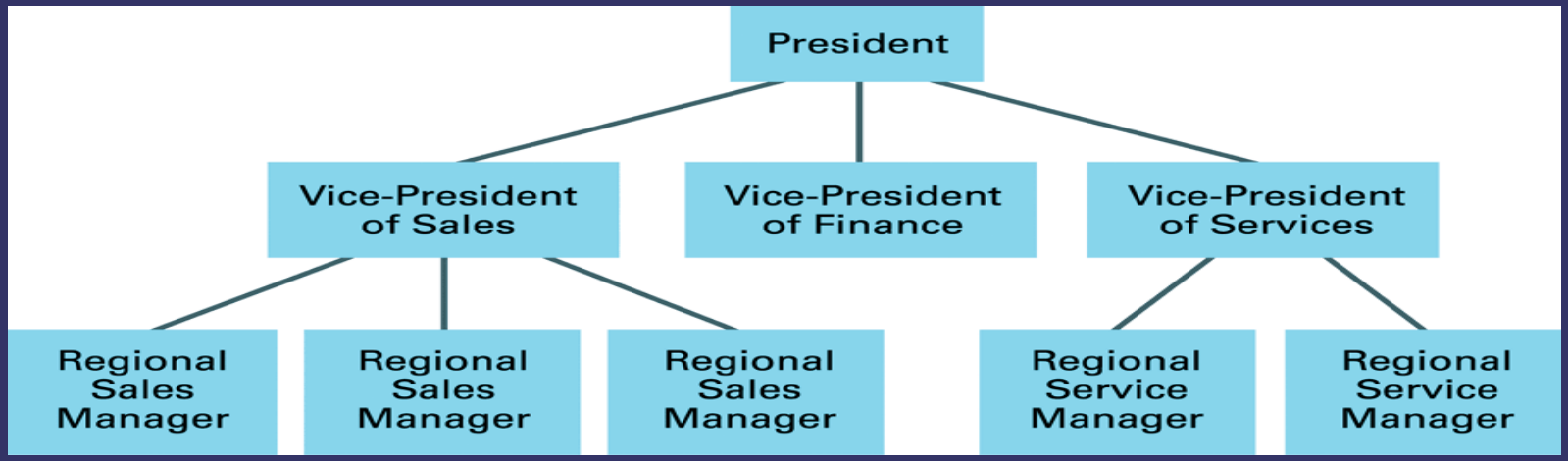
Alberi

