

ALGORITMI  
E STRUTTURE  
DATI

# ANALISI DI ALGORITMI

## TEMPI DI CALCOLO

I problemi da risolvere hanno una dimensione che dipende dalla grandezza dei dati in ingresso in funzione alle operazioni ad esse collegate.

Sono considerate operazioni elementari: operazioni logiche, aritmetiche e di confronto.

Il costo è generalmente valutato nel caso pessimo.

## ORDINE DI GRANDEZZA E COMPLESSITÀ

Il tempo di calcolo di  $T(n)$  di una procedura avviene valutando il numero di operazioni in ordine di grandezza: funzione che limita  $T(n)$  al tendere all'infinito della dimensione  $n$ , trascurando costanti additive e moltiplicative. Si usano nozioni:

- (alpha)  $\mathcal{O}(f(n))$ : si dice che "una funzione  $g(n)$  è di ordine  $\mathcal{O}(f(n))$ " per indicare una limitazione SUPERIORE al comportamento asintotico di  $g(n)$
- (omega)  $\mathcal{\Omega}(f(n))$ : se una funzione  $g(n)$  è di ordine  $\mathcal{\Omega}(f(n))$  indica una limitazione INFERIORE
- (theta)  $\mathcal{\Theta}(f(n))$ : se una funzione  $g(n)$  è di ordine  $\mathcal{\Theta}(f(n))$  indica sia una limitazione SUPERIORE che INFERIORE

## ESEMPIO - Classificazione degli ordini di grandezza

- |   |  |
|---|--|
| - $\mathcal{\Theta}(1)$ : costante              | - $\mathcal{\Theta}(n^2)$ : quadratica                   |
| - $\mathcal{\Theta}(\log n)$ : logaritmico      | - $\mathcal{\Theta}(n^3)$ : cubico                       |
| - $\mathcal{\Theta}(n)$ : lineare               | - $\mathcal{\Theta}(2^n)$ : esponenziale (base 2)        |
| - $\mathcal{\Theta}(n \log n)$ : pseudo lineare | - $\mathcal{\Theta}(n^{\alpha})$ : esponenziale (base n) |

# ALGORITMI EFFICIENTI ED INEFFICIENTI

esempio con vettore  $A = [7, 5, 3, 1, 8, 3, 5]$

insertionSort(ITEM[] A, integer n)

```

for integer i ← 2 to n do
    ITEM temp ← A[i]
    integer j ← i
    while j > 1 and A[j - 1] > temp do
        A[j] ← A[j - 1]
        j ← j - 1
        A[j] ← temp
    
```

i=1 2 3 4 5 6 7

7 ⑦ 2 1 8 3 5

5 7 ② 1 8 3 5

2 5 7 ① 8 3 5

1 2 4 7 ⑧ 3 5

1 2 4 7 8 ③ 5

1 2 3 4 7 8 ⑤

1 2 3 4 5 7 8

Iterationi ciclo for

O = elemento puntato da i

Partendo dal secondo elemento in poi, copiato in "tmp", si cerca di far traslare a destra ogni elemento maggiore di "tmp", e successivamente, inserire "tmp" nel primo posto in cui non è possibile spostare un valore a destra, ovvero quando termina il ciclo while.

COSTO :  $O(n^2)$  nel caso peggiore con A ordinato alla rovescia

$O(n)$  nel caso migliore con A già ordinata.

il costo è quindi  $O(n^2)$  nel caso peggiore e  $O(n)$  nel caso migliore.

$$(n+1) \cdot O(n) = O(n^2)$$

# COMPLESSITÀ DI PROBLEMI E ALGORITMI

Se si riesce a dimostrare che qualunque algoritmo per il problema in esame deve avere complessità  $\Omega(f(n))$ , allora si è stabilito una complessità inferiore al problema.

Analogamente se si trova un particolare algoritmo con complessità  $O(g(n))$ , allora si è stabilito una complessità superiore al problema.

Se  $g(n) = f(n)$  allora la soluzione al problema è detta ottima.

