Appunti di Strutture Dati

1 – Strutture dati astratte

Una struttura dati astratta viene specificata tramite una serie di operazioni ammesse sulla struttura e di condizioni che la struttura deve rispettare prima e dopo l'operazione. L'implementazione della struttura dati consiste nel modo in cui le operazioni vengono eseguite. La complessità computazionale delle operazioni dipende dall'implementazione.

1 – Esempio. Pila (stack) - operazioni:
- interrogazione se la struttura è vuota;
- inserimento di un dato;
- prelevamento dell'ultimo dato inserito (condizione: pila non vuota).

2 – Esempio. Coda (queue) - operazioni:
- interrogazione se la struttura è vuota;
- inserimento di un dato;
- prelevamento del dato inserito da più tempo (condizione: coda non vuota).

3 – Esempio. Coda di priorità (priority queue): ad ogni dato è associata una chiave (un numero intero) che definisce la priorità, più basso il numero più alta la priorità - operazioni:
- interrogazione se la struttura è vuota;
- inserimento di un dato con chiave specificata;
- lettura del dato a maggiore priorità (condizione: coda non vuota);
- prelevamento del dato a maggiore priorità (condizione: coda non vuota);
- lettura del dato con chiave specificata (condizione: presenza del dato);
- prelevamento del dato con chiave specificata (condizione: coda non vuota);
- diminuzione della chiave di un dato con chiave specificata (condizione: presenza del dato e chiave attuale maggiore di quella finale).

2 – Implementazione di una pila

Implementazione come vettore (array) illimitato: sia A il vettore e $A[i]$ il dato presente nella posizione i ($i := 0, 1, 2, \ldots$) del vettore. Sia p un puntatore (indice) alla posizione dove verrà inserito il prossimo dato. Inizialmente $p := 0$.

Interrogazione se la struttura è vuota:

boolean function emptystack(A) := ($p = 0$).

Inserimento del dato d:

procedure push(A, d)
begin
    $A[p] := d$;
    $p := p + 1$
end.

Prelevamento dell'ultimo dato inserito:

procedure pop(A)
begin
    $p := p - 1$;
    if p=0 **then return**("empty stack") **else return**($A[p]$)
end.

Le tre procedure hanno tutte complessità (temporale) costante (indipendentemente dalla dimensione di A). Quando si vuole prelevare un dato conviene comunque verificare che la pila non sia vuota.

Un aspetto critico dell'implementazione è dovuto alla limitatezza inevitabile dell'array A. Quindi è possibile che una procedura d'inserimento possa fallire per 'overflow' della pila. In molti problemi è nota a priori una limitazione superiore alla dimensione massima della pila e quindi basta dimensionare A in modo adeguato. Se invece non è nota una limitazione superiore si può verificare il valore di p prima dell'immissione di una dato. Questo però ha un prezzo computazionale e quindi si tratta di valutare cosa conviene fare a seconda dei casi.

3 – Implementazione di una coda

Implementazione come vettore (array) di dimensione N: sia A il vettore e $A[i]$ il dato presente nella posizione i ($i := 0, 1, 2, \ldots, N-1$) del vettore. Sia p un puntatore (indice) alla posizione dove verrà inserito il prossimo dato e q un puntatore alla posizione del dato inserito da più tempo. Inizialmente $p =: 0$, $q := 0$.

Interrogazione se la struttura è vuota:

boolean function emptyqueue(A) := ($p = q$).

Inserimento del dato d:

procedure queue(A, d)
begin
    $A[p] := d$;
    $p := p + 1 \bmod N$
end.

Prelevamento dell'ultimo dato inserito:

procedure dequeue(A)
begin
    if $p = q$ **then return**("empty queue") **else return**$(A[q])$;
    $q := q + 1 \bmod N$
end.

Le tre procedure hanno tutte complessità (temporale) costante (indipendentemente da N). Anche in questo caso si dà per scontato che N sia sufficientemente grande (in modo che non si verifichi mai $p = q$ dopo l'inserimento di un dato). Cosa succede se si inseriscono dati e dopo una delle inserzioni si ha $p = q$?

4 – Implementazione di una coda con priorità come vettore non ordinato

Implementazione come vettore non ordinato di dimensione N: sia A il vettore e $A[i, 0]$ e $A[i, 1]$ rispettivamente la chiave e il dato presente nella posizione i ($i := 0, 1, 2, \ldots, N - 1$) del vettore. Sia p un puntatore alla posizione dove verrà inserito il prossimo dato. Inizialmente $p := 0$.

Interrogazione se la struttura è vuota:

boolean function emptyprqueue$(A) := (p = 0)$.

Inserimento del dato d con chiave k:

procedure insert(A, d, k)
begin
    $A[p, 1] := d$;
    $A[p, 0] := k$;
    $p := p + 1$
end.

Lettura del dato a maggiore priorità (condizione: coda non vuota);

procedure getmin(A)
begin
    minkey:= $A[0, 0]$;
    minindex:= 0;
    for $i := 1$ **to** $p - 1$ **do**
        if $A[i, 0]$ <minkey
        then begin
            minkey:= $A[i, 0]$;
            minindex:= i;
        end;

```
        return(A[minindex])
    end.
```

Prelevamento del dato a maggiore priorità (condizione: coda non vuota);

```
    procedure getdelmin(A)
    begin
        minkey:= A[0, 0];
        minindex:= 0;
        for i := 1 to p − 1 do
            if A[i, 0] <minkey
            then begin
                minkey:= A[i, 0];
                minindex:= i;
            end;
        return(A[minindex]);
        A[minindex]:=A[p − 1]
        p := p − 1
    end.
```

Lettura del dato con chiave specificata (condizione: presenza del dato);

```
    procedure getkey(A, k)
    begin
        found:=False;
        i := 0
        while ¬ found do
        begin
            found:= (A[i, 0] = k) ;
            i := i + 1;
        end
        return(A[i − 1]);
    end.
```

Prelevamento del dato con chiave specificata (condizione: presenza del dato);

```
    procedure getdelkey(A, k)
    begin
        found:=False;
        i := 0
        while ¬ found do
        begin
            found:= (A[i, 0] = k) ;
            i := i + 1;
        end
        return(A[i − 1]);
        A[i − 1] := A[p − 1];
        p := p − 1
    end.
```

Diminuzione della chiave di un dato con chiave specificata (condizione: presenza del dato e chiave attuale k maggiore di quella finale h).

```
    procedure deckey(A, k, h)
```

```
begin
    found:=False;
    i := 0
    while ¬ found do
    begin
        found:= (A[i,0] = k) ;
        i := i + 1;
    end;
    A[i − 1, 0] := h ;
end.
```

5 – Implementazione di una coda con priorità come vettore ordinato

Implementazione come vettore ordinato di dimensione N: sia A il vettore e $A[i,0]$ e $A[i,1]$ rispettivamente la chiave e il dato presente nella posizione i. I dati sono presenti nelle posizioni

$$I := \{i : q \leq i < p \text{ se } p > q\,;\ (i \geq q) \vee (i < p) \text{ se } p < q\}$$

. Deve valere $A[i,0] < A[i+1 \bmod N, 0]$ per ogni $i \in I$, $i \neq p-1$ (ordinamento circolare). Inizialmente $p := 0$, $q := 0$.

