Algoritmi e Strutture Dati

È proibita qualunque riproduzione di questo fascicolo, anche parziale, in libri,

pubblicazioni anche telematiche, cd, dvd, siti web e ogni altra forma di pubblicazione

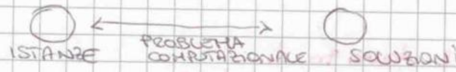
senza il consenso scritto dell'autore.

In particolare, è proibita la vendita di questo fascicolo o di parti di esso in qualunque forma.

Algoritmi e Strutture Dati

DEFINIZIONI DI BASE

- PROBLEMA COMPUTAZIONALE: è la relazione che lega input ed output
  - ISTANZA: insieme dei valori di input
  - SOLUZIONE: insieme dei valori di output

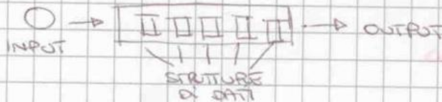


- ALGORITMO: procedura di calcolo ben definita, che a partire da un insieme di valori input, produce valori in output
  → un algoritmo è detto corretto per un problema computazionale p se ∀ istanza x di p si ha:
  1) algoritmo termina
  2) algoritmo produce un output y corretto

- RAM (Random Access Machine): astrazione di un calcolatore più elementare della macchina di Von Neumann → possiede le seguenti caratt.:
  1) RAM (Random Access Memory): costituita a sua volta da un n° di registri arbitrario, indirizzabili individualmente
  2) Input ed output sono sequenze di dati
  3) Ogni operazione ha costo unitario
  4) Le istruzioni vengono eseguite in maniera sequenziale

- PSEUDOCODICE: Linguaggio elementare traducibile in qualsiasi linguaggio di programmazione (traduzione = implementazione)

- STRUTTURA DI DATI: contenitore di dati contiene il dato nella forma più utile e veloce per produrre l'output. È un micro-processo di un algoritmo



I dati in memoria possono essere organizzati secondo diverse discipline → queste discipline prendono il nome di
- STRUTTURE DATI -

PSEUDOCODICE:

- VARIABILI & ASSEGNAZIONI
  - Variabili in pseudocodice hanno un tipo ma non necessitano di dichiarazione
  - Assegnazioni sono denotate con il simbolo (=) o (←)
  - Nello pseudocodice sono possibili assegnazioni multiple
  - Per scrivere un commento si usano i simboli: /*...*/, //..., ▷...
  - Nello pseudocodice non si mette il ";" finale

- ISTRUZIONI CONDIZIONALI:

```
If condizione then
    corpo
else
    corpo
```

- Le parentesi nello pseudocodice non servono. Per far capire che più istruzioni fanno parte dello stesso corpo basta indentare bene
- else è opzionale
- then si può anche omettere

ES:
```
1. if i<0 then
2.    ▷ prendo il valore assoluto di i (il modulo)
3.    i = -i
```

ES:
```
1. if i<0 then
2.    abs = -i
3.    minore_zero = true
4. else
5.    abs = i
6.    minore_zero = false
```

- ISTRUZIONE RIPETIVA FOR
  - for assegnazione to punto di arresto (valore massimo)
    corpo
    (incremento sottointeso)
  - for assegnazione down to punto di arresto (valore minimo)
    corpo
    (decremento sottointeso)

ES:
```
1. for i = a.length - 1 down to 1
2.        ▷ traslo in avanti tutti i valori di A
3.     A [i] = A [i-1]
4.     ▷ quando i < 1 esce dal for
5. A [0] = new        ▷ inserisco il valore
```

- ISTRUZIONE RIPETITIVA REPEAT ~~UNIT~~ UNTIL:
  il blocco viene eseguito finché la condizione è falsa, quando è vero esce dal ciclo

```
repeat
        corpo
until   condizione
```

ES:
```
1. ▷ PONGO A zero tutte le posizioni di A
2. i = 0
3. repeat
4.        A [i] = 0
5.        i = i+1
6. until  i > A.length - 1
```

- ISTRUZIONE RIPETITIVA WHILE :
  il blocco viene eseguito finché vale la condizione quando la condizione è falsa esce dal ciclo

```
while   condizione   do
        corpo
```

ES:
```
while     i < a.length and not trovato do
               if A[i] == quello_che_cerco then
                    trovato = false
          else
                    i = i+1
```

ES:
```
▷ pongo tutte le posizioni a zero
i = 0
while    i < a.length   do
             A[i] = 0
             i = i+1
```

- ARRAY
  - si possono definire array di qualunque tipo
  - la lunghezza dell'array è data da A.length
  - A[i] identifica l'elemento dell'array A in posizione i
  - le posizioni dell'array vanno da 0 a A.length-1
  - A [0 ... 3] indica il sotto array A[0], A[1], A[2], A[3]
  - La lunghezza dell'array va specificata in un commento

Algoritmi e Strutture Dati

2/

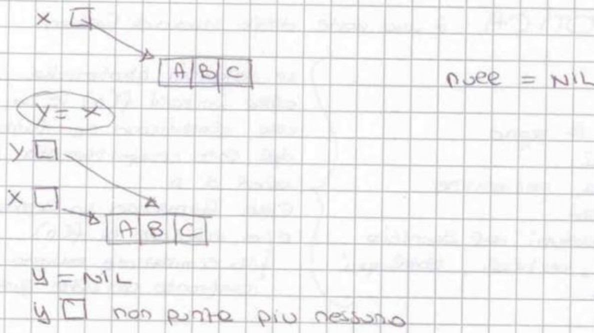
- ES: A[0] = 1 ▷ A è un array di 10 interi
  A[a.length - 1] = 0 ▷ l'ultimo elemento è 0

- ES: ▷ A è un array contenente 3 array di 10 interi
  A[2][9] = 0 ▷ l'ultimo elemento di A[2] è 0

- ESPRESSIONI BOOLEANE: sono espressioni booleane
  1) true, false
  2) il risultato di operatori booleani come: and, or, not (&, |, !) — posso scriverlo in entrambi i modi
  3) il risultato di operatori relazionali come: ==, >, ≥, <, ≤

ES:
```
1. if i < max and trovato == false then
2.     i = i + 1
```

- OGGETTI:
  ogni oggetto ha delle variabili dette attributi (o campi)
  si accede all'attributo campo dell'oggetto x tramite il costrutto x.campo
  gli attributi sono puntati dall'oggetto



y = NIL

y □ non punta più nessuno

- PROCEDURE (metodi di java)
  - la procedura riceve una copia dei parametri, se si modificano all'interno della procedura i parametri copiati, la modifica non ha effetto all'esterno
  - se il parametro è un oggetto, allora non viene riprodotto, ma gli viene passato una copia del riferimento (puntatore) all'oggetto
  - le procedure riportano un solo valore

ES: somma (a, 3) ▷ chiamata della procedura

```
somma (a, b)
   c = a + b        } PROCEDURA
   return c
```

ES:
```
1. massimo (a, b)
2.    max = a
3.    if a < b then
4.       max = b
5.    return max
```

ES
```
1. massimo (A)
2.    max = A[0]
3.    for i = 1 to a.length - 1
4.       if A[i] > max then
5.          max = A[i]
6.    return max
```

ES:
```
1. massimo (M)
2.     somma = 0
3.     for i=0 to M.length - 1
4.         for j=0 to M[i].length - 1
5.             somma = somma + M[i][j]
```

ES:
```
1. positivo (A)
2.     verifica = true
3.     for i=0 to A.length - 1
4.         if A[i] < 0 then
5.             verifica = false
6.     return verifica
```

ES:
```
1. posizione - massimo (A)
2.     max = A[0]
3.     posizione = 0
4.     for i=0 to a.length - 1
5.         if max < A[i] then
6.             max = A[i]
7.             posizione = i
8.     return posizione
```

NOTAZIONE ASINTOTICA: è una parte dello studio di funzioni

STUDIO DI FUNZIONI:
1) Dominio
2) Intersezioni con gli assi & segno
3) Simmetria e periodicità
4) Continuità, discontinuità, derivazione
5) Massimi, minimi e flessi
6) Comportamento agli estremi del dominio
   → asintoti orizzontali, verticali, obblIqui
   → notazione asintotica

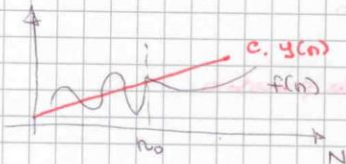
le notazioni asintotiche si applicano alle funzioni $f(n)$ il cui $D = \mathbb{N}$; esse classificano le funzioni a seconda del loro comportamento per grandi valori di n.
esse forniscono un limite superiore e/o inferiore a $f(n)$
{la limitazione avviene mediante il confronto con altre funzioni}

• O-GRANDE (TETTO)

$O(g(n))$ - o grande di g di n è l'insieme delle $f(x)$ limitate superiormente da $g(n)$

$f(n) \in O(g(n)) \iff \exists$ 2 cost > 0 c e n_0 / $\forall n \geq n_0 \Rightarrow 0 \leq f(n) \leq c \cdot g(n)$

$O(g(n)) = \{ f(n) : \exists c > 0, n_0 > 0 \ / \ 0 \leq f(n) \leq c \cdot g(n), \forall n \geq n_0 \}$



osservazione: $O(g(n)) = 0 \Rightarrow g(n)$ è una $f(x)$ asintoticamente < 0 (convenzionalmente $g(n)$ non è mai asintoticamente < 0)

c e n_0 dipendono da $f(n)$ ma qual è il ruolo di c?

$2n \in (n^2)$

$2n \notin (n)$ ⟹ avremmo troppi insiemi ⟹ invece noi vogliamo trascurare le costanti

$n^2 + 1 \in (n^2)$

PROPRIETÀ RIFLESSIVA: $g(n) \in O(g(n))$

FUNZIONI INCOMMENSURABILI: è sempre vero che $f(n) \in O(g(n))$ oppure ~~g(n) ∈~~ $g(n) \in O(f(n))$?

↳ no! vedi come esempio senx e cosx basta che uno sia dispari e l'altro pari e nessuno ammetterà mai come tetto l'altra funzione

Algoritmi e Strutture Dati

3)

- PROPRIETÀ TRANSITIVA: se $f(n) \in O(h(n))$ e $h(n) \in O(g(n)) \Rightarrow f(n) \in O(g(n))$
- REGOLA DEI FATTORI COSTANTI POSITIVI: se $f(n) = d \cdot h(n)$ e $h(n) \in O(g(n))$ e $d > 0 \Rightarrow f(n) \in O(g(n))$
- REGOLA DELLA SOMMA: se $f(n) = h(n) + k(n)$ e $h(n) \in O(g(n))$ e $k(n) \in O(g(n)) \Rightarrow f(n) \in O(g(n))$

ES:

$f(n) = n^2 + n^4 \in O(n^4)$? sì
$f(n) = 0{,}5\,n^2 + 3n \in O(n)$? NO
$f(n) = 4n^2 + 3\log(n) \in O(n)$? NO
$f(n) = n\log(n) \in O(n^2)$? sì
$f(n) = 5 \in O(n)$? sì
$f(n) = 5 \in O(1)$? sì
$f(n) \in O(n^2) \ldots f(n) \in O(n\log(n))$? non è detto
$f(n) \in O(n) \ldots f(n) \in O(n\log(n))$? sì
$h(n) \in K(n) \in O(g(n)) \ldots f(n) = h(n) + K(n) \in O(g(n))$? sì
$f(n) = \log(n) \in O(2^n)$? sì
$f(n) \in O(\log(n)) \ldots f(n) \in O(n)$? sì
$f(n) \in O(n^2) \ldots f(n) \in O(n^3)$? sì
$f(n) \in O(g(n)) \Rightarrow g(n) \in O(f(n))$? NO
$f(n) \notin O(g(n)) \Rightarrow g(n) \in O(f(n))$? NO
$f(n) = n^3 + 3n \in O(n^2)$? NO
$O(23) = O(1)$? sì
$f(n) = 1 \in O(n)$? sì
$f(n) = 2n + 3n^2 \in O(n)$? NO

- OMEGA (pavimento): $\Omega(g(n))$ omega di $g(n)$ è l'insieme delle $f(x)$ limitate inferiormente da $g(n)$

[$f(n) \in \Omega(g(n)) \Leftrightarrow \exists$ 2 costanti > 0 c e n_0 / $\forall n \geq n_0 \Rightarrow 0 \leq c \cdot g(n) \leq f(n)$
$\Omega(g(n)) = \{ f(n) : \exists c > 0, n_0 > 0 \,/\, 0 \leq c \cdot g(n) \leq f(n), \forall n \geq n_0 \}$]

↳ è analogo a scrivere $0 \leq g(n) \leq f(n) \cdot c$

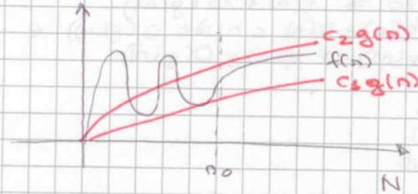
- osservazione: $f(n) \in \Omega(g(n)) \Leftrightarrow g(n) \in O(f(n))$

- anche per Ω $\exists$ $f(x)$ incommensurabili
- anche per Ω vale la proprietà riflessiva
- PROPRIETÀ TRANSITIVA: se $f(n) \in \Omega(h(n))$ e $h(n) \in \Omega(g(n)) \Rightarrow f(n) \in \Omega(g(n))$
- REGOLA DEI FATTORI COSTANTI POSITIVI: $f(n) = d \cdot h(n)$ e $h(n) \in \Omega(g(n))$ e $d > 0 \Rightarrow f(n) \in \Omega(g(n))$
- REGOLA DELLA SOMMA: $f(n) = h(n) + k(n)$ e $h(n) \in \Omega(g(n))$ e $K(n) \in \Omega(g(n)) \Rightarrow f(n) \in \Omega(g(n))$

ES:
$f(n) = n^2 + n^4 \in \Omega(n^3)$? sì
$f(n) = 3 \in \Omega(n)$? NO
$f(n) \in \Omega(n^3) \Rightarrow f(n) \in \Omega(n^2)$? sì
$f(n) \in \Omega(n)$ $h(n) \in \Omega(n) \Rightarrow f(n) = k(n) + h(n) \in \Omega(n)$? sì
$f(n) \in O(g(n)) \Rightarrow g(n) \in \Omega(f(n))$? sì
$n^2 \in \Omega(n)$? sì
$f(n) = 4n + 6n^2 + 8n^3 \in \Omega(n^2)$? sì

- TETA: (Gerarchia): $\Theta(g(n))$ - teta di $g(n)$ è l'insieme delle $f(x)$ lim. superiormente e inferiormente da $g(n)$

$f(n) \in \Theta(g(n)) \Leftrightarrow \exists$ 3 cost. >0 $c_1, c_2, n_0 / \forall n \geq n_0 \Rightarrow 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$
$\Theta(g(n)) = \{f(n) : \exists n_0 > 0, c_1 > 0, c_2 > 0 / 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n), \forall n \geq n_0\}$

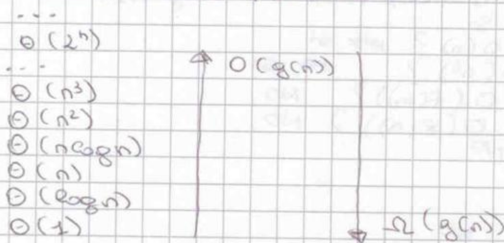


OSS: $f(n) \in \Theta(g(n)) \Leftrightarrow \begin{cases} f(n) \in O(g(n)) \\ f(n) \in \Omega(g(n)) \end{cases}$

- per Θ vale la proprietà riflessiva e tutte le proprietà di O-grande e Ω

PROPRIETÀ SIMMETRICA: $f(n) \in \Theta \Leftrightarrow g(n) \in \Theta(f(n)) \Rightarrow f(n) \in g(n)$ hanno una sorta di relazione di equivalenza appartengono ad uno stesso classe di $f(x) \Rightarrow \Theta$ consente di classificare le $f(x)$ in classi di equivalenza

