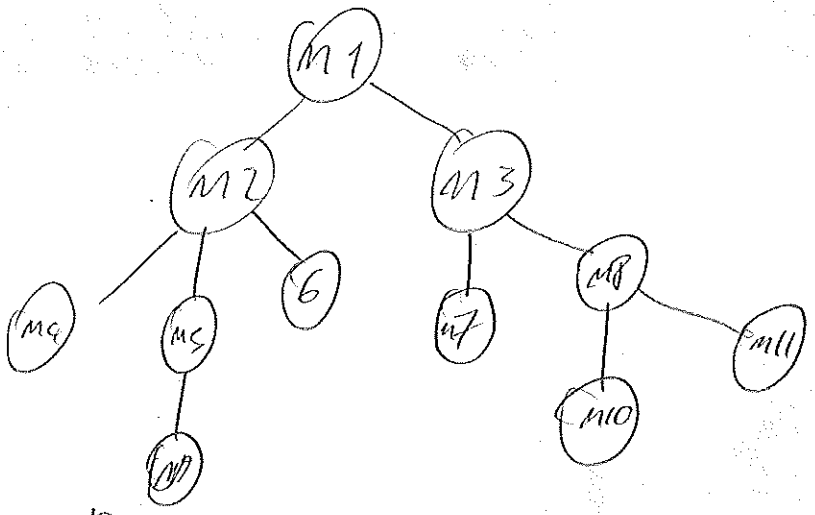
ALBERI

Un albero può essere definito come un insieme di "NODI" collegati da "ARCHI" aventi le seguenti proprietà:

1) Uno dei nodi viene distinto e chiamato "RADICE"
2) Ogni nodo c, diverso dalla radice, è connesso mediante un arco ad un altro nodo f, detto "GENITORE" di c.
3) Ogni albero è connesso, nel senso che si può andare da un qualsiasi nodo diverso dalla radice e ci si sposta nel suo genitore, e poi nel genitore del genitore e così via, per un numero finito di passi raggiungiamo la radice

Se f è genitore di c, allora diciamo che c è figlio di f. Un nodo può avere molti figli, ma anche nessuno. Ogni nodo, diverso dalla radice, ha uno e uno solo genitore.

ESEMPIO



DEFINIZIONE RICORSIVA:

BASE: Un singolo nodo n è un albero ed n è anche la radice.

INDUZIONE: Sia x un nuovo nodo e siano T_1, T_2, T_k suoi k più alberi con radici c_1, c_2, c_k. Ovviamente si chiede che T_1, T_2, T_k non abbiano nodi in comune e che x non appaia in nessuno di questi. Costruiamo un nuovo albero T che ha x come radice e aggiungendo un arco tra x e $c_1, c_2 \ldots c_k$

CAMMINO: Siano m_1, m_2 e m_K una sequenza di nodi in cui m_1 è genitore di m_2, m_2 di m_3 e così. Tale sequenza è detta cammino e $(K-1)$ è la sua lunghezza.
In tale sequenza, m_1 è detto antenato di m_K e m_K discendente di m_1. Se il cammino che unisce m_1 e m_K ha lunghezza 1 allora m_K è un "DISCENDENTE PROPRIO".

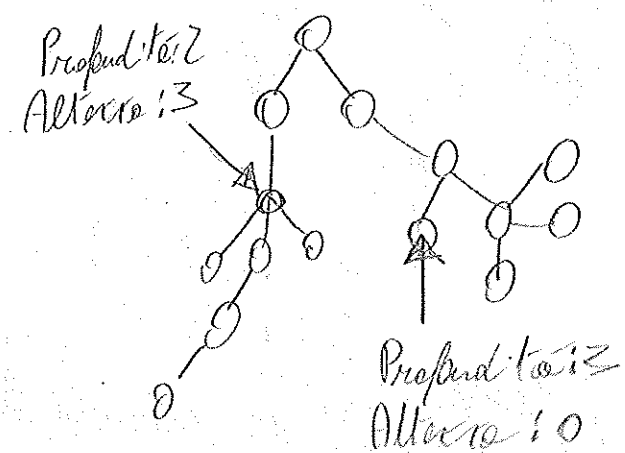
SOTTOALBERO: dato un albero T, un suo sottoalbero è costituito da un nodo n di T insieme a tutti i suoi discendenti propri.

