

Alberi

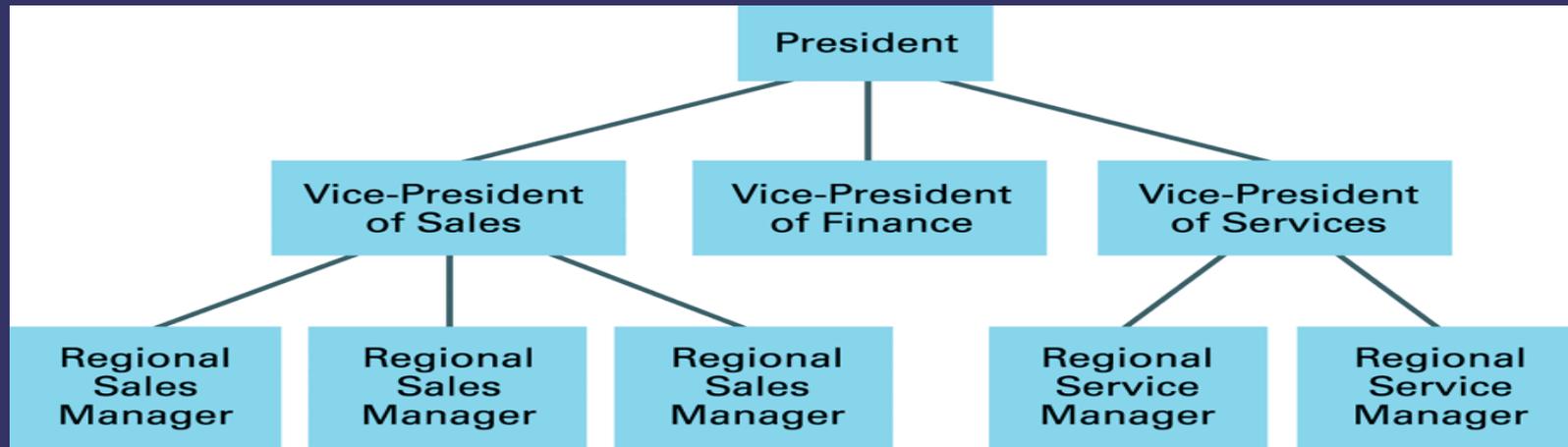
# Struttura dati Albero (Tree)