- GERARCHIA DELLE FUNZIONI:



- PROPRIETÀ DEI POLINOMI: $P_1(n)$, $P_2(n)$ polinomi di grado g_1 e g_2
  - $\Rightarrow g_1 = g_2 \Rightarrow P_1(n) \in \Theta(P_2(n))$ e viceversa
  - $\Rightarrow g_1 < g_2 \Rightarrow P_1(n) \in O[P_2(n)]$
  - $\Rightarrow g_1 > g_2 \Rightarrow P_1(n) \notin O(P_2(n))$
  - $\Rightarrow a > 1 \Rightarrow P_1(n) \in O(a^n)$
  - $\Rightarrow a > 1 \Rightarrow a^n \notin O(P_1(n))$

ES: $f(n) = n^2 + n^4 \in \Theta(n^3)$? NO
$f(n) = n^2 + n^3 \in \Theta(n^3)$? Sì
$f(n) \in O(g(n)) \Rightarrow f(n)$ limitato superiormente da $g(n)$? Sì
$f(n) \in \Omega(g(n)) \Rightarrow f(n)$ limitato inferiormente da $g(n)$? Sì
$f(n) \in \Theta(g(n))$ e $h(n) \in \Theta(g(n)) \Rightarrow f(n) = k(n) + h(n) \in \Theta(g(n))$? Sì
$f(n) \in O(g(n))$, $f(n) \in \Omega(g(n)) \Rightarrow f(n) \in \Theta(g(n))$? Sì
$O(g(n))$ denota l'insieme di $f(x)$ limitate inferiormente da $g(n)$? NO
$f(n) \in \Theta(g(n)) \Rightarrow g(n) \in \Theta(f(n))$? Sì

STRUTTURE DI DATI ELEMENTARI

- STRUTTURE DATI ELEMENTARI (PILE / CODE / LISTE): essi sono insiemi dinamici, ovvero insiemi in cui si possono aggiungere e togliere elementi.
  Gli elementi sono solitamente oggetti, ovvero non contengono gli oggetti, ma il riferimento agli oggetti.
  Tra i campi che costituiscono l'oggetto c'è:
  - $\Rightarrow$ LA CHIAVE: che identifica l'oggetto e consente di ordinare la struttura
  - $\Rightarrow$ DATI SATELLITE: tutti gli altri campi che caratterizzano l'oggetto

- PILE E CODE
  nelle pile e nelle code l'elemento rimosso dalla cancellazione è predeterminato
  PILE (stack): strategia LIFO (Last in first out)
  CODE (queque): strategia FIFO (First in First out)

Algoritmi e Strutture dati

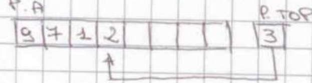
4/

- PILE (STACK):

Operazioni:

1. Consultazione (IS-EMPTY): true ⇒ la pila è vuota / false ⇒ la pila non è vuota
2. Inserimento (PUSH): inserisce un elemento
   (!) può dare un errore di overflow se l'implementazione prevede un n° max di elementi in pila
3. Rimozione e Restituzione (POP): rimuove e restituisce l'elemento affiorante della pila
   (!) da un errore di underflow se la pila è vuota
4. Restituzione (TOP): restituisce l'elemento affiorante della pila senza rimuoverlo
5. Svotamento (EMPTY): svuota la pila
6. Conteggio (SIZE): ritorna il n° di elementi in pila

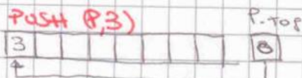
→ IMPLEMENTAZIONE DI UNA PILA:
P pila può essere implementata con un oggetto contenente A Array e P.top come attributi. P.top contiene l'indice dell'elemento affiorante



OSSERVAZIONE: se la pila è vuota P.top = -1

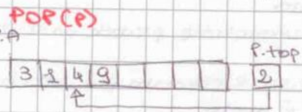
ES:

PUSH (P,3)

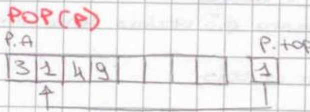


PUSH (P,1), PUSH (P,4), PUSH (P,9)

POP(P)



POP(P)



PUSH: // inserisce un elemento

```
PUSH(P,x)
1. if P.top == A.length-1
2.    then error "overflow"
3. else
4.    P.top = P.top+1
5.    P.A[P.top] = x
```

POP: // rimuove e restituisce un elemento

```
POP(p)
1. if p.top == -1
2.    then error "underflow"
3. else
4.    p.top = p.top - 1
5.    return p.A[p.top + 1]
```

IS-EMPTY: // dice se la pila è vuota

```
IS-EMPTY(p)
1. return p.top == -1
```

EMPTY: // svuota la pila

```
EMPTY(p)
1. p.top = -1
```

TOP: // ritorna l'elemento affiorante

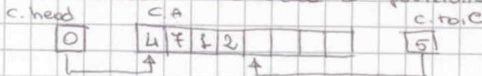
```
TOP(p)
1. return p.A[p.top]
```

SIZE: // ritorna il n° di elementi

```
SIZE(p)
1. return p.top + 1
```

- CODE (QUEUE)
  - operazioni
    1) Consultazione (IS-EMPTY): true ⇒ coda è vuota / false la coda non è vuota
    2) Inserimento (Enqueue): inserisce un elemento
       (!) può dare un errore di overflow se l'implementazione prevede un n° massimo di elementi in coda
    3) Rimozione (Dequeue): rimuove e restituisce l'elemento più vecchio
       (!) da un errore di "underflow" se la coda è vuota
    4) Restituzione (FRONT): restituisce l'elemento più vecchio senza rimuoverlo
    5) Svuotamento (EMPTY): svuota la coda
    6) Conteggio (SIZE): ritorna il n° di elementi in coda

→ IMPLEMENTAZIONE DI UNA CODA:
C coda può essere implementato come un oggetto contenente A array, c.head, c.tail come attributi
c.head: indice dell'elemento più vecchio
c.tail: indice della I^a posizione libera

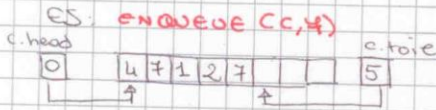


Osservazione: se c.head e c.tail puntano lo stesso indice ⇒ la pila è vuota

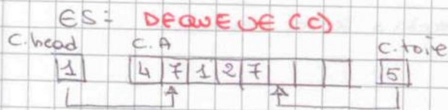
Algoritmi e Strutture Dati

5/

ES: ENQUEUE (C,4)

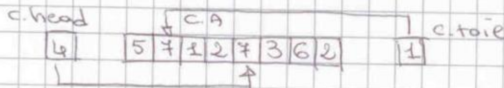


ES: DEQUEUE (C)



OSSERVAZIONE: l'array è gestito come una lista circolare

ES: ENQUEUE (C,3); ENQUEUE (C,6); ENQUEUE (C,2); DEQUEUE (C); DEQUEUE (C); DEQUEUE (C), ENQUEUE (C,5)



Osservazione: la coda non può avere più di n-1 elementi
Poiché se avesse n elementi ⇒ c.toie = c.head anche se non è vuota
⇒ ASSURDO

ENQUEUE: // inserisce 1 elemento

```
ENQUEUE (C,x)
1. if c.head == c.toie+1 or (c.toie == c.A.length-1 and c.head==0)
2.    then error "overflow"
3. else
4.    c.A[c.toie] = x
5.    if c.toie == c.A.length-1
6.       then c.toie = 0
7.    else
8.       c.toie = c.toie +1
```

DEQUEUE: // rimuove e restituisce un elemento

```
DEQUEUE(C)
1. if c.head == c.toie
2.    then error "underflow"
3. else
4.    x = c.A[c.head]
5.    if c.head == c.A.length-1
6.       then c.head = 0
7.    else
8.       c.head = c.head+1
9.    return x
10.
```

IS-EMPTY: // dice se la coda è vuota

```
IS-EMPTY (C)
1. return c.head == c.toie
```

EMPTY : // vuota la coda

```
EMPTY(C)
1. return C.head = C.tail = 0
```

FRONT : // ritorna l'elemento più vecchio senza rimuoverlo

```
FRONT(C)
1. if C.head == C.tail
2.     then error "empty queue"
3. else
4.     return C.A[C.head]
```

SIZE : // ritorna il n° elementi

```
SIZE(C)
1. if C.head ≤ C.tail
2.     return C.tail - C.head
3. else
4.     return C.A.length - C.head + C.tail
```

Algoritmi e Strutture Dati

6/

• LISTE:

Strutture di dati in cui gli oggetti sono disposti in sequenze. Esse possono prevedere tutte le operazioni su insiemi dinamici

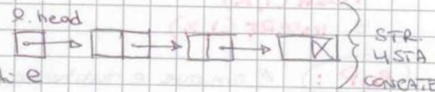
→ OPERAZIONI:

1) INSERT (l, x): inserisce x in testa alla lista l
2) INSERT-BEFORE: inserisce x in l prima di y
3) ADD (l, x): aggiunge x in coda alla lista l
4) ADD-AFTER (l, x, y): aggiunge x in l dopo y
5) DELETE (e, x): elimina x da e
6) EMPTY (l): vuota la lista
7) NEXT (l, x): ritorna l'elemento dopo x oppure NIL se x è l'ultimo element
8) PREV (l, x): ritorna l'elemento prima x, oppure NIL se x è il primo elem
9) FIRST (l): ritorna il 1° elemento di l
10) LAST (l): cerca l'elemento con chiave k in l e ritorna x
11) SEARCH (l, k): ritorna l'ultimo elemento di l
12) IS-EMPTY (l): ritorna true se la lista è vuota altrimenti false

→ IMPLEMENTAZIONE LISTA: ∃ diverse implementazioni per una lista

1) LISTA CONCATENATA:

- ogni nodo ha i seguenti attributi:
  - x.next: fa riferimento all'elemento successivo oppure NIL
  - x.key: chiave
  - x.info: dati satellite
- la lista ha un attributo:
  - l.head: fa riferimento al 1° elemento di l



osservazione: l.head = NIL ⇒ lista vuota ☒ l.head

IS-EMPTY:

```
IS-EMPTY(l)
1. return l.head == NIL
```

EMPTY:

```
EMPTY(l)
  l.head = NIL
```

FIRST:

```
FIRST(l)
  return l.head
```

NEXT:

```
NEXT(l, x)
  return x.next
```

INSERT:

```
INSERT(l, x)
  x.next = l.head
  l.head = x
```

DELETE-FIRST: elimina il primo nodo di l

```
DELETE-FIRST(l)
1. if l.head == NIL
2.     then error "lista vuota"
3. else
4.     l.head = l.head.next
```

DELETE:

oss: x deve essere presente nella lista

```
DELETE(l,x)
1. x=l.head
2. if x==y
3.     then l.head = l.head.next
4. else
5.     while y.next != x
6.         y = y.next
7.     y.next = y.next.next
```

possiamo implementare una lista come una pila... quindi definiamo PUSH e POP

PUSH // inserisce un elemento in testa alla lista

```
PUSH(l,x)
1. INSERT(l,x)
```

POP: // rimuove e restituisce il 1° elemento di l

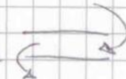
```
POP(l)
1. x = l.head
2. DELETE(l,x)
3. return x
```

possiamo implementare una lista come una coda... definiamo quindi DEQUEUE ENQUEUE

ENQUEUE: // inserisce un elemento in coda alla lista

```
ENQUEUE(l,x)
1. LAST(l).next = x
```

DEQUEUE: // elimina il primo nodo di l e restituisce il suo valore



```
DEQUEUE(l)
1. if l.head == NIL
2.     then error "lista vuota"
3. else
4.     x = l.head
5.     l.head = l.head.next
6. return x
```

Algoritmi e Strutture Dati

71

SEARCH:

asserzione: l'elemento con la chiave k deve esserci

```
SEARCH (l, k)
1. x = l.head
2. while (x != NIL)
3.     if x.key == k
4.        then return x
5.     else
6.        x = x.next
```

INVERTI: inverte gli elementi della lista l

```
INVERTI (l)
1. L = LUNGHEZZA (l)
2. A = TRASFORMA (l, L)
3. INVERTI (A)
4. l = TRASFORMA (A)
5. return l
LUNGHEZZA (l)
1. x = l.head
2. if x == NIL
3.    then return 0
4. else
5.    i = 1
6.    while x.next != NIL
7.        x = x.next
8.        i++
9.    return i
TRASFORMA (l, L)
1. ▷ creo un array di dim L
2.  x = l.head
3.  for i = 0 to L
4.      A[i] = x
5.      x = x.next
6.  return A
INVERTI (A)
1.  META = A.length/2
2.  for i = 0 to META
3.      SCAMBIA (A, i, A.length - 1 - i)
SCAMBIA (A, i, j)
1.  Temp = A[i]
2.  A[i] = A[j]
3.  A[j] = Temp
TRASFORMA (A)
1.  l.head = A[0]
2.  x = l.head
3.  x = x.next
4.  for i = 1 to A.length - 1
5.      x = A[i]
6.      x = x.next
7.  return l
```

CONCAT: // Concatena 2 liste

```
CONCAT (l, p)    ▷ p non essere più modificato
  LAST(l).next = p.head
  return l
```

2) LISTA DOPPIAMENTE CONCATENATA:

→ ogni nodo ha i seguenti attributi
- x.next: fa riferimento all'elemento successivo o NIL
- x.prev: fa riferimento all'elemento precedente o NIL
- x.key: chiave
- x.info: dati satellite

→ ogni lista ha i seguenti attributi
- l.head: fa riferimento al I° elemento di l
- l.tail: fa riferimento all'ultimo elemento di l

INSERT:

```
INSERT (l, x)
1. x.next = l.head
2. if l.head != NIL
3.     then l.head.prev = x
4. l.head = x
5. x.prev = NIL
```

DELETE:

```
DELETE (l, x)
  if x.prev != NIL
    then x.prev.next = x.next
  else
      l.head = x.next
  if x.next != NIL
    then x.next.prev = x.prev
```

INSERT-BEFORE:

```
INSERT-BEFORE (l, x, y)
  x.next = y
  x.prev = y.prev
  y.prev = x
  if x.prev == NIL
    then l.head = x
  else
      x.prev.next = x
```

ADD-AFTER:

```
ADD-AFTER (l, x, y)
1. x.prev = y
2. x.next = y.next
3. y.next = x
4. if x.next == NIL
5.     then l.tail = x
6. else
7.     x.next.prev = x
```

Algoritmi e Strutture Dati

8/

ENQUEUE: // inserisco elemento in coda

```
ENQUEUE(l, x)
1. if l.head == NIL
2.     then x.prev = NIL
3.          x.next = NIL
4.          l.head = x
5.          l.tail = x
6. else
7.     x.prev = l.tail
8.     x.next = NIL
9.     l.tail = x
```

DEQUEUE:

```
DEQUEUE()
1. if l.head == NIL
2.     then error "lista vuota"
3. else
4.     x = l.head
5.     l.head.next.prev = NIL
6.     l.head = l.head.next
7. return x
```

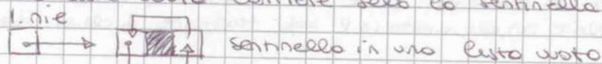
INVERTI // uguale a quello singolarmente concatenato

CONCAT:

```
CONCAT (l, p)   ▷ la lista p dopo non può essere più modificata
1. l.tail.next = p.head
2. p.head.prev = l.tail
3. l.tail = p.tail
4. return l
```

- SENTINELLE: nodo fittizio introdotto in testa alla lista, grazie ad esso le operazioni sono semplificate perché non sono più necessari test al contrario poiché la lista ⇒ lista circolare