Interrogazione se la struttura è vuota:

boolean function emptyprqueue(A) := $(p = q)$.

Per le successive operazioni è utile disporre di una procedura che effettui una ricerca binaria di una chiave k su un insieme ordinato di n dati. Si assume che i dati sono disposti su un vettore di dimensione N dalla posizione q alla posizione $p-1$ in modo circolare (quindi $n = p - q \bmod N$). La procedura restituisce l'indice i dove si trova la chiave k se la chiave è presente (cioè $A[i,0] = k$), oppure, se la chiave non è presente l'indice i dove la chiave dovrebbe essere inserita (cioè $A[i-1,0] < k$ e $A[i,0] > k$)

```
integer function binsearch (A, k, q, p)
begin
    n = p − q mod N;
    if k > A[(p − 1) mod N, 0]
    then return(p)
    else begin
        l = 0;
        r = n − 1;
        while r > l do
        begin
            m = ⌊(l + r)/2⌋;
            if A[(m + q) mod N, 0] ≥ k
            then r := m
            else l := m + 1
        end;
        return((l + q) mod N)
    end
```

end.

Inserimento del dato d con chiave k:

procedure insert(A, d, k)
begin
$n = (p - q) \bmod N$
$r =$ binsearch(A, k, q, p);
$s = (r - q) \bmod N$;
for $i := n - 1$ **downto** s **do** $A[(i + 1 + q) \bmod N] := A[(i + q) \bmod N]$;
$A[r] := (k, d)$;
$p := (p + 1) \bmod N$
end.

Lettura del dato a maggiore priorità (condizione: coda non vuota);

procedure getmin(A)
begin
return$(A$[q]$)$
end.

Prelevamento del dato a maggiore priorità (condizione: coda non vuota);

procedure getdelmin(A)
begin
return$(A$[q]$)$
$q := (q + 1) \bmod N$
end.

Lettura del dato con chiave specificata (condizione: presenza del dato);

procedure getkey(A, k)
begin
$r =$ binsearch(A, k);
return$(A[r])$;
end.

Prelevamento del dato con chiave specificata (condizione: presenza del dato);

procedure getdelkey(A, k)
begin
$r =$ binsearch(A, k);
return$(A[r])$;
for $i := r - q + 1 \bmod N$ **to** $p - 1 - q \bmod N$ **do** $A[i - 1 + q \bmod N] := A[i + q \bmod N]$;
$p := p - 1 \bmod N$
end.

Diminuzione della chiave di un dato con chiave specificata (condizione: presenza del dato e chiave attuale k maggiore di quella finale h).

procedure deckey(A, k, h)
begin
$s =$ binsearch(A, k);
$r =$ binsearch(A, h);
$A[p] := A[s]$;

for $i := r + 1 - q \bmod N$ **to** $s - q \bmod N$ **do** $A[i + q \bmod N] := A[i - 1 + q \bmod N]$;
$A[r] := A[p]$
end.

Le complessità sono (n è il numero di dati presenti nell'array).

	array non ordinato	array ordinato
emptyprqueue	$O(1)$	$O(1)$
insert	$O(1)$	$O(\log n + n) = O(n)$
getmin	$O(n)$	$O(1)$
getdelmin	$O(n)$	$O(1)$
getkey	$O(n)$	$O(\log n)$
getdelkey	$O(n)$	$O(\log n + n) = O(n)$
deckey	$O(n)$	$O(2 \log n + n) = O(n)$

6 – Implementazione di una coda con priorità come albero binario di ricerca

Un albero binario (*BT, binary tree*) è una struttura che può essere definita ricorsivamente come:

- un BT è vuoto (NIL oppure 0), oppure
- è un nodo, detto *radice* del BT, con due BT.

I due BT associati ad ogni nodo i vengono detti *BT sinistro* e *BT destro*. Se questi BT non sono vuoti, le loro radici vengono detti figli (destro o sinistro a seconda del caso) e il nodo i è il loro padre. Useremo la notazione $r[i]$ per il figlio destro del nodo i, $l[i]$ per il figlio sinistro e $f[i]$ per il nodo padre, usando il termine di *puntatori* per r, l e f. Inoltre indicheremo con $r[i] = 0$ (oppure $r[i]$ =NIL), il fatto che il BT di destra è vuoto e analogamente per il BT di sinistra. Dalla definizione si ha che $f[i] = 0$ se e solo se i è la radice del BT ed inoltre nel nodo i esiste un puntatore figlio al nodo j, se e solo se nel nodo j esiste un puntatore padre al nodo i ovvero

$$(j = l[i]) \vee (j = r[i]) \Longleftrightarrow i = f[j]$$

Dato un nodo i si dice sottoalbero di i l'insieme $T(i)$ dei nodi discendenti di i (incluso i stesso). I nodi senza figli vengono detti *foglie*. Gli altri nodi vengono detti *interni*. Se contiamo i figli dei nodi interni per tuti i nodi interni, contiamo tutti i nodi tranne la radice. Siccome ogni nodo interno ha uno oppure due nodi, vale la seguente relazione per il numero n_i di nodi interni

$$n_i \leq n - 1 \leq 2\, n_i$$

e siccome $n_f + n_i = n$ (con n_f numero di foglie) si ha

$$n - n_f \leq n - 1 \leq 2\,(n - n_f) \quad \Longrightarrow \quad 1 \leq n_f \leq \frac{n+1}{2}$$

Si ricava inoltre che $n_f = (n + 1)/2$ se tutti i nodi interni hanno esattamente due figli.

Viene definito come livello di un nodo la distanza dalla radice al nodo (la radice è a livello 0) e viene definita come profondità dell'albero il massimo dei livelli dei nodi. Il numero massimo di nodi al livello h è 2^h. Sia d la profondità e sia n il numero di nodi. Allora si ha

$$n \leq \sum_{h=0}^{d} 2^h = 2^{d+1} - 1 \qquad \Longrightarrow \qquad \log_2(n+1) - 1 \leq d$$

Un BT si dice *bilanciato* se, per ogni nodo, le profondità del sottoalbero sinistro e del sottoalbero destro differiscono al più di uno. Quant'è il minimo numero di nodi in un albero bilanciato? Se la profondità è 0, allora l'albero contiene un nodo (in ogni caso). Se la profondità è 1, il numero minimo è 2 (un solo figlio presente). Indichiamo con N_d il minimo numero di nodi di un albero bilanciato di profondità d. Abbiamo quindi notato che $N_0 = 1$ e $N_1 = 2$. In generale dovrà valere la ricorsione

$$N_d = N_{d-1} + N_{d-2} + 1$$

Infatti il numero di nodi totale è dato dai nodi di un sottoalbero, più i nodi dell'altro sottoalbero, più il nodo padre dei due sottoalberi. Inoltre la profondità dell'albero è di uno più grande della profondità del sottoalbero più profondo, e questo a sua volta, per definizione di albero bilanciato, di uno più grande della profondità dell'altro sottoalbero.