FOGLIA: nodo senza figli

NODO INTERNO: un nodo che ha 1 o più figli

ALTEZZA: l'altezza di un nodo n è il cammino più lungo che va da n a una foglia. L'altezza dell'albero è l'altezza della sua radice

PROFONDITÀ: lunghezza del cammino che va dalla radice a n

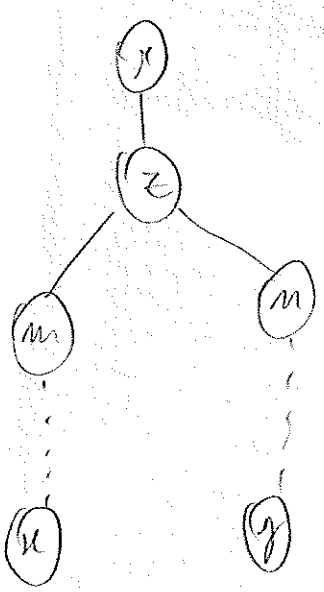


ALBERI ORDINATI

È possibile assegnare un ordine da SX a dX ai nodi.
1) Figli di uno stesso nodo vengono così ordinati da SX a dX.
2) Per ordinare tutto l'albero si stabilisce che se m è a SX di n, allora tutti i discendenti di m stanno a SX di n.
(non antenato/discendente)

Per sapere se due nodi x, y stanno a SX o a dX, prendiamo i cammini che li portano dallo stesso nodo z, i due nodi che portano a z saranno m e n, rispettivamente.
Se m è a SX di n allora, x è a SX di y.

NB: può essere m = x e/o n = y; ma sicuramente m ≠ n perché i due nodi non devono essere nello stesso cammino.

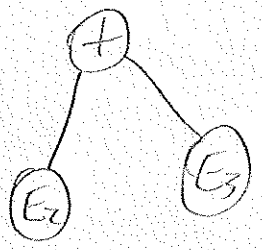


ALBERI ETICHETTATI: sono alberi in cui a ogni nodo viene associato un valore o un etichetta

ALBERI DI ESPRESSIONI

Permettono di eliminare le espressioni.
IDEA: ogni volta che formiamo un'espressione più complessa mettendo insieme due espressioni più piccole costruiamo un nuovo albero aggiungendo un nodo con l'operatore in

in questione e le collegheremo con due sottoalberi che rappresentano le espressioni più piccole.

Es: $E_1 = E_2 + E_3 \Rightarrow$ 

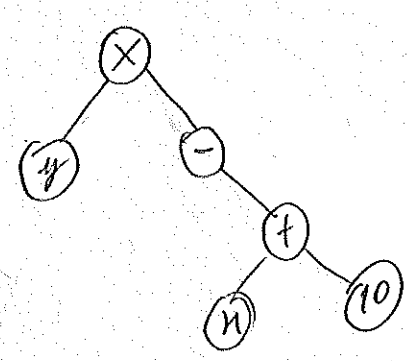
I nodi degli alberi di espressioni possono contenere ovviamente OPERATORI, VARIABILI e NUMERI

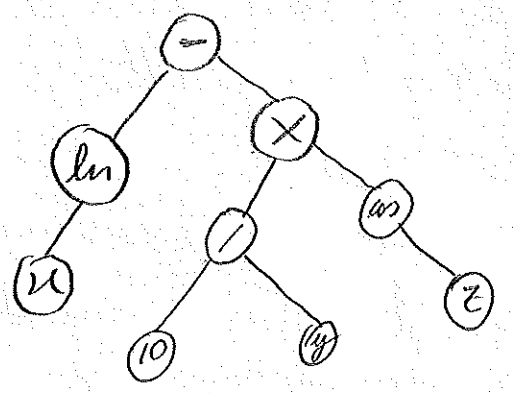
DEFINIZIONE RIGOROSA

BASE: Un unico operando atomico (VARIABILE o NUMERO) è un'espressione e il suo albero è costituito da un unico nodo etichettato con quell'operando.