OSS: la lista vuota contiene solo la sentinella



OSS: non si può cancellare o aggiungere una sentinella

DELETE: (con sentinella)

```
DELETE (l, x)
   x.prev.next = x.next
   x.next.prev = x.prev
```

INSERT: (con sentinella)

```
INSERT (l, x)
   x.next = l.nie.next
   l.nie.next.prev = x
   l.nie.next = x
   x.prev = l.nie
```

SEARCH: (con sentinella)

```
SEARCH (l, k)
1. x = l.nil.next
2. while x != l.nil and x.key != k
3.       do x = x.next
4. return x
```

3) LISTE IN ARRAY:

3 array per 1 lista doppiamente concatenata.

l.next / l.prev / l.key sono 3 array

• Es:

l.next		3	-1		1	
l.key		3	5		2	
l.prev		6	1		-1	

⟹ l.head → [X|2|] ⇄ [|3|] ⇄ [|3|X]

il next di 3 si trova in posizione 3 nell'array ... "5"
il prev di 2 si trova in posizione -1 ⟹ è il 1° elemento della lista

osservazione: un array può contenere più liste in posizioni diverse

osservazione: per inserire 1 elemento nell'array occorre sapere quante posizioni libere ha l'array ⟹ si può creare una seconda lista che memorizzi tutte le posizioni libere (l.free)

l.next	6	3	0	-1	7	2	1	-1
l.key		3	5			2		
l.prev	2	6	5	1	0	-1	-1	4

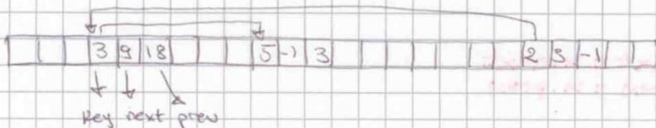
l.head [6]

l.free [5]

osservazione: inserire un elemento in l ⟹ trasferire 1 elem. da l.free a l.head
cancellare un elemento in l ⟹ trasferire 1 elem. da l.head a l.free

osservazione: lista vuota ⟹ tutte le posizioni sono assegnate a l.free

questa implementazione può essere fatta anche con un solo array



Algoritmi e Strutture Dati

COMPLESSITÀ DEGLI ALGORITMI:

Un programma può essere eseguito su una piattaforma opportuna con input opportuni. La sua esecuzione ha un costo espresso tramite: tempo/memoria etc

↳ FATTORI CHE INFLUENZANO IL TEMPO DI CALCOLO:
1) Dimensione INPUT (+ o − grande)
2) Algoritmo (+ o − efficiente) ← MAGGIOR INTERESSE
3) Hardware (+ o − veloce)
4) Linguaggio (+ a basso livello + è vel / pro + alto livello + lento)
5) Compilatore (produce un codice eseguibile + o − velocemente)
6) Programmatore (+ o − esperto)

TEMPO = DENARO ⟹ dobbiamo produrre algoritmi sempre veloci. Purtroppo quando progettiamo non possiamo misurare direttamente l'efficienza dell'algoritmo, dobbiamo però prevedere il tempo di calcolo dell'algoritmo

- ANALISI ASINTOTICA:
  Il tempo di calcolo dipende dalla dimensione dell'INPUT
  1) CASO MIGLIORE: caso in cui l'algoritmo finisce prima. Tempo necessario nel caso più conveniente → non serve a granché
  2) CASO MEDIO: Molto utile ma molto laborioso → dobbiamo avere sia il caso migliore che quello peggiore
  3) CASO PEGGIORE: caso in cui l'algoritmo finisce dopo
     ↳ si preferisce un errore per eccesso che uno per difetto
     ↳ più facile da trovare rispetto al medio e dà più garanzia rispetto al migliore

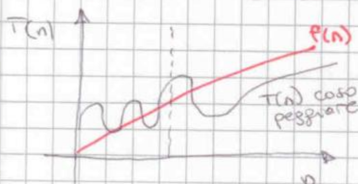
L'ANALISI ASINTOTICA ha bisogno di una funzione
$T(n)$: tempo di calcolo rispetto al caso peggiore

Quanto costa ogni operazione elementare? Definiamo costante il costo di ogni riga (istruzione) di codice
↳ Ci sono 2 strategie per stimare il costo:
1) STRATEGIA + ONEROSA: calcolo esplicito di $T(n)$ a partire dallo pseudocodice
2) STRATEGIA + EFFICIENTE: calcolo il costo asintotico di $T(n)$ [usiamo questa]

$O(f(n))$:

Definizione: − A Algoritmo ha "complessità temporale $O(f(n))$" se $T(n) = O(f(n))$

− T di esecuzione è al max $f(n)$ che è un limite superiore (UPPER-BOUND). $f(n)$ è la quantità "sufficiente" per eseguire A



OSS: se $f(n) = O(g(n)) \Rightarrow T(n) = O(g(n))$ però noi vogliamo un O-grande + possibilmente piccolo

$\Omega(f(n))$

Definizione: - A algoritmo ha "complessità temporale $\Omega(f(n))$" se $T(n) = \Omega(f(n))$

- T di esecuzione è min f(n) che è un "limite inferiore" (LOWER BOUND)
f(n) è la quantità di tempo "necessario" per eseguire A

$\Theta(f(n))$:

Definizione: A algoritmo "ha complessità temporale* $\Theta(f(n))$" se ha complessità temporale $O(f(n))$ e $\Omega(f(n))$

- l'esecuzione di A è f(n), cioè limite superiore e inferiore
f(n) è quantità di T "necessaria e sufficiente"

- CALCOLO EFFICIENTE DEL COSTO ASINTOTICO -

- Istruzione semplice $\Rightarrow O(1)$
- Sequenza di istruzioni semplici $\Rightarrow O(1)$
- Sequenza di istruzioni generiche $\Rightarrow \sum O(\text{istruzioni})$
- If - else $\Rightarrow$ costo max tra parte if e parte else
- Istruzione ripetitiva $O(f(n))$ = N° iterazioni
$O(g(n))$ = costo di ogni iterazione
$O(\text{ciclo}) \Rightarrow O(g(n) \cdot f(n))$

(!) Nelle istruzioni ripetitive va fatta maggiore attenzione soprattutto nel while

Algoritmi e Strutture Dati

ALBERI RADICATI:

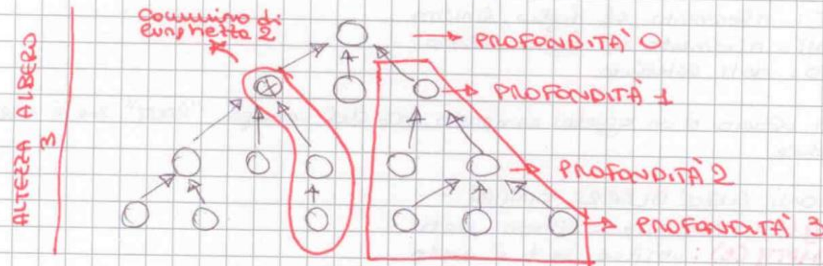
def: si dice "albero radicato" un insieme di nodi su cui è definita una relazione binaria → "x è figlio di y" e "y è figlio di x"

oss: ogni nodo ha 1 solo genitore, tranne la radice che non ha genitori. c'è un cammino diretto da ogni nodo alla radice

TIPI DI ALBERI

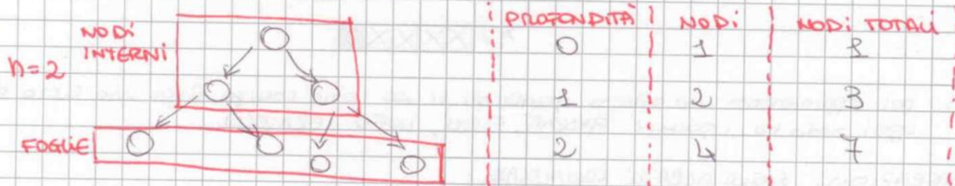
1) BINARI: ogni nodo può avere 2 figli (1 DX, 1 SX) → oss: l'ordine dei figli conta
2) m-ARI: ogni nodo ha al max m figli → oss: l'ordine dei figli non conta
3) QUALSIASI: non è noto a priori il n° MAX degli dei figli

def: 2 nodi con lo stesso genitore si dicono "FRATELLI"
- il n° di figli di un nodo è il suo "GRADO" (del nodo)
- i nodi di grado 0 sono detti "FOGLIE"
- un nodo non foglia è detto "INTERNO"



def: - si dice "cammino" una sequenza di nodi | 1 è il genitore del successivo
- n° degli archi in un cammino è detto "lunghezza" del cammino
- la lunghezza del cammino dalla radice a un nodo è detta "profondità"
- profondità del nodo più profondo è detta "ALTEZZA" dell'albero
- x è "antenato" di y / y è "discendente" di x
- l'insieme costituito da z e tutti i suoi discendenti è detto "sottoalbero radicato a z"
- l'albero è detto "ORDINATO" se l'ordine dei figli è significativo
- l'albero è detto "completo" se ogni livello presenta tutti i nodi possibili

ALBERI BINARI COMPLETI



PROFONDITÀ	NODI	NODI TOTALI
0	1	1
1	2	3
2	4	7

oss: n° nodi in un livello è 2^p dove p = profondità

N° FOGLIE $= 2^h$

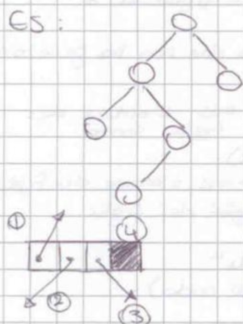
N° NODI INTERNI $= 2^h - 1$

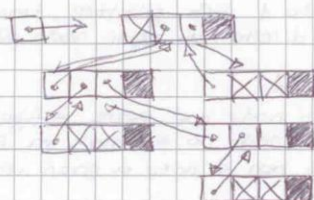
N° NODI TOTALI $= 2^h + 2^h - 1 = 2^{h+1} - 1$

T = tree (albero)
R = root (radice)

Def: un albero binario si dice "quasi completo" se mancano dee delle foglie nella parte destra

Es:



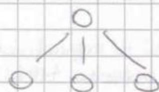


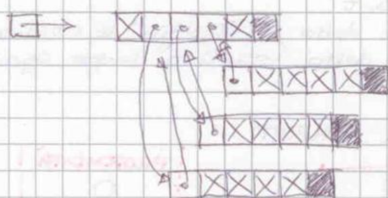
①: PARENT: riferimento al nodo genitore
②: LEFT: riferimento al figlio sinistro
③: RIGHT: riferimento al figlio destro
④: INFO: dati satellite

OSS: un albero è un oggetto composto solo dal campo "ROOT" che è riferito al nodo radice

– OPERAZIONI SUGLI ALBERI BINARI –
1) EMPTY(): restituisce l'albero vuoto
2) IS-EMPTY(t): verifica se t è vuoto
3) ROOT(t): restituisce la radice (NIL se t è vuoto)
4) LEFT(n): restituisce il figlio sinistro
5) RIGHT(n): restituisce il figlio destro
6) INFO(n): restituisce le info di n

Es: m=4 RAPPRESENTAZIONI DEGLI ALBERI M-ARI



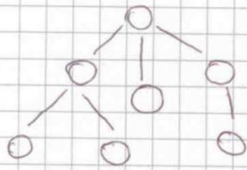


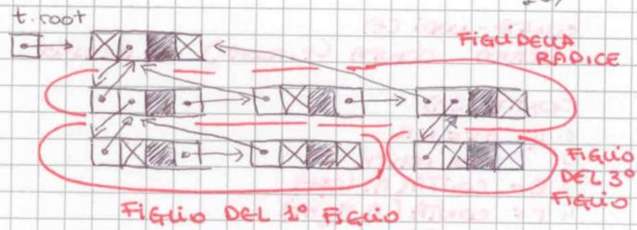
OSS: per rappresentare un albero qualsiasi si usa come campo figli una lista di nodi. ogni nodo ha i campi: PARENT, FIGLI, INFO, FRATELLI

OPERAZIONI SUGLI ALBERI QUALSIASI:
1) EMPTY():
2) IS-EMPTY(t):
3) ROOT(t):
4) FIRST-CHILD(n): restituisce il I° figlio (o NIL se non ho figli)
5) NEXT-SIBLING(n): restituisce il figlio fratello destro di n (o NIL ~~se non ho figli~~ se fratello dx ∄)
6) INFO(n):

Algoritmi e Strutture Dati

11/





VISITE DI ALBERI:

1) PREORDINE: dopo aver processato un n si procede verso i figli (TOP-DOWN)
2) POST-ORDINE: n può essere processato solo quando i figli sono stati processati (BOTTOM UP)
3) SIMMETRIA: se t è binario, processo figlio sinistro, poi n, poi figlio destro

CERCA: // verifica se v è presente in t binario

(A)
```
CERCA(t,v)
1. return PRE-CERCA(t.root, v)   ▷ innesco

PRE-CERCA(n,v)
1. if n == NIL
2.    then return FALSE
3. if n.info == v
4.    then return TRUE
5. return PRE-CERCA(n.left, v) OR PRE-CERCA(n.right, v)
```

(B)
```
CERCA(t,v)
1. return POST-CERCA(t.root, v)   ▷ INNESCO

POST-CERCA(n,v)
1. if {n == NIL}
2.    then return FALSE
3. l = POST-CERCA(n.left, v)
4. r = POST-CERCA(n.right, v)
5. return n.info == v OR l OR r
```

(C)
```
CERCA(t,v)
1. return SIMM-CERCA(t.root, v)   ▷ INNESCO
SIMM-CERCA(n,v)
1. if n == NIL
2.    then return FALSE
3. if n.left.info == v
4.    then return TRUE
5. if n.info == v
6.    then return TRUE
7. if n.right == v
8.    then return TRUE
9. return SIMM-CERCA(n.parent, v)
```

CONTA-NODI: // ritorna il n° nodi di t binario

```
CONTA-NODI (t)
1. return CONTA (t.root) ▷ innesco

CONTA (n)
1. if n == NIL
2.    then return 0
3. l = CONTA(n.left)
4. r = CONTA(n.right)
5. return l+r+1
```

CAMMINO: // verifica se un albero binario è un cammino

```
CAMMINO (t)
1. return CAMM (t.root) ▷ innesco

CAMM (n)
1. if n == NIL
2.    then return TRUE
3. if n.right == NIL AND n.left == NIL
4.    then return TRUE
5. if n.right != NIL AND n.left == NIL
6.    then return CAMM (n.right)
7. if n.right == NIL AND n.left != NIL
8.    then return CAMM (n.left)
9. return FALSE
```

HEIGHT: // calcola l'altezza di t

```
HEIGHT (t)
1. return POST-HEIGHT (t.root)

POST-HEIGHT (n)
1. if n == NIL
2.    then return 0
3. l = POST-HEIGHT (n.left)
4. r = POST-HEIGHT (n.right)
5. return MAX (l,r) + 1
```

AVARAGE: // calcola la media dei valori di un albero binario

```
AVARAGE (t)
1. numnodi = CONTA-NODI (t)
2. somma = SOMMA (t)
3. return somma/numnodi
```

Algoritmi e Strutture Dati

12)

SOMMA:

```
SOMMA (t)
  return  SOM(t.root)

SOM (n)
  if n == NIL
    then return 0
  e = SOM(n.left)
  r = SOM(n.right)
  return n.info + e + r
```

COMPLETO: // verifica se un albero è completo (binario)

```
COMPLETO (t)
  n = CONTA-NODI(t)
  h = HEIGHT(t)
  return n == 2^(h+1) - 1
```

