

Algoritmi 1

COS'È UN ALGORITMO

L'algoritmo è un insieme d'instruzioni non ambigue che consentono di ottenere un risultato attraverso dei valori in input, sono alla base dei programmi perché forniscono il procedimento per arrivare alla soluzione di un problema di calcolo. Un algoritmo dev'essere:

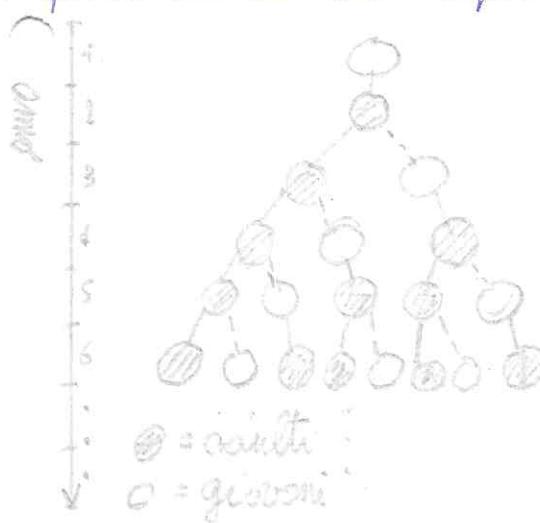
- **Corretto**: devono sempre produrre il risultato desiderato;
- **Efficiente**: Tempo e Spazio devono essere più contenuti possibili.

L'analisi teorica di un algoritmo è importante perché ci aiuta a scegliere tra differenti algoritmi e permette di prevedere le prestazioni prima di scrivere codice.

NUMERI DI FIBONACCI

Problema: Quanto velocemente si espanderebbe una popolazione di conigli in determinate condizioni?

Condizioni: Una coppia di conigli genera un'altra coppia ogni anno; i conigli ponono ripudi a partire dal secondo anno dopo la nascita; i conigli sono immortali.



Nell'anno n vi sono tutte le coppie dell'anno precedente più una coppia per ogni coppia nata due anni fa:

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{se } n \geq 3 \\ 1 & \text{se } n=1 \text{ o } n=2 \end{cases}$$

SOLUZIONE 1

Posiamo considerare la crescita dei conigli come una funzione esponenziale: $a^n = a^{n-1} + a^{n-2} \rightarrow a^{n-2}(a^2 - a - 1) = 0$

Risolvendola si ha che: $F_n = \frac{1}{\sqrt{5}} (\Phi_1^n - \Phi_2^n)$

PSEUDOCODICE

```
Fib1(int n) → int
    return  $\frac{1}{\sqrt{5}} (\Phi_1^n - \Phi_2^n)$ 
```

Il codice funziona ma a lungo andare commette errori d'arredamento.

ALBERO DELLE CHIAVATE

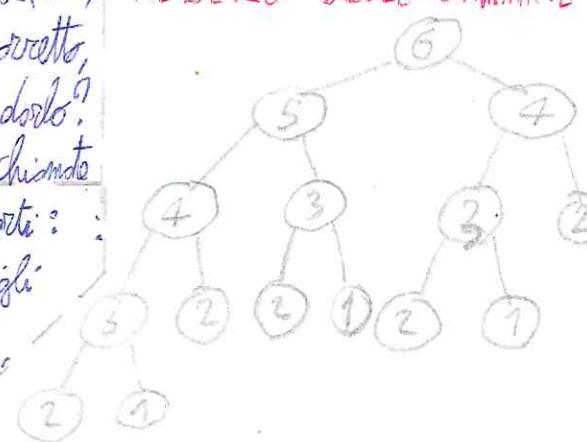
L'algoritmo ora ha un risultato corretto, ma quanto tempo ci metterà a dorlo?

Utilizziamo l'albero delle chiavi. Posiamo dividere l'albero in parti:

- **leffie**: nodi che non hanno figli
- **rimi**: nodi che hanno figli.

Le foglie contengono una istruzione

mentre i rami e, quindi:



$$T_m = F_m + 2(F_{m-1}) = 2F_m - 2$$

L'algoritmo è corretto ma a metà troppo tempo a calcolare una soluzione. In più, fa più volte le stesse operazioni. Una possibile soluzione è la programmazione dinamica, cui si memorizzano i risultati di ogni procedimento svolto.

SOLUZIONE 3

$\text{Fib3(int } n\text{)} \rightarrow \text{int}$

// fib è un array d'interi

$\text{fib}[0] = 1$

$\text{fib}[1] = 1$

for $i=2$ to $n-1$

$\text{fib}[i] = \text{fib}[i-1] + \text{fib}[i-2]$

return $\text{fib}[n-1]$

SOLUZIONE 4

$\text{Fib4(int } n\text{)} \rightarrow \text{int}$

int $a=b=1$

for $i=3$ to n

$c=a+b$

$a=b$

$b=c$

return b

il 3 è più veloce rispetto a Fib2 perché impiega un tempo proporzionale a n e non esponenziale. È possibile migliorare l'algoritmo in altro? (certo)

L'algoritmo ora occupa meno spazio (dato che si serve solo il risultato finale e non quelli intermedi) ma ha una complessità maggiore rispetto a Fib3: $1 + (n-3+1) \cdot 3 + 1 = 3n-4$

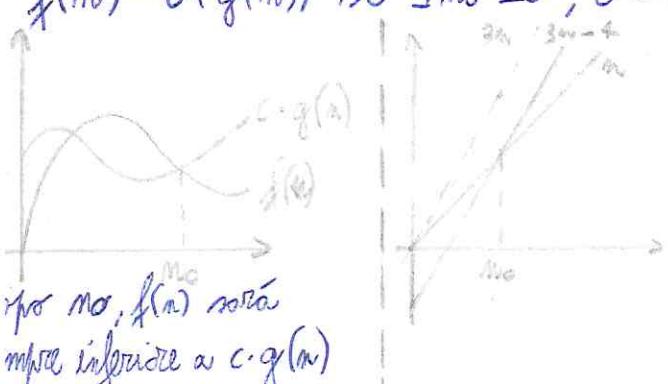
Fib4 è più significativo rispetto a Fib3?

NOTAZIONE ASINTOTICA

rispondere alla precedente domanda, si utilizza

notazione asintotica, ovvero trovare un modo per descrivere $T(n)$ senza considerare le costanti moltiplicative. In questo caso si utilizzerà $O()$.