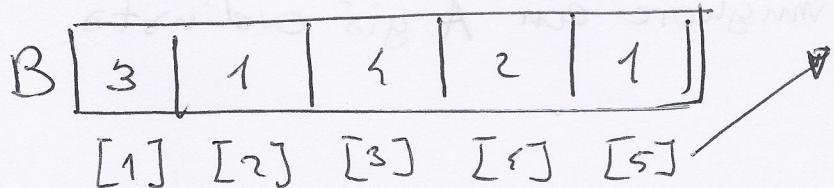
## COUNTING SORT

countingSort(ITEM[] A, integer n, integer k)

```
integer i, j
integer[] B ← new integer[1 ... k]
for i ← 1 to k do B[i] = 0
for i ← 1 to n do B[A[j]] ← B[A[j]] + 1
j ← 1
for i ← 1 to k do
    while B[i] > 0 do
        A[j] ← i
        j ← j + 1
        B[i] ← B[i] - 1
```

Gli elementi da ordinare sono contenuti nel vettore A e sono compresi fra i valori 1 e K.

L'Algoritmo inizialmente scandisce ogni elemento di A e segna nel vettore d'appoggio B[1..K] le occorrenze dei valori trovati.



il valore 5 trovato 1 volta.

Successivamente scrivere nel vettore A il contenuto di  $B[i]$ ,  $i=1..K$ , " $i$ " volte.

costo :  $O(n+k)$

## ANALISI PER LIVELLI

Una tecnica di risoluzione delle ricorrenti è analizzare il costo ad ogni livello, esempio:

$$\begin{cases} T(n) = 1 & n=1 \\ T(n) = 4T(n/2) + n^2 & n = 2^h, h > 0 \end{cases}$$

Ad ogni livello il costo da pagare è dato dalla componente  $n^2$ , ma varia in dimensione durante le chiamate ricorsive.

Dopo la prima chiamata,  $n$  è divisa in 4 parti di dimensione  $n/2$ , quindi si applica  $n/2$  al termine da pagare ad ogni livello, che da origine a  $(n/2)^2 = n^2/2^2$ . Alla seconda chiamata ricorsiva,  $n$  è stato diviso in 16 parti di dimensione  $(n/4)^2$ .

In generale al livello  $i$ -esimo la dimensione è divisa in  $4^i$  parti di costo  $(n/2^i)^2 = n^2/2^{2i}$ .

La ricorsione termina al livello  $h = \log_2 n$  ovvero quando  $n/2^h = 1$ .

LIVELLO	ESPRESSIONE	COSTO	COSTO
0	$n^2$	$n^2$	$n^2$
1	$(n/2)^2$	$(n/2)^2$	$(n/2)^2$
2	$(n/4)^2 \dots (n/4)^2$	$(n/4)^2 \dots (n/4)^2$	$(n/4)^2 \dots (n/4)^2$
$i$	$\frac{n^2}{2^i}$	$\frac{n^2}{2^i}$	$\frac{n^2}{2^i}$
$h$	$T(1)$	$T(1)$	$T(1)$

Sommando i costi si ottiene

$$\sum_{i=0}^{\log_2 n - 1} n^2/2^{2i} \cdot 4^i = n^2 \sum_{i=0}^{\log_2 n - 1} \frac{2^{2i}}{2^{2i}} = n^2 \sum_{i=0}^{\log_2 n - 1} 1 = n^2 \log n$$

Da cui si ricava:  $T(n) = n^2 + n^2 \log n = O(n^2 \log n)$

## RICORRENTE LINEARI

Spesso alcune relazioni di ricorrenza, richiamano se stesse un numero di costante di voto, su dati di dimensione  $n-i$ , con i costante.

In questo caso si può generalizzare con:

$$T(n) = \sum_{i=1}^{d_i} a_i T(n-i) + c n^b \quad \begin{array}{l} d_i > 1 \\ i > 1 \\ c > 0 \\ b \geq 0 \end{array}$$

Con il polinomio  $c n^b$  si indicano le operazioni nel livello di ricorsione.