INDUZIONE: Se E_1 e E_2 sono delle espressioni, l'espressione che si ottiene combinando E_1 e E_2 con un operatore è rappresentata da un albero avente come radice l'operatore e sottoalberi gli alberi di E_1 e E_2

ESEMPI

$(y \times (-(x+10)))$ 

$\left(\ln(x) - \left(\frac{10}{y}\right) \times (\cos z)\right)$ 

N8: Conservare l'ordine degli operatori usando la regola dx, sx

STRUTTURE DATI PER GLI ALBERI

Il modo più semplice di rappresentare un albero è mediante una struttura dati che:

- Contiene l'etichetta
- Contiene "FR" puntatori ai figli del nodo.

FR è detto fattore di ramificazione e può essere una costante oppure può essere dimensionato dinamicamente.
Se lo fanno costante, in generale avrò spreco di memoria perché molti puntatori punteranno a vuoto.

STATICO:

```
TYPE (NODO_ALBERO)
   CHARACTER (LEN=10) :: NOME
   INTEGER, PARAMETER :: FR = 3
   TYPE(NODO_ALBERO), POINTER :: PT_FIGLI(FR)
END TYPE
```

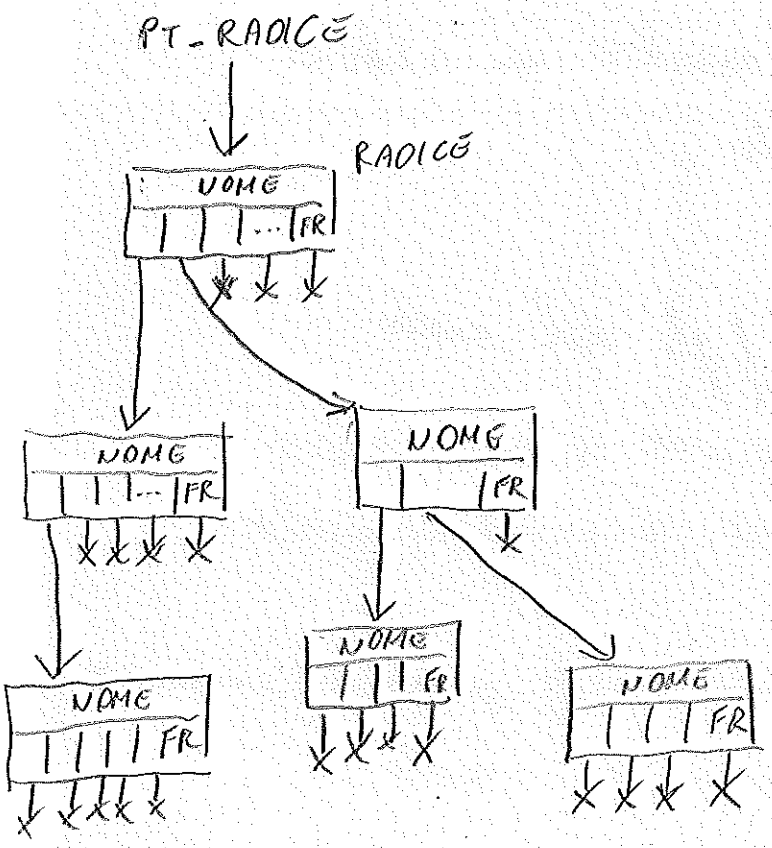
DINAMICO:

```
TYPE (NODO_ALBERO)
   CHARACTER (LEN=10) :: NOME
   INTEGER            :: FR
   TYPE(NODO_ALBERO), POINTER :: PT_FIGLI(:)
END TYPE
```

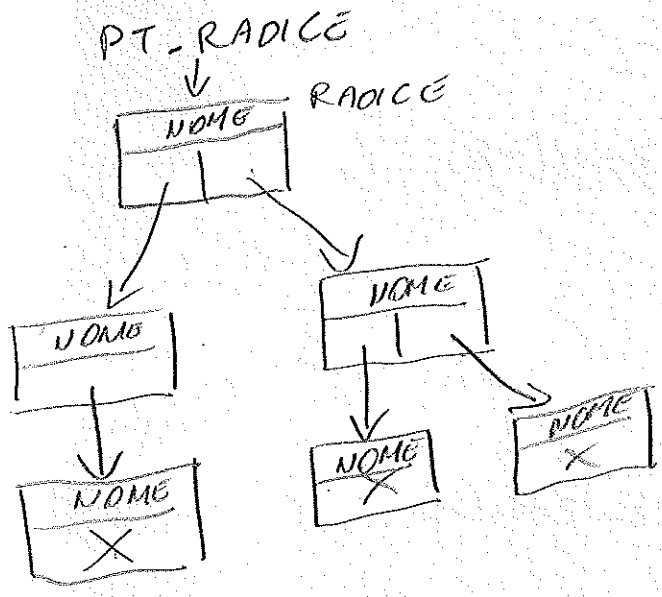
Entrambe le strutture dati (NODO_ALBERO) vengono utilizzate con:

```
TYPE (ALBERO)
   TYPE(NODO_ALBERO), POINTER :: PT_RADICE
END TYPE
```

