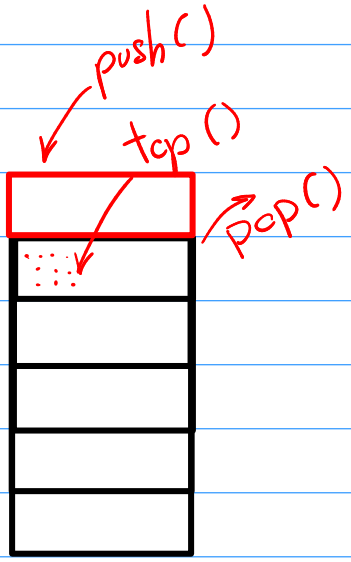
Tipo di dato vs struttura di dati

↓

modello matematico che consiste in una collezione di valori sul quale sono definite certe operazioni.

↓

Pila: dati -> seq. di elementi accedibili da un solo lato



operazioni -> top(), legge in testa
pop() tolgo alla pila
push() aggiungo alla pila

Per definire un tipo di dato si specifica cosa un'operazione deve fare ma non come deve essere realizzata, e non come gli oggetti della mia collezione possono essere organizzati in modo che l'operazione sia efficace e la collezione stessa occupi poco spazio in memoria

↓

Specifica della costruzione del tipo di dato e le sue operazioni. Una particolare organizzazione delle informazioni che permette di rappresentare in modo efficiente le informazioni di un tipo di dato.

Classificazione delle strutture dati

1) Disposizione dei dati

2) Numero " "

3) Tipo " "

1) Distingue tra lineari, non lineari → no sequenze, come grafo, albero

lineari → array, lista, pila, coda

lineari → formati da sequenze (distingue tra 1°, 2°, 10° ecc.)

2) Distingue tra statiche e dinamiche → lista, pila, albero

statiche → array

statiche → n° elementi = costante

dinamiche → n° elementi = variabile

3) Tipi ambigui o non ambigui

ambigui → omogeneo

omogeneo → tutti gli elementi con lo stesso tipo (array)

non ambigui → non omogeneo

non omogeneo → i dati non sono dello stesso tipo

Dizionario: tipo di dato, rappresenta il concetto matematico di relazione univoca, ovvero una funzione mappa elementi di un insieme D, dominio, e gli elementi di un insieme C, codominio.

elementi D = chiavi , elementi C = valori

associazioni chiave-valore

dati del dizionario => insieme di coppie "chiave-valore"

operazioni:

- search (Dizionario s, Chiave k) -> Elemento ∪ {NIL} (NIL → null)
  precondizione: // nessuna
  postcondizione: restituisce il valore associato alla k, se presente in s, NIL altrimenti

- insert (Dizionario s, Elem v, chiave k)
  precondizione: // nessuna
  postcondizione: associa il valore di v alla chiave k

- delete (Dizionario s, Chiave k)
  precondizione: k presente in S
  postcondizione: cancella la coppia con chiave k da S

Realizzazione attraverso array ordinati

dati: un array a, di n dimensione, contenente dei record, con dei campi (key, info) ordinato in base al campo key

Assumo che array abbia un attributo length con la dimensione dell'array.

(spazio) $S(n) = \Theta(n)$

```
search (Dizionario A, Chiave k) -> value {
    // essendo array ordinato, si può applicare una
    // ricerca binaria
    i = searchIndex (A, k, 1, A.length)   ] O(log n)
    if (i == -1) return NIL               ] Θ(1)
    else         return A[i].value
}
```

pre: A[p....r]
post: restituisce $p \le i \le r$ se $k \in A$, altrimenti -1

```
searchIndex (Dizionario A, Chiave k, Int p, Int r) -> int {
    if p > r  return -1   ] impera ] Θ(1)   n = 0
    else
        med = p + r/2     ] divisione
        if A[med].key == k    return med   ] impera
        else
            if A[med].key > k
                return searchIndex (A, k, p, med-1)   ] impera
            else
                return searchIndex (A, k, med+1, r)   ] impera
```

>0 Θ(1)

T(n/2) — dividi

1) La funzione risulta essere ricorsiva

2) Trovo il caso base

- if $p > r$ return -1 $\left(\Theta(1)\right)$

3) Analizzo quando ho le chiamate ricorsive

4) Calcolo complessità

$$T(n) = \begin{cases} \Theta(1) & n = 0 \\ T(n/2) + \Theta(1) & n > 0 \end{cases}$$

n = # elementi della porzione di array
$n = r - p + 1$

uso il Teorema master

5) In quale caso sono?