La ricorsione assomiglia molto alla ricorsione di Fibonacci. Infatti se sommiamo 1 ad entrambi i termini otteniamo

$$(N_d + 1) = (N_{d-1} + 1) + (N_{d-2} + 1)$$

La ricorsione dei numeri $(N_d + 1)$ è ora esattamente quella di Fibonacci. Tuttavia bisogna controllare l'inizio della ricorsione. Si noti che $(N_0 + 1) = 2 = F_3$ e $(N_1 + 1) = 3 = F_4$. Pertanto $N_d + 1 = F_{d+3}$ e quindi il numero di nodi n di un BST bilanciato è sempre limitato da

$$n \geq N_d = F_{d+3} - 1 \geq 2^{(d+3-2)/2} - 1 = 2^{(d+1)/2} - 1 \quad \Longrightarrow \quad d + 1 \leq 2\,\log_2(n+1)$$

Quindi mettendo assieme le due limitazioni

$$\log_2(n+1) - 1 \leq d \leq 2\,\log_2(n+1) - 1$$

abbiamo che in un albero bilanciato $d = \Theta(\log n)$.

Si noti che si può ottenere un simile risultato (con un numero minore di nodi a parità di profondità ma asintoticamente sempre $d = \Theta(\log n)$) assumendo che la differenza fra le profondità dei sottoalberi destro e sinistro sia al più un intero fissato h.

Gli alberi binari si prestano naturalmente ad esecuzioni ricorsive di varie operazioni. Ad esempio si voglia calcolare il numero di nodi di un albero. Si può scrivere la seguente procedura:

integer function size(T)
    if T =NIL **then return**(0) **else return**($1 + T[l(i)] + T[r(i)]$)
end.

Nonostante l'apparenza con il pericoloso calcolo dei numeri di Fibonacci eseguiti in modo ricorsivo, facciamo vedere che la procedura indicata ha complessità lineare. Sia $f(n)$ il numero di chiamate della

procedura per un albero di n nodi. Supponiamo sia vero che $f(n) \leq C\,n + D$. Se $n = 0$ la relazione è vera per $D \geq 1$. Allora per induzione possiamo scrivere, con n_l e n_r il numero dei nodi dei figli sinistro e destro rispettivamente,

$$f(n) = 1 + f(n_l) + f(n_r) \leq 1 + C\,(n_l + n_r) + 2\,D = 1 + C\,(n-1) + 2\,D = C\,n + D + (1 - C + D)$$

Si ha $f(n) \leq C\,n + D$ se $1 - C + D \leq 0$, cioè $D \leq C - 1$. Siccome $D \geq 1$, è sufficiente scegliere $D = 1$ e $C = 2$. Quindi $f(n) \leq 2\,n + 1$.

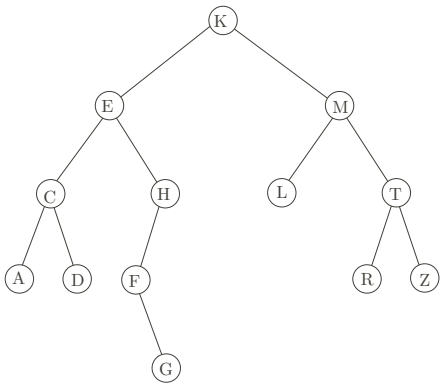
In modo analogo possiamo valutare la profondità di un albero (assumendo -1 la profondità di un albero vuoto).

integer function depth(T)
    if T =NIL **then return**(-1) **else return**$(1 + \max\{T[l(i)], T[r(i)]\})$
end.

A questo punto anche la verifica del bilanciamento di un albero può essere effettuata in modo ricorsivo. La procedura restituisce due valori: un booleano che è vero se l'albero è bilanciato e falso altrimenti e un intero che corrisponde alla profondità dell'albero.

procedure balance(T)
    if T =NIL
    then return(true,-1)
    else begin
        $< b_l, d_l >$:=balance(T_l);
        $< b_r, d_r >$:=balance(T_r);
        return$(b_l \wedge b_r \wedge |d_l - d_r| < 1, 1 + \max\{d_l, d_r\})$
    end
end.

In un *albero binario di ricerca (BST, binary search tree)*, ogni nodo possiede una *chiave* e per ogni nodo i con figlio sinistro $l[i]$ e figlio destro $r[i]$ (eventualmente vuoti), la chiave $k[i]$ di i è migliore di tutte le chiavi in $T(r[i])$ e peggiore di tutte le chiavi in $T(l[i])$.



Siccome si accede ad ogni nodo a partire dalla radice, bisogna che, esternamente alla struttura, sia disponibile un puntatore alla radice, se la struttura non è vuota, oppure un puntatore NIL se la struttura è

vuota. L'implementazione di un BST può essere fatta in vari modi, sia con strutture dinamiche che statiche. Se ad esempio si usa una struttura statica come un array di quaterne, allora i può essere l'indirizzo nell'array mentre $k[i]$, $r[i]$, $l[i]$ e $f[i]$ sono il contenuto dell'array all'indirizzo i. In questo caso è conveniente avere la radice all'indirizzo 1 (se esiste l'indirizzo 0 non conviene usarlo per riservare il valore 0 al significato di NIL), mentre gli altri nodi possono essere collocati in indirizzi aribitrari (conviene naturalmente avere una gestione dell'array il più possibile parsimoniosa in termini di utilizzo di memoria). Ad esempio il BST in figura potrebbe essere rappresentato dal seguente array (nella riga i sono indicati nell'ordine $k[i]$, $l[i]$, $r[i]$ e $f[i]$):

1	K	4	2	0
2	M	3	7	1
3	L	0	0	2
4	E	6	5	1
5	H	8	0	4
6	C	9	13	4
7	T	12	11	2
8	F	0	10	5
9	A	0	0	6
10	G	0	0	8
11	Z	0	0	7
12	R	0	0	7
13	D	0	0	6

Visita di un BT

Si possono elencare i nodi di un BT secondo varie strategie (le procedure indicate assumono che gli alberi da visitare non siano vuoti, per esercizio si modifichino le procedure senza questa ipotesi):
– **in-order** (visita simmetrica):

```
procedure inordervisit(i)
begin
    if l[i] ≠ 0 then inordervisit(l[i]);
    return(i);
    if r[i] ≠ 0 then inordervisit(r[i]);
end.
```

Si ottiene, applicando inordervisit alla radice dell'esempio, la successione

$$(A, C, D, E, F, G, H, K, L, M, R, T, Z)$$

È chiaro dalla definizione che la procedura inordervisit applicata a un BST fornisce le chiavi ordinate.

