Divide et impera su alberi

- **Caso base**: per `u` = `null` o una foglia
- **Decomposizione**: riformula il problema per i sottoalberi radicati nei figli di u.
- **Ricombinazione**: ottieni il risultato con `Ricombina`

```
1  Decomponibile(u):
2    IF (u == null) {
3      RETURN valore base;
4    } ELSE {
5      i=0;
       FOR( ciascun figlio f di u ){

6      risultatiFigli[i] = Decomponibile(f);
7      i=i+1 }
8      RETURN Ricombina(risultatiFigli);
9    }
```

La ricombinazione del risultati della chiamate ricorsive sui figli potrebbe essere effettuata anche nel for man mano che vengono ottenuti i risultati delle chiamate sui figli.

Divide et impera su alberi binari

- **Caso base**: per `u` = `null` o una foglia
- **Decomposizione**: riformula il problema per i sottoalberi radicati nei figli `u.sx` e `u.dx`
- **Ricombinazione**: ottieni il risultato con `Ricombina`

```
1  Decomponibile(u):
2    IF (u == null) {
3      RETURN valore base;
4    } ELSE {
5      risultatoSX = Decomponibile(u.sx);
6      risultatoDx = Decomponibile(u.dx);
7      RETURN Ricombina(risultatoSX, risultatoDx);
8    }
```

Analisi dell'algoritmo Decomponibile

- Assumiamo che il tempo per la decomposizione e la ricombinazione sia costante
- Se escludiamo il tempo impiegato per le chiamate ricorsive, l'algoritmo impiega tempo $O(1+ c_v)$, dove c_v è il numero di figli di v
- Se cominciamo la visita dal nodo w, l'algoritmo viene invocato su tutti i discendenti di w

➔Tempo totale = $\sum_{v \in T_w} O(c_v + 1) = O(|T_w|)$

- La visita di tutto l'albero richiede tempo $O(|\mathbf{T}|)$
- Se l'albero ha n nodi la visita richiede tempo $T(n)= O(n)$

Algoritmi ricorsivi su alberi: dimensione

Calcolo della dimensione d = numero di nodi

- Caso base: albero vuoto $\Rightarrow d = 0$
- Caso induttivo: $d = 1+$ dimensione del sottoalbero sinistro $+$ dimensione del sottoalbero destro

```
1  Dimensione( u ):
2    IF (u == null) {
3      RETURN 0;
4    } ELSE {
5      dimensioneSX = Dimensione( u.sx );
6      dimensioneDX = Dimensione( u.dx );
7      RETURN dimensioneSX + dimensioneDX + 1;
8    }
```

Se si vuole conoscere la dimensione di tutto l'albero, si invoca Dimensione con `u` uguale alla radice

Algoritmi ricorsivi su alberi: altezza

Calcolo dell'altezza h di un nodo:

- caso base per `null` $\Rightarrow h = -1$
- passo induttivo: $h = 1+$ massima altezza dei figli

```
1  Altezza( u ):
2    IF (u == null) {
3      RETURN -1;
4    } ELSE {
5      altezzaSX = Altezza( u.sx );
6      altezzaDX = Altezza( u.dx );
7      RETURN max( altezzaSX, altezzaDX ) + 1;
8    }
```

Per calcolare l'altezza dell'albero, si invoca Altezza con `u` uguale alla radice

Visita di un albero binario: inorder

- **simmetrica** (*inorder*):

```
1  Simmetrica( u ):
2    IF (u != null) {
3      Simmetrica( u.sx );
4      elabora(u);
5      Simmetrica( u.dx );
6    }
```

  $O(n)$ tempo per n nodi

Visita di un albero binario: preorder

- **anticipata** (*preorder*):

```
1  Anticipata( u ):
2    IF (u != null) {
3      elabora(u);
4      Antiticipata( u.sx );
5      Antiticipata( u.dx );
6    }
```

  $O(n)$ tempo per n nodi

Visita di un albero binario: postorder

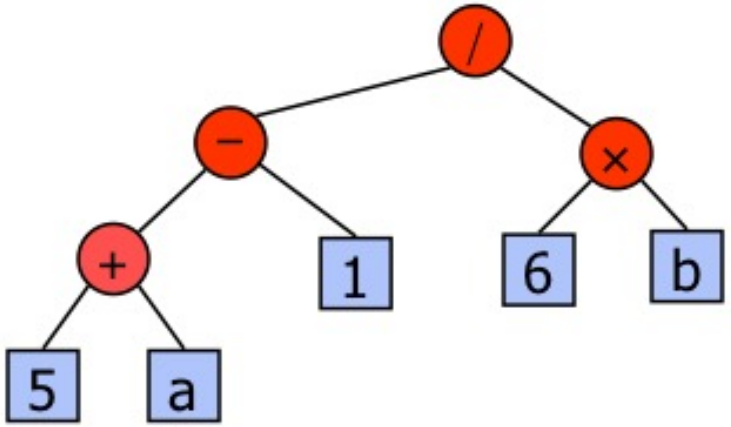
- **posticipata** (*postorder*):

```
1  Posticipata( u ):
2    IF (u != null) {
3      Posticipata( u.sx );
4      Posticipata( u.dx );
5      elabora(u);
6    }
```

  $O(n)$ tempo per n nodi

Esempio dell'uso delle visite: valutazione dell'espressione aritmetica rappresentata da un albero binario

