VISITE DI ALBERI

Esistono due tipologie di viste

- Visita in profondità (depth-first search DFS): si visita partendo dalla radice e andando verso le foglie (esistono tre varianti: anticipata o in preordine, posticipata o in postordine, simmetrica o in ordine)
- Visita in ampiezza (breath-first search BFS): visita i nodi per livelli,a partire dalla radice. Non arriva quindi subito alla foglia, visita un livello alla volta.

Per ogni tipologia di visita esiste una implementazione ricorsiva e una implementazione iterativa.

Non esiste una visita migliore, dipende da ciò che dobbiamo fare.

Visita DFS ricorsiva

Algoritmo visitaDFSRicorsiva(nodo n)

1. visita il nodo *n*
2. for each child *n'* of *n*
   visitaDFSRircorsiva(*n'*)

Sia T un albero non vuoto con radice n e k figli $T_1 \dots T_k$

- In **preordine** visito n e poi $T_1 \dots T_k$ (variante implementata dall'algoritmo)
- In **postordine** visito $T_1 \dots T_k$ e poi n
- In **inordine** visito $T_1 \dots T_i$, visito n, visito $T_{i+1} \dots T_k$ per un prefissato i >= 1.

Visita di un nodo: accesso e elaborazione di un nodo

Esempio di applicazione di visita DFS ricorsiva

Serializzazione di un albero

Con serializzazione si intende la ricerca di una funzione non ambigua che consenta di individuare i sottoalberi di ogni albero. La funzione di serializzazione deve essere quindi invertibile (biunivoca).

$$f: tree \rightarrow stringa$$
$$f^{-1}: stringa \rightarrow tree$$

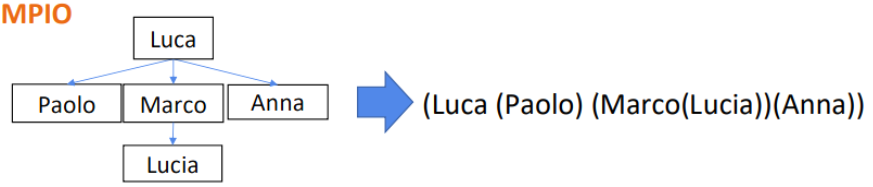
Si può esprimere la serializzazione mediante la rappresentazione di un nodo con una coppia di parentesi bilanciate (...).

FORMALMENTE

Sia *T* un albero non vuoto con radice *n* e *k* figli $T_1,\dots,T_k$

serializzazione(n): (*n* serializzazione(T_1)... serializzazione(T_k))

ESEMPIO



Si tratta di una visita in preordine perchè si stampa prima il valore del nodo e poi si prosegue con la visita dei sottoalberi.

visita del nodo: stampa dell'apertura di parentesi e del contenuto del nodo.

```
void serialize(tree t){
    cout<<"(";
    print(get_info(t));
    tree t1 = get_firstChild(t);
    while(t1!=NULL){
        serialize(t1);
        t1 = get_nextSibling(t1);
    }
    cout<<")";
}
```

Visita del nodo

Chiamata ricorsiva ai figli

Calcolo dell'altezza di un albero

L'altezza misura il massimo delle lunghezze dei cammini che vanno dalla radice alle sue foglie.

- Se T è composto da un solo nodo n allora la sua altezza è 0
- Se T è un albero con radice n e k figli $T_1,\ldots,T_k$ allora la sua altezza è il massimo dell'altezza di $T_1,\ldots,T_k$ incrementata di 1:
  `altezza(T)=max(altezza(T1),…,altezza(Tk))+1`

L'altezza può essere calcolata mediante una DFS post-ordine

Osservazioni

Ogni visita è realizzata come una sequenza di visite ai suoi nodi dove, in ogni passo, ci sono dei nodi aperti.

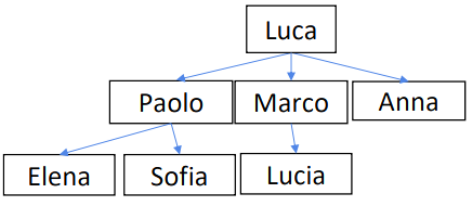
Nodi aperti: nodi che rappresentano i punti di ramificazione rimasti in sospeso e da cui la visita deve proseguire.

Nel caso della DFS i nodi aperti sono i figli di un nodo padre da cui riprende la visita quando un suo sottalbero è esplorato

Visita BFS

In questo caso la lista dei nodi aperti deve essere gestita attraverso una coda.

ESEMPIO



1. Visito Luca
Enqueue Paolo, Marco, Anna
Nodi aperti: Paolo, Marco, Anna
2. Dequeue Nodi aperti: Visito Paolo
Enqueue Elena Sofia
Nodi aperti: Marco, Anna, Elena, Sofia
3. Dequeue Nodi aperti: Visito Marco
Enqueue Lucia
Nodi aperti: Anna, Elena, Sofia, Lucia
3. Dequeue Nodi aperti: Visito Anna
4. Dequeue Nodi aperti: Visito Elena
5. Dequeue Nodi aperti: Visito Sofia
6. Dequeue Nodi aperti: Visito Lucia

Si può implementare il tutto con un algoritmo iterativo.

Algoritmo visitaBFSIterattiva(nodo n)

1. coda C
2. enqueue(C,n)
3. while (C!=NULL)
   a. n=dequeue(C)
   b. visita nodo n
   c. for each child *n'* of *n*
      enqueue(*n'*)

Il tipo di dato CodaBfs è definito nel seguente modo

```
struct elemBFS
{
        node* inf;
        elemBFS* pun ;
};

typedef elemBFS* lista;

typedef struct{
 lista head;
 elemBFS* tail;} codaBFS;
```

Le primitive del modulo sono le seguenti:

```
codaBFS enqueue(codaBFS, node*);
node* dequeue(codaBFS&);
node* first(codaBFS);
bool isEmpty(codaBFS);
codaBFS newQueue();
static elemBFS* new_elem(node*);
```

La struttura dati tree non è adatta a questo tipo di operazioni e sarebbe necessaria una struttura di appoggio.