$n^{\log_b a} = n^{\log_2 1} = n^0 = \Theta(1)$ tempo costante

$f(n) = \{\text{split} + \text{merge}\} = \Theta(1)$ tempo costante

Confronto tra le due, vedo che siamo nel secondo caso!

in quanto abbiamo $f(n) = O\left(n^{\log_b a}\right) = \Theta(1)$

complessità del caso peggiore $= \Theta(n^{\log_b a} \log n) = \Theta(\log n)$!!

tempo di esecuzione = varia molto in base ad n, l'input.

caso + fortunato = al centro , caso peggiore = non c'è.

Tempo di esecuzione di search_index = $O(\log n)$

tempo peggiore!

(tempo di esecuzione = complessità) caso peggiore

tempo costante!

tempo di esecuzione della search = $O(\log_n) + \Theta(1) = O(\log_n)$

Funzione insert (1,15) | (7,29) | (9,5) | (15,30) inserisco (14,59)

insert (Dizionario A, elem v, Chiave k)

```
i = 1
while i < A.length and A[i].key < k        ] i volte * Θ(1)
      i = i + 1                                  (Θ(1))
if i ≤ A.length and A[i].key == k          ] Θ(1)
      A[i].info = v
else
      reallocate (A, A.length + 1)         ] O(n)
      (A.length = A.length + 1)
for j = A.length down to i+1 do            [ n+1-(i+1)+1 volte
      A[j] = A[j-1]   // sposto a destra di 1   [ Θ(1) [
A[i].key = k                               ] Θ(1)
A[i].info = v
```

i Θ(1)

Θ(1) costante

(n-i+1) volte

Calcolo della complessità

1) Calcolo complessità op. * op.

$$T(n) = \Theta(1) + id + \Theta(1) + \Theta(n) + d(n-i+1) + \Theta(1)$$

$$= dn + \Theta(1) + \Theta(n) = \Theta(n)$$ ← caso peggiore

Il tempo di esecuzione è $O(n)$

raddoppia( ) // raddoppio e dimezzo

↓

da un array di dim h,

n = numero elem
h = dim array

per ogni $n > 0$ h soddisfa

$$n \leq h < 4n$$

le prime n celle hanno gli elementi della collezione, mentre il contenuto in altre celle è indefinito

- Inizialmente, n = 0, pongo h = 1,
- Ogni volta che n supera h, l'array raddoppia la dimensione, quindi $h = 2h$

- ogni volta che n scende sotto $h/4$ l'array si dimezza,

$h = h/2$

$S_{spazio} = S(n) = \Theta(h) = \Theta(n)$

Analisi ammortizzata è l'analisi del costo medio dell'esecuzione di m operazioni in una struttura dati ↘ esempio

vettore inizialmente con $n=0$, $h=1$

considero di dover inserire n elementi.

Sia C_i il costo dell'i-esimo inserimento:

$$C_i = \begin{cases} i & \text{se } \exists k : i = 2^k + 1 \\ 1 & \text{altrimenti} \end{cases}$$

$$C_n = \sum_{i=1}^{n} C_i \leq n + \sum_{k=0}^{\lfloor \log_2 n \rfloor} 2^k = n + \frac{2^{\lfloor \log_2 n \rfloor + 1} - 1}{2 - 1}$$

C_n ↓ costo inserimento

$= n + 2 \cdot n - 1 \leq 3n$

Ottengo $C(n) \leq 3n$

Considero il costo medio di 1 inserimento

↓

costo complessivo / n elementi

$\frac{C(n)}{n} \quad \frac{3n}{n} = 3 \rightarrow$ il costo ammortizzato è costante!

