Università di Pisa

Pietro Ducange

Algoritmi e strutture dati
Alberi Binari

a.a. 2020/2021

Si ringrazia la prof. Nicoletta De Francesco per aver messo a disposizione la maggior parte delle slide utilizzate nella presente lezione

Complessità algoritmi

Ripasso veloce

complessità di un algoritmo

funzione (sempre positiva) che associa alla dimensione del problema il costo della sua risoluzione

Costo: tempo, spazio (memoria), ...

dimensione: dipende dai dati

Per confrontare due algoritmi si confrontano le relative funzioni di complessità

complessità dei programmi : esempio

$T_P(n)$ = Complessità del tempo di esecuzione del programma P al variare di n:

```
int max(int a[], int n) {
  int m=a[0];
  for (int i=1; i < n;i++)
     if (m < a [ i ]) m = a[i];
  return m;
}
```

Se tutti i tempi costanti sono uguali a 1:

$$T_{max}(n) = 4n$$

E' necessario trovare un metodo di calcolo della complessità che misuri l'efficienza come proprietà dell'algoritmo, cioè astragga

- **dal computer su cui l'algoritmo è eseguito**
- **dal linguaggio in cui l'algoritmo è scritto**

L'efficienza deve essere misurata indipendentemente anche da specifiche dimensioni dei dati:

la funzione della complessità deve essere analizzata nel suo comportamento asintotico

Notazione O grande (limite asintotico superiore)

f(n) è di ordine O(g(n)) se esistono

un intero n_0 ed una costante $c>0$ tali che

per ogni $n \geq n_0$: $f(n) \leq c\, g(n)$

$O(\ f(n)\ )$ = insieme delle funzioni

di ordine $O(\ f(n)\ )$

$$O(\ n\ ) = \{\ \text{costante},\ n,\ 4n,\ 300n,\ 100 + n,\ ..\ \}$$

$$O(\ n^2\ ) = O(\ n\ ) \cup \{\ n^2,\ 300\ n^2,\ n + n^2,\ \ldots\ \}$$

Classi di Complessità

$O(1)$	costante
$O(\log n)$	logaritmica $(\log_a n = \log_b n \log_a b)$
$O(n)$	lineare
$O(n \log n)$	nlogn
$O(n^2)$	quadratica
$O(n^3)$	cubica
..	
$O(n^p)$	polinomiale
$O(2^n)$	esponenziale
$O(n^n)$	esponenziale

Programmi ricorsivi : definizioni iterative e induttive

Fattoriale di un numero naturale : n!

0!=1

n! = 1 x 2 x … n se n>0 **definizione iterativa**

0!=1

n!=n*(n-1)! se n>0 **definizione induttiva (o ricorsiva)**

fattoriale: algoritmo iterativo

0!=1

n! = 1 x 2 x … n

```
int fact(int n) {
  if (n == 0) return 1;
  int a=1;
  for (int i=1; i<=n; i++) a=a*i;
  return a;
}
```

fattoriale: algoritmo ricorsivo

0!=1

n!=n*(n-1)! se n>0

```
int fact(int x) {
  if (x == 0) return 1;
  else return x*fact(x-1);
}
```

Regole da rispettare

Regola 1

individuare i casi base in cui la funzione è definita immeditamente

Regola 2

effettuare le chiamate ricorsive su un insieme più "piccolo" di dati

Regola 3

fare in modo che alla fine di ogni sequenza di chiamate ricorsive, si ricada in uno dei casi base

Complesssità dei programmi ricorsivi

```
int fact(int x) {
  if (x == 0) return 1;
  else return x*fact(x-1);
}
```

$T(0) = a$
$T(n) = b + T(n-1)$

Relazione di ricorrenza

soluzione

$$T(0) = a$$
$$T(n) = b + T(n-1)$$

$T(0) = a$

$T(1) = b + a$

$T(2) = b + b + a = 2b + a$

$T(3) = b + 2b + a = 3b + a$

.

.