$f(n) = O(g(n))$ se $\exists m_0 \geq 0, c > 0 : \forall n \geq m_0 \quad f(n) \leq c \cdot g(n)$



per m_0 , $f(n)$ sarà
sempre inferiore a $c \cdot g(n)$

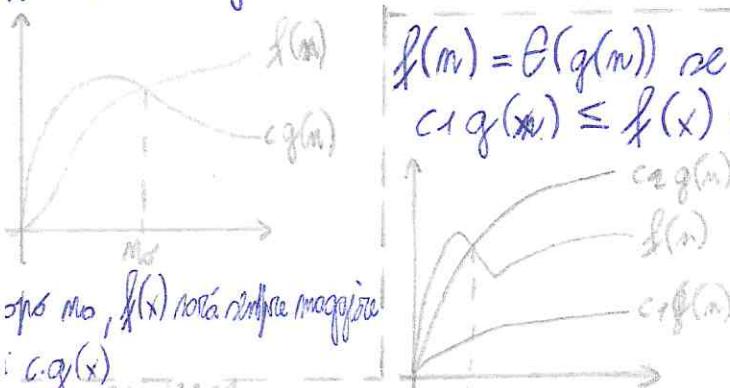
confrontando $3n-4$ e $3n$, si scopre che la prima è minore, quindi si può applicare $O()$.

CONCLUSIONI: $O(n) = 3n-4$ se $c=3$ e $m_0=0$
la complessità in tempo è $O(n)$.

È possibile dire che $O(n^2) = 3n-4$?

Si ma $O(n^2)$ è molto più grande, quindi è più giusto il primo.

$f(n) = \Omega(g(n))$ se $\exists m_0 \geq 0, c > 0 : \forall n > m_0 \quad f(n) \geq c \cdot g(n)$



per m_0 , $f(n)$ sarà sempre maggiore
di $c \cdot g(n)$

$f(n) = \Omega(g(n))$ se $\exists m_0 \geq 0, c_1, c_2 > 0 : \forall n \geq m_0$
 $c_1 g(n) \leq f(n) \leq c_2 g(n)$

$c_2 g(n)$ Dopo m_0 , $f(n)$ sarà sempre compresa tra le due funzioni

CONCLUSIONI:

$3n-4 = O(n)$ con $m_0=0$ e $c=3$

$3n-4 = \Omega(n)$ con $c=1$ e m_0 è il punto interendo

n è sia il limite inferiore che superiore di $3n-4$.

$\sim n-4 - A(n) \quad \sim n \quad \sim -2 \quad \sim 1 \sim 1$

Esistono algoritmi dove vi è sia un caso peggiore che uno migliore.

RICERCA SEQUENZIALE

ricerca(list L, int x) \rightarrow bool
for each ($y \in L$)
if ($y == x$) return trovato
return non trovato

Studiando l'algoritmo, possiamo dire che:

- $T_m(n) = 1$ // CASO MIGLIORE: x in testa alla lista
- $T_p(n) = n$ // CASO PEGGIOR: x in coda alla lista

RICERCA BINARIA

binaryIT(array L, int x) \rightarrow bool

$a = 1$
 $b = \text{len}(L)$
while ($L[\frac{a+b}{2}] \neq x$)
 $m = (a+b)/2$
if ($L[m] > x$) $b = m - 1$
else $a = m + 1$
if ($a > b$) return non trovato
return trovato

PRECONDIZIONE: l'array dev'essere ordinato.

CASO MIGLIORE: $T_m(n) = 1$

CASO PEGGIOR: $T_p(n) = \text{len}(L)$

binaryRIC(array L, int x) \rightarrow bool

$n = \text{len}(L)$
if ($n = 0$) return non trovato
 $i = n/2$
if ($L[i] == x$) return trovato
if ($L[i] > x$) return binaryRIC($L[1:i-1], x$)
else return binaryRIC($L[i+1:n], x$)

CONSIDERAZIONI

$$T(0) = 1$$

$$T_m(n) = 1 + 1 + 1 = 3$$

$$T(n) = C + T\left(\frac{n}{2}\right)$$

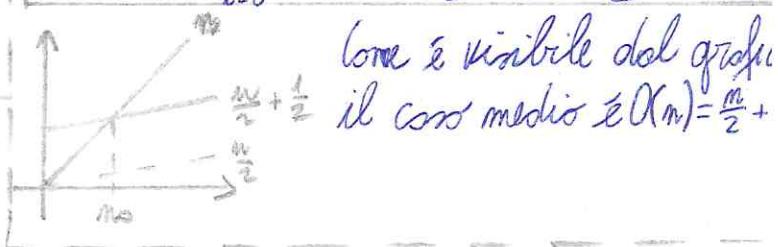
Il valore di C è irrilevante perché il "percorso" della ricorsione non ha importanza.

È possibile calcolare un costo medio? (erto!)

PROCEDIMENTO

Il costo medio è la somma di tutti i costi di ogni caso rapportata alla probabilità di essi; ha quindi che:

$$T_c(n) = \frac{1}{n} \sum_{i=0}^{n-1} i = \frac{1}{n} \cdot \frac{n(n+1)}{2} = \frac{n+1}{2} = \frac{n}{2} +$$



CASO MEDIO

La funzione "divisirà" il vettore in 2 parti ogni volta che non trova quello intermedio, pon quindi riassumere il tutto con una tabella

confronti elementi: $T_c(n) = \frac{1}{n} \sum_{i=1}^{\log_2 n} i \cdot 2^{i-1} = \log n - 1 + \frac{1}{2}$

1	1
2	2
3	4
i	2^{i-1}

Sono determinate situazioni, il costo medio di un algoritmo non varia molto da quello peggiore

ANALISI (METODO DELL'ITERAZIONE)

Si "ritrada" la ricorsione ottenendo un numero dipendente da n :

$$T(n) = C + T\left(\frac{n}{2}\right)$$

$$T\left(\frac{n}{2}\right) = C + T\left(\frac{n}{4}\right)$$

$$T\left(\frac{n}{4}\right) = C + T\left(\frac{n}{8}\right)$$

$$T(n) = C + C + C + \dots + T\left(\frac{n}{8}\right) = 3C + T\left(\frac{n}{8}\right)$$

Possiamo concludere che:

$$T(n) = i \cdot C + T\left(\frac{n}{2^i}\right) = \frac{n}{2^i} = 1$$

$$= \log_2 n + T(1) = i = \log_2 n$$

$$= \log_2 n + 1$$

$$\Theta(\log_2 n)$$

METODO DELLA SOSTITUZIONE

Il metodo di sostituzione consiste nell'individuare la soluzione di una relazione di ricorrenza e provvedere attraverso l'induzione matematica.