ALBERO "STATICO"



ALBERO "DINAMICO"



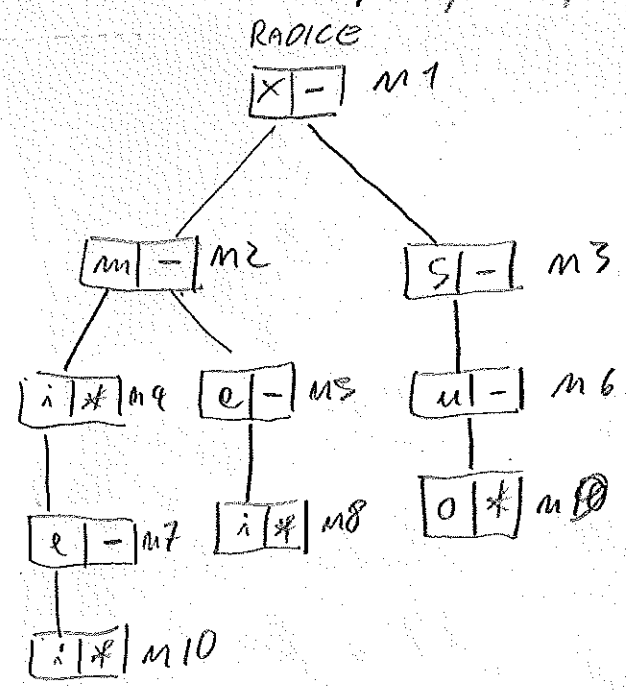
La differenza è che con l'albero dinamico, ogni volta che devo un nuovo figlio, devo decidere quando nodi devo generare

TRIE

Le parole derivano dal termine RETRIEVAL ed è un metodo utile per memorizzare un insieme di parole e per cercare rapidamente se una certa parola appartiene all'insieme

Ad ogni nodo, eccetto la radice, viene associata una lettera e un'etichetta che indica se la parola è finita oppure no.

ESEMPIO: MA, MI, MIEI, SUO (usiamo '*' e '-' per indicare se la parola è completa o no)



Se cerco "mi" lo trovo in due passi

Se cerco "mie" verifico che non esiste in 3 passi

Se cerco "mono" verifico che non esiste in 3 passi.

Se ricerche avranno complessità $O(l)$ con l lunghezza delle parole inserite nel TRIE

Nel caso di un TRIE, se volessimo rappresentare il nostro vocabolario con un albero N-ario connetteremmo un enorme spreco di memoria