$T(n) = nb + a$

$T(n)$ è $O(n)$

Riferimenti Bibliografici

Demetrescu:

Capitolo 1 e 2

Cormen:

Capitolo 1,2,3

Alberi Binari

Alberi binari

- **NULL è un albero binario;**
- **un nodo p più due alberi binari Bs e Bd forma un albero binario**

p è radice

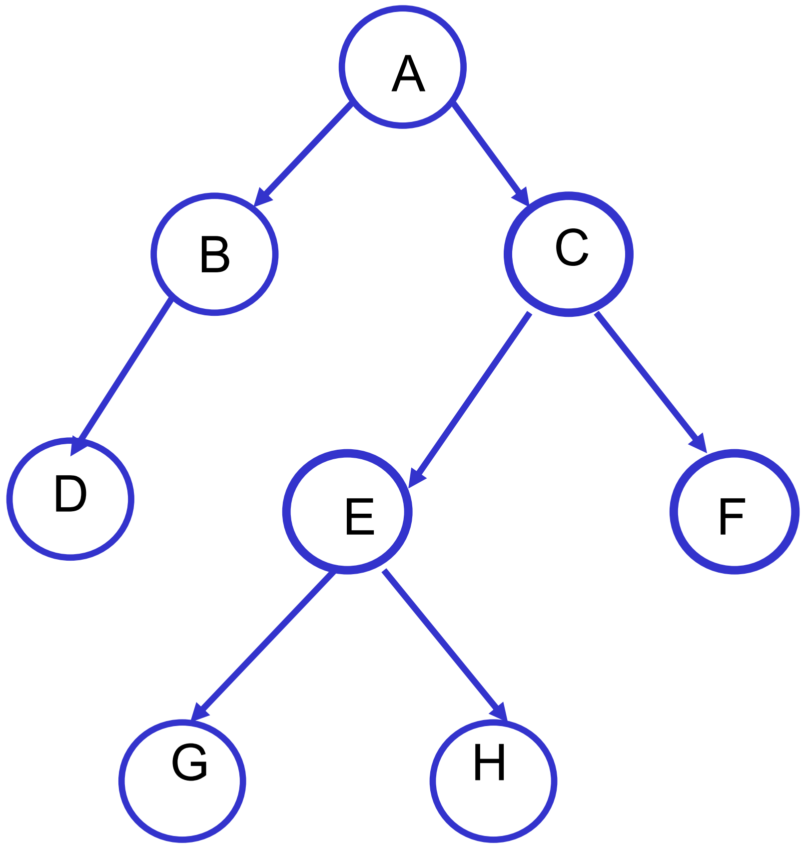
Bs è il sottoalbero sinistro di p

Bd il sottoalbero destro di p

alberi etichettati

Alberi binari

- **padre**
- **figlio sinistro (figlio destro)**
- **antecedente**
- **foglia**
- **discendente**
- **livello di un nodo**
- **livello dell'albero**



