 \leq 3$ Si
- $a[i] \neq b[j] \rightarrow a[2] \neq b[3] \rightarrow 'b' \neq 'b'$? No \rightarrow Ramo else
- $L[i][j] = (L[i-1][j-1]) + 1 \rightarrow L[2][3] = (L[1][2]) + 1 \rightarrow 0+1=1$

| | | j=0 | j=1 | j=2 | j=3 |
|-----|------|------|------|------|------|
| | | b[0] | b[1] | b[2] | b[3] |
| | | c | b | b | |
| i=0 | a[0] | | 0 | 0 | 0 |
| i=1 | a[1] | a | 0 | 0 | 0 |
| i=2 | a[2] | b | 0 | 0 | 1 |
| i=3 | a[3] | b | | | 1 |

- $j=4$ ($j++$)
-

1.2.4° Passaggio ($i=2, j=4, m=3, n=3$)

- $j \leq n \rightarrow 4 \leq 3$ No esco dal ciclo interno e torno al punto 1.2° aumentando i di 1 ($i++$)
-

1.3° Passaggio ($i=3, m=3, n=3$)

- $i \leq m \rightarrow 3 \leq 3$ Si
- $L[i][0]=0 \rightarrow L[3][0]=0$

| | | j=0 | j=1 | j=2 | j=3 |
|-----|------|------|------|------|------|
| | | b[0] | b[1] | b[2] | b[3] |
| | | c | b | b | |
| i=0 | a[0] | | 0 | 0 | 0 |
| i=1 | a[1] | a | 0 | 0 | 0 |
| i=2 | a[2] | b | 0 | 0 | 1 |
| i=3 | a[3] | b | 0 | | 1 |

- Ciclo interno $j=1$

PLSC

... CONTINUO ESEMPIO QUICKLENGTH

1.3.1° Passaggio ($i=3, j=1, m=3, n=3$)

- $j \leq n \rightarrow 1 \leq 3$ Sì
- $a[i] \neq b[j] \rightarrow a[3] \neq b[1] \rightarrow 'b' \neq 'c' ?$ Sì
- $\ln L[i][j] = \max(L[i][j-1], L[i-1][j]) \rightarrow L[3][1] = \max(L[3][0], L[2][1])$
→ Massimo tra 0 e 0 è 0

| | | j=0 | j=1 | j=2 | j=3 |
|-----|------|------|------|------|-----|
| | b[0] | b[1] | b[2] | b[3] | |
| | c | b | b | | |
| i=0 | a[0] | 0 | 0 | 0 | 0 |
| i=1 | a[1] | a | 0 | 0 | 0 |
| i=2 | a[2] | b | 0 | 0 | 1 |
| i=3 | a[3] | b | 0 | 0 | |

- $j=2$ ($j++$)
-

1.3.2° Passaggio ($i=3, j=2, m=3, n=3$)

- $j \leq n \rightarrow 2 \leq 3$ Sì
- $a[i] \neq b[j] \rightarrow a[3] \neq b[2] \rightarrow 'b' \neq 'b' ?$ No → Ramo else
- $\ln L[i][j] = (L[i-1][j-1]) + 1 \rightarrow L[3][2] = (L[2][1]) + 1 \rightarrow 0+1=1$

| | | j=0 | j=1 | j=2 | j=3 |
|-----|------|------|------|------|-----|
| | b[0] | b[1] | b[2] | b[3] | |
| | c | b | b | | |
| i=0 | a[0] | 0 | 0 | 0 | 0 |
| i=1 | a[1] | a | 0 | 0 | 0 |
| i=2 | a[2] | b | 0 | 0 | 1 |
| i=3 | a[3] | b | 0 | 0 | 1 |

- $j=3$ ($j++$)

PLSC

... CONTINUO ESEMPIO QUICKLENGTH

1.3.3° Passaggio ($i=3, j=3, m=3, n=3$)

- $j \leq n \rightarrow 3 \leq 3$ Sì
- $a[i] \neq b[j] \rightarrow a[3] \neq b[3] \rightarrow 'b' \neq 'b' ?$ No \rightarrow Ramo else
- $L[i][j] = (L[i-1][j-1]) + 1 \rightarrow L[3][3] = (L[2][2]) + 1 \rightarrow 1+1=2$

| | | j=0 | j=1 | j=2 | j=3 |
|-----|------|------|------|------|------|
| | | b[0] | b[1] | b[2] | b[3] |
| | | c | b | b | |
| i=0 | a[0] | | 0 | 0 | 0 |
| i=1 | a[1] | a | 0 | 0 | 0 |
| i=2 | a[2] | b | 0 | 0 | 1 |
| i=3 | a[3] | b | 0 | 0 | 1 |

- $j=4$ ($j++$)
-

1.3.4° Passaggio ($i=3, j=4, m=3, n=3$)

- $j \leq n \rightarrow 4 \leq 3$ No esco dal ciclo interno e torno al punto 1.3° aumentando i di 1 ($i++$)
-

1.4° Passaggio ($i=4, m=3, n=3$)

- $i \leq m \rightarrow 4 \leq 3$ No esco dal ciclo interno e torno al punto 1°
-

2° Passaggio

- Ritorno $L[3][3]$ che è 2 il valore della PLSC e chiudo la funzione

| | | j=0 | j=1 | j=2 | j=3 |
|-----|------|------|------|------|------|
| | | b[0] | b[1] | b[2] | b[3] |
| | | c | b | b | |
| i=0 | a[0] | | 0 | 0 | 0 |
| i=1 | a[1] | a | 0 | 0 | 0 |
| i=2 | a[2] | b | 0 | 0 | 1 |
| i=3 | a[3] | b | 0 | 0 | 1 |

PLSC
 $L[m][n] \rightarrow L[3][3]$

PLSC

LETTURA PLSC-PROGRAMMAZIONE DINAMICA

```
void print(char *a, char *b, int i=m, int j=n){ //0
    if((i==0) || (j==0)) //1
        return;
    if(a[i] == b[j]){ //2
        print(a, b, i-1, j-1); //3
        cout<<a[i];
    }
    else if(L[i][j] == L[i-1][j]) //5
        print(a, b, i-1, j); //6
    else
        print(a, b, i, j-1); //7
}
```

- 0- 'i=m' e 'j=n' stanno a significare che parto dalla fine della mia matrice, cioè dall'ultimo elemento → Vanno passati al momento della chiamata sui parametri attuali
- 1- Caso in cui mi fermo, e cioè quando sono sulla riga con tutti 0 oppure sulla colonna con tutti 0
- 2- Controllo se caratteri (partendo dalla fine) delle due stringhe PLSC sono uguali
- 3- Se così fossero, torno indietro sulla riga e colonna precedente (vado a vedere in diagonale), per cercare i precedenti della PLSC. Faccio così poiché ogni volta che trovavo un valore uguale nella **quickeight** ho impostato che sarei sceso di riga spostandomi in diagonale e, invece qua salgo diagonalmente.
- 4- Arrivato qua significa che stampo il carattere e poi chiudo le chiamate ricorsivamente stampando il resto (dall'inizio alla fine)
- 5- Se i caratteri (partendo dalla fine) invece, non siano uguali → Vado a controllare se il carattere in cui mi trovo sia uguale a quello della riga precedente e colonna attuale
- 6- Allora richiamo ricorsivamente la funzione sulla riga precedente e colonna attuale, cioè sopra (vado a vedere in verticale)
- 7- Vuol dire che non vale il punto 5) e quindi richiamo ricorsivamente la funzione sulla riga attuale ma sulla colonna precedente, cioè a fianco (vado a vedere in orizzontale)