Precondizione: k nel dizionario

delete (Dizionario A, Chiave k)

```
i = search_index(A, k, 1, A.length)      ] Θ(log n)
for j = i to A.length - 1                ] Θ(n)
    A[j] = A[j+1]
```

Θ(n) [reallocate(A, A.length - 1) // ricordo come visto sopra, ma al contrario!
       (A.length = A.length - 1)

↓

min(vecchio spazio, nuovo spazio)

Tempo di esecuzione

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

$O(\log n) \rightarrow O(n)$

Realizzazione basata su strutture collegate:
record e puntatori

dati: una collezione L di n record contenente (key, info, next, prev) con next, prev puntatori al record successivo/precedente

Un attributo L.head punta al primo elemento della lista

potrei avere doppioni

ottimale = costante = inserimento in testa

Operazioni:

insert (Dizionario L, Elem v, chiave k)

↓

crea un record p in (key, info)

inserimento in testa, costo = $\Theta(1)$

```
p.next = L.head
if L.head != nil
    L.head.prev = p
L.head = p
p.prev = nil
```

$T(n) = \Theta(1)$

search (Dizionario L, Chiave k) // doppioni, cerco prima occorrenza

```
x = L.head
while x ≠ NIL and x.key ≠ k
    x = x.next;
if x ≠ NIL return x.info
else       return nil
```

Guardia (prog. succ.e)

$\Theta(n)$

Tempo esecuzione: $T(n) = O(n)$

↓

n = # elementi nella lista

Correttezza: la search restituisce la PRIMA occorrenza

Invariante è un asserzione prima, dopo e ad ogni iterazione del ciclo che risulta vera

Per un'invariante si deve dimostrare:

1) inizializzazione: è vera prima della prima iterazione nel ciclo
2) conservazione: se è vera prima della prima iterazione, rimane vera prima anche dell'iterazione successiva

   Inv ∧ guardia -> Inv [dopo esecuzione del ciclo]

3) Conclusione: quando il ciclo termina, l'inv. fornisce la proprietà che aiuta a dimostrare che l'algoritmo sia corretto

   Inv ∧ ¬ guardia -> asserzione finale

esempio, nella search, o si ferma alla prima iterazione, o invece finisce la linked list [guardia] = condizione terminazione del ciclo

se guardia vera, il codice continua

Funzione di terminazione, è una funzione a valori naturali che decresce strettamente ad ogni iterazione del ciclo.

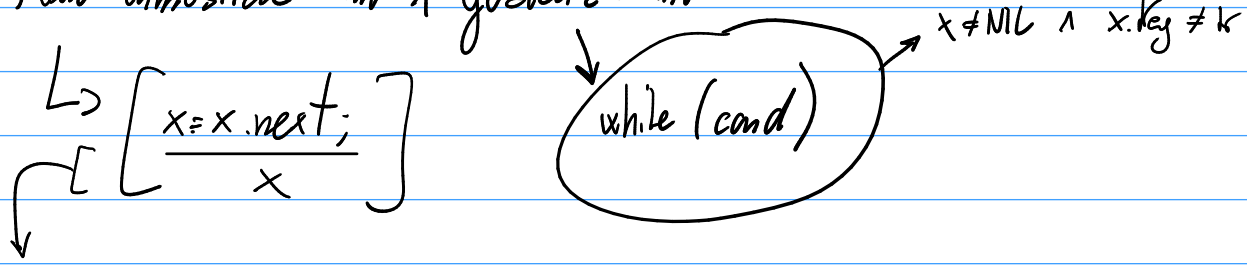
↳ esempio: per il search, la funzione di terminazione corrisponde al numero di elementi ancora da visitare

Inv in search := gli elem. da L.head a x non compresa hanno chiave diversa da (k) ∀∀

(scorro fino a trovare un elemento con chiave = k)

1) x = L.head (pre ciclo) (asserzione vera)
perché non ci sono elementi

2) devo dimostrare Inv ∧ guardia ⇒ Inv



per ogni occorrenza. Gli elementi da L.head ad x.next non compresi hanno
dunque chiave diversa da (k)

vero! fino a x.key == k

3) Conclusione: Inv ∧ ¬ guardia → asserzione finale

x == NIL allora esco
∨
x.key = k allora esco

// per cosa riceve un'invariante che tenga il numero di
// iterazioni di una chiave nell'array.