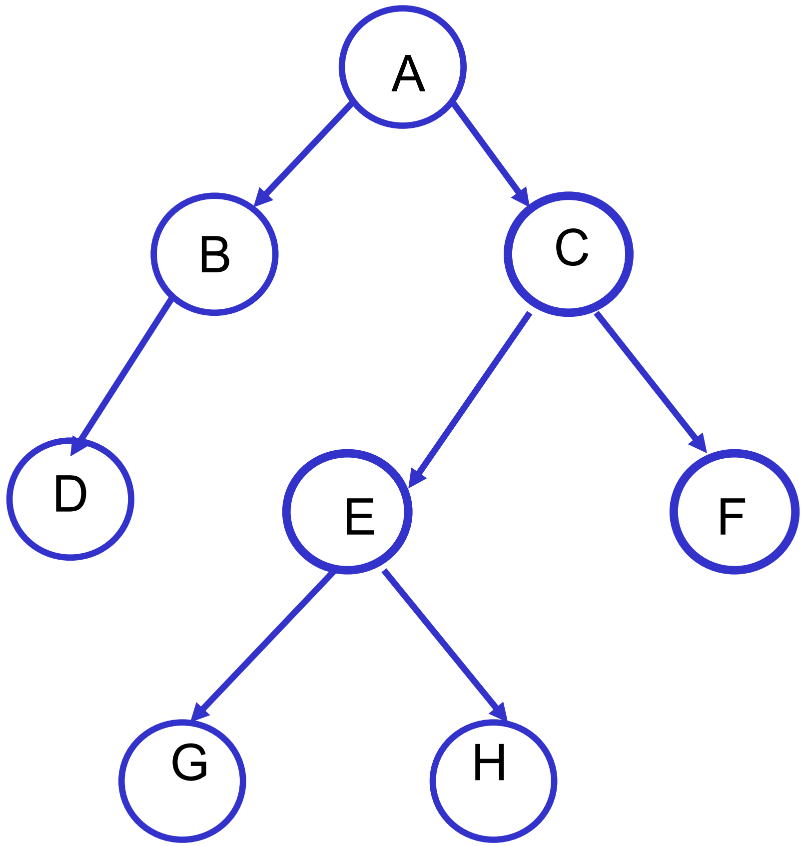
Un Esempio

Assumiamo che il livello di un albero vuoto sia −1

Il livello della radice è 0

Il livello dell'albero è il più lungo cammino fra la radice e una foglia

Un **albero binario etichettato** è un albero binario in cui ad ogni nodo è associato un nome, o etichetta.

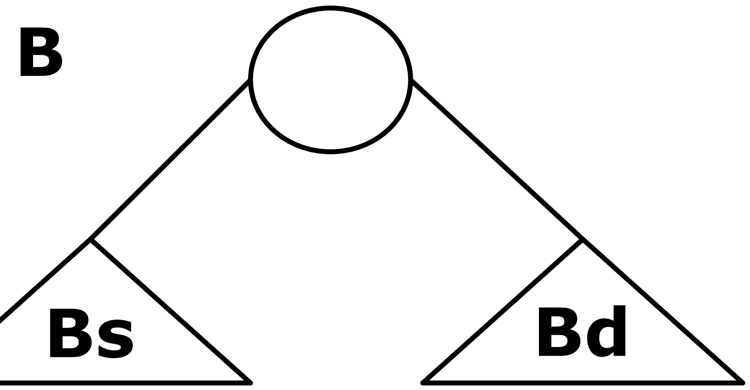


Ricorsione su alberi binari

caso base — albero vuoto (NULL)

caso ricorsivo — radice + due sottoalberi

B = vuoto



Visite di Alberi Binari

Le operazioni più comuni sugli alberi sono quelle di linearizzazione, ricerca, inserimento, e cancellazione di nodi.

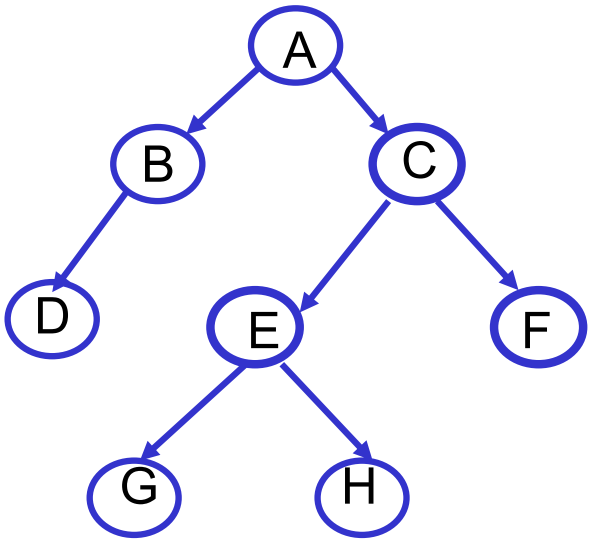
Una linearizzazione di un albero è una sequenza contenente i nomi dei suoi nodi.