STAMPA SIMMETRICA: // per alberi binari

```
STAMPA-SIMM(t)
  STAMPA-NODO-SIMM(t.root)

STAMPA-NODO-SIMM(n)
  if n == NIL
    then print ('(')
  else
    PRINT ('(')
    STAMPA-NODO-SIMM(n.left)
    PRINT (n.info)
    STAMPA-NODO-SIMM(n.right)
  PRINT (')')
```

IL PROBLEMA DELL'ORDINAMENTO

tipologie di ordinamento:
1) ALGORITMI GREEDY (Selection Sort)
2) ALGORITMI ITERATIVI (Insertion Sort)
3) ALGORITMI DIVIDE ET IMPERA (Merge Sort)

- SELECTION SORT: scegliere sempre l'alternativa che al momento sembra migliore (scelta localmente ottima, ma non comporta una scelta globalmente ottima) Prende l'elemento più piccolo e lo mette al primo posto. Prende l'elemento più piccolo tra i rimanenti e lo mette al II° posto ecc

```
SELECTION - SORT (A)
1. for i = 0 to A.lenght - 2
2.     min = i
3.     for J = i+1 to A.lenght - 1
4.         if A[J] < A[min]
5.             then min = J
6.     SCAMBIA (A, i, min)
```

```
SCAMBIA (A, i, min)
1 temp = A[i]
2. A[i] = A[min]
3 A[min] = temp
```

- COMPLESSITÀ : $\Theta(n^2)$

Def: un algoritmo si dice che opera "in loco" se non c'è la necessità di copiare l'input in altre strutture dati

oss: il selection sort opera in loco

Def: un algoritmo si dice "STABILE" se non modifica l'ordine degli elementi con lo stesso valore

oss: il selection-sort è stabile

- INSERTION SORT : cominciando da un singolo elemento, inserisco un elemento alla volta e lo ordino rispetto agli altri ⇒ il sottoinsieme ordinato cresce, finché tutti gli elementi sono inseriti ⇒ ORDINATI

```
INSERTION - SORT (A)
1. for i = 1 to A.lenght - 1
2.     Key = A[i]
3.     J = i - 1
4.     while J > -1 AND A[J] > Key
5.         do A[J+1] = A[J]
6.            J = J - 1
7.     A[J+1] = Key
```

OSS: INSERTION-SORT è IN LOCO e STABILE

COMPLESSITÀ:
- CASO PEGGIORE : $\Theta(n^2)$
- CASO MIGLIORE : $\Theta(n)$ già ordinato e non entra nel WHILE
- CASO MEDIO : $\Theta(n^2)$

oss: L'insertion è efficiente in piccole istanze, è più efficiente selection perché nel caso migliore ha $\Theta(n)$, per di più è ONLINE (può essere utilizzato quando i numeri arrivano 1 alla volta)

- MERGE - SORT: divide il problema finché non si trova una soluzione e poi attua la fusione

DIVIDE (divide l'istanza corrente)
IMPERA (conquista su istanze più piccole)
COMBINA (fusione)

MERGE - SORT (A, p, r) (p = inizio, r = fine)

```
1. if p<r
2. then q = (p+r)/2
3. MERGE-SORT (A, p, q)
4. MERGE-SORT (A, q+1, r)
5. MERGE (A, p, q, r)
```

MERGE (A, p, q, r)

```
1.  l1 = q-p+1              ▷ lunghezza 1 array
2.  l2 = r-q                ▷ lunghezza 2 array
3.  for i=0 to l1-1
4.      L[i] = A[p+i]       ▷ array L è array di sinistra
5.  for j=0 to l2-1
6.      R[i] = A[q+1+j]     ▷ array R è array di destra
7.      i=0     L[l1] = +oo  (serve per iterazione
8.      j=0     R[l2] = +oo   successiva)
9.  for k=p to r
10.     if L[i] ≤ R[j]
11.         then A[k] = L[i]
12.                  i = i+1
13.     else    A[k] = R[j]
                     j = j+1
```

COMPLESSITÀ: $\Theta(n \log n)$ calcolato con il MASTER THEOREM

MASTER THEOREM

Hp: $T(n) = \begin{cases} \Theta(1) & \text{se } n=0 \\ aT(n/b) + f(n^k) & \text{se } n>0 \end{cases}$

Nel MERGE-SORT $a = 2$, $b = 2$, $f(n^k) = n$, $k = 1$ $\Rightarrow a = b^k \Rightarrow \Theta(n \log n)$

Th

$a < b^k \Rightarrow T(n) = \Theta(n^k)$
$a = b^k \Rightarrow T(n) = \Theta(n^k \log n)$
$a > b^k \Rightarrow T(n) = \Theta(n^{\log_b a})$

Oss. non è in loco perché ho bisogno di altre strutture dati, ma è stabile poiché in caso di elementi uguali non esegue nulla

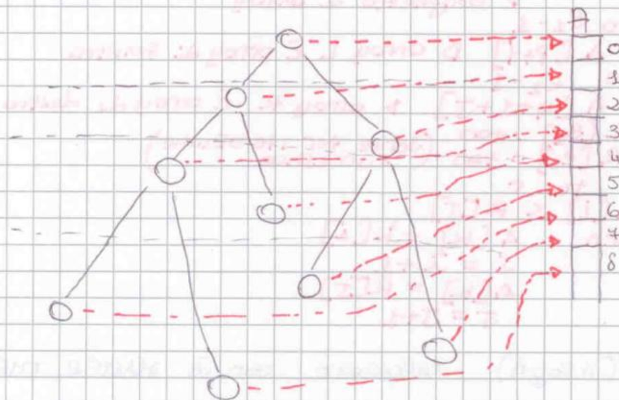
CODE DI PRIORITÀ

Def: si dice "coda di priorità" una collezione di S elementi. Ad ogni elemento è associato una chiave. Sulle chiavi è definito un ordinamento

- OPERAZIONI:
  1) EMPTY(S): inizializza la coda vuota
  2) INSERT(S,x): inserisce x nella coda S
  3) MAXIMUM(S): restituisce l'elemento di S con la chiave più grande
  4) EXTRACT-MAX(S): restituisce l'elemento di S con chiave più grande e lo rimuove
  5) NULL(S): TRUE (coda vuota) / FALSE (coda non vuota)

OSS: Gli elementi della coda è un determinato processo. Ogni processo richiede una coda di priorità

Def: Si dice "HEAP" una struttura di dati utilizzata per realizzare una coda di priorità è un array, i cui valori sono in rapporto con la loro posizione nell'array. (è un array letto come un albero quasi completo)



NODO GENITORE: $\frac{i-1}{2}$ / NODI FIGLI: $2i+1$ & $2i+2$

OSS: A è lungo A.length ma i valori significativi sono tra 0 e A.heap-size

- OPERAZIONI (2):
  6) PARENT(i): ritorna l'indice del genitore
  7) LEFT(i): ritorna l'indice del figlio sinistro
  8) RIGHT(i): ritorna l'indice del figlio destro

PARENT

```
PARENT(i)
1. return ⌊(i-1)/2⌋
```

($\lfloor \dots \rfloor$ = INT)

LEFT

```
LEFT
1. return 2i+1
```

RIGHT

```
RIGHT (i)
  return 2i + 2
```

Def: si definisce "MAX - HEAP" un heap in cui il valore di un elemento è ≥ al valore dei figli

9) MAX - HEPIFY // se LEFT(i) e RIGHT(i) sono MAX-HEAP => trasforma il sotto-albero radicato in un MAX - HEAP

```
MAX - HEPIFY (A, i)
1. l = LEFT (i)
2. r = RIGHT (i)
3. if  l ≤ A.heap-size-1  AND  A[l] > A[i]
4.     then  massimo = l
5. else
6.     massimo = i
7. if  r ≤ A.heap-size - 1  AND  A[r] > A[massimo]
8.     then  massimo = r
9. if  massimo != i
10.    then  SCAMBIA-CASELLE (A, i, massimo)
11.          MAX - HEPIFY (A, massimo)
```

OSS: complessità del MAX - HEAPIFY (A, i) è $O(\log n)$

10) BUILD - MAX - HEAP: // trasforma un array A in un Heap

```
BUILD-MAX-HEAP (A)
   A.heap-size = A.length
   for i = ⌊A.length/2⌋ - 1  down to 0     ▷ tutti i nodi interni
       do MAX - HEPIFY (A, i)
```

OSS: complessità del BUILD - MAX - HEAP(A) è $O(n \log n)$

EMPTY

```
EMPTY (A)
  A.heap-size = 0
```

NULL

```
NULL (A)
  return A.heap-size == 0
```

MAXIMUM

```
MAXIMUM (A)
  return A[0]
```

(EMPTY, NULL, MAXIMUM) COMPLESSITÀ $\Theta(1)$

ESCTRACT - MAX :

```
EXTRACT-MAX (A)
1. if NULL(A)
2.    then return error "heap underflow"
3. max = A[0]
4. A[0] = A[A.heap-size-1]
5. A.heap-size = A.heap-size -1
6. MAX-HEPIFY (A,0)
7. return max
```

oss: complessità dell EXTRACT-MAX(A) è O(log n)

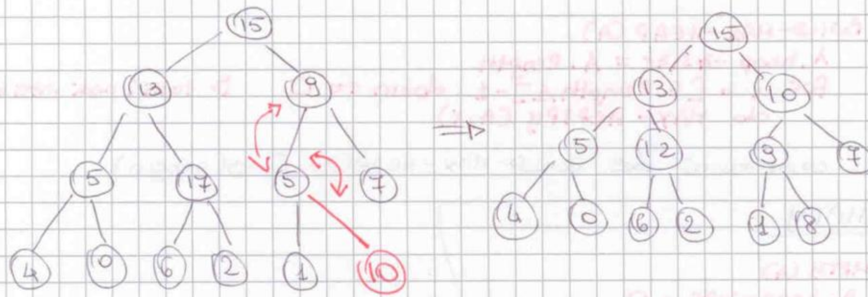
INSERT

```
INSERT (A, key)
1. A.heap-size = A.heap-size + 1
2. i = A.heap-size - 1
3. while   i>0   AND   A[PARENT(i)] < key
4.         do A[i] = A[PARENT(i)]
5.            i = PARENT(i)
6. A[i] = key
```

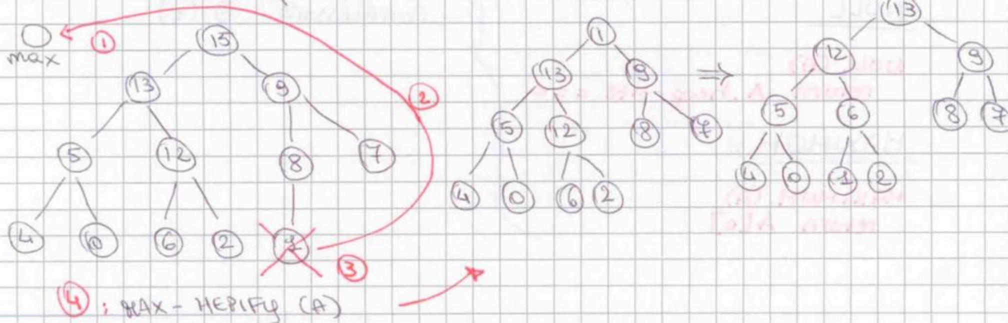
oss: O(log n)

ES: Illustra le operazioni di INSERT(A,10)

A = < 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1 >



ES: Illustra le operazioni di EXTRACT-MAX(A)



Algoritmi e Strutture Dati

15/

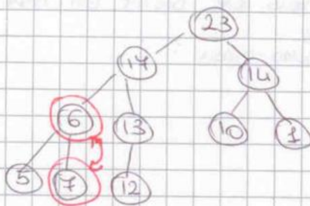
11) HEAP - SORT : procedura di ordinamento tramite heap

```
HEAP-SORT (A)
1. BUILD-MAX-HEAP(A)
2. for i = A.length - 1 to 1
3.     do SCAMBIA-CASELLE (A, 0, i)
4.        A.heap-size = A.heap-size - 1
5.        MAX-HEAPIFY (A)
```

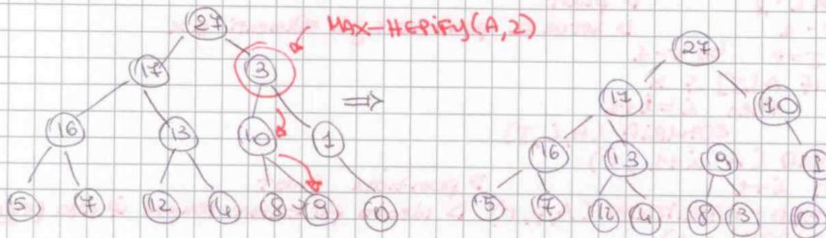
oss: complessità $O(n \log n)$
non è stabile (inverte le posizioni di num uguali) al contrario è in loco

- DOMANDE:

a) N° min e N° max di elementi in un heap di altezza h? n+1
N° max = N° nodi tot di un albero completo = $2^{h+1} - 1$
N° min = N° nodi interni + 1 = $2^h - 1 + 1 = 2^h$ → unico figlio sinistro
b) In un max-heap l'elemento più piccolo dove si trova? In una foglia
c) Un array ordinato in ordine inverso è un MAX-HEAP? SI
d) < 23, 17, 14, 6, 13, 10, 1, 5, 7, 12 > è un MAX-HEAP? NO



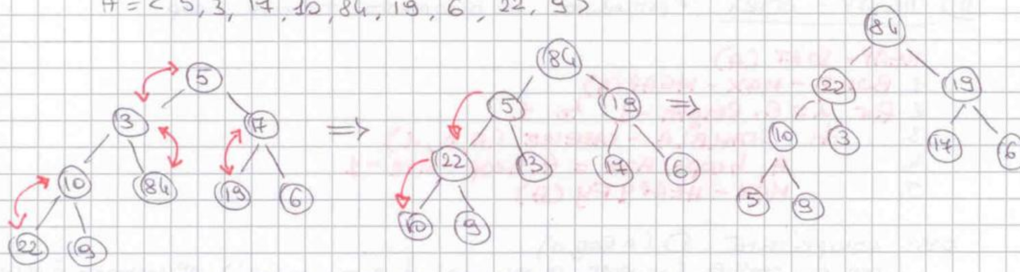
e) Qual è l'effetto di MAX-HEAPIFY (A, i) se l'elemento A[i] è il max? nessuno
f) Qual è l'effetto di MAX-HEAPIFY (A, i) se i > A.heap-size/2 - 1? nessuno



ES: ILLUSTRA HEAP-SORT(A)
A = < 5, 13, 2, 25, 7, 17, 20, 8, 4 > ⇒ A' = < 2, 4, 5, 7, 8, 13, 17, 20, 25 >

ES: ILLUSTRA BUILD-MAX-HEAP(A)

A = < 5, 3, 17, 10, 84, 19, 6, 22, 9 >



A' = < 84, 22, 19, 10, 3, 17, 6, 5, 9 >

QUICK SORT

Algoritmo di ordinamento in loco ma non stabile

- COMPLESSITÀ:
  - CASO PEGGIORE: $\Theta(n^2)$ → PIVOT è l'elemento max o min di A
  - CASO MEDIO & MIGLIORE: $\Theta(n \log n)$ → PIVOT mediano