– **pre-order** (visita anticipata)

```
procedure preordervisit(i)
begin
    return(i);
    if l[i] ≠ 0 then preordervisit(l[i]);
    if r[i] ≠ 0 then preordervisit(r[i]);
end.
```

Questa successione corrisponde ad una visita in profondità dell'albero. Si ottiene, applicando preordervisit alla radice dell'esempio, la successione

$$(K, E, C, A, D, H, F, G, M, L, T, R, Z)$$

– **post-order** (visita posticipata)

```
procedure postordervisit(i)
begin
    if l[i] ≠ 0 then postordervisit(l[i]);
    if r[i] ≠ 0 then postordervisit(r[i]);
    return(i);
end.
```

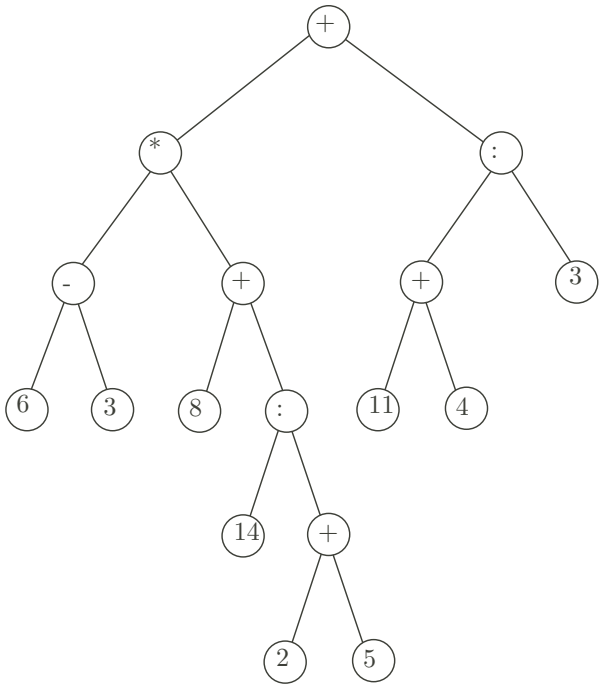
Si ottiene, applicando postordervisit alla radice dell'esempio, la successione

$$(A, D, C, G, F, H, E, L, R, Z, T, M, K)$$

Per operazioni di ordinamento dei dati la visita più utile è ovviamente la visita in-order. Per altri scopi sono comunque importanti anche le altre due visite. Ad esempio si supponga di dover eseguire la seguente operazione aritmetica

$$(6-3)*(8+\frac{14}{2+5})+\frac{11+4}{3}$$

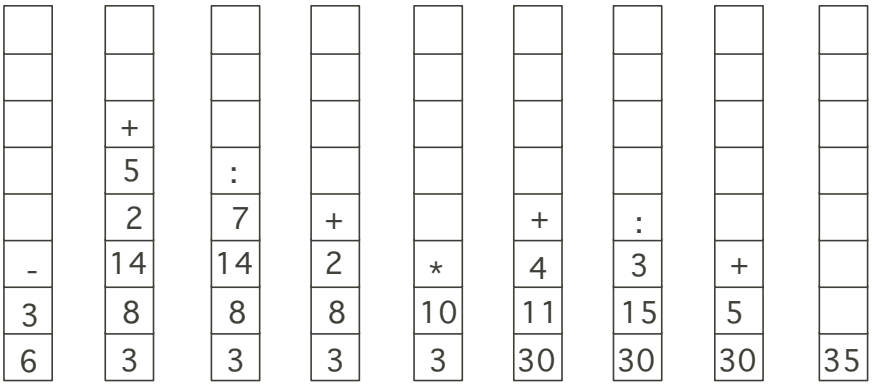
L'esecuzione può essere rappresentata dal seguente BT in cui le chiavi possono essere sia simboli di operazioni che numeri e ogni nodo ha due figli oppure nessuno.



Se un nodo contiene un simbolo la sua chiave va sostituita con il risultato dell'operazione indicata dal simbolo applicata ai suoi due figli (nell'ordine sinistro-destro). La procedure ricorsiva è la seguente (dove xiy è il risultato dell'operazione i applicata a x e y)

```
procedure compute(i)
begin
    if l[i] ≠ 0
    then
    begin
        x :=compute(l[i]) ;
        y :=compute(r[i]) ;
        return(xiy) ;
    end
    else return(i);
end.
```

che corrisponde ad una visita post-order con immissione delle chiavi in una pila. Non appena nella pila si immette un simbolo di operazione, questa va subito eseguita sui due dati sottostanti, il puntatore della pila viene decrementato di due e la visita post-order continua (in figura si vedono alcune fasi del riempimento della pila)



Per trovare la complessità di una delle tre visite dell'albero, indicando con $T(n)$ il numero di operazioni necessario per un albero con n chiavi, le ricorsioni indicate danno una relazione del tipo:

$$T(n) = T(n_l) + T(n_r) + 1$$

dove n_l e n_r sono il numero di nodi nei sottoalberi di sinistra e destra rispettivamente. Per calcolare una limitazione superiore a $T(n)$ ipotizziamo una limitazione lineare che poi verificheremo per induzione. Sia quindi $T(n) \leq c\,n$ con c costante opportuna da determinare. Allora si ha

$$T(n) = T(n_l) + T(n_r) + 1 \leq c\,n_l + c\,n_r + 1 \leq c\,n_l + c\,n_r + c = c\,n$$

Siccome $T(1) = 3$, si ha $T(n) \leq 3\,n = O(n)$. Quindi ottenere le chiavi ordinati ha complessità lineare, *purché però il BST sia stato costruito.* Come vedremo, la costruzione di un BST ha complessità $O(n\,d)$ con d profondità dell'albero.