OSS:

- Restituisco solo una delle PLSC

COMPLESSITÀ

O(n+m)

// O decremento la riga, o decremento la colonna oppure entrambe, al massimo mi sposto n+m volte nei casi peggiori

- In poche parole, estraggo la più lunga PLSC andando a ritroso

PLSC

ESEMPIO PRINT

- Gli passo tra i parametri le e due strighe a='cbb' e b='abb' (e sto lavorando sulla matrice)
- Gli passo le dimensioni m=3 e n=3 che verranno assunte da i e da j tra i parametri

| | | j=0 | j=1 | j=2 | j=3 |
|-----|------|------|------|------|------|
| | | b[0] | b[1] | b[2] | b[3] |
| | | c | b | b | |
| i=0 | a[0] | | 0 | 0 | 0 |
| i=1 | a[1] | a | 0 | 0 | 0 |
| i=2 | a[2] | b | 0 | 0 | 1 |
| i=3 | a[3] | b | 0 | 0 | 1 |

1° Passaggio (i=3, j=3)

- i=0 || j=0 No
- a[i]=b[j] → a[3]=b[3] → 'b'='b' Sì
- Ricorsione su riga e colonna precedente (diagonale) diminuendo i e j → i=2, j=2

| | | j=0 | j=1 | j=2 | j=3 |
|-----|------|------|------|------|------|
| | | b[0] | b[1] | b[2] | b[3] |
| | | c | b | b | |
| i=0 | a[0] | | 0 | 0 | 0 |
| i=1 | a[1] | a | 0 | 0 | 0 |
| i=2 | a[2] | b | 0 | 0 | 1 |
| i=3 | a[3] | b | 0 | 0 | 1 |

1.1° Passaggio (i=2, j=2)

- i=0 || j=0 No
- a[i]=b[j] → a[2]=b[2] → 'b'='b' Sì
- Ricorsione su riga e colonna precedente (diagonale) diminuendo i e j → i=1, j=1

| | | j=0 | j=1 | j=2 | j=3 |
|-----|------|------|------|------|------|
| | | b[0] | b[1] | b[2] | b[3] |
| | | c | b | b | |
| i=0 | a[0] | | 0 | 0 | 0 |
| i=1 | a[1] | a | 0 | 0 | 0 |
| i=2 | a[2] | b | 0 | 0 | 1 |
| i=3 | a[3] | b | 0 | 0 | 1 |

PLSC

... CONTINUO ESEMPIO PRINT

1.1.1° Passaggio ($i=1, j=1$)

- $i=0 \parallel j=0$ No
- $a[i]=b[j] \rightarrow a[1]=b[1] \rightarrow 'a'='c'$ No
- Ramo else L[i][j] = L[i-1][j] $\rightarrow 0=0$ Sì e mi sposto in verticale
- Ricorsione su stessa colonna ma riga precedente (diminuisce solo i $\rightarrow i=0$)

| | | j=0 | j=1 | j=2 | j=3 |
|-----|------|------|------|------|------|
| | | b[0] | b[1] | b[2] | b[3] |
| | | c | b | b | |
| i=0 | a[0] | | 0 | 0 | 0 |
| i=1 | a[1] | a | 0 | 0 | 0 |
| i=2 | a[2] | b | 0 | 0 | 1 |
| i=3 | a[3] | b | 0 | 0 | 1 |

1.1.1.1° Passaggio ($i=0, j=1$)

- $i=0 \parallel j=0$ Sì chiudo questa ricorsione andando al passaggio precedente 1.1.1°
-

1.1.2° Passaggio ($i=1, j=1$)

- Stampo $a[i] \rightarrow a[1] \rightarrow 'b'$, chiudo questa ricorsione e vado al passaggio 1.1°
-

1.2° Passaggio ($i=2, j=2$)

- Dopo aver ottenuto 'b' dal passaggio 1.1.2°, stampo $a[i] \rightarrow a[2] \rightarrow 'b'$ e chiudo anche questa ricorsione e tutta la funzione con l'output della PLSC = 'b b'

PLSC

FUNZIONAMENTO PRINT VISIVA

| | | j=0 | j=1 | j=2 | j=3 | j=4 | j=5 | j=6 |
|-----|------|------|------|------|------|------|------|------|
| | | b[0] | b[1] | b[2] | b[3] | b[4] | b[5] | b[6] |
| | | c | b | a | b | a | c | |
| i=0 | a[0] | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| i=1 | a[1] | a | 0 | 0 | 0 | 1 | 1 | 1 |
| i=2 | a[2] | b | 0 | 0 | 1 | 1 | 2 | 2 |
| i=3 | a[3] | c | 0 | 1 | 1 | 1 | 2 | 3 |
| i=4 | a[4] | a | 0 | 1 | 1 | 2 | 2 | 3 |
| i=5 | a[5] | b | 0 | 1 | 2 | 3 | 3 | 3 |
| i=6 | a[6] | b | 0 | 1 | 2 | 2 | 3 | 3 |
| i=7 | a[7] | a | 0 | 1 | 2 | 3 | 3 | 4 |

PROCEDURA

- Parto da $L[m][n]$, dove m e n sono stati assunti rispettivamente da i,j e faccio i seguenti controlli:

- Controllo se $i==0$ e $j==0$, cioè se sono nella riga o colonna di 0, quando mi trovo qua mi fermo
- Controllo se i caratteri sono uguali, e se sono uguali decremento i e j e \rightarrow MI SPOSTO IN DIAGONALE
- Controllo se il valore della PLSC nella riga e colonna in cui mi trovo è uguale al valore della PLSC di stessa colonna ma riga diversa, e se così fosse \rightarrow MI SPOSTO IN VERTICALE
- Altrimenti \rightarrow MI SPOSTO IN ORIZZONTALE

- Infine, arrivo alla cima della PLSC e la leggerò solo nei punti quando gli spostamenti sono stati in DIAGONALE!