Esempio

TEOREMA MASTER

$$\left\{ \begin{array}{l} T(n) = T\left(\frac{n}{2}\right) + n \\ T(1) = 1 \end{array} \right.$$

$$\text{Ipotesi: } T(n) = O(n)$$

da l'ipotesi, dobbiamo dimostrare che $T(n) \leq c \cdot n$:

$$\text{ASO BASE: } T(1) = c \cdot 1 \quad \forall c$$

$$\text{ASO INTERMEDIO: } T(n) = n + T\left(\frac{n}{2}\right) \leq c \frac{n}{2} + n$$

$$\text{dato che: } T(n) \leq c \cdot n \rightarrow T\left(\frac{n}{2}\right) \leq c \frac{n}{2}$$

$$\text{quindi: } c \frac{n}{2} + n = \left(\frac{c}{2} + 1\right) n$$

$$\text{SOLUZIONE: } T(n) \leq c \cdot n \text{ con } c \geq 2$$

$$\text{quindi } T(n) = O(n).$$

Il teorema master è un metodo d'analisi basato sulla tecnica "divide et impera": Si divide il problema in sottoproblemi grandi $\frac{n}{b}$, si calcolano le relative soluzioni e infine si ricombinano.

Le relazioni di ricorrenza basate su questa tecnica possono essere facilmente risolte con l'utilizzo di questo problema. Le relazioni di ricorrenza hanno questo "struttura":

$$\left\{ \begin{array}{l} T(n) = aT\left(\frac{n}{b}\right) + f(n) \\ T(1) = 1 \end{array} \right.$$

dove a è il numero totale di chiamate ricorsive, b implica la dimensione di base e $f(n)$ è il tempo per dividere/ricondurre

le istanze di dimensione n .

Come è possibile vedere dal grafico, il numero di chiamate equivale sempre ad a e la dimensione di ciascuna volta che "si scende", è n sufficientemente grande.

SOLUZIONI POSSIBILI

$$1) T(n) = \Theta(n^{\log_b a}) \text{ se } f(n) = O(n^{\log_b a - \epsilon}) \text{ per } \epsilon > 0$$

$$2) T(n) = \Theta(n^{\log_b a} \log n) \text{ se } f(n) = \Theta(n^{\log_b a})$$

$$3) T(n) = \Theta(f(n)) \text{ se } f(n) = \Omega(n^{\log_b a + \epsilon}) \text{ per } \epsilon > 0$$

$$\text{diminuisce di } b \text{ ogni volta che "si scende"} \text{ e } a \cdot f\left(\frac{n}{b}\right) \leq c \cdot f(n) \text{ per } c < 1$$

$$\text{e } n \text{ sufficientemente grande.}$$

APPLICAZIONI

$$1) T(n) = n + 2T\left(\frac{n}{2}\right)$$

$$a=2, b=2, f(n)=n$$

$$n^{\log_2 2} = n^1 = n = f(n) \rightarrow \text{CASO 2}$$

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

$$2) T(n) = c + 3T\left(\frac{n}{3}\right)$$

$$a=3, b=3, f(n)=c$$

$$n^{\log_3 3} = n^1 = n > f(n) \rightarrow \text{CASO 1}$$

$$c = n^{\frac{1}{2}} - \epsilon \rightarrow \epsilon = \frac{1}{2}$$

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



$$1) T(n) = n + 3T\left(\frac{n}{3}\right) \quad a=3, b=3, f(n)=n$$

$$n^{\log_3 3} = n^1 = \sqrt{n} < f(n) \rightarrow \text{CASO 3}$$

$$n = n^{\frac{1}{2}} + \epsilon \rightarrow \epsilon = \frac{1}{2}$$

$$3 \frac{n}{3} \leq c n \rightarrow c = \frac{1}{3}$$

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

STRUTTURE DATI ELEMENTARI

Una struttura dati è un'organizzazione di dati che supporta le operazioni di un tipo di dato in modo meno ruivo di calcoli possibili, esse possono essere:

- **Indiretta**: viene utilizzato un array di dimensione nota per contenere i dati; vi è un accesso immediato ai dati attraverso gli indici ma la dimensione è fissa e la realizzazione richiede un tempo lineare;
- **Collegate**: i dati sono contenuti in record e collegati attraverso puntatori, la dimensione è variabile (il tempo d'aggiunta/rimozione di dati è costante) ma l'acceso ai dati è sequenziale



1 2 3 4 5 6 7 ← contenuto

n-1 ← indice

DIZIONARIO

Il dizionario è una struttura dati i cui dati sono rappresentati in coppie (chiave, valore).

Le operazioni supportate sono:

- **INSERT**: aggiunge una nuova coppia.
- **SEARCH**: restituisce l'elemento corrispondente alla chiave se presente, altrimenti restituisce null;
- **DELETE**: cancella una coppia appoggiandosi alla ricerca.

Complessità delle funzioni

Le funzioni del dizionario hanno complessità differente in base all'implementazione utilizzata: se implementiamo il dizionario con un vettore,

abbiamo che **insert** ha un costo sempre costante ($\Theta(1)$) dato che deve solo inserire un nuovo elemento, **search** nel caso peggiore ricorre all'intero vettore ($\Theta(n)$) e lo stesso vale per **delete**. Per il vettore ordinato, invece, la situazione è differente: **insert** deve inserire il valore rispettando l'ordine preesistente, causando un aumento del costo nel caso peggiore ($\Theta(n)$).

In contrario, è possibile utilizzare la ricerca binaria per **search**, diminuendo la complessità ($\Theta(\log n)$).

Questo non è possibile per **delete** perché, nonostante si appoggia alla ricerca, deve "ricreatuare" il vettore dopo l'eliminazione del dato.

Con la lista vi è un caso simile al vettore:

- **insert** ha un costo costante ($\Theta(1)$) dato che bisogna solo inserire un elemento;
- **search** costa $\Theta(n)$ a causa dell'acceso sequenziale dei dati;
- **delete** costa $\Theta(n)$ dato che si appoggia alla ricerca.

	VETTORE	VETTORE ORDINATO	LISTA
INSERT	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
SEARCH	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
DELETE	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$

PILA

La pila è una struttura dati in cui i dati sono inseriti attraverso la logica LIFO: l'ultimo a entrare è il primo a uscire. La pila utilizza le seguenti operazioni:

- **isEmpty**: controlla se la pila è vuota;
- **push**: inserisce un elemento in cima alla pila;
- **pop**: toglie un elemento dalla cima della pila;
- **top**: restituisce l'elemento in cima alla pila senza toglierlo.

CODA

La coda è una struttura dati in cui i dati vengono inseriti secondo la logica FIFO: il primo a entrare è anche il primo a uscire. La coda utilizza le seguenti operazioni:

- **isEmpty**: controlla se la coda è vuota;
- **enqueue**: inserisce un elemento in fondo alla coda;
- **dequeue**: restituisce il primo elemento della coda;

IMPLEMENTAZIONE LISTA

Per capire l'implementazione di pila e coda, occorre ripetere quella su lista:

```

typedef struct elem {
    data d;
    struct elem *next;
} node;
typedef node *list;

```

L'implementazione scritta a destra ha due compi:
 - `d` è il campo contenente il dato effettivo del nodo, il tipo `data` deve essere implementato con un ulteriore `struct`;
 - `next` è un puntatore al nodo successivo.
 Il secondo `typedef` permette l'utilizzo di "`->`" al posto di "`*next, campo`". Infatti: $list \rightarrow next = (*node).next$.