Operazioni su un BST

— successore-predecessore: per trovare il successore (secondo l'ordine delle chiavi) di un elemento dato si può pensare di simulare le operazioni della procedura inordervisit fra due successive esecuzioni di "return". Se l'elemento dato ha un figlio destro si visitano ricorsivamente i figli sinistri fino a fermarsi al primo nodo senza figli sinistri. Se però l'elemento non ha figli destri, bisogna risalire di padre in padre fino a fermarsi non appena il figlio è sinistro (questo corrisponde nella procedura inordervisit a risalire alle procedure chiamanti).

```
procedure successor(i)
begin
    if r[i] ≠ 0
    then
    begin
        i := r[i];
        while l[i] ≠ 0 do i := l[i];
```

```
        return(i);
    end
    else
    begin
        while f[i] ≠ 0 ∧ i = r[f[i]] do i := f[i]
        return(f[i]);
    end
end.
```

Per ottenere l'elemento predecessore basta invertire i ruoli di destro e sinistro.

```
procedure predecessor(i)
begin
    if l[i] ≠ 0
    then
    begin
        i := l[i];
        while r[i] ≠ 0 do i := r[i];
        return(i);
    end
    else
    begin
        while f[i] ≠ 0 ∧ i = l[f[i]] do i := f[i]
        return(f[i]);
    end
end.
```

La complessità di trovare il successore o il predecessore è quindi $O(d)$.

— inserimento di un nodo j con chiave $k[j]$ specificata: si esegue una ricerca binaria a a partire dalla radice

```
procedure bstinsert(k[j], i)
begin
    if k[i] > k[j]
    then if l[i] = 0
        then begin l[i] := j; f[j] := i end
        else bstinsert(k[j], l[i])
    else if r[i] = 0
        then begin r[i] := j; f[j] := i end
        else bstinsert(k[j], r[i])
end.
```

La complessità è $O(d)$. Se il BST è implementato con un array e p è un puntatore che indica il primo elemento disponibile in scrittura dell'array, allora il valore j della procedura è p e alla fine p viene incrementato di uno. Quindi una procedura che inserisce una nuova chiave k può essere la seguente

```
procedure keyinsert(K)
begin
    k[p] := K;
    bstinsert[k[p],1];
    p := p + 1;
end.
```

— lettura del dato a maggiore priorità: basta fornire il primo dato che si ottiene con la visita in-order; la

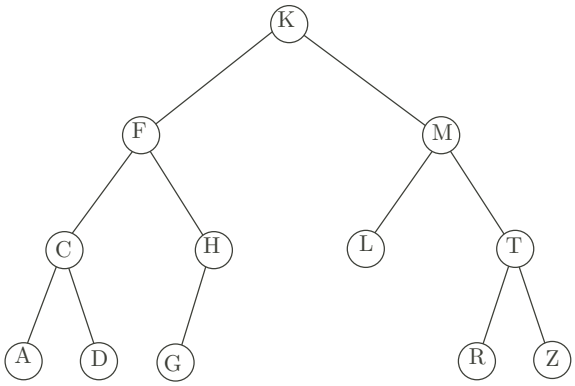
complessità è $O(d)$.

— eliminazione di un nodo i generico: vi sono tre casi: 1) il nodo è senza figli: basta eliminarlo togliendo il puntatore del nodo padre, cioè ponendo $r[f[i]] := 0$ se $r[f[i]] = i$ e $l[f[i]] := 0$ se $l[f[i]] = i$; 2) il nodo ha un figlio, ad esempio $r[i]$: basta cambiare il puntatore del "nonno" direttamente al "nipote", cioè ponendo $r[f[i]] := r[i]$ e $f[r[i]] := f[i]$ se $r[f[i]] = i$ e $l[f[i]] := r[i]$ e $f[r[i]] := f[i]$ se $l[f[i]] = i$; 3) il nodo ha due figli: non è difficile vedere che se un nodo ha due figli il suo successore non può avere due figli; quindi basta spostare il successore, rimuovendolo dal suo posto applicando il caso 1) o 2), al posto del nodo. L'eliminazione di un nodo ha complessità $O(d)$, dato che bisogna determinare il successore, altrimenti la complessità sarebbe costante.

```
procedure delete(i)
begin
    if l[i] = 0 ∧ r[i] = 0
    then if i = r[f[i]] then r[f[i]] := 0 else l[f[i]] := 0
    else
        if l[i] = 0
        then begin
            f[r[i]] := f[i]; if r[f[i]] = i then r[f[i]] := r[i] else l[f[i]] := r[i]
        end
        else if r[i] = 0
            then begin
                f[l[i]] := f[i]; if r[f[i]] = i then r[f[i]] := l[i] else l[f[i]] := l[i]
            end
            else begin
                j := successor(i);
                k[i] := k[j];
                delete(j);
            end
end.
```

Seil BST è implementato con un array e si vuole recuperare lo spazio lasciato disponibile all'indirizzo del nodo eliminato, allora conviene spostare il nodo memorizzato ultimo nell'array e spostarlo nella posizione rimasta vuota. Sia i il nodo da eliminare e sia n il nodo ultimo nell'array. Bisogna allora eseguire le seguenti operazioni: se $r[f[n]] = n$ allora $r[f[n]] := i$ e se $l[f[n]] = n$ allora $l[f[n]] := i$ (il nodo padre di n adesso punta a i e non a n), poi $f[l[n]] := i$ se $l[n] \neq 0$ e $f[r[n]] := i$ se $r[n] \neq 0$ (i nodi figli di n adesso puntano a i e non a n), poi $k[i] := k[n]$, $r[i] := r[n]$, $l[i] := l[n]$, $f[i] := f[n]$ (il contenuto dell'indirizzo n viene copiato nell'indirizzo i) e infine il decremento di uno del puntatore p dell'array al primo indirizzo disponibile in scrittura.

Ad esempio si supponga di eliminare il nodo con chiave E dell'esempio. Il nodo E ha due figli e il suo successore è F. Pertanto l'eliminazione avviene in due fasi: sostituzione di E con F e poi eliminazione del 'vecchio' F. Dal punto di vista dell'implementazione si vedano i quattro array: il primo è quello del BST iniziale; nel secondo si è semplicemente copiato la chiave F dall'indirizzo 8 all'indirizzo 4 (lasciando inalterati i valori di tutti i puntatori); nel terzo array si elimina il nodo 8 (indirizzo obsoleto della chiave F) rendendo il nodo 10 figlio del nodo 5 (si cambiano i puntatori di questi nodi); nel quarto array si recupera la memoria in 8 spostando il nodo 13 in 8 (cambiando un puntatore in 6, padre di 13, ora diventato 8) e copiando il contenuto di 13 in 8.