Una relazione di ricorrenza di questo tipo è detta LINEARE A COEFFICIENTI COSTANTI.

## TEOREMA - MASTER THEOREM

Siano  $a_1, a_2, \dots, a_h$  costanti intere non negative,  $h$  costante positiva,  $c < b$  costanti reali t.c.  $c > 0$ ,  $b \geq 0$  e  $T(n)$  definito

$$\begin{cases} T(n) = \text{const} & n \leq m \leq h \\ T(n) = \sum_{1 \leq i \leq h} a_i T(n-i) + c n^b & n > m \end{cases}$$

Ponendo  $\alpha = \sum_{1 \leq i \leq h} a_i$

(1)  $T(n)$  è  $O(n^{b+1})$  se  $\alpha = 1$

(2)  $T(n)$  è  $O(\alpha^n n^b)$  se  $\alpha > 1$

$$(n \log n)^{\alpha} = n^{\alpha} \log^{\alpha} n + n^{\alpha} = (n^{\alpha}) \cdot \log^{\alpha} n$$

Gli algoritmi che invece dividono il problema originario di dimensione  $n$  in  $q$  sottoproblemi di dimensione  $n/b$ , sono detti divide-et-impara. Sono espresse in termini di:

$$T(n) = \begin{cases} q > 1 \\ b > 2 \\ c > 0 \\ \beta > 0 \end{cases} T(n/b) + cn^\beta$$

Queste ricorrenze lineari sono definite: LINARI A PARTIZIONI BILANCIATE

### TEOREMA

Siano  $a, b$  costanti intere t.c.  $a \geq 1, b \geq 2 < c, \beta$  costanti reali t.c.  $c > 0$  e  $\beta \geq 0$ . Sia  $T(n)$  data dalla relazione di ricorrenza:

$$\begin{cases} T(n) = d & n=1 \\ T(n) = aT(n/b) + cn^\beta & n > 1 \end{cases}$$

Ponendo  $\lambda = \log a / \log b$  allora

- (1)  $T(n) \in O(n^\lambda)$  se  $\lambda > \beta$
- (2)  $T(n) \in O(n^\lambda \log n)$  se  $\lambda = \beta$
- (3)  $T(n) \in O(n^\beta)$  se  $\lambda < \beta$

# STRUTTURE DATI ELEMENTARI

## LISI CON PUNTATORI

La realizzazione di liste basate su puntatori permette di ottenere complessità  $O(1)$  per quasi tutte le operazioni (con l'esclusione di `prev()` e `tail()` che richiedono  $O(n)$ ) a discapito di un leggero uso di memoria in più.

L'idea di base è di memorizzare una lista di  $n$  elementi in  $n$  record, dette celle, tali che, l' $i$ -esima cella contenga il valore  $i$ -esimo della lista, e l'indirizzo della cella contenente l'elemento successivo e precedente. Se non esistono questi elementi per convenzione vengono settati a `nil`.

LIST		
<code>LIST pred</code>	%Predecessore	<code>boolean finished(Pos p)</code>
<code>LIST succ</code>	%Successore	<code>  return (p = this)</code>
<code>ITEM value</code>	%Elemento	
<code>LIST List()</code>		<code>ITEM read(Pos p)</code>
<code>LIST t ← new List</code> <code>t.pred ← t</code> <code>t.succ ← t</code> <code>return t</code>		<code>  return p.value</code>
 		<code>write(Pos p, ITEM v)</code>
<code>boolean empty()</code>		<code>  p.value ← v</code>
<code>  return pred = succ = this</code>		 
 		<code>Pos insert(Pos p, ITEM v)</code>
<code>Pos head()</code>		<code>LIST t ← List()</code>
<code>  return succ</code>		<code>t.value ← v</code>
 		<code>t.pred ← t.pred</code>
<code>Pos tail()</code>		<code>t.pred.succ ← t</code>
<code>  return pred</code>		<code>t.succ ← p</code>
 		<code>p.pred ← t</code>
<code>Pos next(Pos p)</code>		<code>return t</code>
<code>  return p.succ</code>		 
 		<code>Pos remove(Pos p)</code>
<code>Pos prev(Pos p)</code>		<code>p.pred.succ ← p.succ</code>
<code>  return p.pred</code>		<code>p.succ.pred ← p.pred</code>