FUNZIONI LISTA

```

bool isEmptyL(list l)
{ return l==NULL;
}

```

```

void insertHead(data d, list *l)
{ list m = newNode();
    m->d = d;
    m->next = (*l);
    (*l) = m;

```

```

data first(list l)
{ if(l!=NULL) return l->d;
else
{ perror("Lista vuota");
    return 0;
}

```

```

data disposeHead(list *l)
{ data d;
list n;
if((*l)!=NULL)
{ d = first(l); n = (*l);
    (*l) = (*l)->next; free(n);
}
return d;

```

```
list makeList()
```

```
{ return NULL;
}
```

```
list newNode()
```

```

{ list l=(node *) malloc(sizeof(node));
if(l==NULL){perror("Creazione nodo fallita"); exit(1);}
else return l;
}

```

IMPLEMENTAZIONE PILA

La pila non è altro che una lista in cui è possibile inserire/estrare elementi solo in cima, quindi l'implementazione sarà molto simile:

```

typedef list stack;
void push(stack *stK, data d){insertHead(d, stK);}
data pop(stack *stK){return disposeHead(stK);}
stack makeStack(){return makeList();}
data top(stack stK){return first(stK);}
bool isEmptyS(stack stK){return isEmptyL(stK);}

```

IMPLEMENTAZIONE CODA

La coda non è altro che una lista formata da due puntatori che puntano rispettivamente alla fine e all'inizio della lista.

FUNZIONI

```

bool isEmptyQ(queue q){return isEmptyL(q.list);}
data first(queue q){return first(q.list);}
void makeQueue(queue*q)
{ q->first=NULL; q->last=NULL; }

```

```

void enQueue(queue*q, data d)
{
    list l = newNode();
    copyData(&l->d, d);
    l->next = NULL;
    if (q->first == NULL) q->first = l;
    else (*q).last->next = l;
    (*q).last = l;
}

```

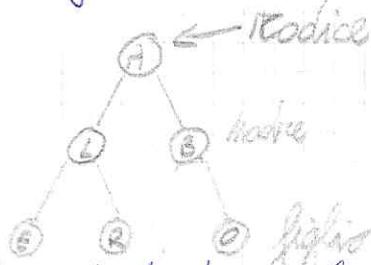
```

data deQueue(queue *q)
{
    return dequeue(q->first);
}

```

ALBERI

Un albero è un tipo di dato in cui i nodi sono collegati gerarchicamente attraverso gli archi.



L'albero è strutturato a livelli, si partono dalla radice (livello 0) fino alle foglie. L'altezza di un albero è il numero di livelli massimo partendo dalla radice. Un albero è detto m-ario quando tutti i nodi, a eccezione delle foglie, hanno m figli.

FUNZIONI ALBERO

```

int numNodi(tree t) // restituisce il numero di nodi
int gradi(tree t) // restituisce il numero di figli di un nodo
tree padre(tree t) // restituisce il padre di un nodo
tree add(tree t) // aggiunge un nodo come figlio di t
tree addTree(tree t) // aggiunge un albero come figlio di t
tree removeTree(tree t) // toglie un albero dal nodo t

```

IMPLEMENTAZIONE

```

typedef struct branch
{
    data d;
    struct branch *left;
    struct branch *right;
} branch;

```

```
typedef branch *tree
```

```
int numNodi(tree t)
```

```

if (t == NULL) return 0;
else if (t->left == NULL && t->right == NULL) return 1;
else return numNodi(t->left) + numNodi(t->right);
}

```

```
int gradi(tree t)
```

```

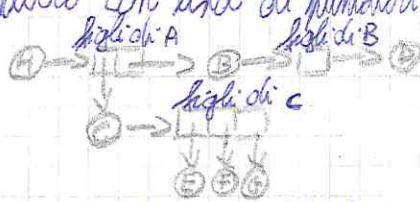
if (t == NULL) return -1;
else if (t->left == NULL && t->right == NULL) return 0;
else if (t->left != NULL && t->right != NULL) return 2;
else return 1;
}

```

GENERALIZZAZIONE DI ALBERI

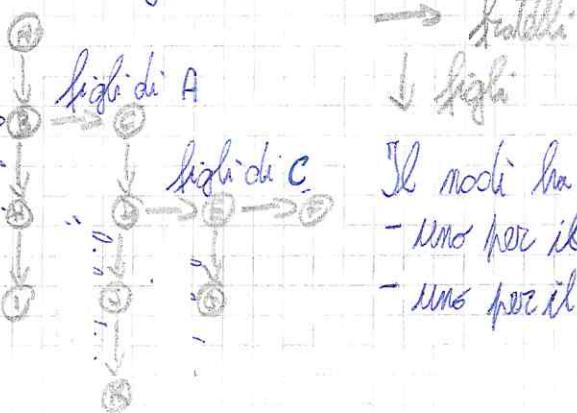
Per semplicità di strutturazione e utilizzo, si generalizza l'albero in due diverse implementazioni:

- Albero con lista di puntatori



Il nodo punta a una lista di puntatori che puntano ai figli.

- Albero figlio - fratello



Il nodo ha due puntatori:

- uno per il figlio;
- uno per il fratello;

VISITA DI ALBERI

La visita di un albero consiste nell'accendere ai nodi/archi in modo sistematico partendo dalla radice fino ad arrivare alle foglie. I diversi algoritmi di visita si distinguono invece dall'ordine d'accordo ai nodi.

Visita generica

L'algoritmo di visita generica utilizza uno stack per contenere i nodi in ogni istante. Se il nodo non è nullo, verrà visitato e vengono inseriti nello stack quelli successivi.

visitagen(tree t)

```

    stack s;
    s.push(t);
    while (!s.isEmpty())
    {
        m = s.pop();
        if (m != NULL)
        {
            visita(m);
            s.push(m.left);
            s.push(m.right);
        }
    }

```

Complessità: $O(n)$

Ordine Visita: ABRELO

La visita fatta in questo modo è anche detta visita in preordine perché visita il nodo e dopo inserisce i figli nello stack.

Visita in ampiezza

La visita in ampiezza consiste nel visitare i nodi solo se quelli al livello successivo non già stati visitati.