- Albero binario associato ad una espressione:
  - Nodi interni: operatori
  - Nodi esterni: operandi
- Esempio: $((5 + a) - 1) / (6 \times b)$



Uso della visita postorder per valutare l'espressione aritmetica rapprentata da un albero binario

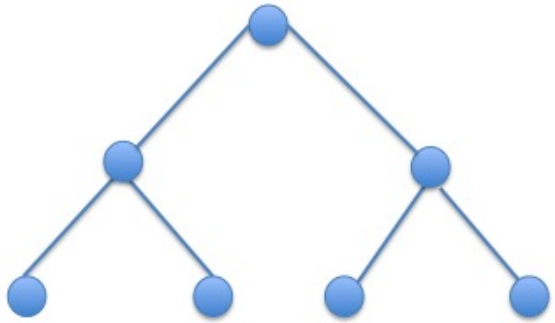
```
   Valuta( u ):
2  1  IF (u==null) {
3       RETURN null;
4     }
5     IF (u.sx == null && u.dx==null) {
6       RETURN u.dato;
7     } ELSE {
8       valSinistra=Valuta( u.sx );
9       valDestra= Valuta( u.dx );
10      ris= Calcola(u.dato,valSinistra ,valDestra);
11      RETURN ris;
12    }
```

- La funzione Calcola invocata su *u.dato*, *valSinistra* e *valDestra*, applica l'operatore memorizzato nel nodo interno *u* ai valori *valSinistra* e *valDestra*.
- N.B.: la condizione del primo if è soddisfatta (*u* è null) solo se inizialmente la funzione Valuta è invocata su null. Se inizialmente Valuta è invocata su un nodo $u \neq null$ allora la condizione del primo if non sarà mai soddisfatta perché quando è invocata su una foglia, la funzione restituisce il contenuto della foglia.

Algoritmo per verificare se un albero binario è completamente bilanciato

Definizioni:

- Albero binario **completo**: ogni nodo interno ha sempre due figli non vuoti
- Albero **completamente bilanciato**: albero completo con tutte le **foglie** alla **stessa profondità**
  Esempio:



Algoritmo per verificare se un albero binario è completamente bilanciato

Indichiamo con $T(u)$ il sottoalbero di T radicato in u

- Risolviamo un problema più generale per $T(u)$, calcolandone anche l'altezza oltre che a dire se è completamente bilanciato o meno
- La ricorsione restituisce una coppia (booleano, intero)
- Tempo di risoluzione: $O(n)$ tempo per n nodi

```
1  CompletamenteBilanciato( u ):
2    IF (u == null) {
3      RETURN <TRUE, -1>;
4    } ELSE {
5      <bilSX,altSX> = CompletamenteBilanciato( u.sx );
6      <bilDX,altDX> = CompletamenteBilanciato( u.dx );
7      bil = bilSX && bilDX && (altSX == altDX);
8      altezza = max(altSX, altDX) + 1;
9      RETURN <bil,altezza>;
10   }
```

Algoritmi ricorsivi su alberi: profondità di un nodo

- La radice ha profondità 0
- I figli della radice hanno profondità pari a 1, e così via
- Un nodo ha profondità p ha i figli a profondità $p + 1$

Versione iterativa dell'algoritmo per calcolare la profondita' di un nodo u

```
p = 0;
WHILE (u.padre != null) {
  p = p + 1;
  u = u.padre;
}
```

Definizione ricorsiva di profondità di un nodo:

- La radice ha profondità 0
- I nodi diversi dalla radice hanno profondità pari alla profondità del padre + 1

Versione ricorsiva dell'algoritmo per calcolare la profondita' di un nodo u

```
1 Profondita( u ):
2   IF (u.padre==null) {
3     RETURN 0;
4   }
5   RETURN profondita(u.padre)+1;
```

Trasmissione dell'informazione tra chiamate ricorsive

- **postorder** : l'informazione è trasferita dalle foglie alla radice
  – la soluzione del problema per $T(\mathtt{u})$ può essere ottenuta dalla soluzioni dei sottoproblemi per $T(\mathtt{u.sx})$ e $T(\mathtt{u.dx})$
- **passaggio dei parametri** : informazione passata attraverso i parametri dalla radice alle foglie
  – la soluzione del problema per $T(\mathtt{u})$ può esserre ottenuta utilizzando l'informazione raccolta dalla radice fino al nodo u

Esempio: stampa la profondità di tutti i nodi

```
1  Profondita( u, p ):
2    IF (u != null) {
3      PRINT profondità di u è pari a p;
4      Profondita( u.sx, p+1 );
5      Profondita( u.dx, p+1 );
6    }
```

Il parametro p indica la profondità del nodo u. Se vogliamo stampare le profondità di tutti i nodi dobbiamo invocare la funzione con u uguale all'indirizzo della radice dell'albero e $p = 0$.

Algoritmo per trovare i nodi cardine

Trasferiamo informazione simultaneamente dalle foglie alla radice e dalla radice verso le foglie combinando i due approcci della slide precedente

- Nodo u è cardine se e solo se $\text{profondita}(u) = \text{altezza}(T(u))$

```
1  Cardine( u, p ):
2    IF (u == null) {
3      RETURN -1;
4    } ELSE {
5      altezzaSX = Cardine( u.sx, p+1 );
6      altezzaDX = Cardine( u.dx, p+1 );
7      altezza = max( altezzaSX, altezzaDX ) + 1;
8      IF (p == altezza) PRINT u.dato;
9      RETURN altezza;
10   }
```