// In questo caso è CBBA

ALGORITMO DI HUFFMAN

FUNZIONAMENTO:

- Gli alberi (partendo da quelli iniziali) sono memorizzati in un **minheap**, dove questo albero è di ordinamento inverso, cioè la radice è l'elemento più piccolo (una specie di "coda con priorità")
- Si fa un ciclo di $n-1$ iterazioni dove per ogni estrazione:
 - Vengono estratti 2 alberi con radice minore (all'inizio la radice è rappresentata dalla frequenza)
 - Questi due alberi si fondono in un nuovo albero avente come etichetta della radice → La somma delle due radici
 - L'albero risultante poi è inserito nello heap (in coda) → Ed andrà a posizionarsi al posto giusto
 - Alla fine del ciclo nello heap mi rimarrà solamente un elemento che ritornerò che rappresenterà l'albero secondo Huffman già sistemato → Dall'unione fatta dei nodi nei passaggi precedenti, con le relative frequenze

OSS: Il valore contenuto nella radice del **minheap** è < del valore contenuto nei sottoalberi

Il ciclo avendo n iterazioni, ogni iterazione ha complessità $O(\log n)$:

→ Somma di 3 complessità $O(\log n)$ ciascuna per le 2 estrazioni e per l'inserimento

COMPLESSITÀ

$O(n \log n)$

STRUTTURA

```
struct NodeH {  
    char symbol;                                //0  
    int freq;                                   //1  
    NodeH *left;  
    NodeH *right;  
}
```

- 0- Carattere dell'alfabeto che ha senso solo per le foglie, mentre per i nodi interni no
- 1- Frequenza carattere, per i nodi intermedi è la somma delle frequenze minori

ALGORITMO DI HUFFMAN

```
Node* huffman(Heap H, int n){ //0
    for(int i=0; i<n-1; i++){ //1
        NodeH *t = new NodeH(); //2
        t->left = H.extract(); //3
        t->right = H.extract(); //4
        t->freq = (t->left->freq) + (t->right->freq); //5
        H.insert(t); //6
    }
    return H.extract(); //7
}
```

- 0- Gli passo lo heap che sarebbe la coda con priorità(**minheap**), dove ho inserito i nodi iniziali con le apposite frequenze, sottoschema di albero
- 1- Faccio il ciclo fino a n-1, poiché mi servirà un passaggio per il punto 7)
- 2- Creo un nuovo nodo (del tipo di Huffman Node H)
- 3- A questo nuovo nodo, a sinistra gli darò l'elemento dell'albero con frequenza minore a quello successivo, che sarà posizionato prima nella coda con priorità(**minheap**)
- 4- A destra invece quello con frequenza maggiore, che si trova dopo a quello del punto 3)
- 5- Alla frequenza faccio la somma delle radici (delle frequenze dai passaggi 3 e 4)
- 6- Inserisco nell' **minheap**, il nuovo nodo che andrà ad accodarsi, trovando la posizione giusta
- 7- Alla fine del ciclo avrò l'albero già costruito dentro il **minheap** e non dovrò far altro che estrarlo, e ci ritornerà la radice dell'albero (tutto intero)

ALGORITMO DI HUFFMAN

ESEMPIO HUFFMAN

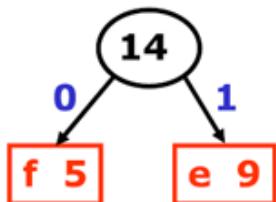
FASE INIZIALE → Ho tutti gli alberi con un solo nodo (SX=Carattere, DX=Frequenza)



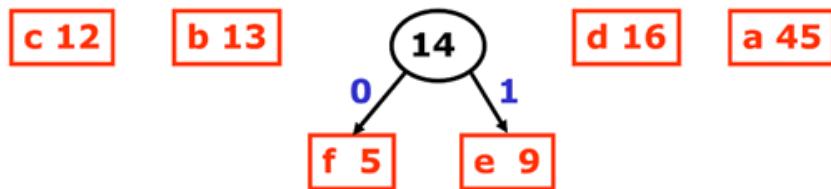
- Questo è l'heap H (coda con priorità) che ha dimensione n=6
- Ciclo i=0

1°) FASE → Scelgo i più piccoli nodi da combinare e metto 0 a SX e 1 a DX per codifica finale, e ne prendo sempre 2 alla volta di nodi per la costruzione dell'albero (CICLO → i=0)

- $i < n-1 \rightarrow 0 < 5$ Sì
- Creo un nuovo Nodo Huffman t
- A SX ci metto H.extract → La prima cosa che estraggo, cioè f(con frequenza 45)
- A DX ci metto H.extract → La seconda cosa che estraggo, cioè e(con frequenza 9)
- $T->freq = (t->left->freq + t->right->freq) = 5+9=14$



- H.insert(t) = Lo metto nel Heap H (nella coda con priorità) al suo posto



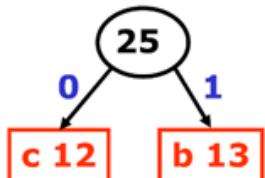
- $i=1$ ($i++$)

ALGORITMO DI HUFFMAN

... CONTINUO ESEMPIO HUFFMAN

2°) FASE → Ripeto il passaggio precedente per i nodi rimanenti (i=1)

- $i < n-1 \rightarrow 1 < 5$ Sì
- Creo un nuovo Nodo Huffman t
- A SX ci metto H.**extract** → La prima cosa che estraggo cioè, c (con frequenza 12)
- A DX ci metto H.**extract** → La seconda cosa che estraggo cioè, b (con frequenza 13)
- $T \rightarrow freq = (t \rightarrow left \rightarrow freq + t \rightarrow right \rightarrow freq) = 12 + 13 = 25$



- H.**insert(t)** = Lo metto nel Heap H (nella coda con priorità) al suo posto



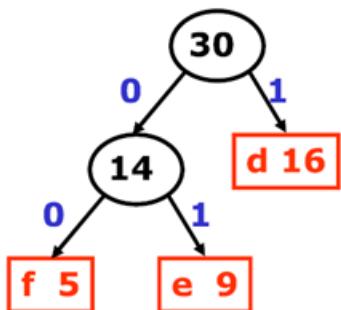
- $i = 2$ ($i++$)

ALGORITMO DI HUFFMAN

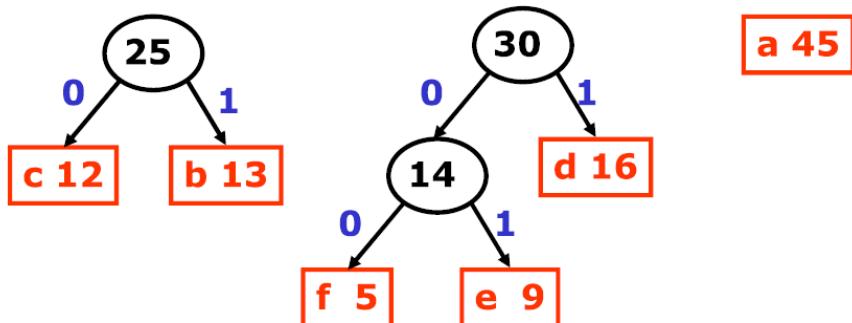
... CONTINUO ESEMPIO HUFFMAN

3° FASE → Ripeto il passaggio precedente per i nodi rimanenti (i=2)