## PILE E CODE

Una pila è una sequenza di elementi a cui è possibile aggiungere o togliere elementi solo dalla testa della sequenza. Meccanismo LIFO.

STACK	
ITEM[] A	%Elementi
integer n	%Cursore
integer m	%Dim. Massima
STACK Stack(integer dim)	boolean isEmpty()
STACK t $\leftarrow$ new STACK	return n = 0
t.A $\leftarrow$ new integer [1...dim]	
t.m $\leftarrow$ dim	
t.n $\leftarrow$ 0	
return t	
ITEM top()	ITEM pop()
precondition: n > 0	precondition: n > 0
return A[n]	ITEM t $\leftarrow$ A[n]
	n $\leftarrow$ n - 1
	return t
	push(ITEM v)
	precondition: n < m
	n $\leftarrow$ n + 1
	A[n] $\leftarrow$ v

Una coda è una sequenza di elementi a cui è possibile aggiungerne in fondo e toglierne dalla cima. Meccanismo FIFO.

QUEUE	
ITEM[] A	%Elementi
integer n	%Dim. attuale
integer testa	%Testa
integer m	%Dim. Massima
QUEUE Queue(integer dim)	boolean isEmpty()
QUEUE t $\leftarrow$ new QUEUE	return n = 0
t.A $\leftarrow$ new integer [0...dim - 1]	
t.m $\leftarrow$ dim	
t.testa $\leftarrow$ 0	
t.n $\leftarrow$ 0	
return t	
ITEM top()	ITEM dequeue()
precondition: n > 0	precondition: n > 0
return A[testa]	ITEM t $\leftarrow$ A[testa]
	testa $\leftarrow$ (testa + 1) mod m
	n $\leftarrow$ n - 1
	return t
	enqueue(ITEM v)
	precondition: n < m
	A[(testa + n) mod m] $\leftarrow$ v
	n $\leftarrow$ n + 1

# ALBERI

Sia  $T$  un albero ordinato di  $n$  nodi e di radice  $r$ .  
Sia  $T_1 \dots T_k$  gli insiemi disgiunti non vuoti in cui sono partitionati i rimanenti  $n-1$  nodi di  $T$ , orente come radice  $r_1 \dots r_k$ . Ciascun  $T_i$  è detto sottoalbero di  $T$  radicato in  $r_i$ , mentre ciascun figlio  $r_i$  è detto figlio di  $r$ . Un nodo senza figli è detto foglia, mentre la radice dell'albero è l'unico nodo senza padre. La radice è al livello zero e così via. L'altezza dell'albero è il massimo livello delle sue foglie.