OSS: n nel quick sort è un numero molto piccolo

- come nel MERGE-SORT questo algoritmo si basa sul DIVIDE ET IMPERA:
  1) DIVIDE: divide l'array in 2 sotto-array
  2) IMPERA: ordina in maniera ricorsiva i 2 sotto-array
  3) COMBINA: fusione

```
QUICK-SORT (A, p, r)        p → inizio, r → fine
1.  if p < r
2.     then q = PARTITION (A, p, r)
3.          QUICK-SORT (A, p, q-1)
4.          QUICK-SORT (A, q+1, r)
PARTITION (A, p, r)
5.  x = A[r]                 ▷ pivot
6.  i = p - 1                ▷ serve a posizionare gli elementi ≤ x
7.  for j = p  to r-1
8.      if A[j] ≤ x
9.         then i = i+1
10.             SCAMBIA (A, i, j)
11. SCAMBIA (A, i+1, r)
12. return i+1               ▷ posizione pivot
RANDOMIZED-PARTITION (A, p, r)  ▷ ideato per evitare sempre il caso peggiore
13. i = RANDOM(p, r)
14. SCAMBIA (A, r, i)
15. return PARTITION (A, p, r)
RANDOM-QUICK-SORT (A, p, r)
16. if p < r
17.    then q = RANDOMIZED-PARTITION (A, p, r)
18.         RANDOMIZED-QUICK-SORT (A, p, q-1)
19.         RANDOMIZED-QUICK-SORT (A, q+1, r)
```

OSS: complessità PARTITION $\Theta(n)$

Algoritmi e Strutture Dati

16/

- COMPLESSITÀ DEI PROBLEMI -

un algoritmo è un algoritmo che termina e produce un output accettabile
[∃ gli algoritmi corretti per un problema, ma alcuni sono + o - efficienti, l'efficienza è misurata con la complessità]

- $O(f(n))$ è il limite superiore (UPPER BOUND) - condizione sufficiente
- $\Omega(f(n))$ è il limite inferiore (LOWER BOUND) - condizione necessaria
- $\Theta(f(n))$ è il limite superiore & inferiore (CONVERGENZA) - condizione sufficiente e necessaria

OSS: esistono problemi con complessità ignota

Def: si dice "ALGORITMO DI ORDINAMENTO PER CONFRONTO" se il flusso delle operazioni dipende dal confronto di 2 elementi della sequenza

OSS: l'esecuzione di un algoritmo di ordinamento per confronto è analoga alla discesa di un albero immaginario di decisione

- tutte le possibili permutazioni : FOGLIE (n)
- n° confronti eseguiti nel caso peggiore: cammino più lungo tra 1 foglia e la radice
- con vari calcoli matematici svolti da STIRLING si può "dimostrare" la complessità di tali algoritmi ($\log_2 n$)

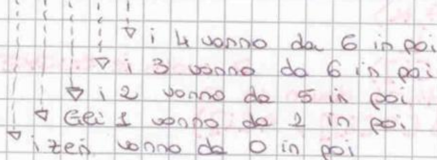
- ORDINAMENTO IN TEMPO LINEARE -

finora conoscevamo algoritmi di ordinamento $\Omega(n \log n)$ ma ∃ anche algoritmi lineari.
Questi algoritmi non funzionano sempre ma presuppongono dei vincoli sull'input

- COUNTING SORT
  - RESTRIZIONE: ogni valore dell'input deve essere $\leq K \in O(n)$
  - STRATEGIA: $\forall i \leq k$ conto quanti sono gli elementi $= i$; conto quanti sono gli elementi $< i$, scorro l'array e posiziono i

ES:

INPUT	4 1 4 0 2 0 1 1	$n = 8$ (n° elem) $K = 4$ (val. max)
i	0 1 2 3 4	SPAZIO CAMPIONE
elem. = ad i	2 3 1 0 2	CONTA QUANTI ELEM DI UN TIPO CI SONO IN INPUT
elem < ad i	0 2 5 6 6	CONTA QUANTI ELEM CI SONO PRIMA DI i



OUTPUT | 0 | 0 | 1 | 1 | 1 | 2 | 4 | 4 |

```
COUNTING-SORT (A, B, K)      ▷ A = input / B = output / K = val max
1. for i = 0 to K
2.     C[i] = 0              ▷ n° elementi = i
3.     D[i] = 0              ▷ n° elementi < i
4. for i = 0 to A.length - 1
5.     C[A[i]] = C[A[i]] + 1
6. for i = 1 to K
7.     D[i] = C[i] + D[i-1]
8. for i = 1 to K
9.     D[i] = C[i] + D[i-1]
10. for i = 0 to A.length - 1
11.    B[D[A[i]] = A[i]
12.    D[A[i]] = D[A[i]] + 1
```

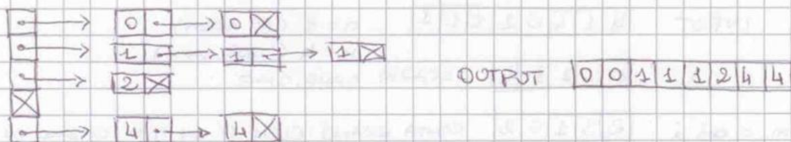
```
COUNTING-SORT-2 (A, B, K)
1. for i = 0 to K
2.     C[i] = 0
3. for i = 0 to A.length - 1
4.     C[A[i]] = C[A[i]] + 1      ▷ n° di elem. = i
5. for i = 0 to K
6.     C[i] = C[i] + C[i-1]       ▷ n° di elem < i
7. for i = A.length - 1 down to 0
8.     C[A[i]] = C[A[i]] - 1
9.     B[C[A[i]]] = A[i]
```

OSS: il COUNTING SORT è lineare, stabile ma non è in LOCO ⇒ non è ONLINE (occorre conoscere l'input)

- BUCKET - SORT:

RESTRIZIONE: ogni valore dell' input deve essere ≤ K ∈ O(n)

STRATEGIA: creo una lista ∀ i ≤ K. Scorro l'array e metto l'elemento corrente nella lista corrispondente. Infine svuoto le liste posizionando gli elementi nell' ARRAY OUTPUT




```
BUCKE-SORT (A, K)
1. for i = 0 to K
2.     EMPTY(B[i])          ▷ vuoto il BUCKET (B)
3. for i = A.length - 1 down to 0
4.     INSERT(B[A[i]], A[i])
5. i = 0
6. for J = 0 to K
7.     while not IS-EMPTY(B[J])
8.         A[i] = EXTRACT-FIRST(B[J])
9.         i = i + 1
```

complessità: $O(n)$ ma $O(n \log n)$ nel caso peggiore (tutti gli elementi sono in un solo bucket)

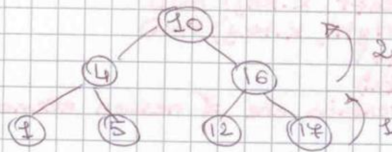
Algoritmi e Strutture Dati

~~Def~~ - ALBERI BINARI DI RICERCA:

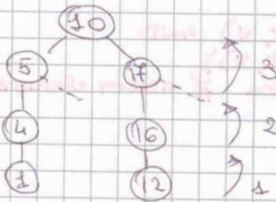
Def: si dice "ALBERO BINARIO DI RICERCA" (ABR) una struttura dati utilizzata per implementare un dizionario. È un albero radicato dove ogni nodo rappresenta un elemento, con i campi (PARENT, LEFT, RIGHT, KEY)

(!) tutti i nodi del sotto-albero sinistro di x hanno chiave ≤ a quella di x;
tutti i nodi del sotto-albero destro di x hanno key ≥ a quella di x;

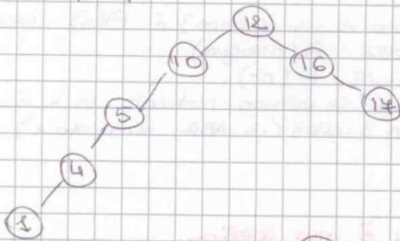
- ES: Disegna un ABR di $h=2$ su $A = \{1, 4, 5, 10, 16, 17, 12\}$



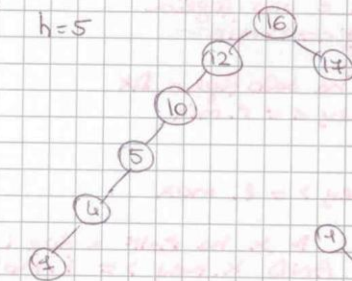
$h = 3$



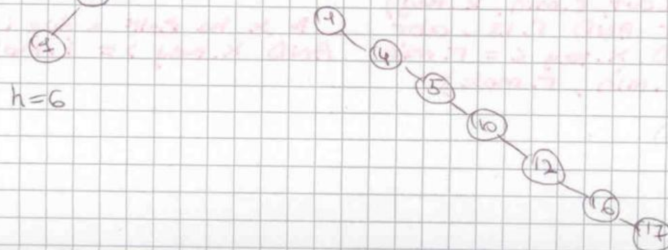
$h=4$



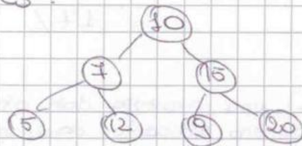
$h=5$



$h=6$



· ES:



è un ALBERO? NO 12 > 10 e 9 < 10

VERIFICA DI UN ALBERO

```
ABR-PRE(x)                      ▷ verifico in pre-ordine se l'albero radicato
  if x == NIL                     in x è un ABR
     then return TRUE
  else
     return (NO-MAGGIORE(x.left, x.key) AND
             NO-MINORE(x.right, x.key) AND
             ABR-PRE(x.left)
             ABR-PRE(x.right))
NO-MAGGIORE(x,v)     ▷ verifico che ∄ nessun elemento > v
  if x == NIL
     then return TRUE
  else
     return ((x.key <= v) AND
             NO-MAGGIORE(x.left, v) AND
             NO-MAGGIORE(x.right, v))
NO-MINORE(x,v)       ▷ verifico che ∄ nessun elemento < v
  if x == NIL
     then return TRUE
  else
     return ((x.key >= v) AND
             NO-MINORE(x.left, v) AND
             NO-MINORE(x.right, v))
```

OSS: - NO-MAGGIORE/NO-MINORE (esplorazione di un albero) è $\Theta(n)$ complex
complessità di ABR-PRE { CASO MIGLIORE: $\Theta(n \log n)$; CASO PEGGIORE: $\Theta(n^2)$ }

```
ABR-POST(x)   // verifico in POST-ORDINE se albero radicato in x è un albero
  if x == NIL      ritorna 1 oggetto con 3 valori (is_ABR, min, max)
     then return NIL
  l = ABR-POST(x.left)
  r = ABR-POST(x.right)
  if l == NIL AND r == NIL          ▷ x è una foglia
     then return (TRUE, x.key, x.key)
  if l == NIL AND r != NIL          ▷ x ha solo figlio DX
     then out = r.is_abr AND x.key <= r.min
          return (out, x.key, r.max)
  if l != NIL AND r == NIL
     then out = l.is_abr AND x.key >= l.max
          return (out, l.min, x.key)
  out = l.is_abr AND r.is_abr          ▷ x ha tutti e due i figli
  out = out AND x.key <= r.min AND x.key >= l.max
  return (out, l.min, r.max)
```

OSS: costo $\Theta(n)$

```
ABR-SYM(x)            ▷ verifica in visita simmetrica che
1. n = CONTA-NODI(x)     l'albero radicato sia un ABR
2. ▷ crea l'array con n posizioni
3. TREE-TO-ARRAY(A, x, 0)      ▷ Riversa l'ALB in A
4. return IS-SORTED(A)
```

OSS: complessità: $\Theta(n)$

```
CONTA-NODI(x)
1. if x == NIL
2.    then return 0
3. l = CONTA-NODI(x.left)
4. r = CONTA-NODI(x.right)
5. return l+r+1

IS-SORTED(A)
1. for i=0 to A.length-2
2.     if A[i+1] < A[i]
3.        then return FALSE
4. return TRUE
```

Complessità: $\Theta(n)$

```
TREE-TO-ARRAY(A, x, i)      ▷ uso a partire da i
  if x == NIL
     then return i
  i = TREE-TO-ARRAY(A, x.left, i)
  A[i] = x.key
  i = TREE-TO-ARRAY(A, x.right, i)
  return i
```

OSS: complessità: $\Theta(n)$

MAX & min IN UN ABR:

```
TREE-MINIMUM(x)
  while x.left != NIL
        x = x.left
  return x

TREE-MAXIMUM(x)
  while x.right != NIL
        x = x.right
  return x
```

RICERCA IN UN ABR:

```
ITERATIVE-TREE-SEARCH(x, k)
  while x != NIL AND k != x.key
        if k < x.key
           then x = x.left
        else
           x = x.right
  return x
```

```
RECURSIVE-TREE-SEARCH (x,K)
1. if x == NIL | K == x.key
2.    then return x
3. if K <= x.key
4.    then return RECURSIVE-TREE-SEARCH (x.left, K)
5. else
6.    return RECURSIVE-TREE-SEARCH (x.right, K)
```

oss: compleità - CASO PEGGIORE (albero sbilanciato): $O(n)$
- CASO MIGLIORE (albero bilanciato): $O(\log n)$

INSERIMENTO:

```
MK-TREE-ELEM (K)          ▷ crea un nodo con chiave K
1. ▷ x è un oggetto con campi (p, left, right, key)
2. x.p = x.left = x.right = NIL
3. x.key = K
4. return x
5.
TREE-INSERT (t, K)        ▷ t vuoto
1. if t.root == NIL
2.    then t.root = MK-TREE-ELEM (K)
3. else
4.    TREE-INS (t.root, MK-TREE-ELEM(K))
5.
TREE-INS (x, new)
1. if new.key < x.key
2.    then if x.left == NIL
3.           then x.left = new
4.                new.p = x
5.         else
6.              TREE-INS (x.left, new)
7. else
8.    if x.right == NIL
9.       then x.right = new
10.           new.p = x
11.   else
12.      TREE-INS (x.right, new)
```

oss: complessità $O(h)$

CANCELLAZIONE:

```
TREE-DELETE (t,x)
1. if x.left != NIL AND x.right != NIL
2.     then y = TREE-MINIMUM (x.right)
3.          x.key = y.key
4. else
5.     y = x
6. TREE-BYPASS (t,y)
```

complessità: $\Theta(h)$

```
TREE-BYPASS (t,x)                 ▷ x ha al max un figlio
1. if x.left != NIL
2.    then figlio = x.left
3. else
4.    figlio = x.right            ▷ NIL se non ho figli
5. if figlio != NIL
6.    then figlio.p = x.p
7. if x.p != NIL
8.    then if x == x.p.left       ▷ x era figlio sinistro
9.            then x.p.left = figlio
10.        else
11.           x.p.right = figlio
12. ▷ dealloca x se il linguaggio lo prevede
```

complessità: $O(1)$

HASHING

Def: si dice "DIZIONARIO" un insieme di elementi legati da una chiave, alla quale corrisponde un oggetto, questa chiave è unica nel dizionario

→ nel dizionario si può inserire un nuovo elemento, rimuovere un elem, o cercare

→ nel dizionario il n° di chiavi utilizzate << n° delle chiavi possibili (universo)

Come si può creare un dizionario?

1) TABELLE AD INDIRIZZAMENTO DIRETTO
uso un array T (tabella) con tante posizioni m quanti sono gli elementi di U (universo)
oss: se $U >> n \Rightarrow$ c'è uno spreco e una troppo elevata complessità spaziale

2) TABELLE HASH:
uso un array T e una funzione h (→ hashing) : $m << U$ & $m \simeq n$ (→ dim array)

h: definisce una corrispondenza tra U e gli indici di T. Queste f(x) generano lo stesso risultato... come lo risolviamo?