- $i < n-1 \rightarrow 2 < 5$ Sì
- Creo un nuovo Nodo Huffman t
- A SX ci metto $H.\text{extract}$ → La prima cosa che estraggo, cioè il nodo (albero) con frequenza 14
- A DX ci metto $H.\text{extract}$ → La seconda cosa che estraggo, cioè d (con frequenza 16)
- $T \rightarrow \text{freq} = (t \rightarrow \text{left} \rightarrow \text{freq} + t \rightarrow \text{right} \rightarrow \text{freq}) = 14 + 16 = 30$



- $H.\text{insert}(t)$ = Lo metto nel Heap H (nella coda con priorità) al suo posto



- $i = 3$ ($i++$)

OSS

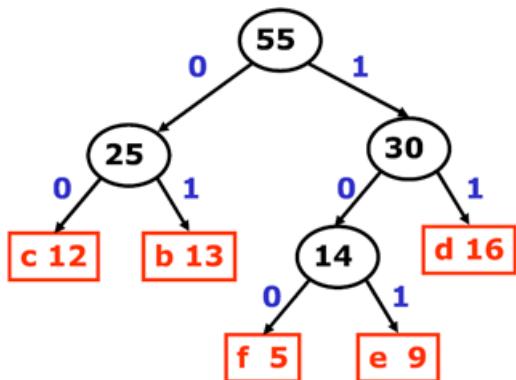
- Con l'unione dei vari nodi, quelli che in un risultato precedente erano nella parte dei più grandi, possono andare nella parte dei più piccoli
(vedi nodo 25 che dalla parte DX va alla parte SX)

ALGORITMO DI HUFFMAN

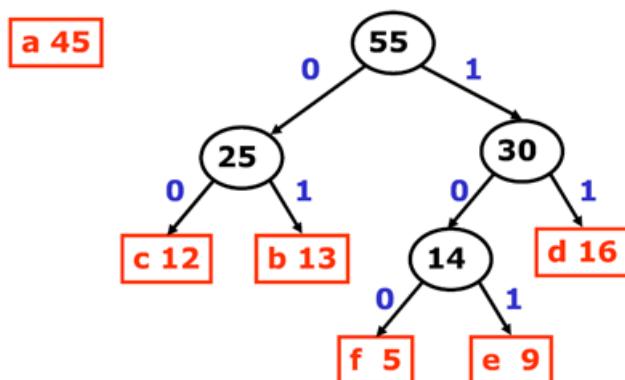
... CONTINUO ESEMPIO HUFFMAN

4°) FASE → Ripeto il passaggio precedente per i nodi rimanenti (i=3)

- $i < n-1 \rightarrow 3 < 5$ Sì
- Creo un nuovo Nodo Huffman t
- A SX ci metto $H.\text{extract}$ → La prima cosa che estraggo, cioè il nodo (albero) con frequenza 25
- A DX ci metto $H.\text{extract}$ → La seconda cosa che estraggo, cioè il nodo (albero) con frequenza 30
- $T \rightarrow \text{freq} = (t \rightarrow \text{left} \rightarrow \text{freq} + t \rightarrow \text{right} \rightarrow \text{freq}) = 25 + 30 = 55$



- $H.\text{insert}(t) =$ Lo metto nel Heap H (nella coda con priorità) al suo posto



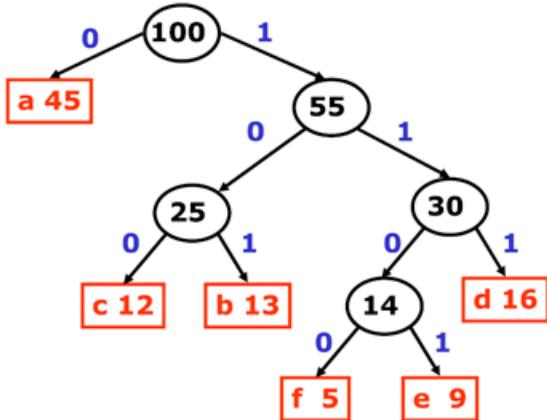
- $i = 4$ ($i++$)

ALGORITMO DI HUFFMAN

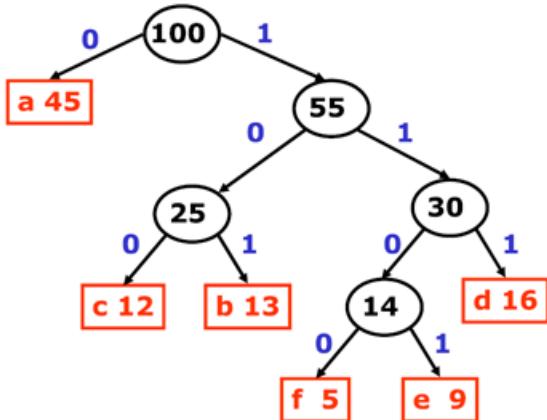
... CONTINUO ESEMPIO HUFFMAN

5°) FASE → Ripeto il passaggio precedente per i nodi rimanenti (i=4)

- $i < n-1 \rightarrow 4 < 5$ Sì
- Creo un nuovo Nodo Huffman t
- A SX ci metto H.extract → La prima cosa che estraggo, cioè a (con frequenza 45)
- A DX ci metto H.extract → La seconda cosa che estraggo, cioè il nodo (albero) con frequenza 55
- $T \rightarrow freq = (t \rightarrow left \rightarrow freq + t \rightarrow right \rightarrow freq) = 45 + 55 = 100$



- $H.insert(t) =$ Lo metto nel Heap H (nella coda con priorità) al suo posto



//Anche se è ripetuta l'immagine vuol dire che ho terminato la mia coda con priorità

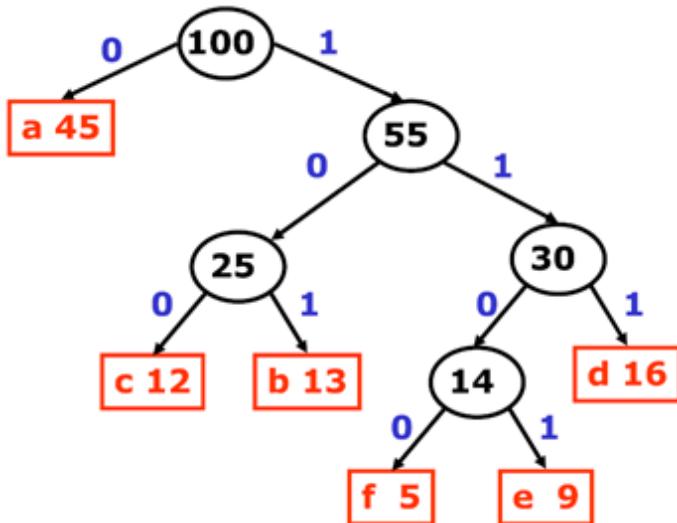
- $i = 5$ ($i++$)