- Organigramma

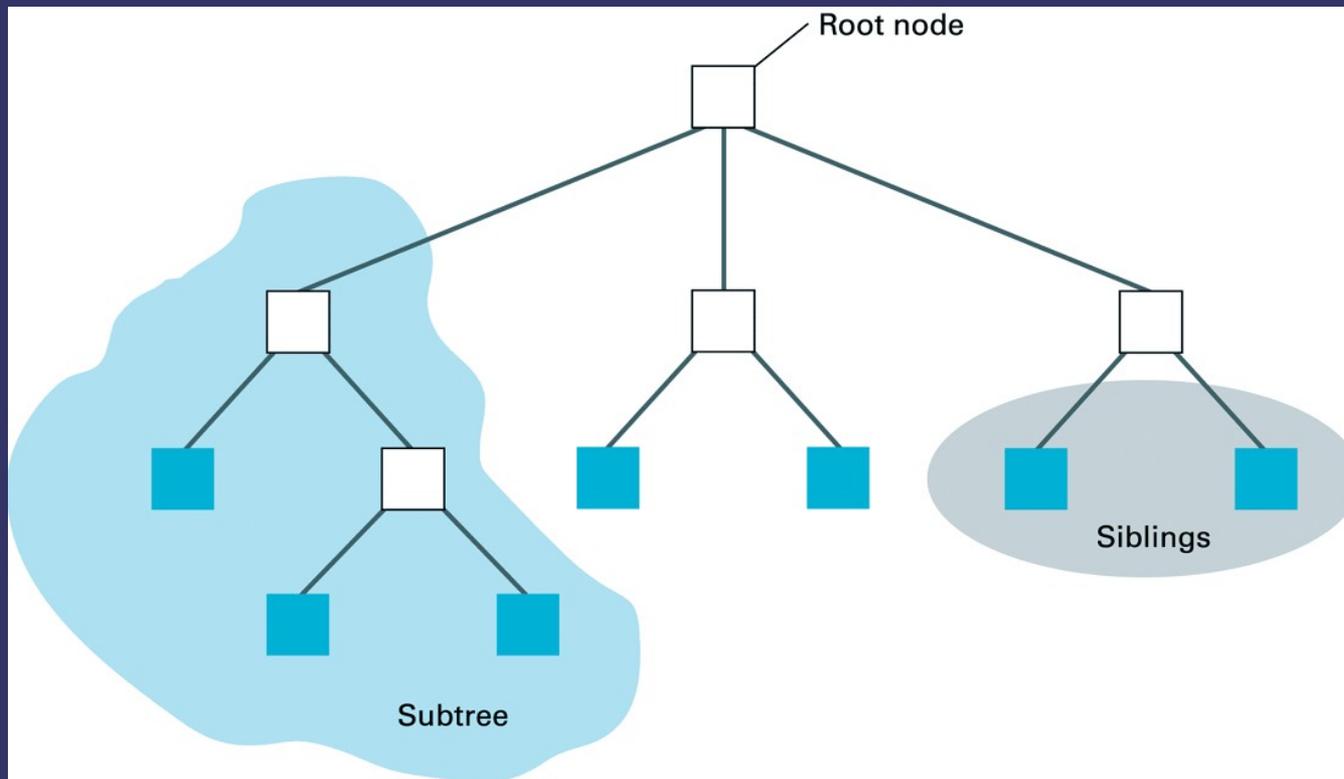
- Gerarchia

- Nessuna persona può avere più di un superiore
- Ogni persona può essere superiore di altre

# Esempio di un organigramma di un'azienda



# Tree terminology



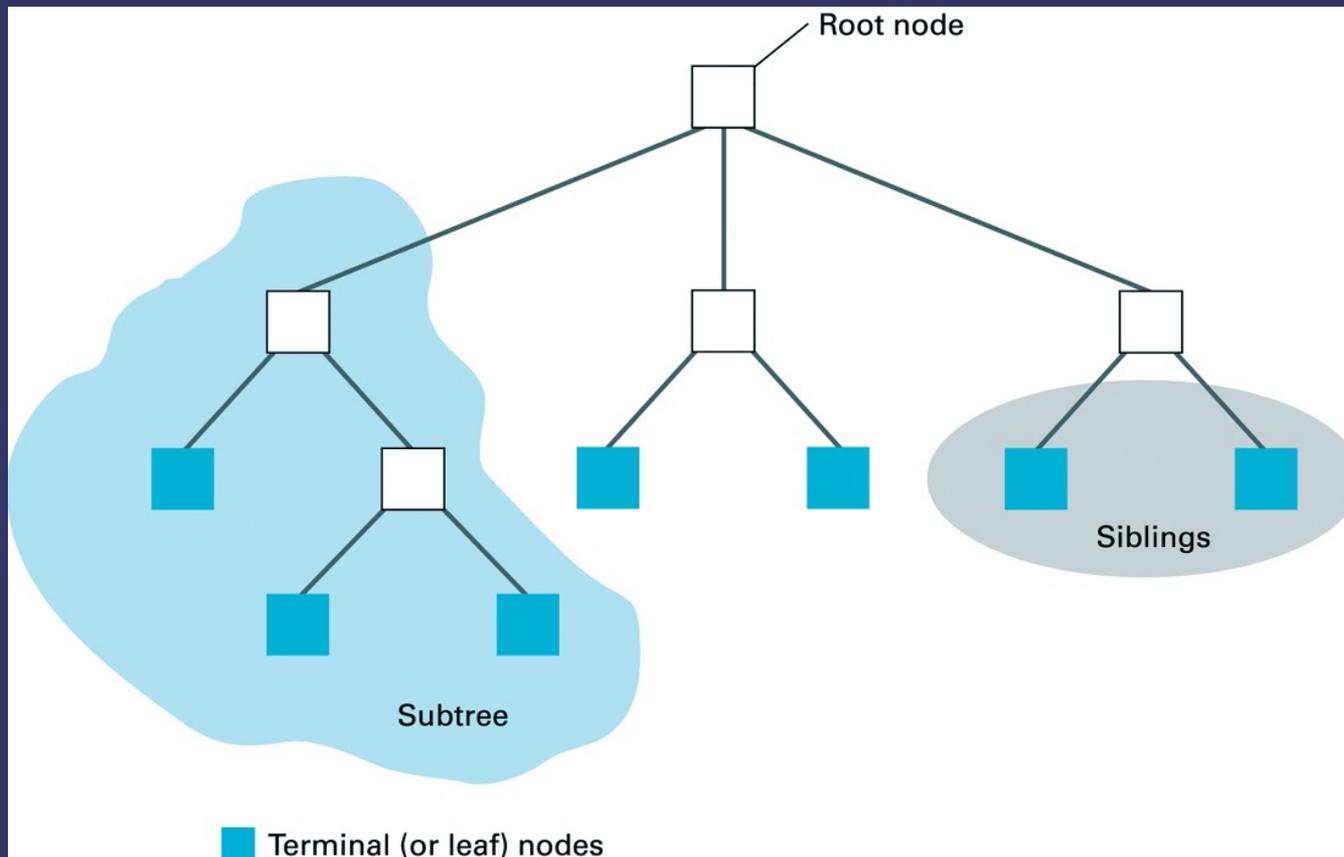
**Nodo (node)** – ogni elemento dell'albero