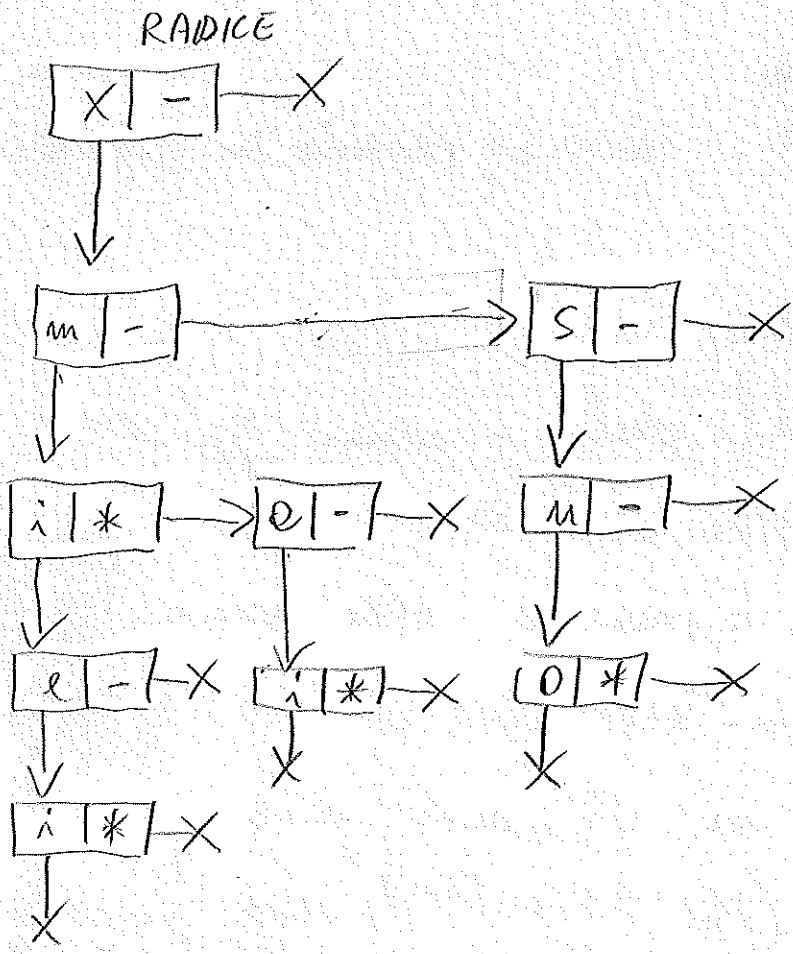
Per esempio, supponendo di usare 21 lettere dell'alfabeto, se volessimo creare un vocabolario di parole lunghe n potremmo usare (21^n+1) nodi e $21(21^n+1)$ puntatori (eventualmente null), Poiché ogni nodo ha 1 solo genitore (tranne la radice) i puntatori non null sarebbero 21^n !!

n	$N=(21^n+1)$	Tot. Archi $=21(21^n+1)$	Archi effettivi 21^n
1	22	462	21
2	442	9282	441
3	9262	194502	9261
4	408402	8576142	4084101
10	$1.668 \cdot 10^{13}$	$3.503 \cdot 10^{14}$	$1.668 \cdot 10^{13}$
20	$2.782 \cdot 10^{26}$	$5.843 \cdot 10^{27}$	$2.782 \cdot 10^{26}$

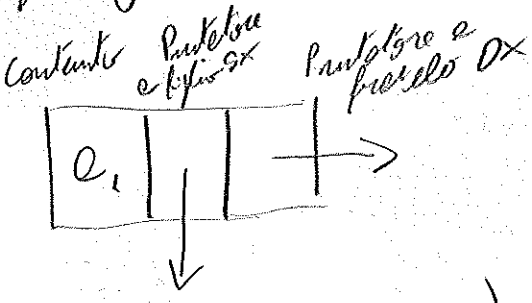
In generale usando solo i nodi utili a partire dalle parole, diciamo $N+1$ (1 per la radice) con un albero N-ario ovvero $21(N+1)$ puntatori di cui solo N saranno non null.

Esiste una rappresentazione binaria. Per il TRIE o per altre con i concetti usare una lista concatenata per memorizzare i figli di un nodo

In questa realizzazione, oltre FIGLIO SINISTRO/FRATELLO (5)
A DESTRA, ogni nodo avrà due puntatori uno al figlio + a SX e l'altro al fratello della stessa DX



In generale il nodo generico sarà rappresentato così




```
TYPE (NODO_ALBERO)
      CHARACTER (LEN=10) :: NOME
      TYPE (NODO_ALBERO), POINTER :: FIGLIO_SX, FRATELLO_DX
END TYPE
TYPE (ALBERO)
   TYPE(NODO_ALBERO), POINTER :: PT_RADICE
END TYPE
```

PUNTATORI AI GENITORI: talvolta può essere necessario "RISALIRE" dalle foglie alla radice e in tal caso si usa anche un puntatore al genitore di un nodo.

```
TYPE (NODO_ALBERO)
    CHARACTER (LEN=10) :: NOME
    TYPE (NODO_ALBERO), POINTER :: FIGLIO_SX, FRATELLO_DX, GENITORE
END TYPE
```