ALGORITMO DI HUFFMAN

... CONTINUO ESEMPIO HUFFMAN

6°) FASE → Ripeto il passaggio precedente per i nodi rimanenti (i=5)

- $i < n-1 \rightarrow 5 < 5$ No
- **H.extract** → Estraggo l'unico elemento rimasto nella coda con priorità e quindi l'Heap ordinato secondo le frequenze creato nei passaggi precedenti, e lo faccio ritornare alla funzione chiudendola



Codifica binaria dei caratteri

| | |
|---|------|
| A | 0 |
| B | 101 |
| C | 100 |
| D | 111 |
| E | 1100 |
| F | 1101 |

GRAFI ORIENTATI

```
class Graph{
    struct Node{
        int nodeNumber;                                //0
        Node *next;
    };
    Node *graph[N];                                  //1
    NodeType nodeLabels[N];                         //2
    int mark[N];                                    //3

    void nodeVisit(int i){                           //4
        mark[i]=1;                                   //5
        <esamina nodeLabels[i]>;                  //6
        Node *g;                                     //7
        int j;
        for(g=graph[i]; g; g=g->next){           //8
            j=g->nodeNumber;                      //9
            if(!mark[j])                          //10
                nodeVisit(j);                     //11
        }
    }
public:
    void depthVisit(){
        for(int i=0; i<N; i++)      mark[i]=0;    //12
        for(int i=0; i<N; i++)          if(!mark[i])
            nodeVisit(i);
    }
    // ... Altri metodi
};
```

GRAFI ORIENTATI

- 0- ID del nodo
- 1- Lista di adiacenza, cioè ai successori di ogni nodo (vettore di puntatori ai nodi)
- 2- Etichetta di tipo NodeType associata ad ogni nodo
- 3- Marcatore che mi indica se ogni nodo è stato visitato, e viene posto a 1 se è stato visitato, 0 altrimenti
- 4- Funzione privata che non mi interessa esporre, e gli passo l'indice del nodo che al momento di questa chiamata voglio visitare
- 5- Essendo entrato dentro questa funzione, vuol dire che il nodo lo sto visitando e quindi marco l'indice del nodo a 1
- 6- Funzione che vede il valore del nodo, e può farne la stampa del contenuto, che si trova in nodeLabels
- 7- Dichiaro un puntatore a Node che mi servirà per scorrere la lista dei successori del nodo specifico
- 8- **for:** g conterrà il puntatore al primo elemento passato in ingresso i → (graph[i]) il quale mi servirà per scorrere la lista di adiacenza e, finché ho l'esistenza di g (cioè finché è diverso da null), in g metto il successivo (g->next), della lista di adiacenza
- 9- In j metto l'ID del successore
- 10- Controllo se non è marcato
- 11- Allora lo marco, richiamando la chiamata ricorsiva su j (che rappresenterà i successori)
- 12- Per lavorare sulla visita in profondità azzero le marcature, nel caso se in qualche visita precedente possono essere state fatte delle visite, e quindi marcate a 1
- 13- Scorro fino a N e vedo se il nodo non è visitato, lo visito e lo marco!

OSS:

- Nella Struct posso inserirci anche il campo che considera il peso di ogni arco cioè → “ArcType arcLabel”, dove ArcType=int
- Complessità **nodeVisit**

O(m)

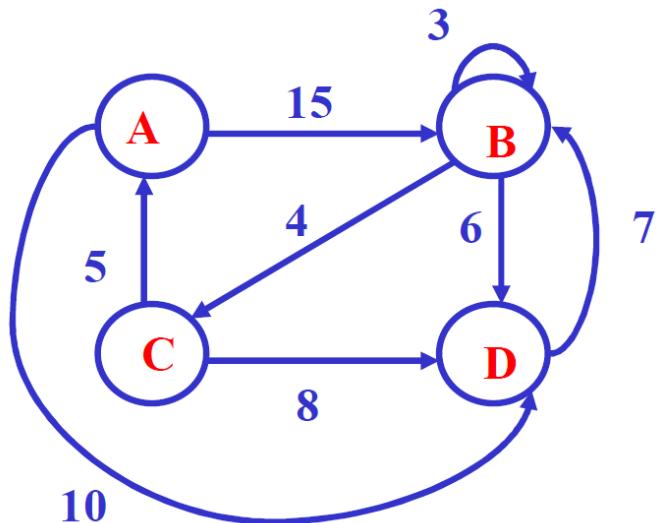
- Complessità **depthVisit**

O(m+n)

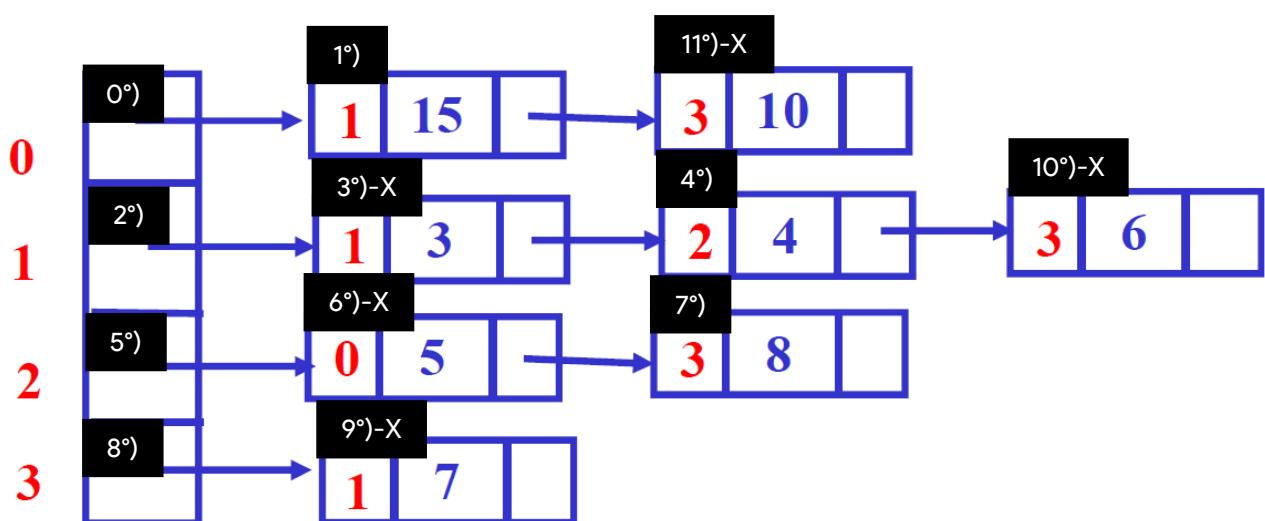
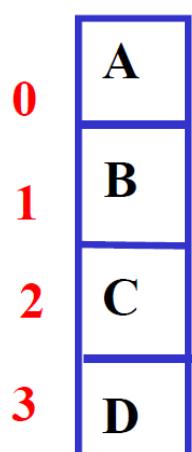
// dove io faccio n stampe e m verifiche di marcatura, perché per ogni arco devo verificare se ogni nodo puntato da quell'arco sia stato visitato o meno