→ TABELLE HASH CON LISTE DI TRABOCCO: ogni elemento ∈ T è una lista in cui vengono archiviati tutti gli elementi che tramite h danno come risultato il medesimo indice

- OPERAZIONI BASE:
  I) CHAINED-HASH-INSERT (T, x): inserisco x in testa alla lista in T[h(x.key)]
  II) CHAINED-HASH-DELETE (T, x): cancello x da T[h(x.key)]
  III) CHAINED-HASH-SEARCH (T, K): restituisce il puntatore all'elemento con chiave K nella lista in T[h(K)]

complessità di ricerca:
- CASO MIGLIORE: tutte le K utilizzate corrispondono a una posizione ≠ ($\Theta(1)$)
- CASO PEGGIORE: tutte le K utilizzate corrispondono alla stessa posizione ($\Theta(n)$)

Def: si dice "FATTORE DI CARICO" $\alpha = \frac{n}{m}$ → n° elem memorizzati / n° posiz. disponibili

- $\alpha < 1$ ⇒ T sottoutilizzato
- $\alpha = 1$ ⇒ T piena
- $\alpha > 1$ ⇒ T sovrautilizzato (qui avrò sicuramente almeno una lista di trabocco)

h: oltre ad essere deterministica dovrebbe distribuire le chiavi in maniera pseudo casuale tra [0; m-1] in maniera uniforme
→ funzione di hash

∃ ≠ tipi di h:

a) METODO DELLA DIVISIONE: $h(K) = K \bmod m$
- questo metodo m (valore critico) non deve essere una potenza di 2 o 10; è preferibile essere un n° primo

b) METODO DELLA MOLTIPLICAZIONE: $h(K) = \lfloor m(KA - \lfloor KA \rfloor) \rfloor$
- A = cost ∈ [0,1] AD ES: KNUTH propone $A = \frac{\sqrt{5}-1}{2} = 0{,}6180$
  qui non è più critico ⇒ si usa solitamente 1,2

• TABELLE HASH CON INDIRIZZAMENTO APERTO - in questo modo non uso le liste ma se la posizione è occupata ne calcolo un'altra finché ne trovo una libera.
Così facendo $\alpha \in [0,1]$ e posso utilizzare meno memoria. ∃ più modi per gestire una collisione. Il modo più semplice è andare nella casella successiva
  oss: l'array è ciclico
- Se andassimo alla casella successiva senza nessun legame con il valore, non riuscirei più a capire da quale elem ∈ U proviene ⇒ ho bisogno di una funzione di scansione

$h(K, i)$: funzione di scansione

```
HASH-INSERT (T, K)
1.  i = 0                          ▷ I° tentativo
2.  repeat  J = h(K, i)
3.          if T[J] == NIL
4.            then T[J] = K
5.                   return J
6.          else
7.                 i = i+1
8.  until i == m
9.  error "OVERFLOW SULLA TABELLA HASH"
```

```
HASH-SEARCH(T,K)
1. i = 0
2. repeat   J = h(K,i)
3.          if T[J] == K
4.             then return J
5.          else
6.             i = i+1
7. until T[J] == NIL | i == m
8. return NIL
```

→ funzione di scansione

∃ ≠ tipi di h : funzione di SCANSIONE

a) SCANSIONE LINEARE: $h(K,i) = (h(K) + i) \bmod m$
oss: purtroppo questo algoritmo produce un eccessivo addensamento primario

b) SCANSIONE QUADRATICA: $h(K,i) = (h(K) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$
$c_1, c_2 = cost \neq 0$ (valori ottimi come ci scelgo? grazie AVADI, M, h $c_1 = c_2 = 1/2$)

oss:

se $c_1 = c_2 = 1/2 \Rightarrow h(K,0) = h(K)$
$h(K,1) = h(K,0) + 1$
$h(K,2) = h(K,1) + 2$
$h(K,3) = h(K,2) + 3$

(!) le scansioni lineari & quadratiche sono affette dal problema dell'addensamento secondario: se $h(K_1) = h(K_2) \Rightarrow h(K_1, i) = h(K_2, i)$
⇒ non troverei mai posto

c) DOPPIO HASHING: $h(K,i) = (h(K) + i \cdot h_2(K)) \bmod m$
$h_1(K)$ e $h_2(K)$: 2 funzioni ma sono ausiliarie

GRAFI

Def: si dice "GRAFO ORIENTATO" (DIRETTO), $G(V,E)$, costituito da insiemi, nodi ed archi
oss: ogni arco è una coppia ordinata di nodi
$\begin{cases} n = |V| \ (n° \text{ nodi}) \\ m = |E| \ (n° \text{ archi}) \end{cases}$

Def: Dato un dat nodo u:
"ARCO USCENTE" è l'arco $(u,v) \in E$
"ARCO ENTRANTE" è l'arco $(v,u) \in E$
"NODO ADIACENTE" è il nodo V per cui ∃ l'arco $(u,v) \in E$
"GRADO DI USCITA" è il n° di archi uscenti
"GRADO DI INGRESSO" è il n° di archi entranti
"SORGENTE" è un nodo che non ha archi entranti
"POZZO" è un nodo che non ha archi uscenti
"CAMMINO" è una sequenza di nodi (è detto semplice se tutti i nodi sono distinti)
"LUNGHEZZA" è il n° archi di un cammino
"CICLO" è un cammino non semplice in cui il I° e l'ultimo nodo coincidono
(il ciclo è detto semplice se gli unici nodi che coincidono sono il primo e l'ultimo)
"CAPPIO" (LOOP) è un ciclo di un solo nodo e un solo arco
"GRAFO SEMPLICE" grafo senza cappi

GRAFO "~~ACICLIO~~ ACICLICO" è un grafo senza cicli

<u>Def</u>: si dice "GRAFO NON ORIENTATO" (INDIRETTO) un grafo in cui $\forall (u,v)\ \exists (v,u)$

<u>Def</u>: si dice "MATRICE DI ADIACENZA" una matrice quadrata $n \times n$ in cui ogni posizione è 0 o 1. 1 è nella posizione in cui l'arco riga x colonna ∃!

OSS: così si può accedere direttamente a tutti gli archi però occupa $\Theta(n^2)$

<u>Def</u>: si dice "LISTA DI ADIACENZA" un array di liste, in cui gli elementi delle liste sono le chiavi degli oggetti puntati dall' oggetto i (indice dell' array)

OSS: così ~~si può accedere direttamente a tutti gli archi~~ per vedere se ∃ un arco mi devo scorrere l'array ma occupa - spazio $(O(n) + O(m))$ e nel caso peggiore è $O(n^2)$) - GRAFO DENSO : $m \simeq n^2$

<u>Def</u>: si dice "GRAFO PESATO" un grafo in cui ad ogni arco è associato un peso (usati ad esempio in una mappa stradale in cui il peso può essere la distanza)
→ MATRICE ADIACENZA

```
LISTE (A)    ▷ Converte la matrice in un array di liste
1. ▷ B è un array di liste lungo A.length
2. for i=0  to A.length-1
3.     for J=0 to A.length-1
4.         if A[i][J] != 0
5.           then INSERISCI (B, i, J)
6. return B
```

```
INSERISCI (B, i, J)
1. ▷ x è un nodo nuovo
2. x.info = J
3. x.prev = NIL
4. x.next = B[i]
5. if B[i] != NIL
6.   then B[i].prev = x
7. B[i] = x
```

```
GRAFO-SEMPLICE (A)    ▷ verifica che non ho cappi
1. for i=0 to A.length-1
2.     if A[i][i] == 1
3.       then return FALSE
4. return TRUE
```

```
VERIFICA-ARCO (A, u, v)    ▷ verifico se ∃ l'arco (u,v) per liste di adiacenza
1. x = A[u]
2. while (x != NIL
3.     if x.info == v
4.       then return TRUE
5.     x = x.next
6. return FALSE
```

```
VERIFICA-NON-ORIENTATO (A)
1. for i=0 to A.length-1
2.     x = A[i]
3.     while x != NIL
4.         if !VERIFICA-ARCO (A, x.info, i)     ▷ da rivedere non è giusto
5.             then return FALSE
6.         x = x.next
7. Return TRUE
```

```
VERIFICA-POZZO (A, u)
1. return A[u] == NIL
```

```
VERIFICA-SORGENTE (A, u)
1. for i=0 to A.length-1
2.     x = A[i]
3.     while x != NIL
4.         if x.info == u
5.             then return FALSE
6.         x = x.next
7. return TRUE
```

```
VERIFICA-UNIONE (A1, A2)   ▷ verifico che ∀ arco questo ∃ in almeno 1 dei 2
1. for i=0 to A.length-1
2.     for j=0 to A.length-1
3.         if !verifica-ARCO (A1, i, j) AND !VERIFICA-ARCO(A2, i, j)
4.             then return FALSE
5. return TRUE
```

VISITE DI GRAFI

Def: un nodo v si dice "RAGGIUNGIBILE" da un nodo u se ∃ cammino diretto da u a v. GRAFO "FORTEMENTE CONNESSO": ogni coppia di nodi u, v sono raggiungibili
↳ per grafi orientati

Def: un nodo v si dice "RAGGIUNGIBILE" da un nodo u se ∃ un cammino da u a v. GRAFO "CONNESSO": se ∀ coppia u, v ∃ un cammino da u a v
↳ per grafi non orientati

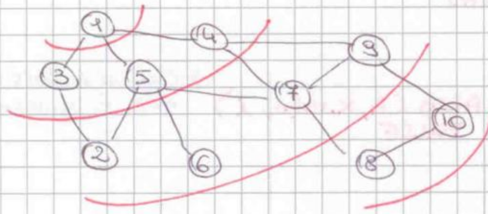
OSS: lo scopo degli algoritmi di visita di grafo è quello di visitare tutti i nodi raggiungibili da un nodo

Def: si dice "MARCATORE" un valore associato ad un determinato nodo per segnare che quel nodo è stato raggiunto o meno (BOOLEANO / COLORE)

```
▷ "color" è UN ARRAY di int con n pos
for i=0 to color.length-1
    color[i]=0                ▷ inizializzo l'array con zero
...
    color[6]=1                ▷ coloro il nodo di indice 6
```

∃ ≠ tipi di visite

1) VISITA IN AMPIEZZA (BREADTH-FIRST SEARCH)



questa visita parte da un nodo v e visita i nodi raggiungibili da v (prima i più vicini e poi i più lontani)

○ = MAI RAGGIUNTO
◐ = RAGGIUNTO
● = ESPLORATO

- STRATEGIA: facciamo uso di una coda, in cui ci inseriamo v.
i nodi messi in coda li coloriamo di grigio.
inseriamo i nodi raggiunti
finché la coda non è vuota, estrai un nodo dalla coda, lo coloriamo di nero,
consideriamo tutti i suoi adiacenti e se sono raggiunti per la I^a volta li coloriamo di grigio e li mettiamo in coda

```
BFS (A,v)
1.  for i=0 to A.length-1
2.      color [i] =0                    ▷ 0=BIANCO=NON RAGGIUNTO
3.  QUEUE-EMPTY (q)                     ▷ creo una coda vuota
4.  color [v] =1                        ▷ 1=GRIGIO = RAGGIUNTO
5.  ENQUEUE (q,v)
6.  while ! QUEUE-VOID (q)              ▷ finché la coda non è vuota
7.        i= DEQUEUE (q)                ▷ estraggo un indice dalla coda
8.        x= A [u]
9.        while x!=NIL
10.             k = x.key
11.             if color [k] ==0
12.                then color [k] =1
13.                     ENQUEUE (q,K)
14.             x. x.next
15.       color[u]=2                    ▷ 2=NERO = ESPLORATO
```

2) VISITA IN PROFONDITÀ (DEPTH-FIRST SEARCH)

// a partire da v si percorre il grafo fino a che si trovano nodi non visitati. Quando tutti i vicini sono visitati si torna indietro verso v perché ∄ nodi vicini non visitati

```
DFS (A,v) - GRAFO CONNESSO
1. for i=0 to A.length-1
2.       color (i] =0                   ▷ BIANCO =0 = non raggiunto
3. DFS-VISIT (A,v, color)
DFS-VISIT (A,i, color)
1. color [i] =1                         ▷ GRIGIO= 1 = RAGGIUNTO
2. x= A[i]
3. while x!=NIL
4.       v= x.key
5.       if color [v] ==0
6.          then DFS-visit (A,v, color)  ▷ continua a visitare da v
7.       x = x.next
8. color [i] =2                         ▷ 2 = NERO = ESPLORATO
DFS (A) - GRAFO NON CONNESSO
1. for i=0 to A.length -1
2.      color [i] = 0
3. for i=0 to A.length -1
4.    if color [i] ==0
5.       then DFS- VISIT( A,i, color)
```

Algoritmi e Strutture Dati

ALGORITMI - GRAFI

22/

→ INSERT (A, u, v)
```
1. ▷ x è un nuovo nodo
2. x.info = v
3. x.prev = NIL
4. x.next = A[u]
5. if A[u] != NIL
6.     then A[u].prev = x
7. A[u] = x
```

→ VERIFICA-ARCO (A, u, v)
```
1 x = A[u]
2 while x != NIL
3.     if x.info == v
4.         then return TRUE
5.     x = x.next
6. return FALSE
```

→ VERIFICA-NON-ORIENTATO (A)
```
1. for i=0 to A.length -1
2.     x = A[i]
3.     while x != NIL
4.         if !VERIFICA-ARCO (A, x.info, i)
5.             then return FALSE
6.         x = x.next
7. return TRUE
```

→ VERIFICA-POZZO (A, u)
```
1. return A[u] == NIL
```

→ VERIFICA-SORGENTE (A, u)
```
1. for i=0 to A.length -1
2.     x = A[i]
3.     while x != NIL
4.         if x.info == u
5.             then return FALSE
6.         x = x.next
7. return TRUE
```

→ VERIFICA-UNIONE (A1, A2)
```
1 for i=0 to A.length -1
2.     for j=0 to A.lenght -1
3.         if !VERIFICA-ARCO (A1, i, j) AND !VERIFICA-ARCO (A2, i, j)
4.             then return FALSE
5. return TRUE
```

→ DFS (A, u) - GRAFO CONNESSO
```
1. for i=0 to A.length -1
2.     color[i] = 0
3. DFS-VISIT( A, u, color)
```

```
→ DFS-VISIT (A, u, color)
1. color[u] = 1
2. x = A[u]
3. while x != NIL
4.       v = x.key
5.       if color[v] == 0
6.          then DFS-VISIT (A, v, color)
7.       x = x.next
8. color[u] = 2

→ DFS (A, u) - GRAFO NON CONNESSO
1. for i=0 to A.length-1
2.      color[i] = 0
3. for i=0 to A.length-1
4.      if color[i] == 0
5.         then DFS-VISIT (A, i, color)

→ AGGIUNGI-ARCHI-OPPOSTI (A, u)
1. x = A[u]
2. while x != NIL
3.       if !VERIFICA-ARCO (A, x.info, u)
4.          then INSERT (A, x.info, u)
5.       x = x.next

→ GRAFO-INDIRETTO (A)
1. for i=0 to A.length-1
2.      AGGIUNGI-ARCHI-OPPOSTI (A, i)

→ CONTA-RAGGIUNTI (A, u)
1. for i=0 to A.length-1
2.      color[i] = 0
3. DFS-VISIT (A, u, color)
4. n = 0
5. for i=0 to A.length-1
6.      if color[i] == 2
7.         then n++
8. return n

→ VERIFICA-MAGGIORI (A, u)
1. x = A[u]
2. while x != NIL
3.       if x.info <= u
4.          then return FALSE
5.       x = x.next
6. return TRUE

→ VERIFICA-TUTTI-MAGGIORI (A)
1. for i=0 to A.length-1
2.      if !VERIFICA-MAGGIORI (A, i)
3.         then return FALSE
4. return TRUE

→ RIMUOVI-ARCHI-ENTRANTI (A, u)
1. for i=0 to A.length-1
2.      if VERIFICA-ARCO (A, i, u)
3.         then RIMUOVI (A, i, u)
```

Algoritmi e Strutture Dati

23)

→ RIMUOVI (A,u,v) ▷ ARCO u→v ∃

```
1. x = A[u]
2. if x.info == v
3.    then if x.next == NIL
4.            then A[u] = NIL
5.         else
6.            A[u] = x.next
7.            x.next.prev = NIL
8. else
9.    while x.next != NIL
10.         if x.info == v
11.            then x.prev.next = x.next
12.                 x.next.prev = x.prev
13.         x = x.next
14. if x.info == v
15.    then x.prev.next = NIL
```