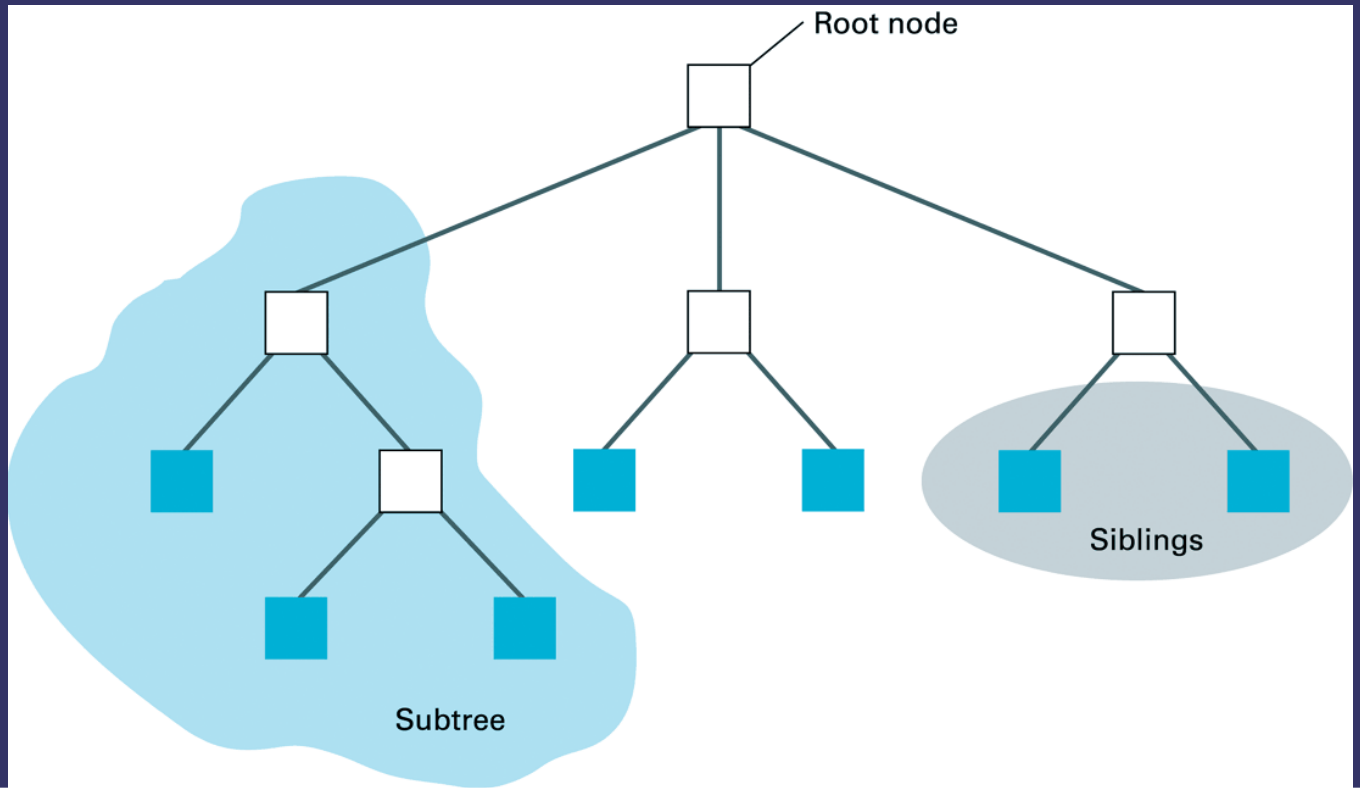
Struttura dati Albero (Tree)