**Radice (root)** - il nodo che non ha predecessori

**Nodi terminali o foglie (terminal nodes or leaf nodes):** nodi che non hanno successori

**Sottoalbero (subtree)** – un nodo e tutti i nodi che lo seguono

# Tree terminology

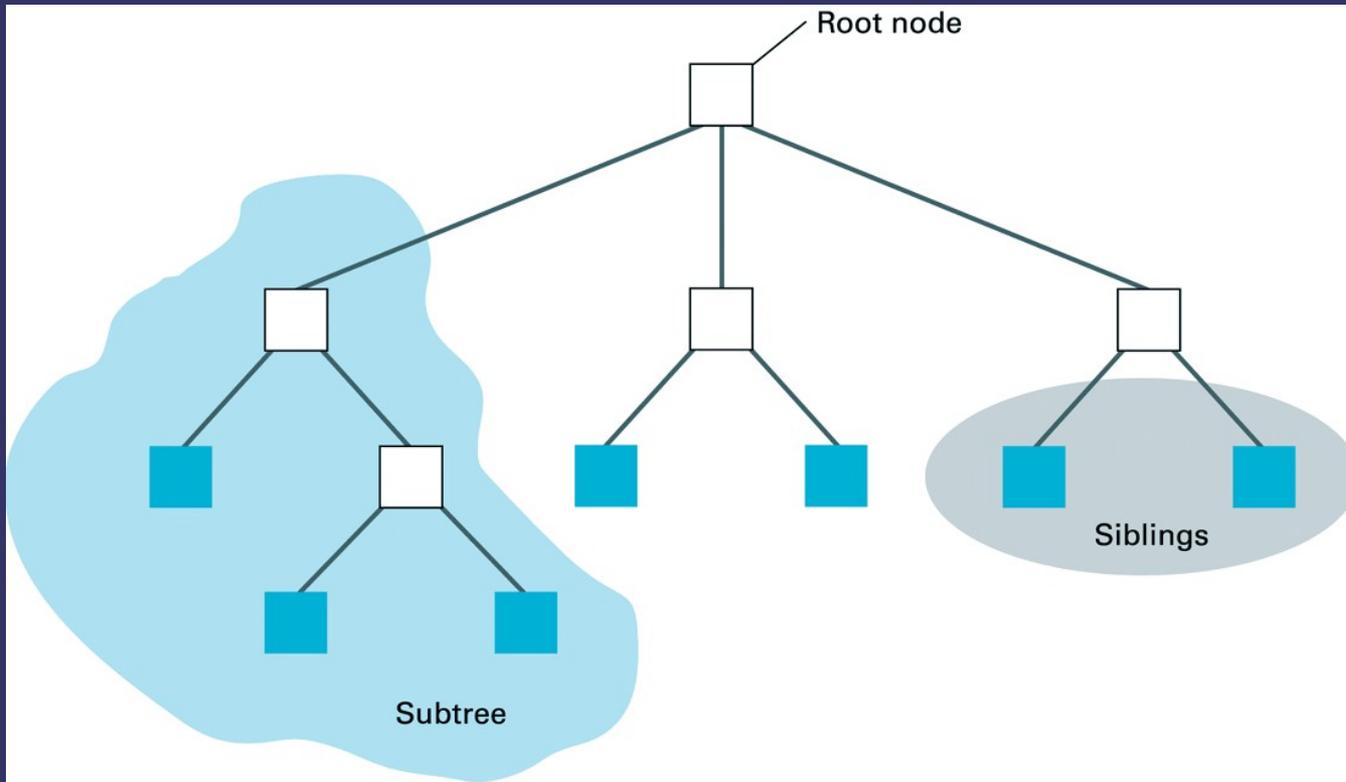


**Nodo padre (parent node):** il nodo immediatamente precedente

**Nodo figlio (child node):** il nodo immediatamente seguente

**Fratelli (siblings)** – nodi che hanno lo stesso padre

# Tree terminology



**Cammino (path) fra due nodi:** – elenco dei nodi che devono essere attraversati per andare da un nodo all'altro

**Profondità di un nodo:** lunghezza del cammino dalla radice al nodo

**Profondità di un albero (depth of tree):** profondità del nodo di profondità massima in un albero

# Alberi

- Alberi binari: ogni nodo ha al più due figli
- Alberi generali: ogni nodo ha un numero  $n$  di figli

# Alberi binari

Una albero è un insieme ordinato, non limitato, di elementi dello stesso tipo, caratterizzato dal fatto che ogni elemento può avere al più due successori o un predecessore.