1	K	4	2	0
2	M	3	7	1
3	L	0	0	2
4	E	6	5	1
5	H	8	0	4
6	C	9	13	4
7	T	12	11	2
8	F	0	10	5
9	A	0	0	6
10	G	0	0	8
11	Z	0	0	7
12	R	0	0	7
13	D	0	0	6

K	4	2	0
M	3	7	1
L	0	0	2
F	6	5	1
H	8	0	4
C	9	13	4
T	12	11	2
F	0	10	5
A	0	0	6
G	0	0	8
Z	0	0	7
R	0	0	7
D	0	0	6

K	4	2	0
M	3	7	1
L	0	0	2
F	6	5	1
H	10	0	4
C	9	13	4
T	12	11	2
F	0	10	5
A	0	0	6
G	0	0	5
Z	0	0	7
R	0	0	7
D	0	0	6

K	4	2	0
M	3	7	1
L	0	0	2
F	6	5	1
H	8	0	4
C	9	8	4
T	12	11	2
D	0	0	6
A	0	0	6
G	0	0	8
Z	0	0	7
R	0	0	7

– prelevamento del dato a maggiore priorità: il dato a maggiore priorità può avere un figlio (destro) o nessun figlio, quindi una volta ottenuto il dato basta eliminare il nodo corrispondente secondo i casi 1) e 2) visti prima. Complessità è $O(d)$

– lettura del dato con chiave h specificata (condizione: presenza del dato): si effettua con ricerca binaria

```
procedure bstsearch(h, i)
begin
    if h = k[i]
    then return(i)
    else if k[i] > h
        then bstsearch(h, l[i])
        else bstsearch(h, r[i])
end.
```

Complessità $O(d)$

– prelevamento del dato con chiave specificata (condizione: coda non vuota): vedi i casi precedenti.

– diminuzione della chiave di un dato con chiave specificata (condizione: presenza del dato e chiave attuale maggiore di quella finale): basta trovare il nodo, eliminarlo e inserirlo ex-novo.

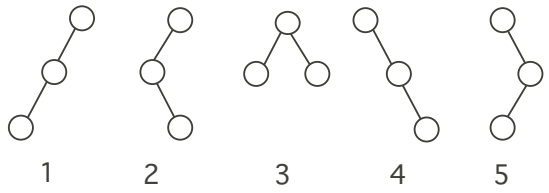
Riassumendo tutte le operazioni hanno complessità $O(d)$ che, nel caso di alberi bilanciati diventa $O(\log n)$. Quindi la tabellina, per alberi bilanciati diventa

	array non ordinato	array ordinato	BST bilanciato
insert	$O(1)$	$O(\log n + n) = O(n)$	$O(\log n)$
getmin	$O(n)$	$O(1)$	$O(\log n)$
getdelmin	$O(n)$	$O(1)$	$O(\log n)$
getkey	$O(n)$	$O(\log n)$	$O(\log n)$
getdelkey	$O(n)$	$O(\log n + n) = O(n)$	$O(\log n)$
deckey	$O(n)$	$O(2 \log n + n) = O(n)$	$O(\log n)$

7 – Profondità media di un BST

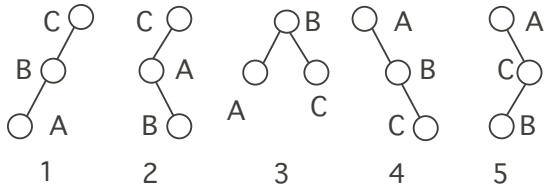
Cosa si intende per albero casuale? Possiamo definire due modi alternativi:

1) sia $T(n-1)$ un albero binario con $n-1$ nodi ($T(1)$ è sempre la radice). Da $T(n-1)$ viene generato un albero $T(n)$ nel seguente modo ricorsivo (sia r la radice): $i := r$, si sceglie sinistra o destra con probabilità $1/2$, se esiste il figlio j (sinistro o destro a seconda della scelta) di i, si pone $i := j$ e si ripete altrimenti il nodo n-mo diventa il figlio j.



Ad esempio, per $n = 4$ gli alberi possibili sono i cinque indicati in figura con probabilità $p_1 = p_2 = p_4 = p_5 = 1/8$ mentre $p_3 = 1/2$. Secondo questo schema di generare alberi casuali, il valore atteso della profondità è $2 \cdot 4 \cdot 1/8 + 1 \cdot 1/2 = 3/2$

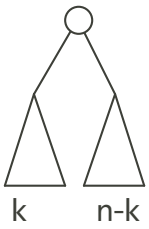
2) Si considerino tutte le permutazioni di n chiavi. Ogni permutazione genera un albero, inserendo le chiavi nell'ordine della permutazione. Assegnando probabilità uniforme alle permutazioni ($1/n!$), ad ogni albero viene assegnata la probabilità $1/n!$ per il numero di permutazioni che generano il medesimo albero.



Le sei permutazioni generano i seguenti alberi:

$$(ABC) \to 4,\ (ACB) \to 5,\ (BAC) \to 3,\ (BCA) \to 3,\ (CAB) \to 2,\ (CBA) \to 1$$

per cui le probabilità degli alberi sono $p_1 = p_2 = p_4 = p_5 = 1/6$ e $p_3 = 1/3$. Secondo questo schema di generare alberi casuali, il valore atteso della profondità è $2 \cdot 4 \cdot 1/6 + 1 \cdot 1/3 = 5/3$.



In generale possiamo considerare il problema di come si ripartiscono n nodi nei due sottoalberi della radice. Secondo il modello 1) la probabilità di avere k nodi nel sottoalbero di sinistra è

$$\frac{1}{2^n}\binom{n}{k}$$

Secondo il modello 2) la probabilità che un nodo venga inviato nel sottoalbero di sinistra dipende dal valore della chiave nella radice. Se vi sono $n+1$ chiavi e la chiave nella radice è la chiave p-ma, allora necessariamente $p-1$ vanno nel sottoalbero di sinistra e $n-p+1$ in quello di destra. Siccome la radice può essere una chiave qualsiasi (con $p := 1, \ldots, n+1$) con probabilità uniforme, la probabilità di avere k nodi (con $k := 0, \ldots, n$) nel sottoalbero di sinistra è $1/(n+1)$. Si ottiene il medesimo risultato con un modello in cui la successione di chiavi viene generata casualmente ed una chiave è un reale nell'intervallo $[0, 1]$. Infatti se la chiave è il numero $x \in [0, 1]$, la probabilità che su n numeri generati a caso k siano inferiori a x è

$$x^k\,(1-x)^{n-k}\,\binom{n}{k}$$

Quindi la probabilità di avere un sottoalbero di sinistra con k nodi è

$$\int_0^1 x^k\,(1-x)^{n-k}\,\binom{n}{k}\,dx = \frac{1}{n+1}$$

In base ai due modelli sia $f_1(n)$ la profondità attesa secondo il modello 1 di un albero con n nodi e $f_2(n)$ la profondità attesa secondo il modello 2. Si ha quindi

$$f_1(0) = f_2(0) = 0; \qquad f_1(1) = f_2(1) = 0;$$

e ricorsivamente

$$f_1(n+1) = 1 + \sum_{k=0}^{n} \frac{1}{2^n}\binom{n}{k}\ \max\{f_1(k)\,;\, f_1(n-k)\} =$$