RICORSIONE SUGLI ALBERI

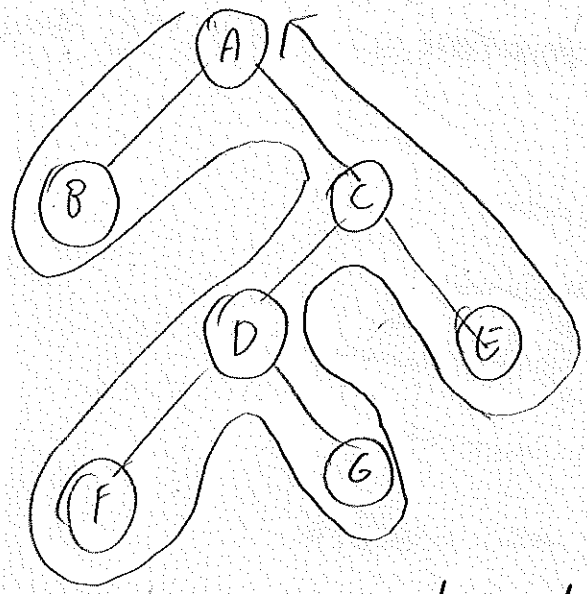
Gli alberi sono molto utili perché si possono applicare facilmente procedure RICORSIVE su di essi.

Ovviamente la procedura ricorsiva avrà come argomento principale un "NODO" dell'albero

Detto P la procedura e n il nodo generico e C_1, C_2, C_K i suoi figli (da SX a DX) una procedura ricorsiva funziona in questo modo:

```
P(n):
    A0 ← Azione 0
    P(C1) ← Chiamata sul figlio C1
    A1 ← Azione 1
    P(C2) ← Chiamata sul figlio C2
    A2 ← Azione 2
    :
    P(CK) ← Chiamata al figlio CK
    AK ← Azione K
```

ESEMPIO:



I nodi vengono visitati nell'ordine indicato in figura.

ORDINE ANTICIPATO:
Tutte le azioni sono nulle tranne $A_0 \Rightarrow$ Eseguo la mia azione alla prima visita di un nodo.
Se per esempio l'azione A_0 fosse la stampa del contenuto con un ordine anticipato otterrei:

ABCDFGE

ORDINE POSTICIPATO: Tutte le azioni sono nulle tranne $A_K \Rightarrow$ Eseguo la mia azione quando ho eseguito tutte le mi chiamate ai figli.
Se A_K fosse la stampa del contenuto del nodo corrente otterrei:

BFGDECA

VALUTAZIONE DI UN'ESPRESSIONE

BASE: Se un nodo è una foglia, l'espressione è il valore.
INDUZIONE: Calcoliamo l'espressione del sottoalbero che ha radice n, come la combinazione, secondo l'operatore contenuto in n delle espressioni che hanno radice nei figli di n.

```
TYPE(NODO_ALBERO_ESPRESSIONE)
    CHARACTER(LEN=1)  :: OPERAZIONE
    REAL(KIND=8)      :: VALORE
    TYPE(NODO_ALBERO_ESPRESSIONE), POINTER :: FIGLIO_SX, FRATELLO_DX
END TYPE
```

OPERAZIONE: '+' → Esegui addizione
'-' → Esegui sottrazione
'*' → Esegui moltiplicazione
'/' → Esegui divisione
'X' → È contenuto un valore

CALCOLA ALTEZZA

BASE: l'altezza di una foglia = 0

INDUZIONE: l'altezza di un nodo interno è pari all'altezza del suo figlio + alto, più 1.

+ algoritmi comuni

BASE: nel caso di una foglia "altezza" è 0

INDUZIONE: calcola l'altezza di tutti i figli ricorsivamente e ne estrai il massimo. Aggiungi 1 al risultato.

```
TYPE (NODO_ALBERO)
    INTEGER :: ALTEZZA
    TYPE(NODO_ALBERO), POINTER :: FIGLIO_SX, FIGLIO_DX
END TYPE
```

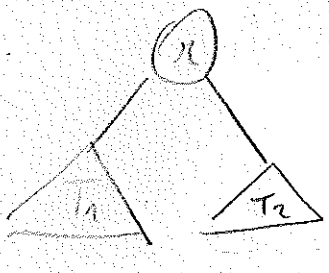
Alberi Binari

Un albero binario è un albero i cui nodi hanno al più 2 figli, figlio SX e figlio DX

Definizione induttiva:

Base: l'albero nullo è un albero binario

Induzione: Sia "n" un nodo e siano T_1 e T_2 due alberi binari. Esiste un albero binario T che ha come radice "n" e sottoalbero SX T_1 e DX T_2. Se T_1 è nullo allora T non ha figlio SX, se T_2 è nullo allora T non ha figlio DX



Struttura dati Fortran 90 per un Albero Binario

```
TYPE NODO-ALBERO_BIN
    CHARACTER(LEN=10) :: NOME
    TYPE (NODO_ALBERO_BIN), POINTER :: FIGLIO_SX, FIGLIO_DX
END TYPE
TYPE ALBERO_BIN
    TYPE(NODO_ALBERO_BIN), POINTER :: PT_RADICE
END TYPE
```

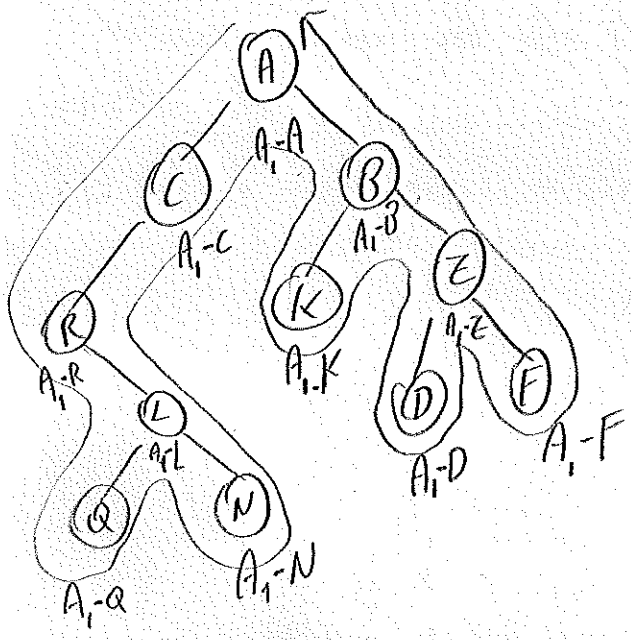
Ricorsione su Alberi Binari

Sia P una procedura ricorsiva che ha come INPUT il nodo generico dell'albero binario, la cui struttura tipo sarà

```
P(n):
    AZIONE A0
    P(n_SX)      <-- chiamata di figli SX e DX
    AZIONE A1
    P(n_DX)
    AZIONE A2
```

Con gli Alberi binari è possibile, oltre alla visita in ordine ANTICIPATO, POSTICIPATO, la visita in ordine SIMMETRICO.
Seguendo tale visita l'unica stringa non nulla è A_1.

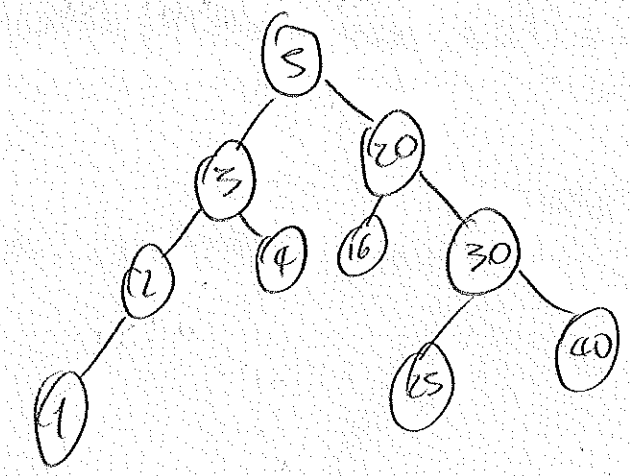
ESEMPIO



R
Q
L
N
C
A
K
B
D
E
F

Gli ALBERI BINARI sono molto utili per realizzare DIZIONARI con ALBERI BINARI DI RICERCA (ABR)
DEFINIZIONE: Un ABR è un albero binario etichettato in cui per ogni nodo x vale la proprietà: tutti i nodi che appartengono al sottoalbero SX hanno etichetta minore di x, quelli appartenenti al sottoalbero DX hanno etichetta maggiore.
N8: non esistono elementi duplicati in un ABR.
Sugli ABR è possibile realizzare le operazioni "CERCA" "INSERISCI" ed "ELIMINA" in maniera molto efficiente.

ESEMPIO



ESEMPIO: SCRIVERE UNA FUNCTION RICORSIVA PER LA VALUTAZIONE DI UNA PILA.