Post Visita (tree t)

```

queue q;
q.enqueue(t);
while (!q.isEmpty())
{
    m = q.dequeue();
    if (m != NULL)
    {
        postVisita(m->left);
        postVisita(m->right);
        visita(m);
    }
}

```

Ordine Visita: ERLOBA

simVisita(tree t)

```

queue q;
q.enqueue(t);
while (!q.isEmpty())
{
    m = q.dequeue();
    if (m != NULL)
    {
        simVisita(m->left);
        visita(m);
        simVisita(m->right);
    }
}

```

Ordine Visita: ELRABO

Confronto

tipo	risultato	costo
pre ordina	ALERBO	$O(n)$
simmetrica	ELRABO	$O(n)$
postordine	ERLOBA	$O(n)$

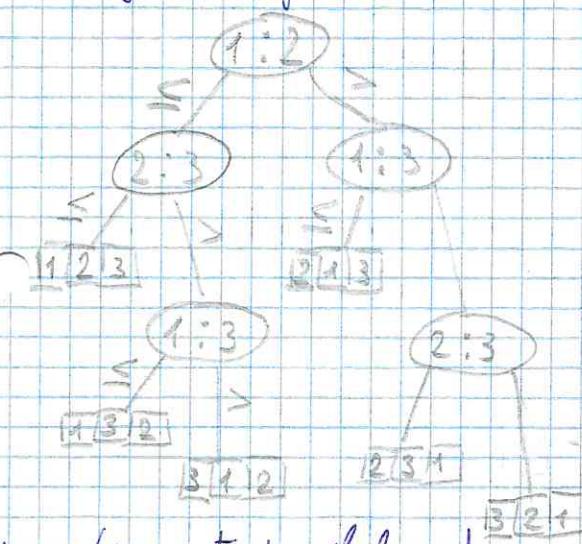
ORDINAMENTO

Gli algoritmi d'ordinamento sono algoritmi che permettono ad array, liste e altri tipi di strutture dati di essere ordinate, in questo modo è possibile utilizzare determinate funzioni che, grazie ad esso, permettono di diminuire il costo computazionale.

NOTA: NON ESISTE algoritmo la cui complessità è inferiore a $n \log n$.

funzionamento

array di lunghezza n $\boxed{1} \boxed{2} \boxed{3}$



ordinamenti possibili: $m!$

confronti: $\Omega(m \log m)$ CASC PEGGIORE

l'altezza di quest'albero è maggiore al numero di nodi dell'albero. Cogni nodo ha entrambi 2 figli.

$$h \geq \log_2 K$$

se l'albero ha un solo nodo, allora

$h=0$, quindi:

$$0 \geq \log_2 K \rightarrow K=1$$

di conseguenza, se erominiamo il sottoalbero della radice:

$$h' = \log_2 \frac{k}{2} = \log_2 K - \log_2 2 = \log_2 K - 1$$

quindi:

se $h=h'+1$ allora:

$$h=h'+1 \geq 1 + \log_2 K + 1$$

In conclusione:

$$h \geq \log_2 K \rightarrow h \geq \log_2 m!$$

Complessità

$$T(n) \geq \log_2 m!$$

$$\begin{aligned} \log_2 m! &= \log_2 (1, 2, \dots, m) = \\ &= \log_2 1 + \log_2 2 + \dots + \log_2 \frac{m}{2} + \dots + \log_2 m \end{aligned}$$

Butto via i primi $\frac{m}{2}-1$ logaritmi perché influiscono sul risultato:

$$\log_2 \frac{m}{2} + \dots + \log_2 m \geq \frac{m}{2} \log_2 \frac{m}{2} \rightarrow \Omega(m \log m)$$

NOTA: la somma degli $\frac{m}{2}$ logaritmi è maggiore/uguale alla somma del primo elemento $\frac{m}{2}$ volte (in questo caso il primo elemento è $\frac{m}{2}$).

Algoritmi Quadratici

Gli algoritmi quadratici sono algoritmi la cui complessità è $O(n^2)$, ne esistono di 2 gruppi:

- algoritmi incrementali: ottenuti attraverso la tecnica induttiva.
- algoritmi a bolle: vengono confrontati l'elemento i -esimo e il successivo.

SELECTION SORT

Il Selection Sort prende l'elemento più piccolo e lo sposta in prima posizione, dopodiché lo esclude e lavora sul resto dell'array.

selectionSort(int a[], int n)

for $i=0$ to $n-1$:

$$m = i+1$$

for $j=i+2$ to n :

$$\text{if } (a[j] < a[m]) m=j$$

scambia $a[i:m]$ con $a[i+1:m]$

COMPLESSITÀ

L'algoritmo Selection Sort scorre l'intero Selection Sort scorre continuamente l'array tutto l'array dalla posizione i contando una posizione in più ogni volta, a ogni ciclo, quindi:

$$\begin{aligned} T(n) &= [m - (0+2)+1] + \dots + [m - (m-2+2)+1] = \\ &= \sum_{i=0}^{m-2} m - i - 2 - 1 = \sum_{i=0}^{m-2} m - 1 - \sum_{i=0}^{m-2} i = \\ &= (m-1)^2 - \frac{1}{2}(m-2)(m-1) = \\ &= m^2 + 1 - \frac{1}{2}(m^2 - 3m + 2) = \\ &= \frac{m^2}{2} - \frac{3m}{2} \rightarrow O(m^2) \end{aligned}$$

COMPLESSITÀ

$$T(n) = \sum_{i=0}^{m-1} i = \frac{1}{2}m(m-1) = \frac{m^2}{2} - \frac{1}{2} \rightarrow O(m^2)$$

RAGIONAMENTO

- 1) prendo il valore successivo a i ;
- 2) deiscolemento J fino a quando $a[J]$ è maggiore del valore prece;
- 3) se i è minore di J , sposta a sinistra tutti gli elementi da i a $J+1$;
- 4) assegna alla casella $J+1$ l'elemento prece.

RAGIONAMENTO

- 1) confronto elemento in J con elemento in m , se il primo è minore allora m diventa uguale a J ;
- 2) ricopre il punto 1 fino alla fine del ciclo;
- 3) scombihi elementi in m con l'elemento in $i+1$;
-) ricopre i primi 3 punti fino a quando $i <= m-2$

BUBBLE SORT

Il bubble sort confronta l'elemento i -esimo col successivo scombinandoli quando il primo è più grande del secondo, in questo modo gli elementi più grandi vengono spostati alla destra dell'array.

bubbleSort (int a[], int n)

for $i = 0$ to $n-1$

 for $J = 1$ to $n-i-1$

 if ($a[J-1] > a[J]$)

 scombihi elemento in $J-1$ con quello in J

COMPLESSITÀ

Il bubble sort scorre gli elementi da 0 a n considerandone uno in meno a ogni ciclo, quindi:

$$T(n) = \sum_{i=0}^{n-1} n = n^2 \rightarrow O(n^2)$$

RAGIONAMENTO

- 1) confronta l'elemento i -esimo col successivo, se il primo è maggiore, li scombihi;
- 2) ricopre il punto 1 con la coppia successiva;
- 3) quando arriverà alla fine, ricorda l'array dall'inizio senza considerare l'ultimo elemento.