$$\begin{cases} 1 + \dfrac{1}{2^{n-1}} \displaystyle\sum_{k=0}^{\lfloor n/2 \rfloor} \binom{n}{k}\ f_1(n-k) & n \text{ dispari} \\ 1 + \dfrac{1}{2^{n-1}} \displaystyle\sum_{k=0}^{n/2-1} \binom{n}{k}\ f_1(n-k) + \dfrac{1}{2^n}\binom{n}{n/2}\ f_1(n/2) & n \text{ pari} \end{cases}$$

$$f_2(n+1) = 1 + \sum_{k=0}^{n} \frac{1}{n+1}\ \max\{f_2(k)\,;\, f_2(n-k)\} =$$

$$\begin{cases} 1 + \dfrac{2}{n+1} \displaystyle\sum_{k=0}^{\lfloor n/2 \rfloor} f_2(n-k) & n \text{ dispari} \\ 1 + \dfrac{2}{n+1} \displaystyle\sum_{k=0}^{n/2-1} f_2(n-k) + \dfrac{1}{n+1}\, f_2(n/2) & n \text{ pari} \end{cases}$$

Limitazione superiore per f_2. Sia $f_2(n) \le C\, \log_2 n$. Allora da

$$f_2(n+1) \le 1 + \frac{2}{n+1} \sum_{k=0}^{n/2} C\, \log_2(n-k) = 1 + \frac{2\,C}{n+1}\ \log_2 \frac{n!}{n/2!} =$$

$$1 + \frac{2\,C}{n+1}\ \log_2 \frac{n^n\, e^{-n}\, \sqrt{2\,\pi\, n}}{(n/2)^{n/2}\, e^{-n/2}\, \sqrt{\pi\, n}} = 1 + \frac{2\,C}{n+1}\ \log_2(n^{n/2}\, 2^{n/2}\, e^{-n/2}\, \sqrt{2}) =$$

$$1 + \frac{C}{n+1}\ \log_2(n^n\, 2^{n+1}\, e^{-n}) \le 1 + \frac{C}{n+1}\ \log_2((n+1)^{n+1}\, 2^{n+1}\, e^{-n-1}) = 1 + C\, \log_2((n+1)\, 2\, e^{-1}) =$$

$$C\, \log_2 2^{1/C} + C\, \log_2((n+1)\, 2\, e^{-1}) = C\, \log_2(2^{1/C}(n+1)\, 2\, e^{-1})$$

se

$$2^{1/C}\, 2\, e^{-1} < 1 \quad \Longrightarrow \quad 2^{1/C+1} < e \quad \Longrightarrow \quad \frac{1}{C} + 1 < \log_2 e \quad \Longrightarrow \quad C > \frac{1}{\log_2 e - 1} = 2.5889$$

allora $f_2(n+1) < C\, \log_2(n+1)$. Siccome $f_2(1) = 0 = \log_2(1)$, $f_2(2) = 1 = \log_2(2)$, la limitazione superiore vale sempre.

Il calcolo appena eseguito tiene conto della profondità dell'albero, cioè del massimo livello a cui può arrivare una chiave che venga inserita. Questo calcolo è però leggermente pessimistico perché non è detto che una chiave casuale debba scendere fino al massimo livello. Volendo tener conto anche di questo si ha la seguente analisi: sia $A(n)$ il numero medio di confronti che l'n-ma chiave dovrà mediamente fare se inserita in un albero che abbia già $n-1$ chiavi. Eseguiamo il ragionamento pensando che n chiavi sono già inserite e dobbiamo inserire la $(n+1)$-ma. Assumiamo che la radice sia la k-ma chiave delle prime n (k-ma secondo l'ordine delle chiavi, non secondo l'ordine di inserimento in quanto la radice è sempre la prima chiave inserita). Allora $(k-1)$ chiavi stanno nel sottoalbero di sinistra e $(n-k)$ in quello di destra. Con probabilità $k/(n+1)$ la chiave $(n+1)$-ma andrà nel sottoalbero di sinistra e con probabilità $(n+1-k)/(n+1)$ andrà nel sottoalbero di destra. Quindi ricorsivamente indicando con $A(n+1|k)$ il numero medio di confronti condizionato al fatto che la radice è la k-ma chiave

$$A(n+1|k) = 1 + \frac{k}{n+1}\, A(k) + \frac{n+1-k}{n+1}\, A(n+1-k)$$

Siccome la radice può essere in posizione k con probabilità $1/n$ si ha

$$A(n+1) = 1 + \frac{1}{n} \sum_{k=1}^{n} \frac{k}{n+1}\, A(k) + \frac{n+1-k}{n+1}\, A(n+1-k) = 1 + \frac{2}{n\,(n+1)} \sum_{k=1}^{n} k\, A(k)$$

Per risolvere la ricorsione la si riscriva per $A(n)$

$$A(n) = 1 + \frac{2}{n\,(n-1)} \sum_{k=1}^{n-1} k\, A(k)$$

e si riscrivano le due espressioni come

$$(n+1)\,A(n+1) = n+1+\frac{2}{n}\sum_{k=1}^{n} k\,A(k)$$

e

$$(n-1)\,A(n) = n-1+\frac{2}{n}\sum_{k=1}^{n-1} k\,A(k)$$

sottraendo si ha

$$(n+1)\,A(n+1)-(n-1)\,A(n) = 2+\frac{2}{n}n\,A(n)$$

ovvero

$$A(n+1) = A(n)+\frac{2}{n+1}$$

Siccome $A(1)=0$ si ha

$$A(n) = \frac{2}{2}+\frac{2}{3}+\frac{2}{4}+\frac{2}{5}+\cdots+\frac{2}{n} = 2\,(H_n-1)$$

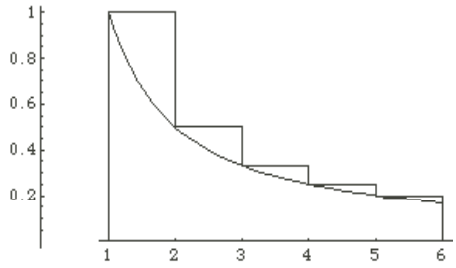
dove i valori $H(n)$ sono i *numeri armonici* definiti da $H_n = \sum_{k=1}^{n} 1/k$.