## TREE

NODE <i>parent</i>	% Puntatore al padre
NODE <i>child</i>	% Puntatore al primo figlio
NODE <i>sibling</i>	% Puntatore al successivo fratello
ITEM <i>value</i>	% Valore del nodo
Tree(ITEM <i>v</i> )	% Crea un nuovo nodo
TREE <i>t</i> $\leftarrow$ new TREE <i>t.value</i> $\leftarrow$ <i>v</i> <i>t.parent</i> $\leftarrow$ <i>t.child</i> $\leftarrow$ <i>t.sibling</i> $\leftarrow$ nil return <i>t</i>	
insertChild(TREE <i>t</i> )	% Inserisce <i>t</i> prima dell'attuale primo figlio
<i>t.parent</i> $\leftarrow$ this <i>t.sibling</i> $\leftarrow$ <i>child</i> <i>child</i> $\leftarrow$ <i>t</i>	
insertSibling(TREE <i>t</i> )	% Inserisce <i>t</i> prima dell'attuale fratello
<i>t.parent</i> $\leftarrow$ this <i>t.sibling</i> $\leftarrow$ <i>sibling</i> <i>sibling</i> $\leftarrow$ <i>t</i>	
deleteChild()	
NODE <i>newChild</i> $\leftarrow$ <i>child.rightSibling()</i> delete( <i>child</i> ) <i>child</i> $\leftarrow$ <i>newChild</i>	
deleteSibling()	
NODE <i>newBrother</i> $\leftarrow$ <i>sibling.rightSibling()</i> delete( <i>sibling</i> ) <i>sibling</i> $\leftarrow$ <i>newBrother</i>	
delete(TREE <i>t</i> )	
NODE <i>u</i> $\leftarrow$ <i>t.leftmostChild()</i> while <i>u</i> $\neq$ nil do TREE <i>next</i> $\leftarrow$ <i>u.rightSibling()</i> delete( <i>u</i> ) <i>u</i> $\leftarrow$ <i>next</i> delete <i>t</i>	

# VISITE

La visita di un albero può essere effettuata in vari nodi, detti ordinii: tra cui:

- **ORDINE ANTICIPATO** (previsita): il nodo viene prima esaminato e poi si continua ricorsivamente sui suoi figli
- **ORDINE POSTICIPATO** (postvisita): si scorre l'albero ricorsivamente e poi si effettua l'esame del nodo.

visitaProfondità(TREE $t$ )
<pre> <b>precondition</b> <math>t \neq \text{nil}</math> (1) esame "anticipato" del nodo radice di <math>t</math>     TREE <math>u \leftarrow t.\text{leftmostChild}()</math>     <b>while</b> <math>u \neq \text{nil}</math> <b>do</b>         visitaProfondità(<math>u</math>)         <math>u \leftarrow u.\text{rightSibling}()</math> (2) esame "posticipato" del nodo radice di <math>t</math> </pre>

- (1) esame "anticipato" del nodo radice di  $t$   
TREE  $u \leftarrow t.\text{leftmostChild}()$
- while**  $u \neq \text{nil}$  **do**  
        visitaProfondità( $u$ )  
         $u \leftarrow u.\text{rightSibling}()$
- (2) esame "posticipato" del nodo radice di  $t$

- **ORDINE SIMMETRICA** (invisita): prima viene chiamata ricorsivamente sui figli del nodo considerato, poi si esamina il padre.
- **ORDINE PER LIVELLI** (in ampiezza): procedura non ricorsiva per esaminare i nodi dello stesso livello.

invisita(TREE $t$ , integer $i$ )	visitaAmpiezza(TREE $t$ )
-----------------------------------	---------------------------

```

precontion:  $t \neq \text{nil}$ 
TREE  $u \leftarrow t.\text{leftmostChild}()$ 
integer  $k \leftarrow 0$ 
while  $u \neq \text{nil}$  and  $k < i$  do
     $k \leftarrow k + 1$ 
    invisita( $u$ ,  $i$ )
     $u \leftarrow u.\text{rightSibling}()$ 
    esame "simmetrico" del nodo  $t$ 
while  $u \neq \text{nil}$  do
    invisita( $u$ ,  $i$ )
     $u \leftarrow u.\text{rightSibling}()$ 

```

```

precontion:  $t \neq \text{nil}$ 
QUEUE  $Q \leftarrow \text{Queue}()$ 
 $Q.\text{enqueue}(t)$ 
while not  $Q.\text{isEmpty}()$  do
    TREE  $u \leftarrow Q.\text{dequeue}()$ 
    esame "per livelli" del nodo  $u$ 
     $u \leftarrow u.\text{leftmostChild}()$ 
    while  $u \neq \text{nil}$  do
         $Q.\text{enqueue}(u)$ 
         $u \leftarrow u.\text{rightSibling}()$ 

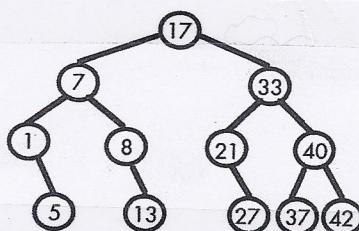
```

# ALBERI BINARI DI RICERCA

Gli alberi sono delle strutture dati più efficienti per la realizzazione di dizionari. Affinché questo possa accadere i nodi dell'albero devono essere dotati di informazioni di tipo chiave-valore.