Delete: (vicordo che ho una precondizione, ovvero che la chiave esiste)

```
delete (DizionarioC L, Chiave K)   // ho + occorrenze, arrivo alla
                                   // fine dell'array
    x = L.head
    while (x ≠ NIL) {
        if (x.key == K)
            if (x.next ≠ NIL)   // ultimo elemento
                x.next.prev = x.prev
            if (x.prev ≠ NIL)   // primo elemento
                x.prev.next = x.next
            else
                L.head = x.next
            temp = x
            x = x.next
            rimuovi (temp)
        else
            x = x.next
    }
```

Tempo di esecuzione $\Theta(n)$ (non $O(n)$ perché deve sempre scorrerlo tutto)

```
A(int n)
    s = 0                          Θ(1)
    for i = 1 to n
        s = s + B(i)
    return s                       Θ(1)
```

$$\sum_{i=1}^{n} \Theta(i) = \sum_{i=1}^{n} c \cdot i$$

costo in B costante

```
B(int m)
    s = 0                          ] Θ(1)
    for i = 1 to m      m-volte    ] Θ(m)
        s = s + 1
    return s                       ] Θ(1)
```

1) Complessità di B (perché innestata in A)

↓

complessità = $\Theta(m)$

2) Complessità di A

↓

Complessità $T_A(n) = \Theta(1) + \sum_{i=1}^{n} c\,i = d + \sum_{i=1}^{n} c\,i$

$= d + c \sum_{i=1}^{n} i$ → somma aritmetica, posso trasformare

$= d + c\left(\frac{(n+1)n}{2}\right) = O(n^2)$

```
A-ott(int n)                       // versione ottimizzata
    return (n-1)n/2                Θ(1)
```

```
foo(n)
  if n <= 2
     return 1
  else
     if n > 321
Θ(1)C   i = n/3
        return 2 * foo(i) + n*n*n*i
     else
        return foo(n-3) + foo(n-2)
```

una sola chiamata, $T(n/3)$ $\rightarrow$ $a=1$, $b=3$

parte che mi interessa (andamento asintotico)

$$T(n) = \Theta(1) + T\left(\frac{n}{3}\right)$$

Applico teorema Master

$n^{\log_3 1} = n^0 = 1$

$f(n) = \Theta(1)$

Secondo caso

$f(n) = \Theta(n^{\log_3 1}) = \Theta(1)$

dunque la complessità risulta essere $T(n) = \Theta(\log n)$

Proc (m) = $\Theta(\sqrt{m})$ determinare la complessità asintotica (caso peggiore) di questa procedura

```
Fun (A, n)

    if n < 1
        return 1
    else
        t = Fun (A, n/2)          ] T(n/2)
        if T > A[n]
            t = t + Fun (A, n/2)  ] T(n/2)

        for j = 1 to n            ] n-volte Θ(√n) = Θ(n√n)
            t = t + A[j] + Proc(n);

        return t
```

$$T(n) = T\left(\frac{n}{2}\right) + T(n/2) + \Theta(n\sqrt{n}) \qquad c < 1$$

$$T(n)\ 2T(n/2) + \Theta(n\sqrt{n})$$

$$a f(n/b) \leq c f(n)$$

$$\frac{n}{8}\sqrt{\frac{n}{2}} \leq c\, n\sqrt{n}$$

$$n^{\log_2 2} = n^1 = n$$

$$n \cdot \frac{\sqrt{n}}{2} \leq c \cdot n\sqrt{n} \qquad \frac{1}{2} \leq c$$

$$\varepsilon = 1/2$$

$n \cdot \sqrt{n} > n$ 3° caso $f(n) = \Omega(n^{1+\varepsilon}) = \Omega(n^{3/2})$

$$T(n) = \Theta(n\sqrt{n})$$

```
compute(int n)
    if n ≥ 10
        for i = n down to n-3
                j = ⌊n/2⌋
        return compute(⌊n/2⌋) . compute(⌊n/2⌋) * 5n

    else
        if n ≥ 20
            return compute(n+1)
        else
            return n
```