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

Esempio di un organigramma di un'azienda

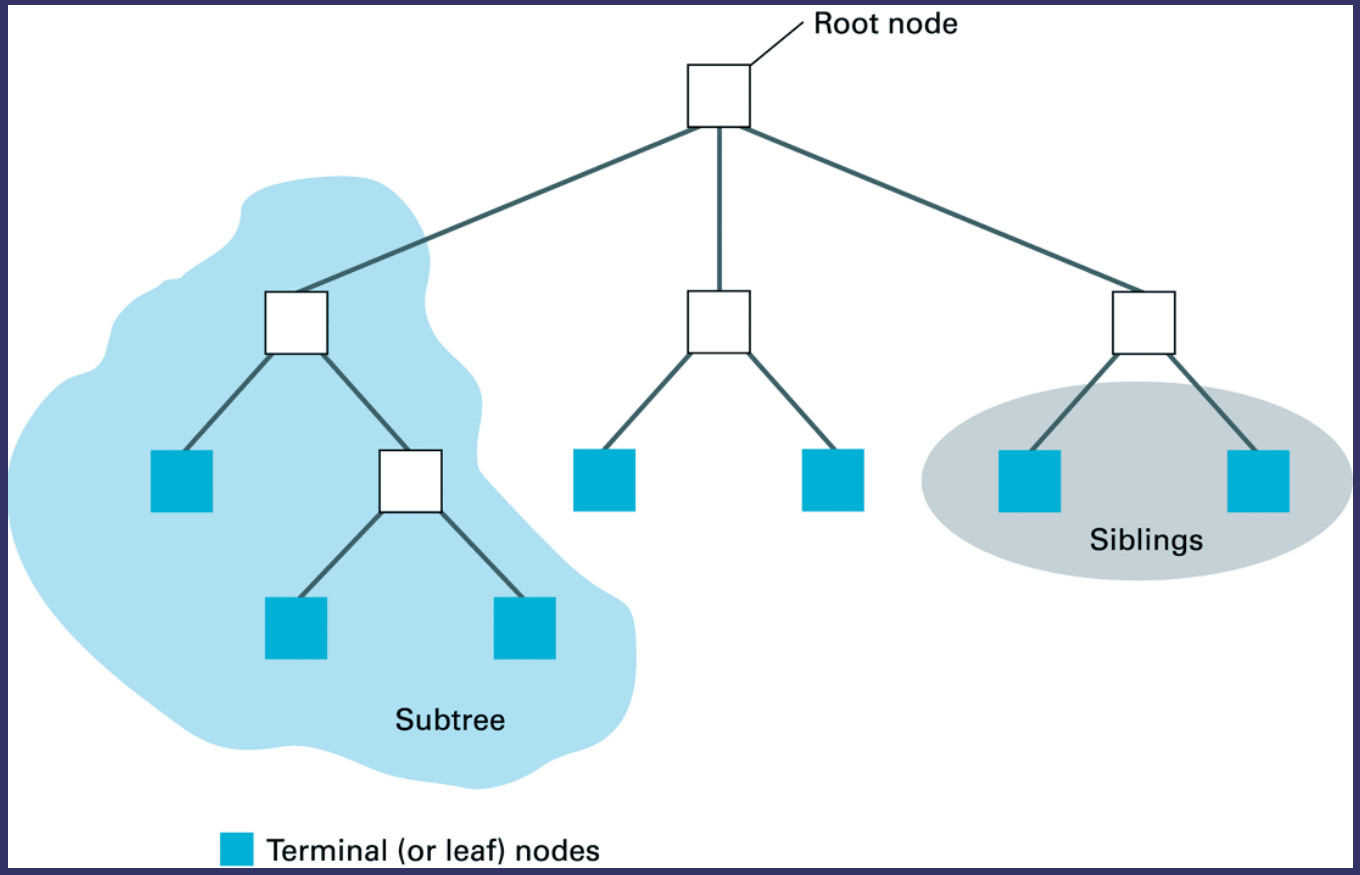


Tree terminology



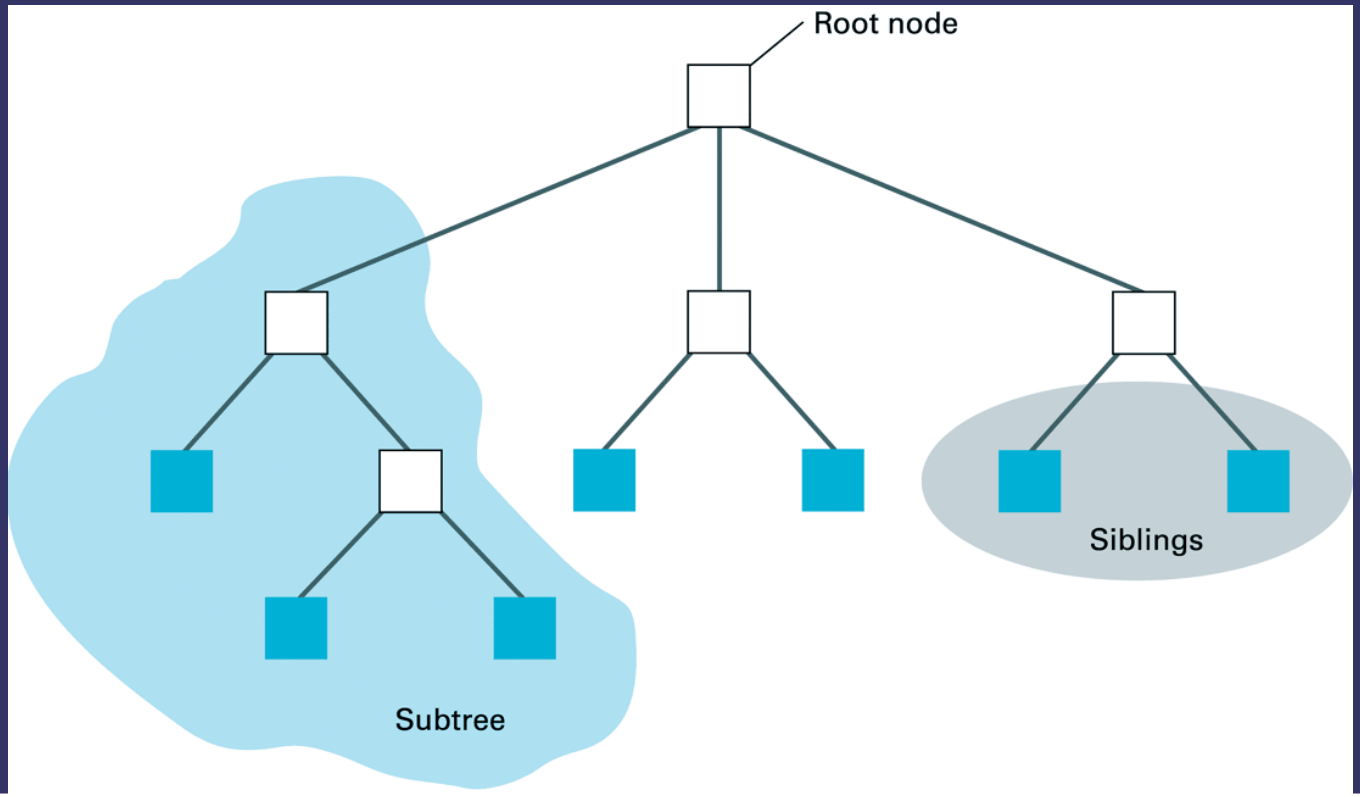
Nodo (node) – ogni elemento dell'albero
Radice (root) - il nodo che non ha predecessori
Nodi terminali o foglie (terminal nodes or leaf nodes): nodi che non hanno sucessori
Sottoalbero (subtree) – un nodo e tutti I nodi che lo seguono

Tree terminology



Nodo padre (parent node): il nodo immediatamente precedente
Nodo figlo (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 Abero binario è un ADT <S,F,C dove