La PILA in questione è dotata di una funzione POP con la seguente interfaccia

```
FUNCTION POP(PILA_IN, ELEMENTO_OUT) RESULT(POP_OK)
IMPLICIT NONE
TYPE(PILA), INTENT(IN) :: PILA_IN
TYPE(OP_VAL), INTENT(OUT) :: ELEMENTO_OUT
LOGICAL :: POP_OK
```

Tale funzione restituisce l'elemento "POPPATO" in ELEMENTO_OUT e .TRUE. in POP_OK se il POP è andato a buon fine.
Restituisce POP_OK = .FALSE. se il POP fallisce (pila vuota)
Ipotizziamo che TYPE(OP_VAL) sarà fatto così:

```
TYPE OP_VAL
    CHARACTER(LEN=1) :: OPER
    REAL(KIND=8)     :: VALORE
END TYPE
```

OPER potrà assumere i valori
- '+' → addizione
- '-' → sottrazione
- '*' → moltiplicazione
- '/' → divisione
- 'V' → valore in VALORE

①

CORREZ. PROVISTE

```
RECURSIVE FUNCTION VALUTA_PILA (PILA_INOUT, VALUTA_OK) RESULT(RIS)
USE CLASS_PILA
IMPLICIT NONE
TYPE(PILA), INTENT(INOUT) :: PILA_INOUT
REAL(KIND=8)              :: RIS      <- LOGICAL, INTENT(OUT) :: VALUTAOK
TYPE(OP_VAL)              :: ELEM
REAL(KIND=8)              :: A,B

! PRENDO ELEMENTO IN CIMA ALLA PILA
VALUTA_OK = POP(PILA-INOUT, ELEM)
IF (.NOT. VALUTA_OK) THEN
   RIS = 0.0
   RETURN
ENDIF
IF (ELEM%OPER .EQ. 'V') THEN
   RIS = ELEM%VALORE

ELSE RIS=0.0
   B = VALUTA_PILA(PILA_INOUT, VALUTA_OK)
   IF (.NOT. VALUTA_OK) RETURN
   A = VALUTA_PILA (PILA_INOUT, VALUTA_OK)
   IF(.NOT. VALUTA_OK ) RETURT
   IF (ELEM%OPER .EQ. '+') THEN
      RIS = A+B
   ELSEIF (ELEM%OPER .EQ. '*') THEN
      RIS = A*B
   ELSEIF (ELEM%OPER .EQ. '-') THEN
      RIS = A-B
   ELSEIF (ELEM%OPER .EQ. '/') THEN
      RIS = A/B
   ENDIF
ENDIF
END FUNCTION VALUTA_PILA
```

Volendo si può realizzare una versione meno "sicura" in cui si interrompe l'esecuzione nel caso di operazione di POP non andata a buon fine.

```
RECURSIVE FUNCTION VALUTA_PILA (PILA_INOUT) RESULT (RIS)
USE CLASS_PILA
IMPLICIT NONE
TYPE (PILA), INTENT (INOUT) :: PILA_INOUT
REAL (KIND=8)               :: RIS
TYPE (OP_VAL)               :: ELEM
REAL (KIND=8)               :: A, B
LOGICAL                     :: OK

OK = POP(PILA_INOUT, ELEM)
IF (.NOT. OK) STOP
IF (ELEM%OPER .EQ. 'V') THEN
    RIS = ELEM%VALORE
ELSE
    B = VALUTA_PILA (PILA_INOUT)
    A = VALUTA_PILA (PILA_INOUT)
    IF (ELEM%OPER .EQ. '+') THEN
       RIS = A+B
    ELSEIF (ELEM%OPER .EQ. '*') THEN
       RIS = A*B
    ELSEIF (ELEM%OPER .EQ. '-') THEN
       RIS = A-B
    ELSEIF (ELEM%OPER .EQ. '/') THEN
       RIS = A/B
    ENDIF
ENDIF
END FUNCTION VALUTA_PILA
```

NB: la versione "sicura" di VALUTA_PILA, nel caso di espressione errata, usciva con .False. in VALUTA_OK e con 0.0 in RIS, così risolveva un problema senza interrompere l'esecuzione. La versione "non sicura" in caso di espressione errata interrompe completamente l'esecuzione del codice!