Il tipo Albero binario è un ADT  $\langle S, F, C \rangle$  dove

$$S = \{\text{albero}, \text{atomo}, \text{boolean}\}$$

albero è il dominio di interesse

atomo è il dominio degli elementi che formano l'albero

# Alberi binari

$F = \{\text{costruisci, radice, null, sin, des}\}$

costruisci : atomo x albero x albero  $\rightarrow$  albero

costruisce un nuovo albero

radice : albero  $\rightarrow$  atomo

ritorna l'informazione contenuta nella radice dell'albero

null : albero  $\rightarrow$  boolean

ritorna il valore vero se l'albero è vuoto

sin : albero  $\rightarrow$  albero

ritorna il sottolabero sinistro

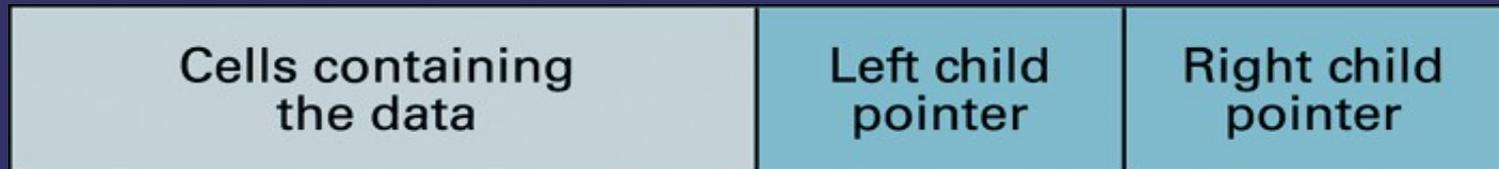
des : albero  $\rightarrow$  albero

ritorna il sottolabero destro

$C = \text{albero è vuoto}$ , è la costante che denota la lista priva di elementi

# Implementazione degli alberi binari utilizzando i puntatori

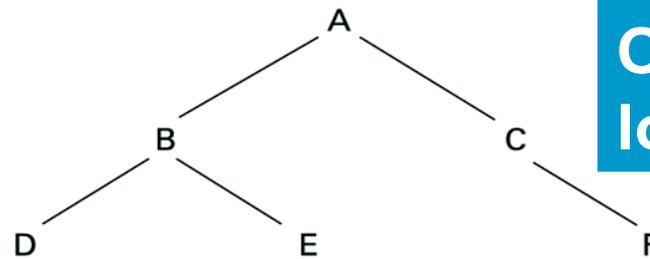
- Ogni nodo ha tre componenti:
  - Dati
  - Figlio sinistro
  - Figlio destro



- Ogni nodo è costituito da celle di memoria contigue
- Uso di struct
- Se non è presente uno dei due figli il puntatore corrispondente è uguale a NULL

# Esempio di un albero

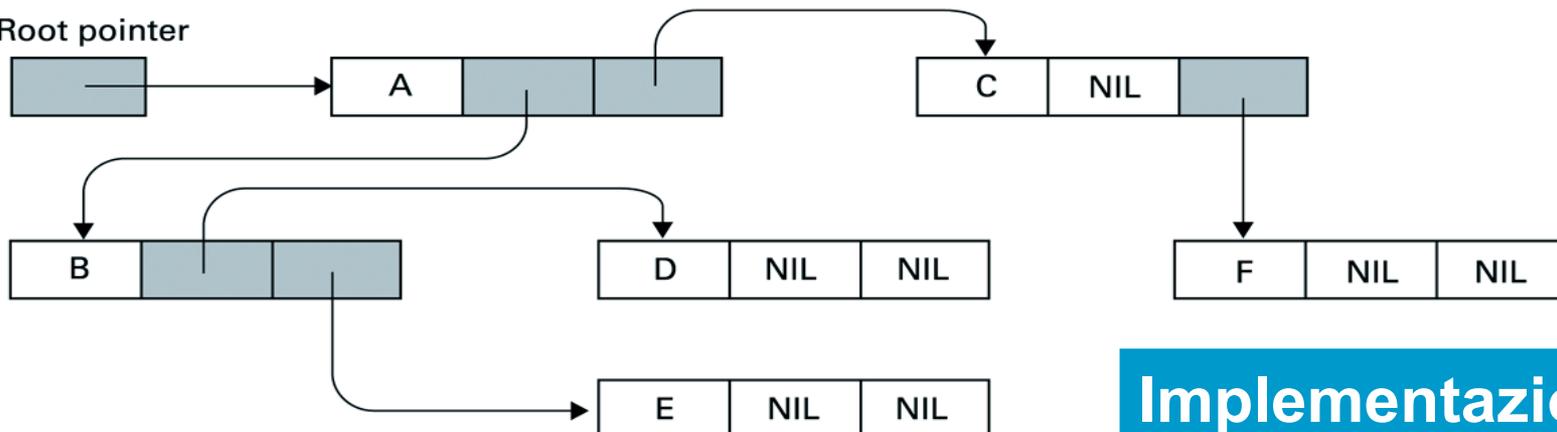
## Conceptual tree



Organizzazione logica

## Actual storage organization

Root pointer



Implementazione

# Implementazione di alberi binari mediante puntatori

```
typedef int Tatomico;  
  
typedef struct elem{  
    Tatomico info;  
    struct elem *sx,*dx;  
}nodo, *albero;
```