Le più comuni linearizzazioni, dette visite, degli alberi binari sono tre:

- **ordine anticipato (preorder)**
- **ordine differito (postorder)**
- **ordine simmetrico (inorder)**

Visita anticipata (preorder)

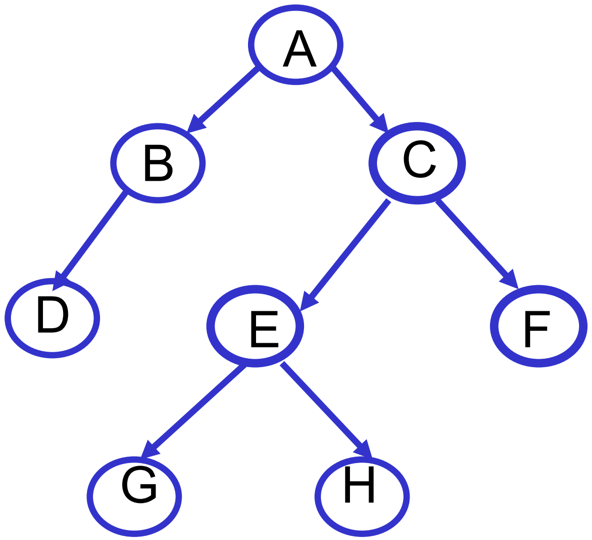
```
void preOrder ( albero ) {
        se l'albero e' vuoto termina;
        altrimenti {
            esamina la radice;
            preOrder ( sottoalbero sinistro);
            preOrder ( sottoalbero destro);
        }
}
```



A B D C E G H F

Visita differita (postorder)

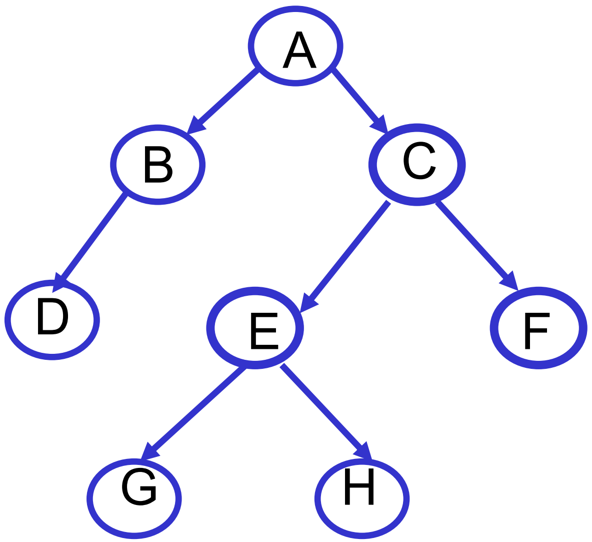
```
void postOrder ( albero ) {
        se l'albero e' vuoto termina;
        altrimenti {
          postOrder ( sottoalbero sinistro);
          postOrder ( sottoalbero destro);
          esamina la radice;
        }
}
```



D B G H E F C A

Visita simmetrica (inorder)

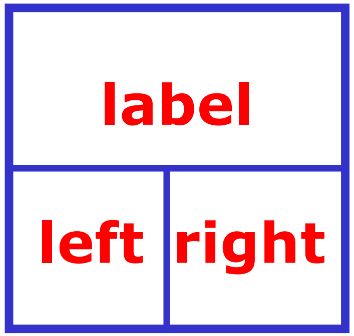
```
void inOrder ( albero ) {
        se l'albero e' vuoto termina;
        altrimenti {
          inOrder ( sottoalbero sinistro);
          esamina la radice;
          inOrder ( sottoalbero destro);
        }
}
```