I numeri armonici furono analizzati già da Eulero. Una dimostrazione che $H_n \to \infty$ per $n \to \infty$ è immediata

$$1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5}+\frac{1}{6}+\frac{1}{7}+\frac{1}{8}+\frac{1}{9}+\frac{1}{10}+\cdots$$

$$\geq 1+\frac{1}{2}+\frac{1}{4}+\frac{1}{4}+\frac{1}{8}+\frac{1}{8}+\frac{1}{8}+\frac{1}{8}+\frac{1}{16}+\frac{1}{16}\cdots = 1+\frac{1}{2}+2\,\frac{1}{4}+4\,\frac{1}{8}+\cdots = 1+\frac{1}{2}+\frac{1}{2}+\frac{1}{2}+\cdots$$

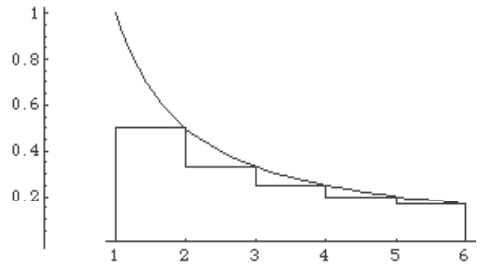
Non è nemmeno difficile ottenere delle espressioni approssimate per H_n. Per ottenere delle limitazioni superiori ed inferiori ad H_n è utile considerare la funzione $1/x$. Per ottenere una limitazione inferiore si consideri la seguente figura dove è riportato il grafico della funzione $1/x$.



Allora si ha che H_{n-1} è uguale alla somma dell'area degli $n-1$ rettangoli da 1 a n ($n=6$ in figura) ed inoltre (ulteriore dimostrazione della divergenza di H_n)

$$H_{n-1} \geq \int_1^n \frac{1}{x}\,dx = \ln n$$

Per ottenere una limitazione inferiore si consideri invece la seguente figura



dalla quale si ha che

$$\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n}\leq\int_1^n\frac{1}{x}\,dx=\ln n$$

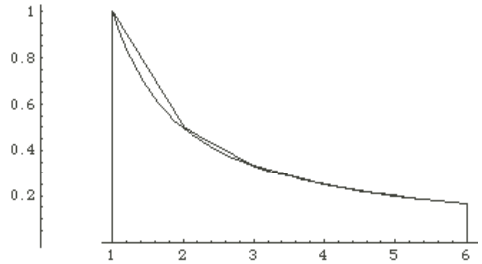
cioè

$$H_n-1\leq\ln n \tag{1}$$

Quindi

$$\ln(n+1)\leq H_n\leq 1+\ln n\quad\Longrightarrow\quad\ln n\leq H_n\leq 1+\ln n$$

Per ottenere una limitazione più stretta possiamo approssimare l'integrale di $1/x$ con dei trapezi come in figura:



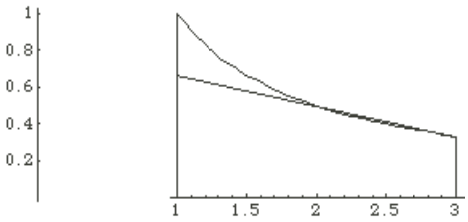
La somma delle aree dei trapezi è

$$\sum_{k=2}^{n}\frac{\frac{1}{k-1}+\frac{1}{k}}{2}=\frac{1}{2}\sum_{k=2}^{n}\frac{1}{k-1}+\frac{1}{2}\sum_{k=2}^{n}\frac{1}{k}=\frac{1}{2}\sum_{k=1}^{n-1}\frac{1}{k}+\frac{1}{2}\sum_{k=2}^{n}\frac{1}{k}=$$

$$\frac{1}{2}+\sum_{k=2}^{n-1}\frac{1}{k}+\frac{1}{2\,n}=\frac{1}{2}-1+\sum_{k=1}^{n}\frac{1}{k}-\frac{1}{n}+\frac{1}{2\,n}=H_n-\frac{1}{2}-\frac{1}{2\,n}$$

quindi

$$H_n-\frac{1}{2}-\frac{1}{2\,n}\geq\ln n$$



Per ottenere una limitazione superiore, consideriamo il prolungamento del segmento congiungente i punti $(n, 1/n)$ e $(n+1, 1/(n+1)$ verso sinistra (in figura $n = 2$) fino ad intersecare la retta verticale $x = n - 1$. Il punto di intersezione ha ordinata $1/n + (1/n - 1/(n+1))$. Il prolungamento verso sinistra è tutto sotto il grafico di $1/x$. Il trapezio che si considera è quindi quello definito dai quattro punti $(n-1, 0)$, $(n, 0)$, $(n, 1/n)$, $(n-1, 1/n + 1/n - 1/(n+1))$ di area

$$\frac{(\frac{1}{n} + (\frac{1}{n} - \frac{1}{n+1})) + \frac{1}{n}}{2} = \frac{3}{2}\frac{1}{n} - \frac{1}{2}\frac{1}{n+1}$$

Quindi

$$\sum_{k=2}^{n} \left(\frac{3}{2}\frac{1}{n} - \frac{1}{2}\frac{1}{n+1}\right) \leq \ln n$$

$$\frac{3}{2}\sum_{k=2}^{n}\frac{1}{n} - \frac{1}{2}\sum_{k=2}^{n}\frac{1}{n+1} = \frac{3}{2}(H_n - 1) - \frac{1}{2}(H_n + \frac{1}{n+1} - \frac{3}{2}) = H_n - \frac{3}{4} - \frac{1}{2(n+1)}$$

da cui

$$H_n \leq \ln n + \frac{3}{4} + \frac{1}{2(n+1)}$$

Le due limitazioni portano quindi al seguente risultato

$$\ln n + \frac{1}{2} + \frac{1}{2n} \leq H_n \leq \ln n + \frac{3}{4} + \frac{1}{2(n+1)} \leq \ln n + \frac{3}{4} + \frac{1}{2n} \tag{2}$$

e asintoticamente

$$\frac{1}{2} \leq \lim_{n\to\infty}(H_n - \ln n) \leq \frac{3}{4}$$

Siccome $H_n - \ln n$ è una funzione decrescente (si veda il modo come si è ottenuta la limitazione $H_n - 1 \leq \ln n$) il limite necessariamente esiste. Il limite, indicato con γ, viene chiamato *costante di Eulero* e vale $0.577216\ldots$. Da (2) si ottiene un'espressione che approssima bene i numeri armonici:

$$H_n \approx \ln n + \frac{1}{2n} + \gamma$$

Per l'analisi sui BST è comunque sufficiente la limitazione fornita da (1) in base alla quale possiamo scrivere

$$A(n) = 2(H_n - 1) \leq 2\ln n = (2/\log_2 e)\log_2 n$$

Quindi costruire un BST ha una complessità media $O(n \log n)$. Il caso peggiore ha ovviamente una complessità $O(n^2)$ (quando le chiavi sono inserite secondo l'ordine). Si può fare in modo che il BST sia sempre bilanciato dopo ogni inserimento con opportune operazioni. Tralasciamo questo argomento. Si noti che

$$C > \frac{1}{\log_2 e - 1} > \frac{2}{\log_2 e}$$

a riprova che l'analisi precedente sulla profondità media di un BST è pessimistica rispetto alla valutazione del numero medio di confronti di una chiave.