S = {albero, atomo, boolean}

albero è il dominio di interesse

atomo è il dominio degli elementi che formano l'albero

Alberi binari

F = {costruisci, radice, null, sin, des}

costruisci : atomo x albero x albero →albero

costruisce un nuovo albero

radice : albero → atomo

ritorna l'informazione contenuta nella radice dell'albero

null : albero → boolean

ritorna il valore vero se l'albero è vuoto

sin : albero →albero

ritorna il sottolabero sinistro

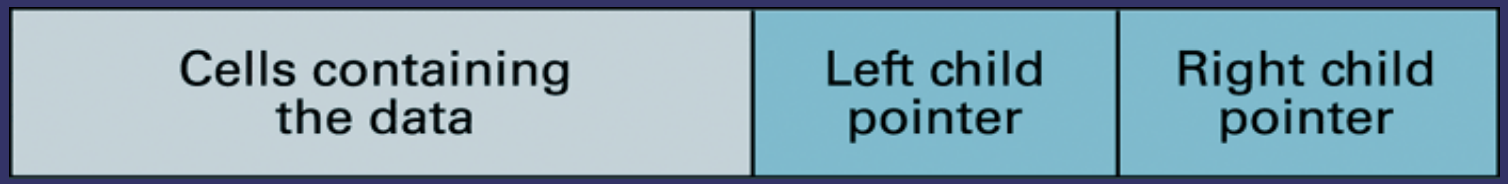
des : albero →albero

ritorna il sottolabero destro

C = albero è vuoto, è la costante che denota la lista priva di elementi

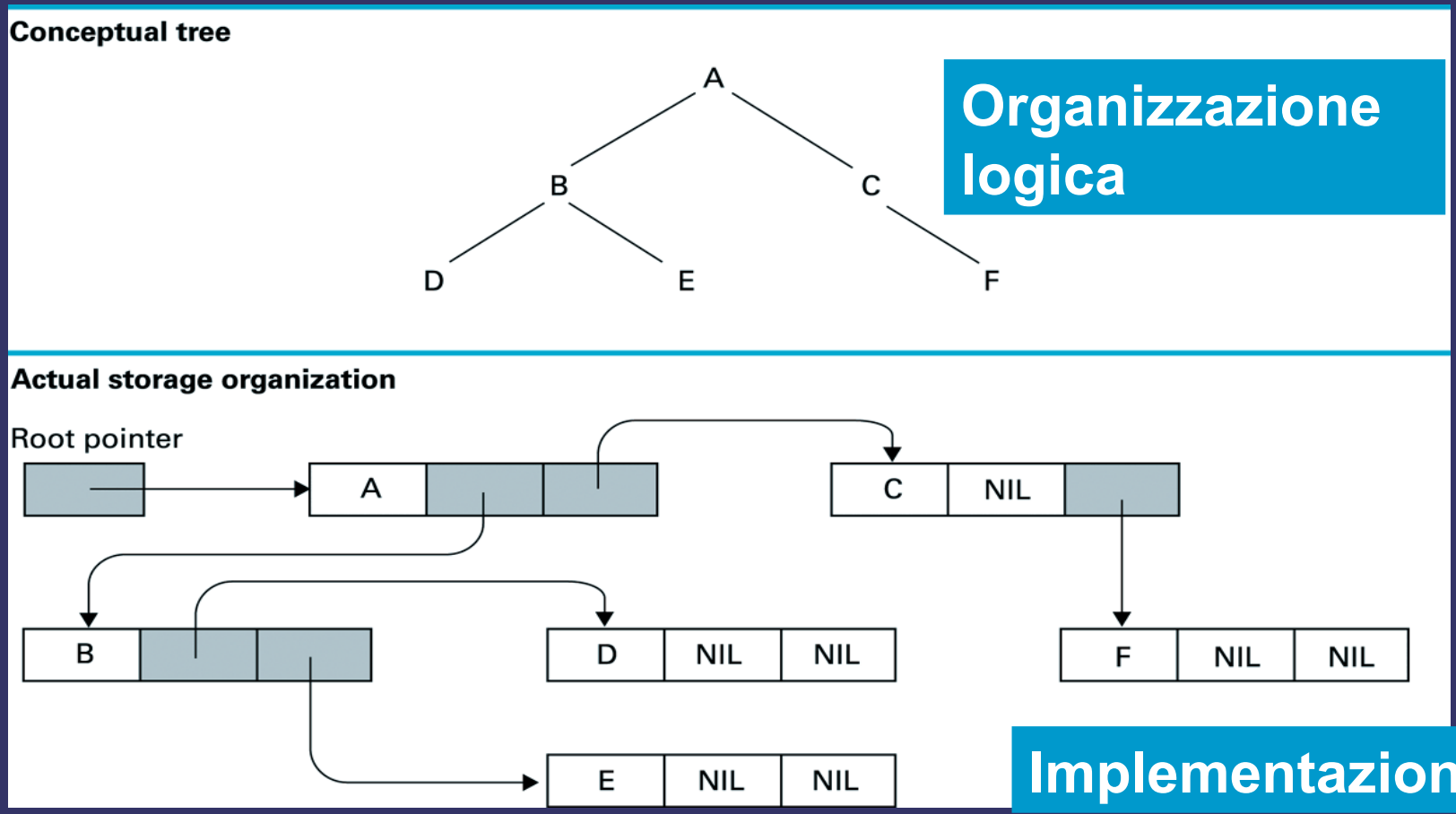
Implementazione degli alberi binari utilizzando I puntatori