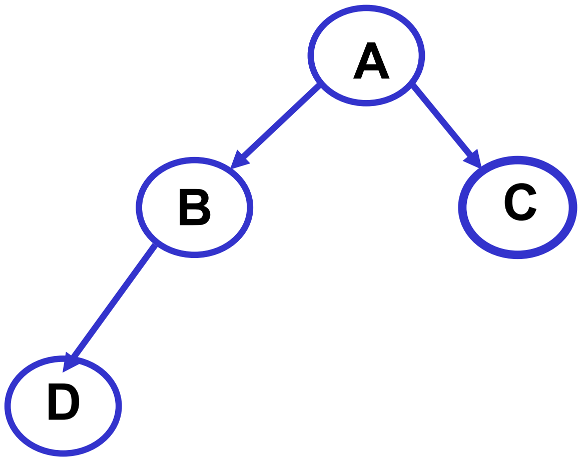


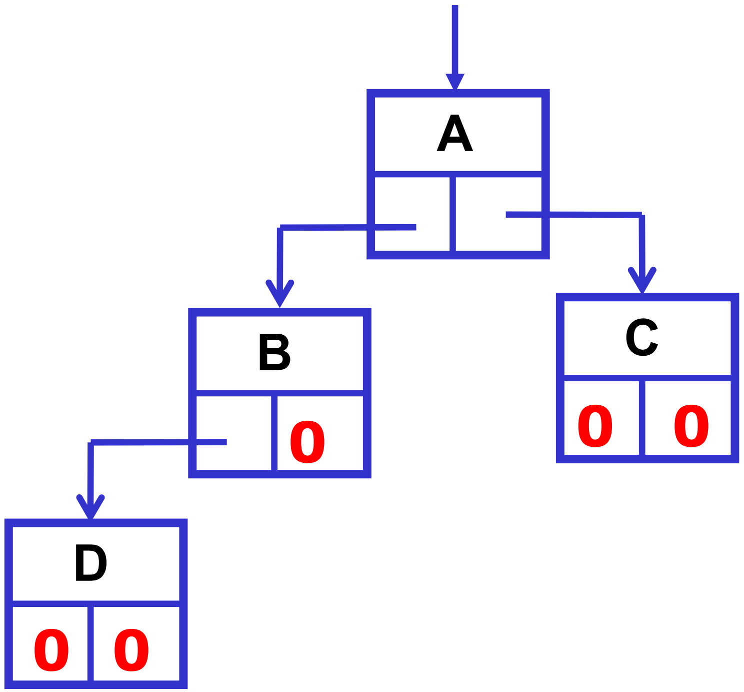
D B A G E H C F

Memorizzazione in lista multipla

```
struct Node {
   InfoType label;
   Node* left;
   Node* right;
 };
```







visite in C++

```
void preOrder(Node* tree)
{   if (!tree) return;
    else {
      <esamina tree->label>;
      preOrder(tree->left);
      preOrder(tree->right);
      }
 }
```

```
void preOrder(Node* tree) {
    if (!tree) return;
    else {
        cout << tree->label;
        preOrder(tree->left);
        preOrder(tree->right);
      }
}
```

Visite in C++

```
void postOrder(Node* tree) {
   if (!tree) return;
   else {
     postOrder(tree->left);
     postOrder(tree->right);
     <esamina tree->label>;
   }
}
```

```
void inOrder(Node* tree) {
   if (!tree) return;
   else {
     inOrder(tree->left);
     <esamina tree->label>;
     inOrder(tree-> right);
   }
}
```

Complessità delle visite

Complessità in funzione del numero di nodi:

$T(0) = a$

$T(n) = b+T(n_s)+T(n_d)$ con $n_s+n_d=n-1$ $n>0$

Caso particolare:

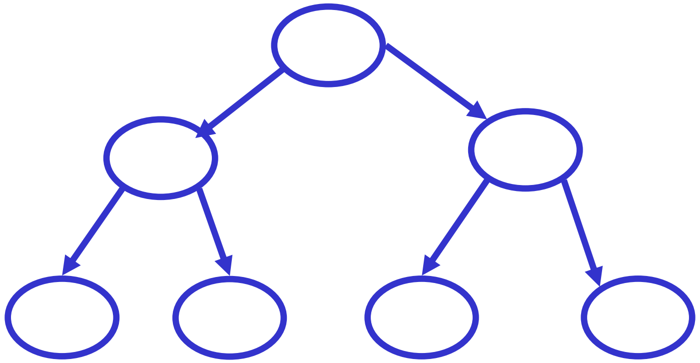
$T(0) = a$

$T(n) = b+2T((n-1)/2)$

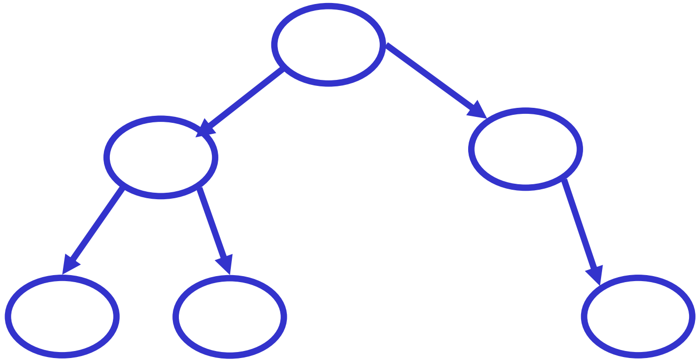
$$\boxed{T(n) \in O(n)}$$

Alberi binari bilanciati

i nodi di tutti i livelli tranne quelli dell'ultimo hanno due figli



bilanciato



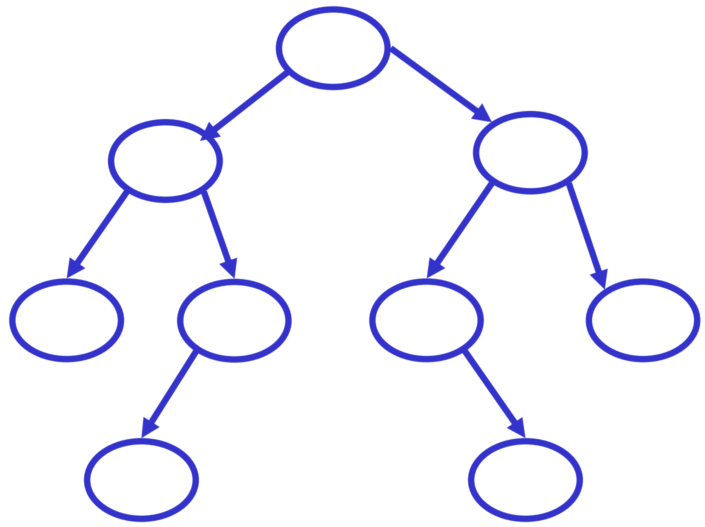
non bilanciato

Un albero binario bilanciato con livello k ha $2^{(k+1)}$ -1 nodi e 2^k foglie

Alberi binari quasi bilanciati

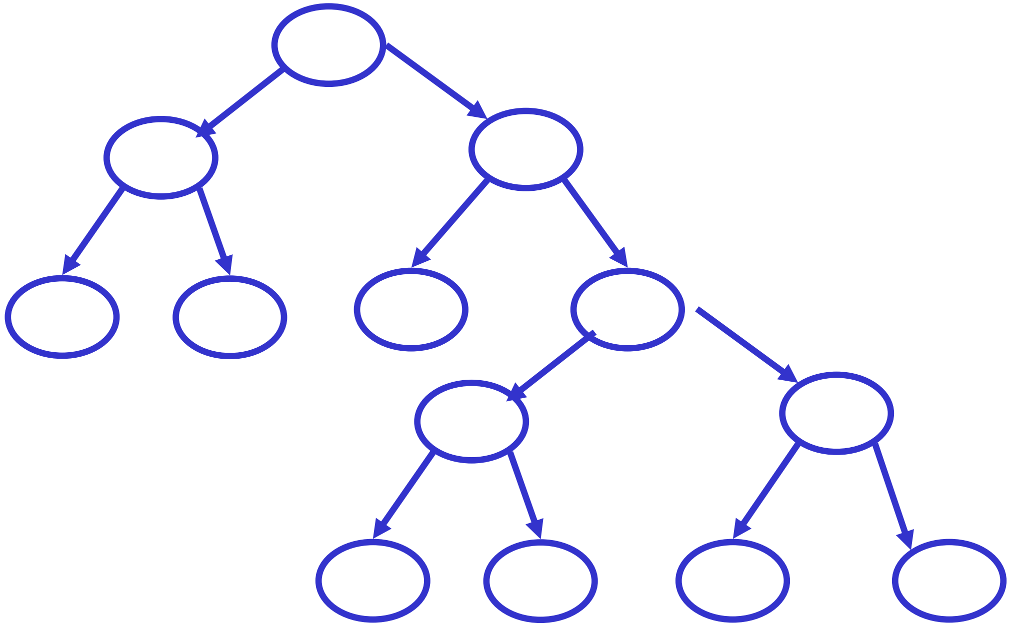
fino al penultimo livello è un albero bilanciato

(un albero bilanciato è anche quasi bilanciato)



Alberi pienamente binari

Tutti i nodi tranne le foglie hanno 2 figli



Un albero binario pienamente binario ha tanti nodi interni quante sono le foglie meno 1

Complessità delle visite nel numero dei livelli

Complessità in funzione dei livelli (se l'albero è bilanciato):

$T(0) = a$

$T(k) = b+2T(k-1)$

$$\boxed{T(k) \in O(2^k)}$$

Funzioni su Alberi

Alberi binari: conta i nodi e le foglie

conta i nodi

```
int nodes (Node* tree) {
    if (!tree) return 0;                    // albero vuoto
    return 1+nodes(tree->left)+nodes(tree->right);
}
```

conta le foglie

```
int leaves (Node* tree) {
    if (!tree) return 0;                    // albero vuoto
    if ( !tree->left && !tree->right ) return 1;  // foglia
    return leaves(tree->left)+leaves(tree->right);
}
```

$$\boxed{T(n) \in O(n)}$$

Alberi binari: cerca un'etichetta

restituisce il puntatore al nodo che contiene l'etichetta n. Se l'etichetta non compare nell'albero restituisce NULL. Se più nodi contengono n, restituisce il primo nodo che si incontra facendo la visita anticipata