→ CONNESSO (A,u)

```
1. for i=0 to A.length-1
2.     color[i] = 0
3. DFS-VISIT (A,u, color)
4. for i=0 to A.length-1
5.     if color[i] != 2
6.        then return FALSE
7. return TRUE
```

→ VERIFICA-NO-ARCHI (A)

```
1. for i=0 to A.length-1
2.     if A[i] != NIL
3.        then return FALSE
4. return TRUE
```

→ VERIFICA-NODO-ISOLATO (A,u)

```
1. return VERIFICA-POZZO (A,u) AND VERIFICA-SORGENTE (A,u)
```

→ INVERTI-ARCHI (A)

```
1. ▷ B è un nuovo array di liste doppiamente concatenate con dim = A.length-1
2. for i=0 to A.length-1
3.     x = A[i]
4.     while x != NIL
5.         INSERT (B, x.info, i)
6.         x = x.next
7. return B
```

→ ESISTE-NODO-ISOLATO (A)

```
1. for i=0 to A.length-1
2.     if VERIFICA-NODO-ISOLATO (A,i)
3.        then return TRUE
4. return FALSE
```

→ VERIFICA-ARCO-USCENTE (A,u) ▷ verifico che u ho un solo arco uscente

```
1. x = A[u]
2. if x != NIL
3.    then if x.next == NIL
4.            then return TRUE
5. return FALSE
```

```
→ VERIFICA-ARCO-ENTRANTE (A,v)    ▷ verifico che v ha un solo arco entran
1. c=0          ▷ contatore di archi entranti in v
2. for i=0 to A.lenght-1
3.     x = A[i]
4.     while x != NIL
5.         if x.info == v
6.             then c++
7.         x = x.next
8. if c==1
9.     then return TRUE
10. return FALSE

→ VERIFICA (A)    ▷ ver. che ogni nodo ha un solo arco entrante e un solo arco uscente
1. for i=0 to A.length-1
2.     if !(VERIFICA-ARCO-ENTRANTE (A,i) AND VERIFICA-ARCO-USCENTE(A,i))
3.         then return FALSE
4. return TRUE
```

Algoritmi e Strutture Dati

24/

INTRODUZIONE AL LINGUAGGIO C

- LINGUAGGI MACCHINA, ASSEMBLEY, E DI ALTO LIVELLO
  - LINGUAGGI MACCHINA: linguaggio del computer (codice binario)
  - LINGUAGGIO ASSEMBLEY: abbreviazioni simili all'inglese per rappresentare operaz. elementari del PC
  - LINGUAGGI AD ALTO LIVELLO: singole istruzioni contenenti notazioni matematiche utilizzate comunemente
- PARADIGMI DI PROGRAMMAZIONE:
  - PROGRAMMAZIONE IMPERATIVA: nel programma le istruz. devono essere eseguite in sequenza
  - PROGRAMMAZIONE ORIENTATA AGLI OGGETTI: il programma modella una realtà di interesse come una collezione di oggetti software che cooperano

  OSS: C è un linguaggio di programmazione imperativo un po ibrido con gli oggetti

- STEPS DI PROGRAMMAZIONE:
  1) PROBLEMA: viene posto un problema e bisogna risolverlo
  2) ALGORITMO: bisogno trovare un modo per risolvere il problema e trasm. risultat
  3) PROGRAMMA: traduce l'algoritmo risolvendo il problema
- COME È FATTA LA MEMORIA:
  - la memoria è rappresentabile da una griglia di celle di memoria, ognuna delle quali può memorizzare un certo valore. Esso può essere acceduto dal suo indirizzo

  OSS: quando creiamo un oggetto e questo non viene più utilizzato nel programma, in JAVA il garbage collector elimina l'oggetto, liberando la cella di memoria occupata. Purtroppo in C non è presente il garbage, saremo quindi noi a dover eliminare esplicitamente l'oggetto inutilizzat
- STRUMENTI PER LA PROGRAMMAZIONE
  1) EDITING: rendere il programma accessibile al calcolatore
  2) COMPILAZIONE: traduce il programma, in un formato eseguibile dal calcolo
  3) ESECUZIONE: far eseguire il programma al calcolatore
- LIBRERIE C: i programmi scritti in C, consistono in moduli chiamati funzioni. Esiste già una collezione di funzioni chiamata "LIBRERIA STANDARD DEL C"
  Perché utilizzare le librerie?
  - RIUSABILITÀ: inutile ricostruire funzioni già esistenti
  - EFFICIENZA: il programma funzionerà meglio
  - PORTABILITÀ: posso usare queste funzioni, indipendentemente dalla piattaforma
- LINGUAGGIO C: è un linguaggio di programmazione strutturato, che permette di scrivere programmi molto compatti e in più permette di accedere direttamente alle risorse dell' ~~HARDA~~ HARDWARE
- AMBIENTE C:
  1) EDITOR: programma creato con l'editor
  2) PREPROCESSORE: preprocessore esegue il codice
  3) COMPILATORE: crea il codice oggetto
  4) LINKER: collega il codice oggetto alle librerie
  5) LOADER: carica in memoria il programma
  6) CPU: esegue un istruzione alla volta
- PROGRAMMAZIONE

- PROGRAMMAZIONE
  - DIRETTIVA DI COMPILAZIONE: include info relative ad una libreria predefinita del C

  # include <stdio.h> contiene funzioni di INPUT/OUTPUT

  main(): identifica il programma principale ed è il punto di ingresso del programma all'inizio dell'esecuzione

  /* ... */: commenti classici

  # define <nome><valore>: va messo all'intestazione del programma per definire costanti

  {...}: delimitano un blocco

(!) le variabili vengono dichiarate come in JAVA

- TIPI DI DATO ELEMENTARI:
  1) Char
  2) Int
  3) Short
  4) Long
  5) Float
  6) Double

(!) ∄ il tipo boolean ma si usa la convenzione che 0 = false e 1 = true (oppure tutti valori diversi da 0)

printf ("<stringa>", <elenco argomenti>); Output Stampa

scanf ("<Stringa>", <elenco argomenti>); Input Lettura

ES:
```
printf ("Risultato = %d \n", fat);
scanf ("%d", & n);
```

% indica il punto in cui vanno sostituiti gli elementi che seguono

il carattere che segue %, indica il tipo dell'argomento

& precede sempre gli argomenti input ed output

- %d Int
- %lf Double
- %l Long
- %f Float
- %c char
- %s Stringhe (anche se la stringa non è un tipo)

<elenco argomenti>: dove vengono elencate le variabili da STAMPARE o inserire

= simbolo utilizzato per l'assegnazione

Operazioni tra interi: *, +, -, /, %
Operazioni tra reali: *, +, -, /

FORME CONTRATTE:
- x++ = x+1
- x-- = x-1
- a+=b ⇒ a = a+b
- a*=b ⇒ a = a*b

(!) La compilazione avviene da destra verso sinistra. Molti compilatori (soprattutto su Windows) compilano da sinistra verso destra

Algoritmi e Strutture Dati

- LINGUAGGIO C -

• ISTRUZIONI CONDIZIONALI:

1) if (<espr>) <istr 1>
   else <istr2>

2) switch (<espr>) {
      case <costante 1> : <istr1> [break]
      case <costante 2> : <istr2> [break]
      ...
      default: <istr> }

Valuta è espressione e il valore dell'espressione (val. di tipo elemento)
Confronta il valore con le costanti (val. di tipo elemento)
Quando trovo che <espr> == <costante i> => entro nel ciclo
Se incontra il Break esce dal ciclo
Se nessuna costante è uguale al valore => entra in default e continua l'esecuzione

break: comando per uscire dall'esecuzione

ES: (*int in questo caso)

```
switch (mese) {
   case 2: n=28; break
   case 4:
   case 6:
   case 9:
   case 11: n=30; break;
   default: n=31
```

• ISTRUZIONI ITERATIVE:

1) while (<espr>)
      <istr>

2) do <istr>
   while (<espr>)

3) for (<init>; <test>; <incr>)
      <istr>

(!) All'interno del for il contatore nell'espressione "inizializzazione" non può essere contemporaneamente anche dichiarato. Va dichiarato prima

- FUNZIONI IN C -

Def: si dice "funzione" un blocco di istruzioni che ha parametri in ingresso (PARAMETRI FORMALI) e restituisce un risultato

```
<tipo> <Nome Funzione> (lista parametri) {
    ...
    blocco istruzioni
    ...
    return <risultato>
}
```

Lista parametri: (<tipo> <nome>, <tipo> <nome>, ...)

oss: nella chiamata i parametri passati corrispondono in ordine ai parametri formali

(!) Non è possibile annidare funzioni:

~~int Fatt (int n)~~
~~int Fatt2 (int n2)~~

(!) Non è possibile passare parametri riferimento

~~int Fatt (int & n)~~

- DEFINIZIONE E IMPLEMENTAZIONE:

DEFINIZIONE: specifica la struttura della funzione (NOME / ORDINE E TIPO DEI PAR) la definizione di una f(x) è detta in C "prototipo" (o HEADER)

Ex:
```
int Fatt (int);
float radice(int);
int somma (int, int);
```

IMPLEMENTAZIONE: specifica completa della funzione

Ex:
```
int Somma(int a, int b) {
    int som;
    som = a+b;
    return som; }
```

oss: con scanf non mettere mai \n => va a capo da solo

- ORGANIZZAZIONE DEI PROGRAMMI C:

- La programmazione C è strutturata => un programma è distribuito su più file
- Per includere un file in un altro si usa

include <...> : per file nel direttorio di sistema (librerie standard)

oppure # include "..." : per file nel direttorio corrente

Algoritmi e Strutture Dati

- LINGUAGGIO C -

→ ORGANIZZAZIONE DEI PROGRAMMI C:

le definizioni vengono poste in file detti HEADER con estensione **.h**
le implementazioni vengono poste in file con estensione **.c**

per cucire programmi che contengono dati oppure f(x) altrove, si includono nel file che contiene P le loro definizioni e si rendono così visibili a P => può compilare separatamente

↳ il LINKER ha il compito di associare le definizioni alle invocazioni

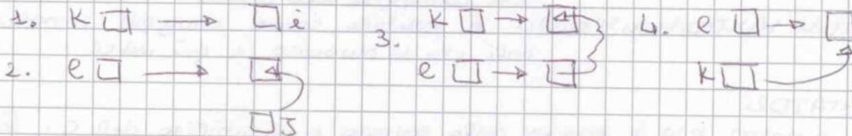
→ PUNTATORI:

- permettono la gestione di strutture dinamiche
- un puntatore è una variabile che contiene l'indirizzo di memoria di un'altra variabile
- Dichiarazione: TIPO (della variabile puntata) * NOME_VAR

  ES: `int *i;`
- L'operatore **&** applicato ad una variabile restituisce il puntatore ad essa
- L'operatore ***** applicato ad un puntatore restituisce la variabile puntata

ES:

```
int i, j;
int *k, *l;
....
1. k = &i    // k punta ad i
2. *l = j    // nella variabile puntata da l va il contenuto di j
3. *k = *l   // nella var. puntata da k va il contenuto della variabile
             //    puntata da l
4. k = l     // k e l puntano la stessa variabile
```



- PROCEDURE: funzioni che restituiscono il tipo void
- PASSAGGI PER RIFERIMENTO: non possiamo modificare ma il suo puntatore

ES:

```
void SommaProd (int i, int j, int *s, int *p) {
    *s = i+j;
    *p = i*j; }
```

- GESTIONE DELLA MEMORIA: la memoria in C viene gestita in 2 modi:
  1) STATICA: viene allocata dal sistema operativo un'area di memoria fissa per tutta l'esecuzione
  2) DINAMICA: vengono allocate 2 aree di memoria
     I) PILA (o STACK): dove le variabili e i parametri locali vengono allocati ~~dente~~ durante l'invocazione di una f(x), ad ogni f(x) corrisponde un record di ATTIVAZIONE (contenente le variabili locali e i parametri locali della f(x)) Quando f(x) termina => RDA viene rimosso ~~e~~ e ~~locali~~ cancellato dallo STACK (gestito dall'elaboratore)
     II) HEAP: la gestione è lasciata al programmatore mediante la creazione e distruzione dinamica di variabili (tramite puntatori) => gestito dall'utente

- RICORSIONE: è possibile definire in C f(x) ricorsive

ES:
```
int Fatt (int n) {
    if (n==0) return 1;
    else return (n* Fatt(n-1)); }
```

ARRAY E PUNTATORI:

Def: si dice "array" una sequenza di elementi omogenei

DICHIARAZIONE: <TIPO DELL'ARRAY> <NOME DELL'ARRAY> [<DIMENSIONE ARRAY>];

ES: `int c[12]; // vettore di dimensione 12`

Def: si dice "vettore" un gruppo di locazioni di memoria correlate dal fatto che hanno tutte lo stesso nome e tipo dato

- il I° elemento ∀ vettore è l'elemento 0
- il n° all'interno delle parentesi (non quelle della dichiarazione) è detto "INDICE"

<nome dell'array> [indice]

ALTRI MODI PIÙ RAPIDI PER DICHIARARE UN VETTORE

```
int vet[5] = {1,2,3,4,5};
int vett[] = {1,2,3,4};   // x. un vett. lo specifico delle dimensioni in
```

↳ è possibile passare come parametro formale di una f(x) un array — questo caso è opzionale

- UTILIZZO DEGLI ARRAY
  - ORDINAMENTO DEI DATI: Insertion Sort, Quick Sort, Merge Sort, Bubble Sort
  - RICERCA NEI VETTORI: se un vettore contiene o no almeno un elemento che corrisponde alla chiave
- VETTORI MULTIDIMENSIONALI: è possibile creare array più complessi multidimensionali che si muovono a più indici

- PUNTATORI:
  - I puntatori sono il segreto della potenza e flessibilità del C, sono l'unico modo per effettuare alcune operazioni → sono la parte più complessa del linguaggio
  - In C ogni variabile è caratterizzata da 2 valori
    - indirizzo di locazione di memoria (che contiene la variabile)
    - valore contenuto
  - Il puntatore è una variabile che contiene l'indirizzo di memoria di un altra variabile

ES:
```
int a = 2, y;
int *pa;

pa = &a
```

`y = *pa // assegna ad y il contenuto della locazione di memoria a cui punta pa`

`*pa = 4 // sostituisce il valore di a con 4`

Algoritmi e Strutture Dati

- LINGUAGGIO C -

ES:

n* n *n // elevo al cubo n

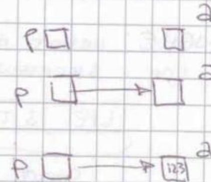
*pa = *pa * *pa* *pa // elevo al cubo la variabile puntata

(!) è molto facile fare confusione fra oggetti puntati e puntatori!

ES: int *p; *p = 123 NO!!

int *p;
int a;

p = &a;
*p = 123; SI!