GRAFI ORIENTATI

VISITA IN PROFONDITÀ (VISIVAMENTE)



nodeLabels



graph

GRAFI ORIENTATI

Passaggi depthVisit

0°) Parto da 0, controllo se il nodo non è marcato, non lo è e lo marco e leggo **A**

1°) Mi sposto sul nodo 1, controllo se il nodo non è marcato → Non lo è e mi sposto su 1

2°) Marco il nodo e leggo **B**

3°) Mi sposto sul nodo 1, controllo se il nodo non è marcato → È marcato X, vado avanti

4°) Mi sposto sul nodo 2, controllo se il nodo non è marcato → Non lo è e mi sposto su 2

5°) Marco il nodo e leggo **C**

6°) Mi sposto sul nodo 0, controllo se il nodo non è marcato → È marcato X, vado avanti

7°) Mi sposto sul nodo 3, controllo se il nodo non è marcato → Non lo è e mi sposto su 3

8°) Marco il nodo e leggo **D**

9°) Mi sposto sul nodo 1, controllo se il nodo non è marcato → È marcato X, vado avanti tornando indietro nella lista del passaggio 2°)

10°) Mi sposto sul nodo 3, controllo se il nodo non è marcato → È marcato X, vado avanti tornando indietro nella lista del passaggio 0°)

11°) Mi sposto sul nodo 3, controllo se il nodo non è marcato → È marcato X, mi fermo perché ho controllato tutti i nodi

→ La visita è **ABCD**

ALGORITMO DI DIJKSTRA

EURISTICA(PSEUDOCODICE)

```
Q = N; //0
per ogni nodo p diverso da p0{
    dist(p) = infinito; //2
    pred(p) = vuoto; //3
}
dist(p0) = 0; //4
while(Q contiene più di un nodo){ //5
    estrai da Q il nodo p con minima dist(p); //6
    per ogni nodo q successore di p{
        lpq = lunghezza dell'arco (p,q); //7
        if (dist(p)+lpq < dist(q)){
            dist(q) = dist(p)+lpq; //9
            pred(q) = p; //10
            re-inserisci in Q il nodo q modificato; //11
        }
    }
}
```

STUDIO DELLA COMPLESSITÀ

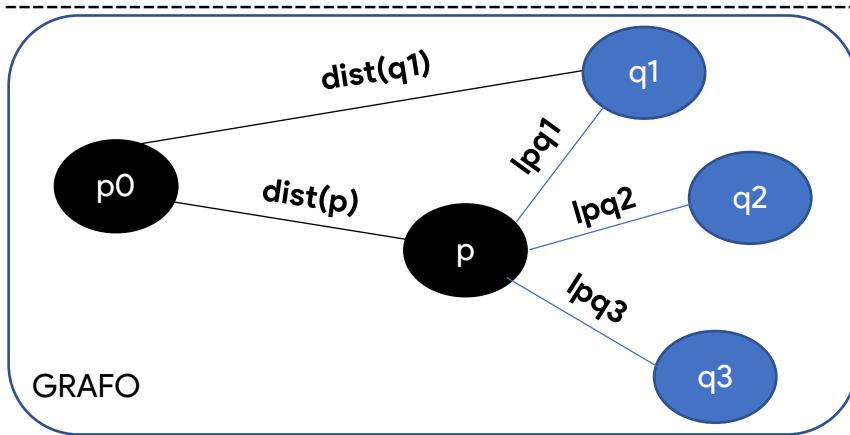
- Il ciclo al punto 1 ha complessità $\rightarrow O(n)$
- L'estrazione al punto 6 ha complessità $\rightarrow O(\log n)$
// Dopo che si fa un'estrazione devo mettere l'ultimo elemento in cima alla coda con priorità, poi devo procedere facendo le down per sistemare gli elementi in modo da mantenere la priorità
- I punti 9, 10 hanno complessità $\rightarrow O(1)$
- Il re-inserimento nella coda con priorità al punto 11 ha complessità $\rightarrow O(\log n)$
- Numero di iterazioni del ciclo while : n
- Complessità per ogni iterazione:
Complessità del punto 6 + $(m/n) *$ Complessità del punto 11 $\rightarrow O(\log n + (m/n)\log n)$
// $(m/n) =$ media del numero di nodi uscenti / media degli archi
- Complessità del ciclo: $O(n *$ complessità per ogni iterazione (precedente))
 $\rightarrow O(n (\log n + (m/n)\log n))$

COMPLESSITÀ TOTALE ALGORITMO DI DIJKSTRA: **$O(n \log n + m \log n)$**

ALGORITMO DI DIJKSTRA

OSSERVAZIONE: L'insieme Q rappresenta una coda con priorità

- 0- Insieme Q memorizzato in un **minHeap** dei nodi non sistemati deve essere pari a tutti i nodi, considerando anche il nodo sorgente, che sono tutti messi in N
- 1- Ciclo, considerando ogni nodo p diverso dal nodo sorgente p_0
- 2- Pongo inizialmente la distanza di ogni nodo p dal nodo sorgente (che sarebbe il cammino minimo) = infinito
- 3- Non gli do ai nodi p il predecessore, ponendolo a vuoto
- 4- Pongo la distanza di p_0 da sé stesso = 0, che sarebbe la distanza dalla sorgente alla sorgente
- 5- Finchè nel mio **minHeap** (coda con priorità) avrò più di un nodo
- 6- Estraggo da questa coda il nodo p con distanza minima dalla sorgente
- 7- Verifico la lunghezza dell'arco da che unisce p a q e lo metto nella variabile lpq
- 8- Controllo se la distanza dalla sorgente del nodo estratto p + la variabile precedente è < della distanza di q dalla sorgente
- 9- Allora come distanza di q dalla sorgente scelgo → La distanza dalla sorgente del nodo estratto p + la variabile lpq
- 10- Aggiorno il predecessore di q dandogli il nodo estratto p
- 11- Reinsisco nella coda con priorità il nodo appena modificato con il cammino minimo minore



1) Estraggo i nodi q_1 , q_2 , q_3 dalla coda con priorità essendo non sistemati per cui è definito un collegamento con p

2) Le tre distanze q_1 , q_2 , q_3 , e p hanno tutte distanze dalla sorgente (per comodità ne scelgo solo una, quella di q_1)

//inizialmente q_1 , q_2 , q_3 hanno distanza infinita

3) Considerando q_1 verifico la condizione che se la $dist(p) + lpq_1 < dist(q_1)$ → Aggiorno la $dist(q_1)$ mettendoci questa distanza → $dist(p) + lpq_1$

