

RICERCA IN LISTA ORDINATA

RICERCA(L, K)

IF L ≠ NIL THEN

 IF L → KEY = K

 RET = L

 ELSE IF L → KEY < K THEN

 RET = RICERCA(L → NEXT, K)

 ELSE

 RET = NIL

 ELSE

 RET = NIL

RETURN RET

Il tempo risulta essere lo stesso della lista non ordinata

ovvero lineare $\forall k \mid \forall x \in L \ x \leq k$

CANCELLAZIONE DI UN ELEMENTO IN UNA LISTA ORDINATA

CANCELLA (L, K)

IF $L \neq \text{NIL}$ THEN

 IF $L \rightarrow \text{KEY} = K$ THEN

$\eta = L \rightarrow \text{NEXT}$

 DEALLOCA L

$L = \eta$

 ELSE

 IF $L \rightarrow \text{NEXT} < K$ THEN

$L \rightarrow \text{NEXT} = \text{CANCELLA}(L \rightarrow \text{NEXT}, K)$

 RETURN L

L'algoritmo e' quasi lo stesso, ma questo crea una nuova lista

INSERIMENTO IN LISTA ORDINATA

I casi base sono due: la lista e' vuota oppure l'elemento da inserire e' piu' piccolo della testa.



INSERT(l, k)

IF $l = \text{NIL}$ OR $l \rightarrow \text{KEY} > k$ THEN

$x = \text{ALLOCATELEMENTO}()$

$x \rightarrow \text{KEY} = k$

$x \rightarrow \text{NEXT} = l$

ELSE

IF $l \rightarrow \text{KEY} < k$ THEN

$l \rightarrow \text{NEXT} = \text{INSERT}(l \rightarrow \text{NEXT}, k)$

RETURN l

Ancora una volta nel caso peggiore il tempo è lineare

STRUTTURA DATI ALBERO

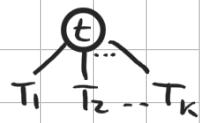
Possiamo implementare il concetto di albero in una struttura

dati partendo dalla lista che avrà però più successori (figli) e un campo che indica il precedente (padre). Rigorosamente:

T è un albero K -ario se

① $T = \text{NIL}$

② $T = \forall 1 \leq i \leq K T_i$ e' un albero K -ario



Possiamo analogamente generalizzare gli algoritmi sulle liste perché funzionino con il nostro albero.

Così come l'albero generalizza la lista, ci concentriremo su **ALBERI ORDINATI** che generalizzano liste ordinate

VISITA DI ALBERI BINARI

A differenza della lista l'albero non è una struttura che può essere rappresentata da una sequenza, quindi lo scorrimento non è sequenziale.

Questo scorrimento dell'albero è detto **VISITA** e può essere effettuata in **PROFOUNDITÀ** (verticale) o in **AMPIEZZA** (orizzontale).

Possiamo vedere la visita come una ricerca sempre al suo caso peggiore, di conseguenza studiare la complessità di visita e ricerca è analoga.

RICERCA DFS Depth first Search

Sfruttiamo la definizione ricorsiva di albero per la ricerca in profondità.

RICERCA DFS (T, k)

IF $T \neq \text{NIL}$ THEN

IF $T \rightarrow \text{Key} = k$ THEN

RET T

ELSE

RET = RICERCA DFS ($T \rightarrow \text{sx}, k$)

IF RET = NIL THEN

Se non era a sinistra

RET = RICERCA DFS ($T \rightarrow \text{dx}, k$)

ricerca a destra

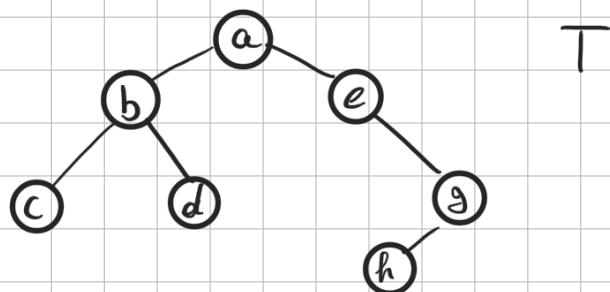
RETURN RET

Se arriva alla foglia e non trova l'elemento, torna sul padre della foglia e va a destra, se non lo trova salire di un livello. Non possiamo fare alcun controllo sul campo Key dei nodi perché non vi è alcuna relazione d'ordine tra i valori, nonostante ci sia una relazione tra padri e figli dell'albero.