INSERTION SORT

l'algoritmo insertion sort adotta il cosiddetto approccio incrementale: si estende l'ordinamento da K a $K+1$ elementi e si posiziona quest'ultimo nella posizione corretta rispetto ai primi.

insertionSort (int a[], int n)

for $i = 1$ to $n-1$

$x = a[i+1]$

 for $J = i$ down to 0

 if ($J > 0$ e $a[J] \leq x$) break

 if ($J < i$) for $t = i$ down to $J+1$

$a[t+1] = a[t]$

$a[J+1] = x$

Algoritmi a complessità logaritmica

Gli algoritmi logaritmici sono algoritmi **COMPLESSITÀ**

la cui complessità equivale a $O(\log n)$ # confronti nel caso peggiore: $n - 1$
 nel caso migliore, solitamente utilizziamo una tecnica chiamata "Divide et impera", cioè si divide il problema principale in sottoproblemi più piccoli (solitamente grandi la metà) e di essi si trovano le relative soluzioni.

Dato che l'algoritmo esegue due chiamate ricorsive che lavorano sul metà array, si può dire

$$T(n) = 2T(n/2) + n - 1$$

Applicando il teorema Master

$$n^{\log_2 2} = n^{\log_2 2} = n \quad \text{CASO DUE}$$

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

MERGE SORT

Il mergesort è un algoritmo che utilizza la tecnica spiegata in precedenza e permette di ordinare un array in un tempo $O(n \log n)$.

```
mergeSort (int a[], int i, int l)
  if (i < l)
    m = (i+l)/2
    mergeSort (a, i, m)
    mergeSort (a, m+1, l)
    merge (a, i, m, l)
```

La funzione merge alla fine dell'algoritmo è quella che realizza l'ordinamento vero e proprio (**IMPERA**) mentre le due chiamate successive dividono l'array a metà (**DIVIDE**)

```
merge (int a[], int i, int m, int l)
  int l-i & -i+1]
```

$$j = i, k = m+1, o = 1$$

while ($i \leq m$ e $k < l$)

if ($a[i] < a[j]$) $b[o++]$ = $a[i]$

else $b[o++]$ = $a[j]$

if ($i \leq m$) copy (a, i, m, b)

else copy ($a, m+1, l, b$)

copy (a, i, l, b)

RAGIONAMENTO

- 1) Divido l'array a metà fino a quando non li ho in singole celle;
- 2) Confronto ogni valore della metà d'array con ogni valore dell'altra, inserisco quello minore nel nuovo array e passo al successivo;
- 3) Quando una delle due metà d'array finisce, copio quella rimasta nel nuovo array;
- 4) Copio tutti gli elementi nel nuovo array in quello vecchio.

QUICK SORT

L'algoritmo quicksort è un algoritmo che ordina in tempo $\Theta(n \log n)$ nel caso migliore e $\Theta(n^2)$ in quelli peggiori utilizzando la tecnica Divide et Impera.

```
quicksort (int a[], int i, int l)
  if (i < l)
```

$$m = \text{partition}(a, i, l)$$

quicksort (a, i, m-1)

quicksort (a, m+1, l)

La funzione partition seleziona un elemento **pivoto** e sposta gli elementi maggiori a destra e minori/uguali a sinistra (**IMPERA**), le due chiamate ricorsive lavoreranno rispettivamente a sinistra e a destra del pivoto.

NOTA: l'unico vantaggio del mergesort è che utilizza un secondo array per ordinare.

partition (int a[], int i, int l)

$$x = a[i], \text{inf} = i + 1, \text{sup} = l$$

while ($\text{inf} < \text{sup}$)

 while ($\text{inf} \leq l$ e $a[\text{inf}] \leq x$) $\text{inf}++$

 while ($a[\text{sup}] > x$) $\text{sup}--$

 if ($\text{inf} < \text{sup}$) swap ($a[\text{inf}], a[\text{sup}]$, inf)

swap ($a[i], a[\text{sup}]$)

return sup

JOTA: Il più grande svantaggio del quicksort è la sua complessità nel caso peggiore ($\Theta(n^2)$), tuttavia questo problema è risolvibile randomizzando il valore del perno.

COMPLESSITÀ

confronti: $n - 1$

CASO MIGLIORE

Il caso migliore si verifica quando divido sempre a metà, quindi:

$$T(n) = 2T\left(\frac{n}{2}\right) + n - 1 \rightarrow \Theta(n \log n)$$

CASO PEGGIORE

Il caso peggiore si verifica quando il perno scelto è il massimo/minimo valore dell'array, quindi: Caso il livello in cui si trovano non sia completo

$$T(n) = T(n-1) + T(0) + n - 1$$

SROTOLAMENTO

$$T(n) = \sum_{i=0}^{n-1} n - \sum_{i=0}^{n-1} i = n^2 - \frac{1}{2}n(n-1) =$$

$$= \frac{n^2}{2} - \frac{n}{2} \rightarrow T(n) = \Theta(n^2)$$

CASO MEDIO

Il caso medio si verifica quando l'array viene diviso con un punto intermedio:

$$T(n) = \frac{1}{m} \sum_{a=0}^{m-1} (n-1 + T(a) + T(n-a-1)) =$$

$$= \frac{1}{m} \sum_{a=0}^{m-1} n-1 + \frac{1}{m} \sum_{a=0}^{m-1} T(a) + \frac{1}{m} \sum_{a=0}^{m-1} T(n-a-1) =$$

$$= n-1 + \frac{1}{m} \sum_{a=0}^{m-1} T(a) + \frac{1}{m} \sum_{a=0}^{m-1} T(n-a-1)$$

$$T(a) = a \alpha \log a \rightarrow T(n) \leq n-1 + \frac{2}{m} \sum_{a=0}^{m-1} a \alpha \log a$$

$$\leq \alpha \alpha \log m$$

RAGIONAMENTO

- 1) si prende il primo valore dell'array;
- 2) incremento inf fino a quando non trovo un valore maggiore del perno o diviso alla fine;
- 3) decremento sup fino a quando non trovo un valore minore del perno;
- 4) se inf è minore di sup , scambio i valori delle relative celle;
- 5) scambio il valore della cella iniziale con quello di sup ;
- 6) lavoro nelle parti d'array a destra e a sinistra di sup .

HEAP SORT

L'algoritmo heapsort è un algoritmo che, grazie all'utilizzo di strutture dati efficienti, permette di ordinare usando l'approccio incrementale del selection sort con tempi minori.

L'algoritmo lavora su una struttura dati chiamata heap, un albero binario completo almeno fino all'ultimo livello, essa è anche una funzione

riparato: i nodi stanno tutti a sinistra nel senso il livello in cui si trovano non sia completo
L'heap è anche rappresentabile attraverso un vettore: i figli di una cella i corrispondono alle celle $2i+1$ e $2i+2$.