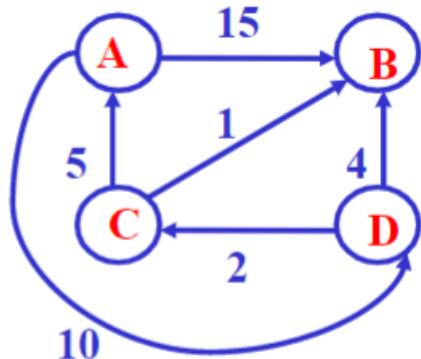
// lpq_1 è l'arco che congiunge p a q_1

"In poche parole applico la scelta greedy, istante per istante, cioè mi conviene passare per p e poi andare a q_1 partendo da p_0 , anziché andare direttamente da p_0 a q_1 "

ALGORITMO DI DIJKSTRA

ESEMPIO APPLICAZIONE ALGORITMO

Considerato il grafo



e la coda con priorità **minHeap** $\rightarrow Q\{A, B, C, D\}$

Estraggo A e sarà il nodo sorgente, quindi

$\rightarrow \text{dist}(A) = 0$ //distanza di A dal nodo sorgente (che è A stesso) = 0

\rightarrow Costruisco la tabella del tipo DIST | PRED

//Dove DIST è la distanza dal nodo indicato alla sorgente A e PRED è il suo predecessore

| A | B | C | D |
|-------|---------|---------|---------|
| 0 - | inf - | inf - | inf - |

OSS: All'inizio imposto infinito (inf) per tutti tranne che per il nodo sorgente

Controllo i successori di A:

B

- Verifico: $\text{dist}(A) + |(A,B)| < \text{dist}(B) \rightarrow 0+15 < \text{inf}$ sì
- Aggiorno: $\text{dist}(B)=15$, $\text{pred}(B)=A$

D

- Verifico: $\text{dist}(A) + |(A,D)| < \text{dist}(D) \rightarrow 0+10 < \text{inf}$ sì
- Aggiorno: $\text{dist}(D)=10$, $\text{pred}(D)=A$

Poiché ho estratto A aggiorno il mio **minHeap** $\rightarrow Q\{B, C, D\}$

Aggiorno la tabella DIST | PRED

| A | B | C | D |
|-------|--------|---------|--------|
| 0 - | 15 A | inf - | 10 A |

ALGORITMO DI DIJKSTRA

... CONTINUO ESEMPIO APPLICAZIONE ALGORITMO

Estraggo D dal **minHeap**, poiché dentro la coda ha distanza minore dall'origine

Controllo i successori di D (tranne quelli estratti):

B

- Verifico: $\text{dist}(D) + |(D,B)| < \text{dist}(B)$ → $10+4<15$ sì
- Aggiorno: $\text{dist}(B)=14$, $\text{pred}(B)=D$

C

- Verifico: $\text{dist}(D) + |(D,C)| < \text{dist}(C)$ → $10+2<\infty$ sì
- Aggiorno: $\text{dist}(C)=12$, $\text{pred}(C)=D$

Poiché ho estratto D aggiorno il mio **minHeap** → Q{B, C}

Aggiorno la tabella DIST | PRED

A B C D

| | | | |
|-------|--------|--------|--------|
| 0 - | 14 D | 12 D | 10 A |
|-------|--------|--------|--------|

Estraggo C dal **minHeap**, poiché dentro la coda ha distanza minore dall'origine

Controllo i successori di C (tranne quelli estratti):

B

- Verifico: $\text{dist}(C) + |(C,B)| < \text{dist}(B)$ → $12+1<14$ sì
- Aggiorno: $\text{dist}(B)=13$, $\text{pred}(B)=C$

Poiché ho estratto C aggiorno il mio **minHeap** → Q{B}

Aggiorno la tabella DIST | PRED

A B C D

| | | | |
|-------|--------|--------|--------|
| 0 - | 13 C | 12 D | 10 A |
|-------|--------|--------|--------|

- Mi fermo perché ho un solo elemento nel **minHeap**

ALGORITMO DI DIJKSTRA

... CONTINUO ESEMPIO APPLICAZIONE ALGORITMO

Costruisco la tabella finale per ogni nodo scelto ad ogni passo

| Nodo scelto | Q | A | B | C | D | |
|----------------|------------|------|------|------|------|--------------|
| | A, B, C, D | 0 /- | i /- | i /- | i /- | Passaggio 0° |
| A | B, C, D | 0 /- | 15/A | i /- | 10/A | Passaggio 1° |
| D | B, C | 0 /- | 14/D | 12/D | 10/A | Passaggio 2° |
| C | B | 0/- | 13/C | 12/D | 10/A | Passaggio 3° |

minHeap

TABELLA DIST/PRED

CAMMINI MINIMI

- Da A a B lunghezza 13: $A \rightarrow D \rightarrow C \rightarrow B$
 - Da A a C lunghezza 12: $A \rightarrow D \rightarrow C$
 - Da A a D lunghezza 10: $A \rightarrow D$

ALGORITMI NON DETERMINISTICI

Soddisfattibilità della formula logica

```
int nsat(Formula f, int *a, int n){ //0
    for(int i=0; i<n; i++) //1
        a[i] = choice({0,1});
    if (value(f,a)) //2
        return 1;
    else return 0;
}
```

- 0- Funzione di tipo *int*, che mi restituisce vero (1) o falso (0). Alla funzione gli si passa la formula logica *f*, e un vettore *a* di dimensione *n* che rappresenta le variabili booleane
- 1- Ciclo in cui assegno ad ogni variabile booleana un valore a caso tra 0 e 1, grazie alla funzione *choice* → Funzione che sceglie un valore tra l'insieme {0,1}
- 2- Se il valore che assume *f* con il vettore *a* è vero, restituisco 1, 0 altrimenti

COMPLESSITÀ T(n)

O(n)

Ricerca in array

```
int nsearch(int *a, int n, int x){
    int i = choice( {0 ... n-1} ); //1
    if (a[i] == x) //2
        return 1;
    else
        return 0;
}
```

- 1- La funzione *choice* in questo caso serve per estrarre un indice ed assegnarlo alla variabile *i*
- 2- Se in posizione *i*-esima appena estratta, trovo l'elemento *x*, restituisco 1, 0 altrimenti

COMPLESSITÀ T(n)

O(1)

ALGORITMI NON DETERMINISTICI

Ordinamento

```
int nsort(int *a, int n){  
    int b[n];                                //1  
    for(int i=0; i<n; i++)  
        b[i] = a[i];  
    for(int i=0; i<n; i++)  
        a[i] = b[choice({0 ... n-1})];  
    if(ordinato(a));                         //4  
        return 1;  
    return 0;
```

- 0- Inizializzo un vettore
- 1- Copio il vettore *a* nel vettore *b*
- 2- Estraggo a caso gli elementi del vettore *b* e li assegno al vettore *a*
- 3- Controllo se il vettore *a* è ordinato, e se lo è restituisco 1, 0 altrimenti

COMPLESSITÀ T(n)

O(n)

RECAP COMPLESSITÀ

Serie di Fibonacci iterativa → **O(n)**

Serie di Fibonacci ricorsiva → **O(2ⁿ)**

Ricerca lineare iterativa e ricorsiva → **O(n)**

Ricerca binaria ricorsiva → **O(log n)**

SelectionSort iterativa e ricorsiva → **O(n²)**

BubbleSort → **O(n²)**

QuickSort:

- Caso medio e migliore → **O(n log n)**
- Caso peggiore → **O(n²)**

MergeSort array e liste → **O(n log n)**

InsertionSort:

- Caso migliore → **O(n)**
 - Caso peggiore e medio → **O(n²)**
-

PreOrder, InOrder, PostOrder BINTREE

- In funzione del numero dei nodi → **O(n)**
- In funzione del numero dei livelli → **O(2ⁿ)**

Nodes BINTREE → **O(n)**

//CONTA NODI

Leaves BINTREE → **O(n)**

//CONTA FOGLIE

FindNode ABR:

- Caso medio e migliore → **O(log n)**
- Caso peggiore → **O(n)**

InsertNode ABR → **O(log n)**

DeleteMin ABR:

- Caso medio e migliore → **O(log n)**
- Caso peggiore → **O(n)**

DeleteNode ABR → **O(log n)**

RECAP COMPLESSITÀ

Up HEAP → $O(\log n)$

Down HEAP → $O(\log n)$

HeapSort → $O(n \log n)$

- Down HeapSort → $O(\log n)$
 - Extract HeapSort → $O(\log n)$
 - BuildHeap HeapSort → $O(n)$
-

CountingSort → $O(n+k)$

RadixSort → $O(d(n+k))$

Lenght PLSC → $O(2^n)$

QuickLenght PLSC → $O(n^2)$

Print PLSC → $O(n+m)$

Huffman → $O(n \log n)$

NodeVisit GRAPH → $O(m)$

DepthVisit GRAPH → $O(n+m)$

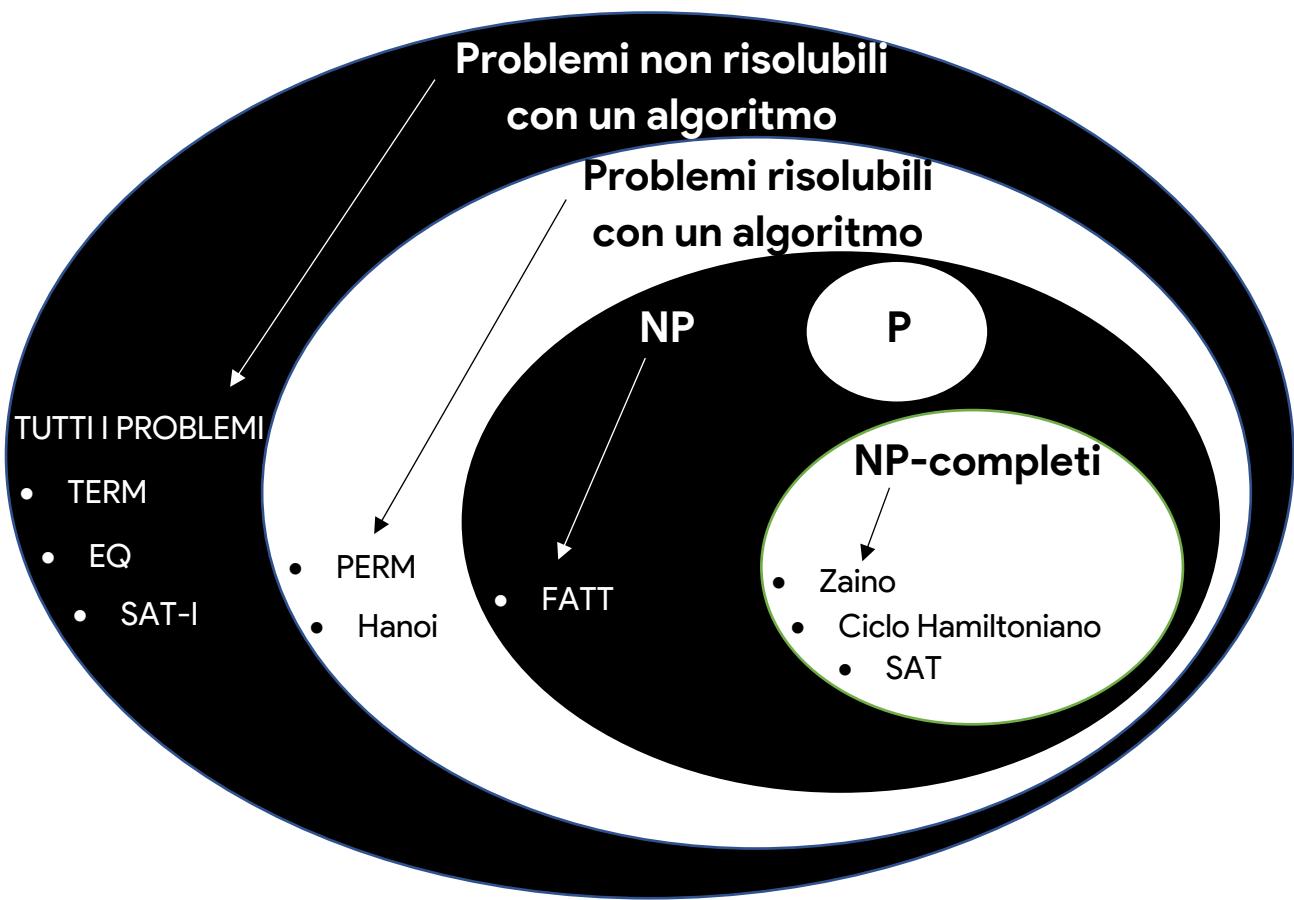
DIJSKTRA → $O(n \log n + m \log n)$

Nsat → $O(n)$

Nsearch → $O(1)$

Nsort → $O(n)$

PROBLEMI



- **TERM:** Decidere la terminazione di un programma su un input
- **SAT-I:** Soddisfattibilità di una formula logica del I ordine
(di formule logiche maggiori al binario)
- **EQ:** Decidere l'equivalenza di due programmi
- **PERM:** Trovare tutte le permutazioni di un insieme
- **Hanoi:** Torre di Hanoi
- **FATT:** Fattorizzazione → Scomposizione di un numero in fattori primi
- **Ciclo Hamiltoniano:** Ciclo di un grafo che tocca tutti i nodi una sola volta
- **SAT:** Soddisfattibilità di una formula logica nei predicati
→ Data una formula logica con n variabili devo trovare se esiste una combinazione di booleani che, assegnate alle n variabili rendono vera la formula
- **Zaino:** Problema di ottimizzazione del riempimento dello zaino
(risolvibile con euristiche greedy)

- **P:** Insieme di tutti i problemi decisionali facili da risolvere, risolvibili in tempo Polinomiale con un algoritmo *deterministico*
- **NP:** Insieme di tutti i problemi decisionali facili da verificare, risolvibili in tempo Polinomiale con un algoritmo *Nondeterministico*
- **NP-completi:** Problemi più difficili della classe NP