```
Node* findNode (Infotype n, Node*tree) {
   if (!tree) return NULL;        //albero vuoto: l'etichetta non c'è
   if (tree->label==n)            // trovata: restituisce il puntatore
       return tree;
   Node* a=findNode(n, tree->left);   // cerca a sinistra
   if (a) return a;               // se trovata restituisce il puntatore
       else return findNode(n, tree->right); // cerca a destra
 }
```

Alberi binari: cancella tutto l'albero

```
void delTree(Node* &tree) {
  if (tree) {
    delTree(tree->left);
    delTree(tree->right);
    delete tree;
    tree=NULL;          }
}
```

alla fine il puntatore deve essere NULL

Alberi binari: inserisci un nodo

inserisce un nodo (son) come figlio di father, sinistro se c='l', destro se c='r'. Restituisce 1 se l'operazione ha successo, 0 altrimenti. Se l'albero è vuoto, inserisce il nodo come radice. Se father non compare nell'albero o ha già un figlio in quella posizione, non modifica l'albero

```
int insertNode (Node* & tree, InfoType son, InfoType father, char c){
      if (!tree) {                    // albero vuoto
      tree=new Node;
      tree ->label=son;
      tree ->left = tree ->right = NULL;
      return 1;
   }
```

Alberi binari: inserisci un nodo (cont.)

```
Node* a=findNode(father,tree);    //cerca father
if (!a) return 0;                 //father non c'è
if (c=='l' && !a->left) {   //inserisci come figlio sinistro e verifica
che non esista già un figlio
    a->left=new Node;
    a->left->label=son;
    a->left->left =a->left->right=NULL; //imposta la foglia
    return 1;
  }
```