- 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



Implementazione dell'albero senza i puntatori utilizzando blocchi di memoria contigui

La radice è memorizzara nella cella #1

I figli sono memorizzati nella celle #2 e #3

I figli del figlio sono memorizzati nelle celle 4, 5, 6, & 7

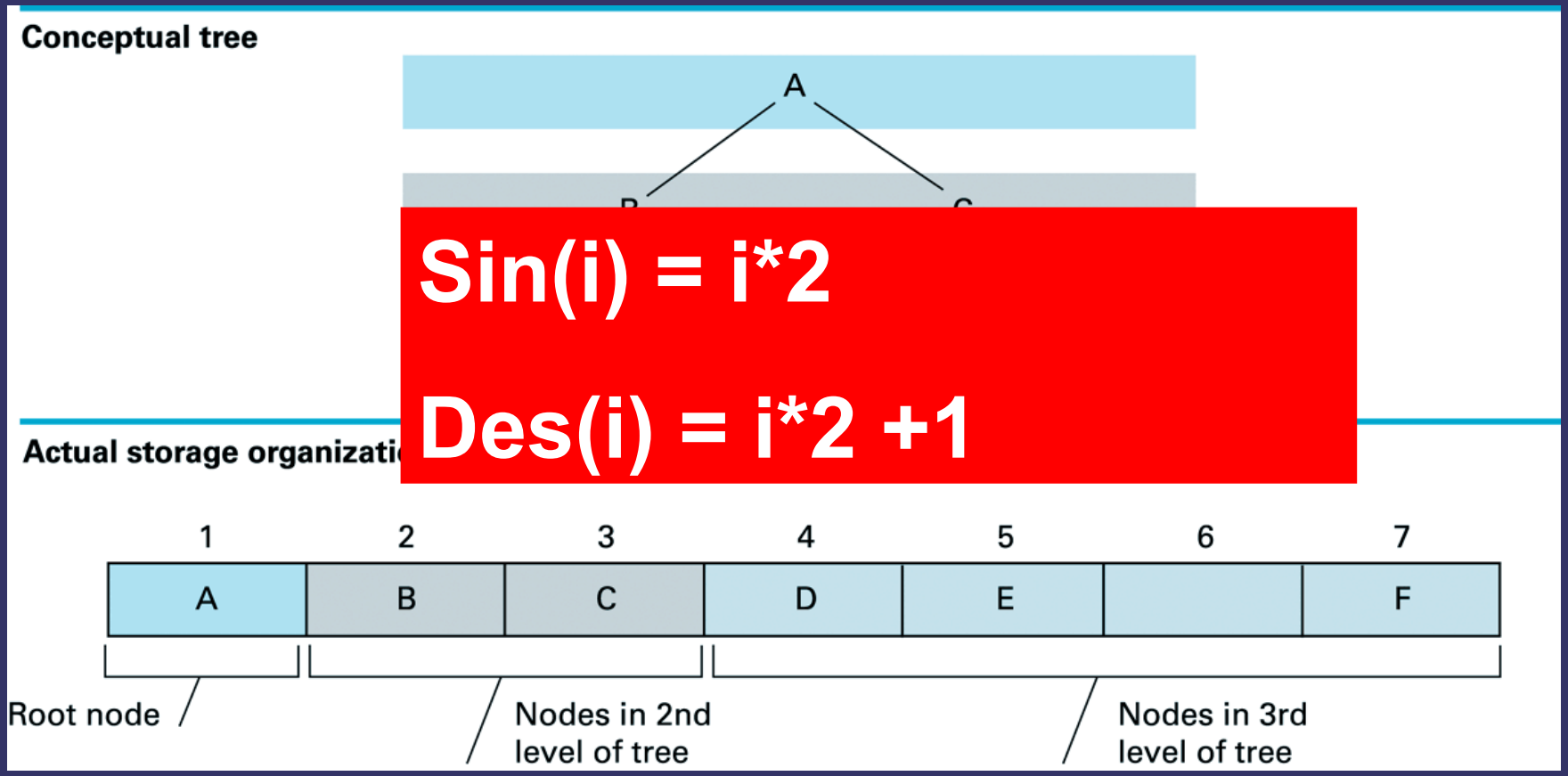
A livellol 0 → 1 cella

A livello 1 → 2 celle

A livello 2 → 4 celle

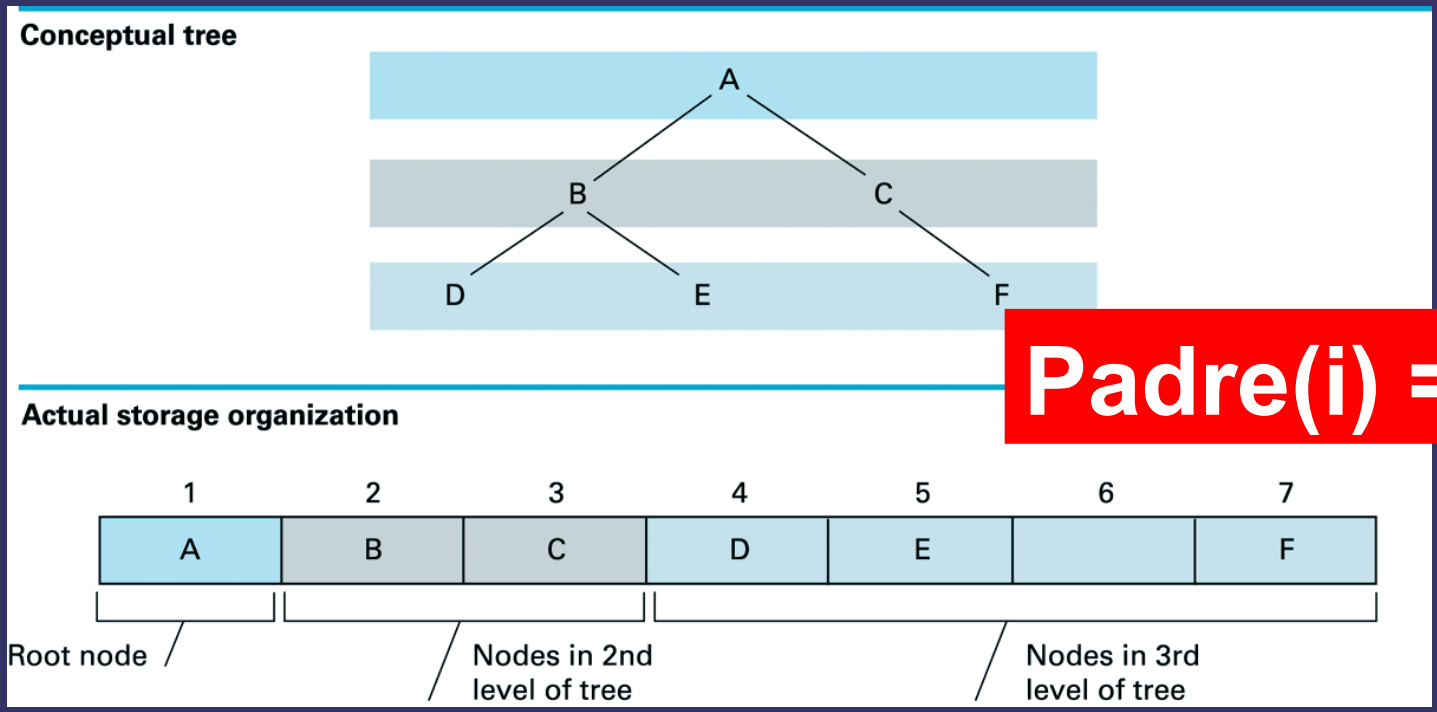
A livello 3 → 8 celle

Ricerca del figlio



È semplice trovare il figlio
Figlio sinistro di B: 2*2 → D
Figlio destro di B: 2*2 +1 → E