# Implementazione di alberi binari

```
int costruisci(albero S, albero D, T atomo root, albero *PA) {  
    *PA = (albero)malloc(sizeof(nodo));  
    if (*PA != NULL){  
        (*PA) ->info = root;  
        (*PA) ->sx = S;  
        (*PA) ->dx = D;  
        return 1;  
    }  
    return 0;  
}
```

# Implementazione di alberi binari

```
Tatomo radice(albero A) {return A->info;}
```

```
int null(albero A) {return (A==NULL);}
```

```
albero sin(albero A) {return A -> sx;}
```

```
albero des(albero A) {return A -> dx;}
```

# VISITA DEGLI ALBERI BINARI

Visita anticipata (implementazione ricorsiva)

```
void visita_anticipata(albero A)
{
    if (!null(A)) {
        stampa(radice(A));
        visita_anticipata(sin(A));
        visita_anticipata(des(A));
    }
}
```

# VISITA DEGLI ALBERI BINARI

Visita anticipata: implementazione iterativa, utilizza una pila in cui il tipo TAtomo è il tipo albero

```
void visita_anticipata2 (albero A)
{
    pila P;
    push(P, A)
    while (!null_pila(P))
        {
            stampa(top(P));
            A = pop(P);
            if (!null(des(A)) push(P, des(A));
            if (!null(sin(A)) push(P, sin(A));
        }
}
```

# VISITA DEGLI ALBERI BINARI

Visita simmetrica (implementazione ricorsiva)

```
void visita_simmetrica(albero A)
{
    if (!null(A)) {
        visita_simmetrica(sin(A));
        stampa(radice(A));
        visita_simmetrica(des(A));
    }
}
```

# VISITA DEGLI ALBERI BINARI

Visita in postordine (implementazione ricorsiva)

```
void visita_post(albero A)
{
    if (!null(A)) {
        visita_post(sin(A));
        visita_post(des(A));
        stampa(radice(A));
    }
}
```

# RICERCA ALBERI BINARI

/\* la funzione cerca il dato nell'albero A e ritorna il puntatore al nodo contenente l'informazione cercata\*/

```
albero ricerca (albero A, Tatomico dato)
{
    albero aux;
    if (null(A)) return NULL;
    if (radice(A) == dato) return A;
    aux = ricerca(sin(A),dato)
    if (!aux) return ricerca(dest(A),dato));
    return aux;
}
```

# CANCELLA ALBERI BINARI

```
void cancella (albero *A){  
  
    if (!null(*A)) {  
        cancella(&sin(*A));  
        cancella(&dest(*A));  
        free (*A)  
    }  
}
```

# RICERCA: ALBERI BINARI DI RICERCA

- Un albero binario è detto di ricerca se:
  - l'informazione della radice è maggiore di tutte le informazioni del sottoalbero sinistro
  - l'informazione della radice è minore di tutte le informazioni del sottoalbero destro
  - Il sottoalbero sinistro è un albero binario di ricerca
  - Il sottoalbero destro è un albero binario di ricerca