Prima di eseguire l'ordinamento vero e proprio, l'heap deve essere preparato attraverso la funzione `heapsify()`:

`heapsify(heap h)`

`heapsify(liglio h sinistro)`

`heapsify(liglio h destro)`

`fixHeap(radice di h, h)`

La funzione `fixHeap` estrae il valore massimo dell'heap e lo porta alla radice.

$\text{fixHeap}(\text{tree } t, \text{ heap } h)$
 if (t non è foglia)
 $m = \text{figlio di } t \text{ con chiave minima}$
 if ($\text{chiave}(m) < \text{chiave}(t)$)
 scambia le due chiavi
 $\text{fixHeap}(m, h)$

COMPLESSITÀ

confronti: $O(h) \rightarrow h = \log n$

nodi: $\sum_{i=0}^{h-1} 2^i \leq n \leq \sum_{i=0}^h 2^i$

quindi si può dire che:

$$\frac{2^h - 1}{2 - 1} \leq n \leq \frac{2^{h+1} - 1}{2 - 1}$$

$$2^h - 1 \leq n \leq 2^{h+1} - 1$$

$$2^h \leq n + 1 \leq 2^{h+1}$$

$$h \leq \log_2(n+1) \leq h+1$$

h e $h+1$ hanno la stessa crescita, quindi:

$$h = O(\log n)$$

COMPLESSITÀ HEAPIFY: $O(n)$

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

RAGIONAMENTO

- 1) heapify: visita tutto l'albero spostando adeguatamente il valore massimo al livello superiore;
- 2) sposta il valore all'inizio dell'array;
- 3) applica la fixHeap senza considerare l'ultima cella dell'array.

ALGORITMI LINEARI

Gli algoritmi lineari sono algoritmi la cui complessità è esattamente $O(n)$.

INTEGER SORT

L'integer sort è un algoritmo che utilizza un array di contatori per ordinare. Questo però permette di ordinare array di interi da 0 a $n+1$.

$\text{integerSort(int a[], int n)}$
 int b[n];
 for i=0 to n-1
 b[i]=0;
 for i=0 to n-1
 b[a[i]]++
 j=0
 for i=0 to n-1
 while (b[i] != 0)
 a[j++] = i;
 b[i]--

COMPLESSITÀ

Se l'array da ordinare comprende solo interi da 0 a $n-1$, la complessità è $O(n)$. Nel caso generale (con valori superiori a n) la complessità è $O(k+n)$.

RAGIONAMENTO

- 1) incremento adeguatamente le celle dell'array di contatori;
- 2) assegno alle celle dell'array iniziale l'indice i in base a quante volte è ripetuto.

BUCKET SORT

L'algoritmo bucket sort è un algoritmo che sfrutta lo stesso principio dell'integer sort, l'unica differenza è che viene creato un array di liste al posto di uno di puntatori.

$\text{BucketSort(int a[], int m, int K)}$
 list b[K]
 for i=0 to K-1 b[i]=NULL
 for i=0 to m-1
 aggiungi a b[i] un nodo contenente a[i]
 for i=0 to K
 copia il valore dei nodi della lista b[i] in a

COMPLESSITÀ: $O(m+K)$ come integerSort

RAGIONAMENTO: analogo a integerSort

RADIXSORT

Il radixsort è un algoritmo tabù che ordina sfruttando la cifra $b-1$ -ima in cui i nodi sono ordinati in modo da stare a sinistra quelli minori e a destra quelli maggiori, in questo modo è possibile creare funzioni più efficienti.

radixSort (int a[])

$t = 0$

while ($\exists m : m[t] \neq 0$)

 bucketSort (a, 10, t++)

 bucketSort (int a[], int b, int t)

 y [b]

 for $i = 0$ to $b-1$ $y[i] = \text{NULL}$

 for $i = 0$ to $a.length$

 c = cifra ($a[i], t$)

 aggiungo a y alla lista $y[c+1]$

 for $i = 0$ to b

 copio gli elementi della lista $y[i]$ in a

COMPLESSITÀ

Il radixsort compie $O(\log_b K)$ chiamate parate di bucketSort, ognuna delle quali richiede un tempo $O(n+b)$, quindi:

$$T(n) = O((n+b) \log_b K)$$

se $K = O(n^c)$ con c costante, l'algoritmo è lineare.

RAGIONAMENTO

- 1) estraggo la cifra t -esima della cella i ;
- 2) inserisco la cella i nella lista della cella cifra + 1;
- 3) copio le liste ordinatamente nell'array;
- 4) ripeto finché ci sono ancora cifre t -esime.

ALBERI BINARI DI RICERCA (ABR)

Un ABR è essenzialmente un albero binario in cui i nodi sono ordinati in modo da stare a sinistra quelli minori e a destra quelli maggiori, in questo modo è possibile creare funzioni più efficienti.

RICERCA DI UN NODO

Dato la struttura dell'ABR, la ricerca funziona confrontando il valore da cercare con quello in radice: se è uguale, finisce la ricerca, altrimenti si prosegue ricorsivamente nel sottoalbero adeguato. Si può concludere che la ricerca, a ogni chiamata ricorsiva, tiene $\frac{m}{2}$ nodi, quindi si ha che:

$$T(m) = 2T\left(\frac{m}{2}\right) + 1 = \Theta(m)$$

INSERIMENTO DI UN NODO

Ogni nuovo elemento dell'albero viene aggiunto come foglia in una posizione specifica, sfruttando le proprietà dell'ABR:

- 1) si confronta il nuovo elemento con la radice
- 2) se andiamo nel sottoalbero adeguato
- 3) se il nodo da confrontare è una foglia, creiamo un nuovo nodo a destra/minore a seconda del valore.

Dato che si percorre l'albero dall'inizio alla fine per realizzare quest'operazione, la complessità è $O(h)$ che equivale a $O(n)$ nel caso peggiore.

CANCELLAZIONE DI UN NODO

Se il nodo da cancellare è una foglia, l'operazione si realizza in tempo costante ($O(1)$), altrimenti:

- se il nodo da cancellare ha un solo figlio lo si collega al "nonno" e si dealloca il nodo
- se ha due figli, si copia il valore del suo predecessore (il minimo del sottoalbero sinistro) nel nodo e poi si cancella la foglia corrispondente. In questi ultimi due casi, la complessità è $O(h)$ aggiornato a $O(n)$.

ALBERI AVL

Gli alberi AVL sono ottimizzazioni degli ABR in cui l'altezza bilanciata, permettendo così di ottenere il costo migliore di una funzione ABR più frequentemente.

Gli AVL sono ottimizzati grazie al fattore di bilanciamento, esso è dato dalla differenza delle altezze del sottoalbero sinistro e destro. Quando il fattore di bilanciamento è minore o uguale a 1, l'albero è bilanciato, altrimenti si ribilancia l'albero con le rotazioni.