Alberi binari: inserisci un nodo (cont.)

```
if (c=='r' && !a->right) {          //inserisci come figlio destro
    a->right=new Node;
    a->right->label=son;
    a->right->left = a->right->right = NULL;
    return 1;
  }
  return 0;                          //inserimento impossibile
}
```

```
int insert(Node*& root, LabelType son, LabelType father, char c) {
    if (!root) {
        root=new Node;
        root->label=son; root->left = root->right = NULL;
        return 1;
    }
    Node* a=findNode(father,root);
    if (!a) return 0;
    if (c=='l' && !a->left) {
        a->left=new Node;
        a->left->label=son; a->left->left = a->left->right = NULL;
        return 1;
    }
    if (c=='r' && !a->right) {
        a->right=new Node;
        a->right->label=son; a->right->left = a->right->right = NULL;
        return 1;
    }
    return 0;
}
```

Class BinTree

```
template<class InfoType>
class BinTree {
   struct Node {
           InfoType label;
           Node *left, *right;
    };
   Node *root;
   Node* findNode(InfoType, Node*);
   void preOrder(Node*);
   void inOrder(Node*);
   void postOrder(Node*);
   void delTree(Node*&);
   int insertNode(Node*&, InfoType, InfoType, char)
```

Class BinTree

```
public:
    BinTree() { root = NULL; };
    ~BinTree(){ delTree(root); };
    int find(InfoType x) { return findNode(x, root); };
    void pre() { preOrder(root); };
    void post(){ postOrder(root); };
    void in()  { inOrder(root); };
    int insert(InfoType son, InfoType father, char c) {
        insertNode(root,son, father,c);
        };
};
```

Riferimenti Bibliografici

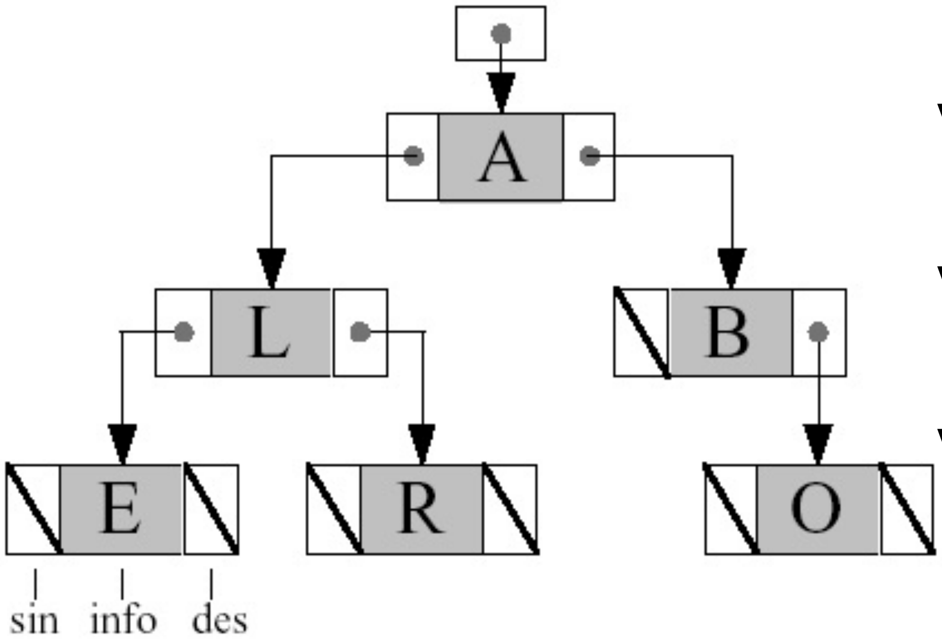
Demetrescu:

Paragrafo 3.3

Cormen:

Capitolo ???

Esercizio



Visita Simmetrica:?

Visita in Preordine:?

Visita in Postordine:?