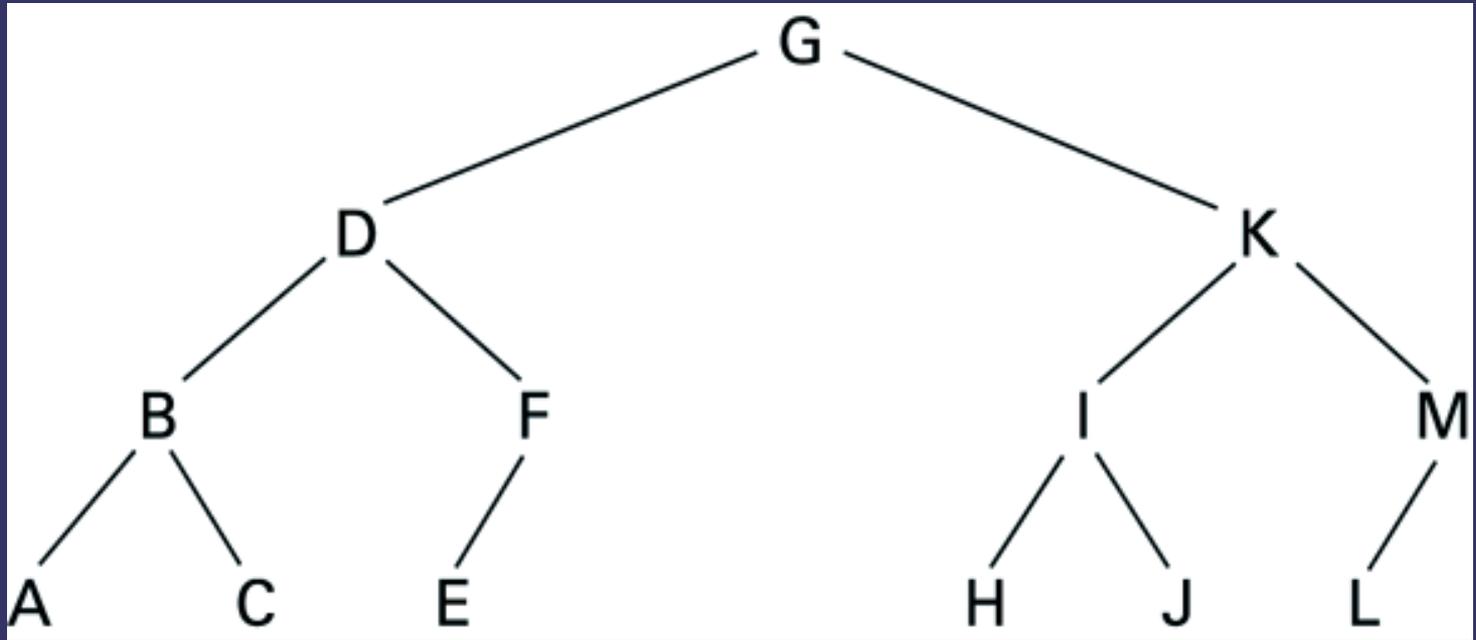
# Esempio

depth 0 →

depth 1 →

depth 2 →

depth 3 →



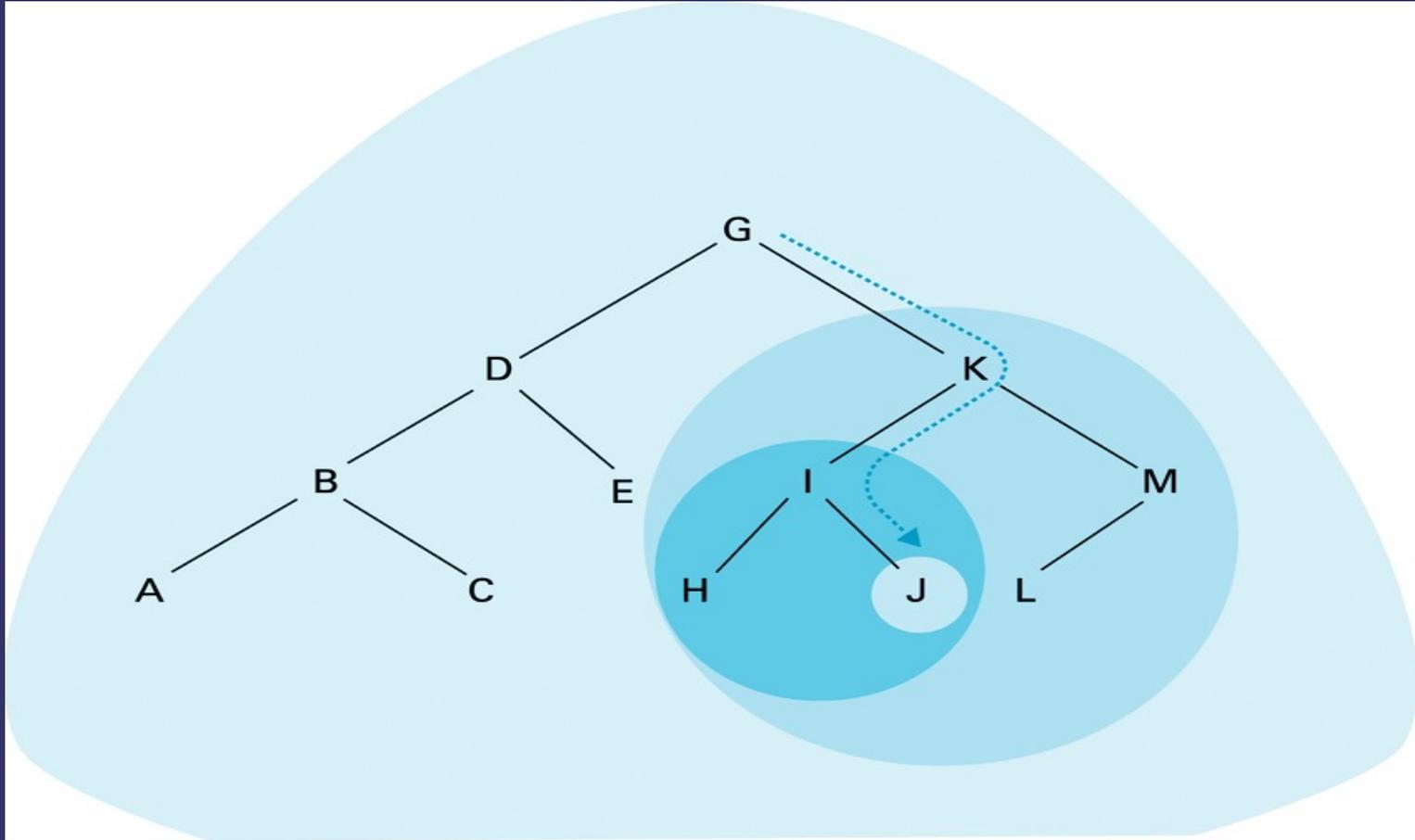
A, **B**, C, D, E, **F**, **G**, H, **I**, J, K, L, **M**  
level: 3 **2** 3 1 3 **2** **0** 3 **2** 3 1 3 **2**