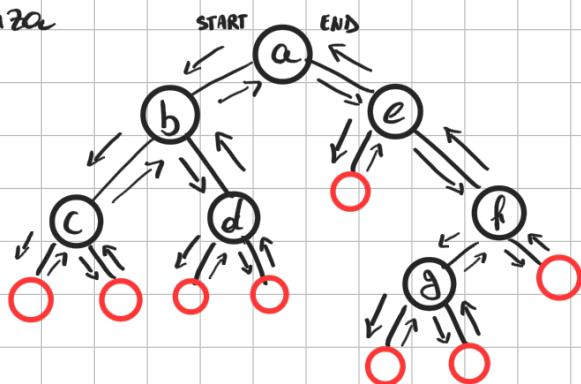
ALBERO DI RICORRENZA DFS (caso peggiore)

Studiamo l'albero di ricorrenza dell'algoritmo di DFS nel caso peggiore dato un albero, notando che l'albero di ricorrenza e l'albero struttura saranno simili in quanto DFS genera solo un figlio alla volta.

K_eT



Albero di ricorrenza



I nodi rossi sono le chiamate con risultato NIL, avremo allora n chiamate (numero di nodi dell'albero struttura T).

Sappiamo quindi che nel caso peggiore $T(m) = \sum (m)$.

Cerchiamo il limite superiore di $T(m)$.

Analizziamo l'albero di ricchezza:

Ha m nodi interni, il numero di nodi totali è dato dalla somma dei nodi interni + il numero di foglie; il numero di nodi totali di un albero binario è $2^{m_f} - 1$ dove m_f è il numero di foglie. Ricaviamo il numero di nodi totali:

$$mn = m + m_f = 2^{m_f} - 1 \rightarrow m = m_f - 1 \rightarrow m_f = m + 1$$

$$mn = 2(m+1) - 1 = 2m + 1$$

Ogni nodo contribuisce localmente a tempo costante quindi

$$T(n) = O(n)$$

Possiamo dire quindi che $T(n) = \Theta(n)$

OSS

Esistono 3 tipi di DFS a seconda dell'ordine di ricerca:

- PRE ORDER (quella vista in precedenza)

Visito la radice e poi i nodi dei sottoalberi

- POST ORDER

Prima visito i sottoalberi e poi la radice

● INORDER

Visito un sottoalbero, poi la radice, poi l'altro sottoalbero

RICERCA BFS Breadth First Search

Vediamo la ricerca in ampiezza in un albero binario.

Ci si pone un problema in quanto i nodi di uno stesso livello non sono tra loro collegati. Dovremo tenere memoria dei nodi poiché, ci aiuterà una CODA.

In particolare l'algoritmo inserisce la radice nella coda, la visita e la rimuove dalla coda, inserendo i due figli della coda, e così via.

BFS (T)

$$Q = \{T\}$$

WHILE ($Q \neq \text{NIL}$) DO

$\Theta(n)$

$x = \text{Testa}(Q)$

Testa della coda

VISITA (x)

Analizza in qualche modo il nodo $\Theta(1)$

FOR EACH $y \in \text{FIGLI}(x)$ DO

Aggiungi i figli $\Theta(2)$

$$Q = \text{ACCODA}(Q, y)$$

alla coda

$\Theta(1)$

$Q = DECODE(Q)$

Togli il primo elemento $O(1)$

Siccome accoda, decoda, testa vengono svolte a tempo costante e l'algoritmo VISTA analizza il contenuto del modo a tempo costante, questo algoritmo ha lo stesso costo di DFS ovvero lineare

$$T(m) = \Theta(m \cdot T_{VISITA})$$

OSS

Seppur il BFS usa la coda, per la natura ricorsiva dell'algoritmo viene occupata memoria nello stack di esecuzione per l'esecuzione del DFS, quindi bisogna eseguire un'analisi più accurata per vedere quale è più efficiente.