Un albero binario di ricerca (ABR) verifica le seguenti proprietà:

- (1) per ogni nodo  $v$ , tutte le chiavi contenute nel sottoalbero radicato nel figlio destro di  $v$  sono minori della chiave contenuta in  $v$
- (2) per ogni nodo  $v$ , tutte le chiavi contenute nel sottoalbero radicato nel figlio sinistro di  $v$  sono maggiori della chiave contenuta in  $v$ .



La procedura `LookupNode` è un approssimativo della ricerca su un ABR.

La procedura continua iterativamente fino alle foglie dell'albero ( $T \neq \text{nil}$ ) oppure non si ha trovato la chiave cercata ( $T.\text{key} \neq x$ ), dopo di che decide quali figli analizzare secondo il confronto fatto sulla chiave del nodo e il valore che sto cercando ( $x \leq T.\text{key}$ ): Complessità:  $O(h)$ ,  $h = \text{altezza ABR}$ .

TREE lookupNode(TREE  $T$ , ITEM  $x$ )

```
while  $T \neq \text{nil}$  and  $T.\text{key} \neq x$  do
|    $T \leftarrow \text{iif}(x < T.\text{key}, T.\text{left}, T.\text{right})$ 
return  $T$ 
```

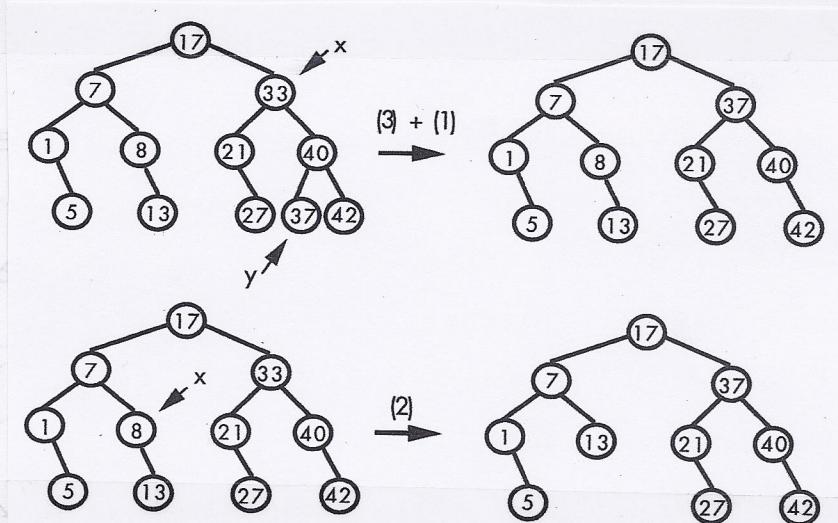
La procedura `insertNode` si occupa di inserire una nuova associazione chiave-valore nell'albero, o di aggiornarla se già presente. Prima esegue una ricerca simile alla `lookupNode` ma invece di restituire `nil` se non viene trovato nulla, restituisce il padre, poi aggiorna il valore o inserisce un nuovo nodo a sinistra o a destra del padre trovato a seconda del valore del nuovo elemento. Complessità  $O(h)$ ,  $h = \text{altezza ABR}$ .