# Ricerca in un albero binario di ricerca - versione ricorsiva -

```
/* la funzione cerca il dato nell'albero A e ritorna il puntatore  
al nodo contenente l'informazione cercata, NULL se  
l'elemento non è presente*/
```

```
albero ricerca (albero A, Tatomico dato) {  
    Tatomico rad;  
    if (null(A)) return A;  
    rad = radice(A);  
    if (rad == dato) return A;  
    if (dato < rad) return ricerca(sin(A),dato);  
    return ricerca(dest(A),dato);  
}
```

# Esempio

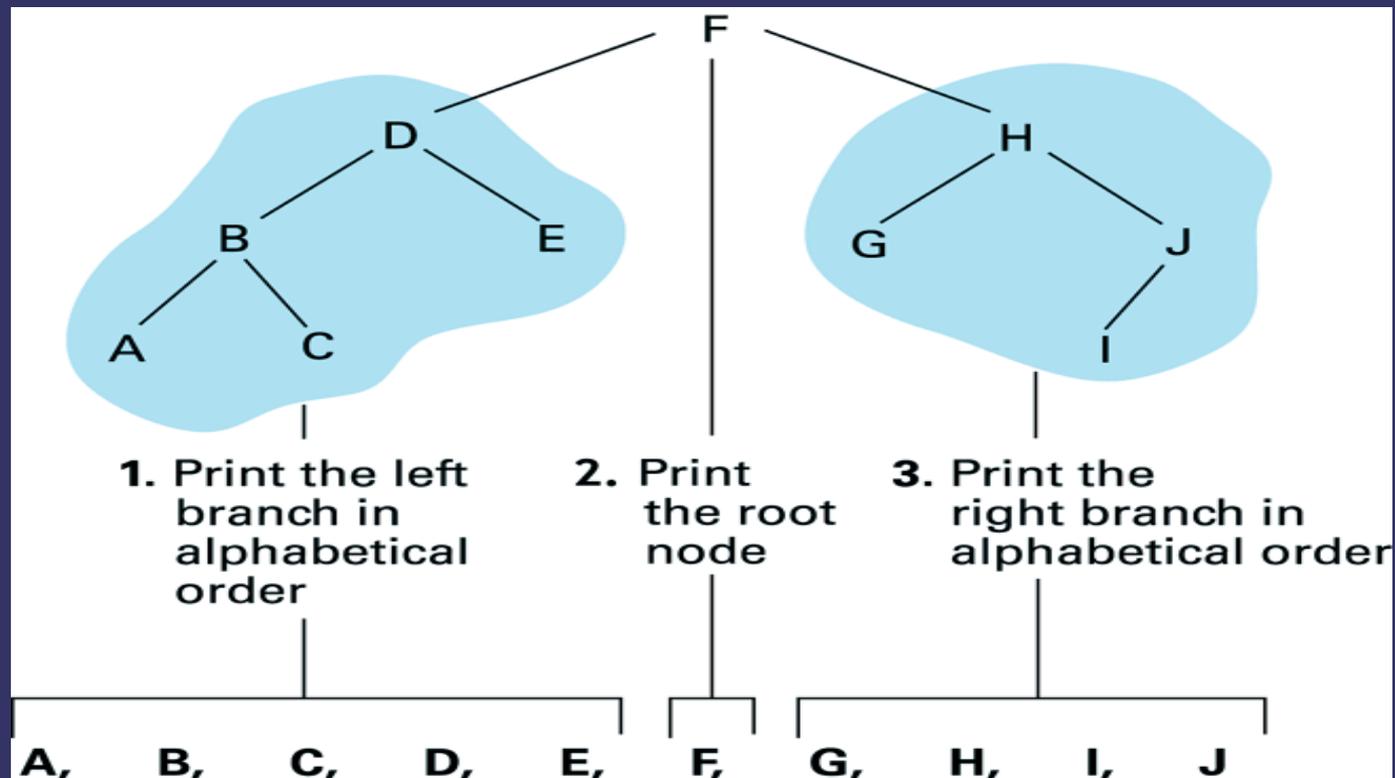


# Ricerca in un albero binario di ricerca - versione iterativa -

```
albero ricerca (albero A, Tatomico dato) {  
    Tatomico rad;  
    while (!null(A)) {  
        rad = radice(A);  
        if (rad == dato) return A;  
        if (dato < rad) A = sin(A);  
        else A = dest(A);  
    }  
    return A;  
}
```