- ARITMETICA DEGLI INDIRIZZI:

- Incrementare un puntatore significa spostarsi di tot Bit da un indirizzo di memoria a un altro. Però le operazioni aritmetiche sui puntatori dipendono dal tipo di variabile puntata
- Gli array sono strettamente connessi con i puntatori ..
Gli elementi dell'array vengono allocati in locazioni contigue della memoria principale. Incrementare di uno p => incrementare di una quantità di Byte pari alla dim di T (tipo della variabile puntata)

OSS: il nome dell'array coincide con l'indirizzo della I^a componente del vettore. Il puntatore ad un elemento dell'array si ottiene incrementando di uno il puntatore all'elemento precedente

quindi ..

int a[5];
a = &a[0];
*a = a[0];
*(a+3) = a[3]
(a+2) = & a[2]

OSS: in C il nome di un array è trattato dal compilatore come un puntatore costante alla I^a locazione di memoria dell'array stesso. Ma a differenza dei vettori l'area di memoria a cui il puntatore punta non è allocata staticamente ma dinamicamente

ES: char s[10]
char *p;

p = s;

→ s[9] = *(s+9) = p[9] = *(p+9)   Sono tutti modi per accedere alla 9 posizione del vettore s puntato da p

st=p NO! non è possibile reindirizzare il puntatore di un ARRAY

LE STRINGHE:

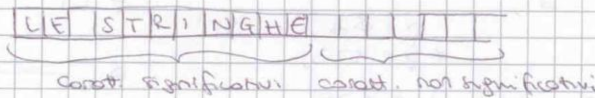
TIPI DI DATO ELEMENTARI: int, double, char, "boolean"
TIPI DI DATO COMPLESSI: Array monodimensionali, Array multidimensionali, Puntatori

- TYPEDEF: C permette di definire nuovi nomi per i tipi di dati mediante il seguente costrutto

```
typedef <tipo> <nuovo_nome_tipo>;
```

ES:
```
typedef int intero;
intero i; // definisce una variabile di tipo int
```

- STRINGHE: vettore di caratteri che contiene il carattere '\0' (terminatore) che indica che i successivi caratteri non sono significativi



ES:
```
# define Dim 5
...
char ch[Dim];
char s[] = "prova";
```

ch[0]	
ch[1]	
ch[2]	
ch[3]	
ch[4]	
s[0]	p
s[1]	r
s[2]	o
s[3]	v
s[4]	a
s[5]	\0

(!) C non ha un sistema semplice per costruire stringhe ma ha una libreria per trattarle

<STRING.h>

#include <string.h> contiene:

1) int strcmp (char *string1, char *string2) — confronta string1 e string2
2) void strcpy (char *string1, char *string2): copia string2 in string1
3) char * strerror (int err);
4) int strlen (char * string);
5) char * strcat (char * string1, char * string2, size_t n): aggiunge "n" caratteri di string2 a string1
6) int strcmp (char * string1, char * string2, size_t n): confronta i primi n caratteri di 2 stringhe
7) char * strncpy (char * string1, char * string2, size_t n): copia i primi n caratt. di string2 in string1
8) char * strnset (char * string, char c, size_t n): setta i primi n caratteri di stringa a 'c'
9) char * strchr (char * string, char c): cerca l'ultima occorrenza del caratt. "c" in string

(!) quando si usa scanf con le stringhe bisogna stare attenti => ricordare che è un vettore e non una semplice variabile => non si usa "&"

```
scanf ("%s", s);
```

(!) attenzione:

La strcmp(s1, s2) restituisce un num < 0 se $s_1 \neq s_2$
num > 0 se $s_1 = s_2$

Algoritmi e Strutture Dati

- LINGUAGGIO C -

i vettori occupano memoria. Il programmatore deve specificare il tipo di ogni elemento e il n° degli elementi, cosicché il PC possa inserire l'appropriata memoria

ES:
```
char s[] = "blu";
char s[] = {'b', 'l', 'u', '\0'}   // in questo caso va specificato
```

OSS: `strcat (s1, s2)` // concatena le 2 stringhe

OSS: per definire il tipo stringa bisogna:

```
#define SMAX 30
typedef char stringa [SMAX];
```

- MANIPOLAZIONE DI STRINGHE:

```
1) void leggiStringa (stringa); // legge una stringa da input
2) void scriviStringa (stringa); // scrive una stringa in output
3) int lung (stringa); // calcola lunghezza di una stringa
4) void assegna (stringa, stringa); // esegue l'assegnamento su stringhe
5) void concatena (stringa, stringa, stringa); // concatena 2 stringhe e pone il risultato in una terza
6) char elemento (stringa int); // restituisce l'i-esimo carattere della stringa
7) getChar () : // funzione per leggere un carattere
8) putChar (s) : // funzione per stampare un carattere s
```

- STRUTTURE DATI

Def: sono dette "Strutture" le collezioni di variabili correlate sotto un unico nome. Esse possono essere collezioni eterogenee di variabili ⇒ le variabili possono essere di diversi ~~tip~~ TIPI. Esse sono tipi di dati derivanti da altri tipi

OSS: la combinazione di Strutture (RECORD) e puntatori danno vita alle liste concatenate (struct ≃ oggetti)

struct = PAROLA CHIAVE CHE INTRODUCE LA DEF. DELLA STRUTTURA

```
struct <nome_struttura> {
    VARIABILI (CAMPI) }
```
] STRUTTURA DI UNA STRUTTURA

```
struct <nome_struttura><nome-var>;
```
] DICHIARAZIONE DI UNA STRUTTURA

"." operatore che permette di accedere ai camp. della struttura

OSS:

```
struct <nome_struttura> *p;
```
p è un puntatore che punta alla struttura
per accedere ai campi della struttura c'è bisogno dell'operatore "→"

ES: `p → <campo>;`

oss: posso dichiarare le strutture mentre le definisco

```
struct <nome_struttura> {
    CAMPI
} <var1>, <var2>;
```

<var1> e <var2> sono 2 strutture

oss: posso creare ≠ strutture con la medesima definizione

```
struct {
    CAMPI
} <nome_struttura1>, <nome_struttura2>;
```

così facendo ho definito 2 strutture con la stessa definizione
(!) questo non è una dichiarazione

oss: posso definire l'oggetto anche con il TYPEDEF

```
typedef struct {
    CAMPI
} <nome_struttura>;
...
<nome_struttura> <var1>;     (DICHIARAZIONE)
```

oss: un campo di una struttura può essere a sua volta una struttura

oss: visto che un campo può essere una struttura... quindi può anche essere un autoriferimento alla stessa struttura (CONCETTO DI LISTA)

- ALLOCAZIONE DINAMICA DELLA MEMORIA -

si usa per gestire di cui non è nota a priori la dimensione
questa gestione avviene mediante puntatori

→ la libreria <stdlib.h> contiene 4 operazioni per gestire la MEMORIA:

1) MALLOC: alloca una zona di memoria della dim specificata. Restituisce un puntatore void all'area di memoria. Se non c'è memoria restituisce NULL
- oss: viene seguito sempre da un CASTING per restituire un puntatore di tipo desiderato
- oss: void *p = un puntatore di tipo qualsiasi

```
void * malloc (size_t size)
```
MALLOC

- oss: con la MALLOC si usa l'operazione SIZE-OF che restituisce la dim di una variabile o di un tipo

```
size_t sizeof(<VAR. O TIPO>)
```
SIZE OF

Algoritmi e Strutture Dati

- LINGUAGGIO C -

ES:
- (int *) malloc (sizeof(int)); // alloco una cella di memoria per un intero
- int *p;
  p= (int *) malloc (sizeof (*p)); ←
- Persona *tizio;
  tizio = (Persona *) malloc (sizeof(persona)); // alloco una cella di memoria per una persona

2) CALLOC: alloca una zona di memoria per memorizzare n oggetti della dimensione specificata

3) REALLOC: ralloca uno spazio di memoria ⇒ modifica un'area di memoria già allocata

OSS: in entrambi i casi viene restituito un puntatore void o NULL se non c'è memoria

CALLOC
```
void *calloc (size_t n_elem, size_t elem_size)
```

REALLOC
```
void *realloc( void *p, size_t size)
```

OSS: size_t n_elem = n° elem. da allocare
size_t size (REALLOC) = la nuova dimensione da riallocare a *p

ES:
```
Persona *grp;            // → alloca memoria per 10 persone
grp = (Persona *) calloc (10, sizeof(persona));

Persona *p;
Studente *s;
p= (Persona *) malloc (sizeof(persona));
s= (Studente *) realloc (realloc (p, sizeof(studente));
```
↳ ralloca la cella di memoria che conteneva una persona (puntata da p)

4) FREE: libera una zona di memoria precedentemente allocata

OSS: dovrebbe essere sempre usata prima della fine di un programma su ogni variabile allocata dinamicamente

FREE
```
void *free (void *p)
```

• LISTE CONCATENATE:

Def: si dice "LISTA CONCATENATA" una collezione lineare di strutture connesse da puntatori (LINK)

Def: si dicono "NODI" di una lista le strutture ricorsive che come campo hanno un puntatore che fa riferimento ad una struttura dello stesso tipo

OSS: l'ultimo elemento punterà a NULL

⇒ ~~PONTATORI~~ PUNTATORE

OSS: struct + puntatori => liste

- MEMORIA SECONDARIA - [FILE]

consente la memorizzazione permanente dei dati e la condivisione tra programmi ≠

Def: si dice "FILE" un contenitore di informazioni permanente. È un insieme di record
oss: la vita di un file è indipendente dalla vita di un programma. Infatti il file, terminato il programma, continua ad ∃

Def: Si dice "METODO DI ACCESSO" A UN FILE, la tecnica usata dal programma per accedere al file

OSS:
C vede i file come sequenze di Byte che terminano con EOF (MARCATORE)
C prima di utilizzare il file deve aprire un "flusso di comunicazione" e dopo aver usato il file deve chiudere il ϕ

| FILE *fp; | DICHIARAZIONE DI UN FILE (Def. una variabile di tipo "PUNTATORE A FILE") |

→ OPERAZIONI PER I FILE: (presenti in <stdio.h>)

1) fopen: → apre un file o restituisce NULL se l'apertura di f non ha successo
2) fclose: → chiude un file o restituisce NULL se la chiusura di f non ha successo
3) fgetc: → legge un carattere da f e lo stampa se ha avuto successo altrimenti EOF
4) fputc: → scrive un carattere in f e lo stampa se ha avuto successo altrimenti EOF
5) fgets: → legge n caratteri da f e li memorizza in una stringa. Restituisce s o NULL
6) fputs: → scrive una stringa in f senza aggiungere '\0'. Restituisce 0 o EOF
7) fprintf: → stampa i caratteri
8) fscanf:
9) fread: → legge n oggetti da f di una certa dim e li memorizza in un vettore, restituisce n° ogg. letti
10) fwrite: → scrive n oggetti in f di una certa dim prelevandoli da un vettore. Restituisce n° oggetti scritti
11) fseek: → definisce una posizione in f da cui partiranno le prossime operazioni I/O restituisce pos o NULL
12) fsetpos:
13) ftell: → restituisce la posizione corrente in f oppure un n° negativo (in caso d'errore)
14) fgetpos:

→ APERTURA DI UN FILE & CHIUSURA: un file può essere aperto in più modalità attraverso dei caratteri:

- r (READ): f già ∃ viene aperto in lettura e restituisce il puntatore a inizio f
- W (WRITE): f viene creato se ∄ oppure viene sovrascritto se già ∃ e restituisce il puntatore a inizio f
- a (APPEND): f viene creato se ∄ oppure viene aperto in scrittura con inserimenti solo dal fondo. Viene restituito il puntatore alla fine del file
- + (UPDATE): concatena le operazioni (MODALITÀ) - (Unica con r)
- b (BINARY): si usa in combinazione con una delle prime modalità - gestione per file binari.

| FILE * fopen (nomefile, modalità) | APERTURA DI UN FILE |

OSS: un file può essere binario o di caratteri. Di Default è di caratteri

`void fclose (FILE *fp)` CHIUSURA DI UN FILE

- LETTURA E SCRITTURA DI UN FILE:

`int fput (int c, FILE *pf)` SCRITTURA DI UN FILE

oss: è vero che restituisce un carattere, allora perchè intero? qualsiasi info è un numero anche i caratteri. Infatti l'alfabeto UNICODE associa ad ogni carattere un numero ⇒ possiamo mettere sia int che char tanto il programma lo traduce

`int fget (FILE *pf)` LETTURA DI UN FILE

`char *fgets (char *s, int n, FILE *fp)` LETTURA DI UNA STRINGA IN UN FILE

`int fputs (char *s, FILE *fp)` SCRIVE UNA STRINGA IN UN FILE

- OPERAZIONI DI I/O:

`int scanf (FILE *pf, str_cont, elementi)`

oss: str_cont = stringa di controllo

`int fprintf (FILE *pf, str_cont, elementi)`

`int fread (void* buf, int size, int n, FILE *fp)` LETTURA di n OGGETTI DA FILE E MEMORIZZAZIONE IN UN VETTORE

*buf: puntatore alla regione di memoria dove vengono memorizzati i dati

`int fwrite (void* buf, int size, int n, FILE *fp)` SCRITTURA DI n OGGETTI IN F da UN VETTORE

- POSIZIONAMENTO SU FILE:

`int fseek (FILE *fp, long s, int o)` POSIZIONAMENTO SU FILE

s: spiazzamento a partire da o ⇒ la posizione è ora a 5 Byte da 0

`long ftell (FILE *fp)` POSIZIONE CORRENTE

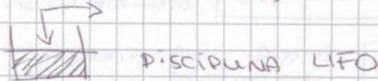
- PROPRIETÀ DELLA SEEK -
1) SEEK_SET: posizionamento parte dall'inizio del file
2) SEEK_CUR: posizionamento parte dall'attuale posizione
3) SEEK_END: posizionamento parte dalla fine

• feof (*f): verifica se il file è arrivato alla fine

- TIPI DI DATO ASTRATTO -

→ FUNZIONI DELLE LISTE:
1) plist init(): restituisce la lista vuota
2) int empty (plist): verifica se è vuota
3) plist cons(p list, tipoElem): aggiunge elem in testa alla lista e la restituisce
4) void Stampa (p list): stampa la lista
5) tipo elem car: restituisce il I° elem della lista
6) plist cdr (plist): elimina il I° elem lista e restituisce la lista

→ PILE (o STACK)


DISCIPLINA LIFO

FUNZIONI DELLE PILE:
1) pstack initS(): vedi init
2) int emptyS(pstack): vedi empty
3) pstack push(pstack, tipoelem): vedi cons
4) tipoelem top(pstack): vedi car
5) p stack pop(p stack): vedi cdr
6) void stampaS(p stack): vedi stampa

→ LE CODE (QUEUE)

disciplina FIFO

oss: la coda ha 2 puntatori. Uno in testa alla lista (head) e uno in coda (tail)

Algoritmi e strutture dati

31/

ALBERI BINARI

def: un albero binario è un insieme finito di nodi e archi orientati dove ogni arco collega il nodo padre al nodo figlio.
- ogni nodo ha esattamente un padre
- ogni nodo ha al più due figli
- il nodo radice non ha genitore

LIVELLO NODO: distanza dalla radice, espressa come numero di archi di cui è composto il cammino dalla radice al nodo. La radice è a livello 0;

PROFONDITÀ: altezza dell'albero

IMPLEMENTAZIONE 1
```
struct bit_node {
        Item info;
        struct bit_node *left;
        struct bit_node *right; }
typedef struct bitnode* nodo;
```

IMPLEMENTAZIONE 2
```
struct bit_node {
        int info;
        struct bit_node *p;
        struct bit_node *left;
        struct bit_node *right; };
typedef struct bit_node * nodo;
```

VISITE ALBERI

INORDER → attraversamento in ordine simmetrico, prima il sottoalbero di sinistra poi la radice infine il sottoalbero di destra

```
void albInorder(nodo n){
    if (n) {
        albInorder (n->left);
        printnode (n);
        albInorder (n->right); }}
```

PREORDER → attraversamento in ordine anticipato: prima la radice poi il sottoalbero di sinistra, infine il sottoalbero di destra

```
void albPreorder(nodo n){
    if (n) {
        print node(n);
        albPreorder (n->left);
        albPreorder( n-> right); }}
```

POSTORDER → prima il sottoalbero di sinistro poi quello di destra ed infine la radice.

```
void aebPostorder(nodo n){
    if(n) {
        aebPostordine(n->left);
        aebPostordine(n->right);
        printnode(n); }}
```