TREE insertNode(TREE  $T$ , ITEM  $x$ , ITEM  $v$ )

```
TREE  $p \leftarrow \text{nil}$  % Padre
TREE  $u \leftarrow T$ 
while  $u \neq \text{nil}$  and  $u.\text{key} \neq x$  do % Cerca posizione inserimento
|    $p \leftarrow u$ 
|    $u \leftarrow \text{iif}(x < u.\text{key}, u.\text{left}, u.\text{right})$ 
if  $u \neq \text{nil}$  and  $u.\text{key} = x$  then % Chiave già presente
|    $u.\text{value} \leftarrow v$ 
else
|   TREE  $n \leftarrow \text{Tree}(x, v)$  % Crea un nodo contenente coppia chiave-valore
|   link( $p, n, x$ )
|   if  $p = x$  nil then return  $n$  % Primo nodo ad essere inserito
return  $T$  % Ritorna albero non modificato
```

Nel caso della rimozione di una chiave  $x$  di un nodo  $u$  dell'albero, possono sorgere tre casi:

- (1) Se  $u$  non ha figli, allora  $u$  viene rimosso
- (2) Se  $u$  ha un solo figlio  $f$ , allora  $u$  è rimosso rendendo  $f$  a padre.
- (3) Se  $u$  ha due figli, ci si può ridurre al caso (1) e (2) sostituendo l'elemento da cancellare  $x$  con il più piccolo elemento  $y$  t.c.  $y < x$ , che si trova nel nodo  $s$  raggiungibile sendendo a sinistra nel sottoalbero radicato nel figlio destro di  $u$ . L'elemento  $y$  prende posto di  $x$  nel nodo  $u$ , mentre viene rimosso il nodo  $s$ , che non può avere un figlio sinistro.



La procedura `link()` è utilizzata per collegare un nodo padre  $p$  ad un figlio  $u$ . Casi particolari:

-  $p == \text{nil}$   $\rightarrow u$  è la radice

-  $u == \text{nil}$   $\rightarrow$  un nodo è stato cancellato e nil sostituisce il figlio corrispondente al padre

Complessità  $O(1)$ .

link(TREE  $p$ , TREE  $u$ , ITEM  $x$ )

```

if  $u \neq \text{nil}$  then  $u.parent \leftarrow p$ 
if  $p \neq \text{nil}$  then
    if  $x < p.key$  then  $p.left \leftarrow u$ 
    else  $p.right \leftarrow u$ 

```

La procedura `removeNode` restituisce il puntatore alla radice dell'albero, che può cambiare se viene cancellata la radice. Si inizia con la `lookupNode` del nodo da cancellare  $u$ , se  $u$  non è stato trovato la procedura termina senza alterare l'albero.

---

#### TREE removeNode(TREE T, ITEM x)

---

```

TREE u ← lookupNode(T, x)
if u ≠ nil then
    if u.left ≠ nil and u.right ≠ nil then
        TREE s ← u.right
        while s.left ≠ nil do s ← s.left
        u.key ← s.key
        u.value ← s.value
        u ← s
    % Caso (3)

    TREE t
    if u.left ≠ nil and u.right = nil then
        t ← u.left
    % Caso (2) - Solo figlio sx
    else
        t ← u.right
    % Caso (2) - Solo figlio dx / Caso (1)
        link(u.parent, t, x)
        if u.parent = nil then T ← t
        delete u
    return T

```

---

Albero è organizzato come segue:

- il caso (3) appare prima in quanto si riduce a (1) e (2); viene identificato il nodo  $s$  e la sua chiave-valore viene copiata in  $u$ , il puntatore di  $u$  assume il valore di  $s$  e si cancella  $s$  tramite il caso (1) o (2);
- il caso (2) con solo un figlio sinistro; il figlio viene collegato al padre di  $u$ ; se questo è nil allora  $s$  diventa la nuova radice;
- il caso (2) con solo un figlio destro è simmetrico. Questa parte gestisce anche il caso (1), dove il figlio destro di  $u$  è nil. Se  $u$  che il figlio destro sono nil, l'albero è vuoto e la radice = nil.