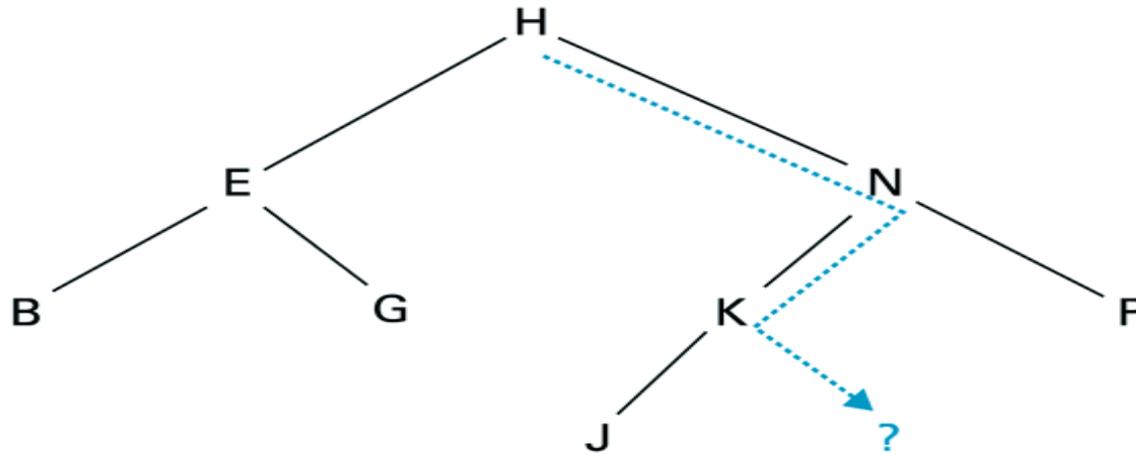
# Visita

La visita in ordine simmetrico di un albero binario di ricerca produce un sequenza ordinata rispetto alla chiave.

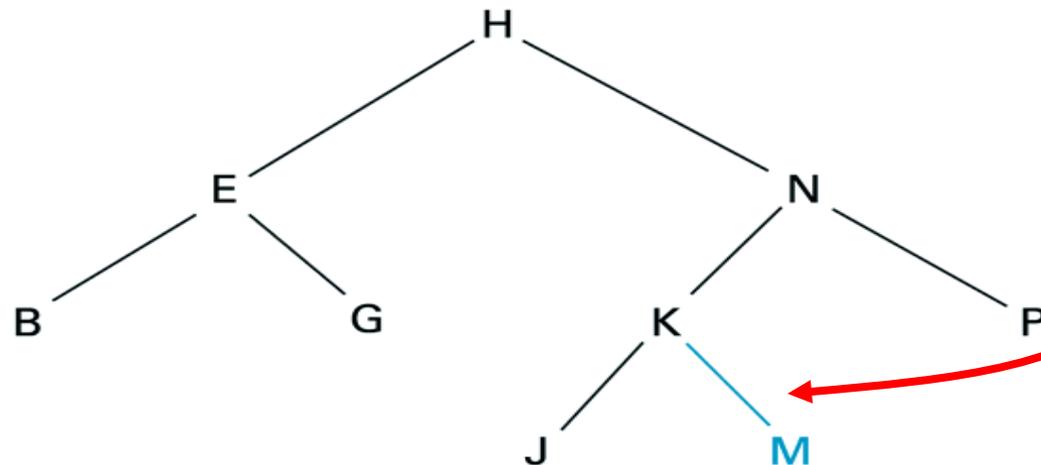


# INSERIMENTO IN ALBERI BINARI

a. Search for the new entry until its absence is detected



b. This is the position in which the new entry should be attached



# INSERIMENTO IN ALBERI BINARI

## - versione ricorsiva -

/\* la funzione inserisce il dato nell'albero A e ritorna 1 se l'elemento è stato inserito con successo \*/

```
int insord (albero *A, Tatomico dato){
    Tatomico rad;
    if (null(*A)) return(costruisci(NULL,NULL,dato,A));
    rad =radice(*A);
    if (rad == dato) return 0;
    if (rad > dato) return insord(&((*A)->sx),dato)
    return insord(&((*A)->dx),dato);
}
```

# INSERIMENTO IN ALBERI BINARI

## - versione iterativa -

```
/* stessa insord ma implementata in modo iterativo*/
int insord_it (albero *A, Tatomico dato){
    albero P, Prec;
    int flag;

    if (null(*A)) return (costruisci(NULL,NULL,dato,A));
    P = Prec = *A;
    while (!null(P)){
        Prec = P;
        if (radice(P) == dato) return 0
        else if (radice(P) > dato) P = sin (P);
        else P = dest(P);
    }
}
```

# CANCELLAZIONE INTERO ALBERO

```
void cancella (albero *A){  
    if (!null(*A)){  
        cancella(&((*A)->sx));  
        cancella(&((*A)->dx));  
        free(*A);  
    }  
}
```