ROTAZIONI

Le rotazioni permettono a un albero AVL di rimanere bilanciato, ne esistono due tipi:

- Rotazione di base: ribilancia l'albero ruotando verso il sottoalbero con altezza minore, eventuali nodi "orfani" vengono riadegnoti seguendo le proprietà dell'ABR;

- Rotazione doppia: ribilancia l'albero eseguendo prima una rotazione preparatoria su un sottoalbero di un sottoalbero e dopo con quella stessa e propria verso il sottoalbero da ribilanciare. Questa tecnica si utilizza solitamente quando la rotazione di base non si utilizzasse oppure peggiorerebbe la situazione.

OPERAZIONI AVL

Le operazioni di un albero AVL sono le stesse di un ABR, tuttavia le funzioni d'inserimento e cancellazione di nodi sottoalbero obbligano a bilanciare l'albero e quindi hanno bisogno d'ulteriori controlli.

ALBERI DUE-TRE

Gli alberi due-tre sono tipi di alberi, simili agli ABR, in cui un nodo può aver un massimo di tre figli un minimo di due. Nel caso un nodo abbia tre figli, esso avrà due valori all'interno:

- il primo corrisponde al massimo del sottoalbero sinistro;
- il secondo corrisponde al massimo del sottoalbero centrale.

PROPRIETÀ

$$1) 2^{h+1} \leq n \leq \frac{3^{h+1}-1}{2}$$

$$h = h-1, m = m - \ell$$

$$2\ell \leq \ell \leq 3\ell$$

per ipotesi induttiva, si ha che:

$$2^h \leq \ell \leq 3^h \rightarrow 2^h \cdot 2 \leq \ell \leq 3^h \cdot 3$$

$$2^h \leq \ell \leq 3^h \leftarrow |$$

$$2^{h+1} - 1 \leq m + \ell \leq \frac{3^{h+1}-1}{2}$$

$$2^h - 1 \leq m \leq \frac{3^h - 1}{2}$$

$$2^h + 2^h - 1 \leq m + \ell \leq \frac{3^h - 1}{2} + 3^h$$

$$2^{h+1} - 1 \leq m + \ell \leq \frac{3^{h+1}-1}{2}$$

$$\text{se } m \geq 2^h \text{ e } \ell \geq 2^h \rightarrow m + \ell \geq 2^{h+1}$$

$$\text{se } m \leq \frac{3^h - 1}{2} \text{ e } \ell \leq 3^h \rightarrow m + \ell \leq \frac{3^{h+1}-1}{2}$$

quindi $h = O(\log n)$

OPERAZIONI

L'operazione search (di complessità $O(h)$) è molto simile a quella dell'ABR con la differenza che deve anche gestire il sottoalbero centrale se presente. L'insert e la delete (anch'esse $O(h)$) non solo modificare la struttura dell'albero unendo o dividendo i nodi, in più, a causa di questo, l'altezza dell'albero può aumentare o diminuire.

TABELLE AD ACCESSO DIRETTO

Le tabelle ad accesso diretto sono strutture di grossa capacità m in cui ogni elemento ha una chiave associata da 0 a $m-1$. Ogni elemento con chiave K viene sempre messo nella posizione K dell'array. Le operazioni hanno tutto costo $O(1)$.

Il fattore di bilanciamento indica lo spazio di memoria e viene ricavato rapportando il numero d'elementi inseriti con la dimensione della tabella.

$$\alpha = \frac{n}{m}$$

Nonostante il costo delle funzioni, lo spazio di memoria, proporzionale a m , può produrre un grande spazio di memoria, in più le chiavi devono essere necessariamente interi da 0 a $m-1$.

TABELLE HASH

Le tabelle hash sono la soluzione ai problemi della tabella ad accesso diretto, infatti le chiavi possono anche non essere numeri e vengono calcolate attraverso la funzione hash. Quindi, dato un elemento con chiave K , esso verrà inserito nella cella $h(K)$. La funzione hash perfetta è iniettiva: da due chiavi differenti ottengono celle differenti.

COLLISIONI

Le tabelle hash possono soffrire del problema delle collisioni, esse si verificano quando un elemento di chiave K da inserire trova una cella già occupata, la quale vedrebbe sovrascritta.

VANTAGGI

Grazie alle funzioni hash, è possibile migliorare il fattore di bilanciamento, oltre alla possibilità d'utilizzare più tipi di chiavi.

UNIFORMITA DELLE FUNZIONI HASH

Per abbattere il numero di collisioni e lo spazio di memoria, si "rompe" il vincolo d'iniettività e si gestiscono tutte le collisioni in modo aperto.

Per gestire le collisioni, esistono due differenti:

- liste di collisione: ogni elemento di chiave K che collida nella cella $h(K)$ viene aggiunto in testa alla lista della medesima;
- indirizzamento aperto: ogni elemento di chiave che collida nella cella $h(K)$ ricalcola una nuova posizione

LISTE DI COLLISIONE

le liste di collisione aggiungono un elemento di chiave K in testa alla lista della cella $h(K)$ quando uno collide.

ESEMPIO: memorizziamo ogni lettera di BAMBOLA:



COMPLESSITÀ

spazio: $\Theta(mn) + \Theta(n)$

tempo:

- insert $\rightarrow O(1)$
- search (uniforme) $\rightarrow O(1 + \frac{m}{n})$
- search (generale) $\rightarrow O(n)$
- delete si appoggia a search

INDIRIZZAMENTO APERTO

la funzione hash ricalcola l'indice della cella quando l'elemento collide. La cella può essere trovata in due modi:

- scansione lineare: a ogni collisione la funzione controlla la cella immediatamente successiva;
- hashing doppio: utilizza lo stesso principio di scansione lineare con la differenza che utilizza due funzioni hash.

D-HEAP

Il d-heap è un'implementazione delle code con priorità ed è rappresentato:

- ~ da un albero radicato d-ario.

PROPRIETÀ

Un d-heap ha struttura completa fino al penultimo livello, ogni nodo ha un elemento e una chiave presa da un dominio ordinato e le chiavi dei padri sono minori o uguali rispetto a quelle dei figli.

L'altezza di un d-heap equivale a $\lceil \log_d n \rceil$, la radice contiene l'elemento di chiave minima e deve essere rappresentata tramite un avvertito particolare.

FUNZIONI D-HEAP

insert $\rightarrow O(\log_d n)$

delete $\rightarrow O(d \log_d n)$

findMin $\rightarrow O(1)$

deleteMin si appoggia a delete e findMin

increaseKey $\rightarrow O(d \log_d n)$

decreaseKey $\rightarrow O(\log_d n)$

FUNZIONI AUSILIARIE

Servono per riadattare le proprietà del dheap:

~ muoviFiglio(tree t)

while (t non è radice e $t < \text{padre}(t)$)

scambia t con suo padre

muoviBono(tree t)

do

$u = \text{figlio di } t \text{ con chiave minima}$

if (u esiste)

scambia t con u

while (t ha figli e $t > u$)