Complessità della procedura  $O(h)$ ,  $h = \text{altezza ABR}$

Un operatore **min**, che restituisce il nodo contenente il minimo, può essere facilmente realizzato in tempo  $O(n)$ ; basta effettuare una ricerca in cui si esegue sempre il figlio sinistro fino a trovare il nodo che non ha figlio sinistro.  
 L'operatore **max** è simmetrico.

TREE **min(TREE T)**

```
while T.left ≠ nil do
| T ← T.left
return T
```

TREE **max(TREE T)**

```
while T.right ≠ nil do
| T ← T.right
return T
```

Dato un nodo  $t$  può essere interessante ottenere il successore/predecessore di  $t$  nell'albero, ovvero il nodo la cui chiave segue/precede  $t.key$  nell'ordinamento. Sono dati due casi:

- se  $t$  ha un figlio destro, il successore è il minimo  $v$  del sottoalbero di destra
- = altrimenti, il successore è il primo shtensto  $v$  di  $t$  per cui  $t$  sta nel sottoalbero sinistro di  $v$ .

TREE **successorNode(TREE t)**

```
if t = nil then
| return t
if t.right ≠ nil then
| return min(t.right())
TREE p ← t.parent
while p ≠ nil and t = p.right do
| t ← p
| p ← p.parent
return p
```

TREE **predecessorNode(TREE t)**

```
if t = nil then
| return t
if t.left ≠ nil then
| return max(t.left())
TREE p ← t.parent
while p ≠ nil and t = p.left do
| t ← p
| p ← p.parent
return p
```

## TABELLE HASH

È un metodo di memorizzazione dei dizionari alternativo agli alberi binari. L'idea è ricavare direttamente dalla chiave la posizione che occuperà nel vettore, questo avrà costo  $O(n)$  nel caso pessimo, ma  $O(1)$  nel caso medio.

La soluzione ideale è avere a disposizione una funzione che data una qualsiasi chiave  $U$  restituisca sempre uno indice distinto + c.

$$H : U \rightarrow \{0, \dots, m-1\}$$

( $m$  è la dimensione del vettore)  $K_1 \in U, K_2 \in U$  con  $K_1 \neq K_2$  e  $H(K_1) \neq H(K_2)$ . Questo meccanismo prende il nome di hash table ad accesso diretto.

Questo approccio funziona se l'universo delle chiavi  $U$  non è troppo grande, altrimenti "m" può diventare grande a tal punto da sprecare memoria; quindi si rinuncia alla iniettività della funzione e si cerca di gestire le collisioni quando:  $K_1 \in U, K_2 \in U, K_1 \neq K_2, H(K_1) = H(K_2)$

$$K_1 \in U, K_2 \in U, K_1 \neq K_2 \text{ e } H(K_1) = H(K_2)$$

Per realizzare efficientemente un dizionario con una tabella hash occorre:

(1) una funzione  $H$ , che sia calcolabile in tempo  $\mathcal{O}(1)$  e che distribuisca le chiavi uniformemente nel vettore  $A$  al fine di ridurre il numero di collisioni.

(2) un metodo di scansione per reperire chiavi che hanno trovato la loro posizione occupata, che permetta di esplorare tutto il vettore  $A$  e non pronchi la formazione di agglomerati.

(3) la dimensione del vettore  $A$  deve essere sovrastimata rispetto al numero di chiavi attese, onde evitare di riempire  $A$  completamente.

$$(s_k)H = (n_k)H \quad s \neq n_k, 0 \leq k < n$$

## FUNZIONI HASH

Una funzione di hash non è casuale, in quanto calcolare  $H(k)$  due volte deve produrre lo stesso risultato.

Una buona funzione di hash deve minimizzare le collisioni; uno dei possibili criteri è detto uniformità semplice. Detta  $p(k)$  la probabilità che una chiave  $k$  sia inserito nella tabella sia

$$Q(i) = \sum_{k: H(k)=i} p(k)$$

La probabilità che una chiave qualsiasi finisca nella cella  $i$ . Una funzione  $H$  gode di questa proprietà se  $\forall i \in \{0, \dots, m-1\}: Q(i) = 1/m$