Ricerca del padre



Padre(i) = i/2

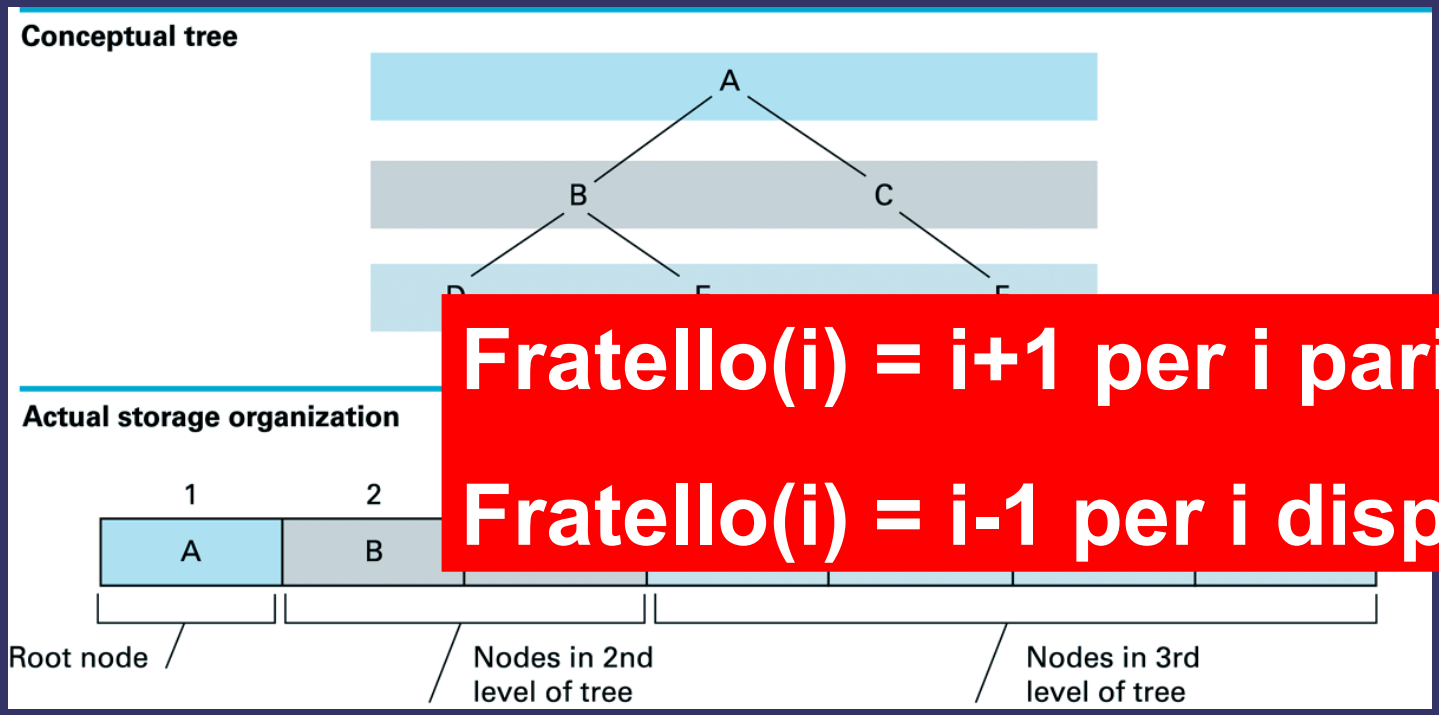
È semplice trovare il padre

Padre di D: 4/2 = 2 → B

Padre di E: 5/2 = 2 → B

Padre di F: 7/2 = 3 → C

Ricerca dei fratelli



Fratello(i) = i+1 per i pari

Fratello(i) = i-1 per i dispari

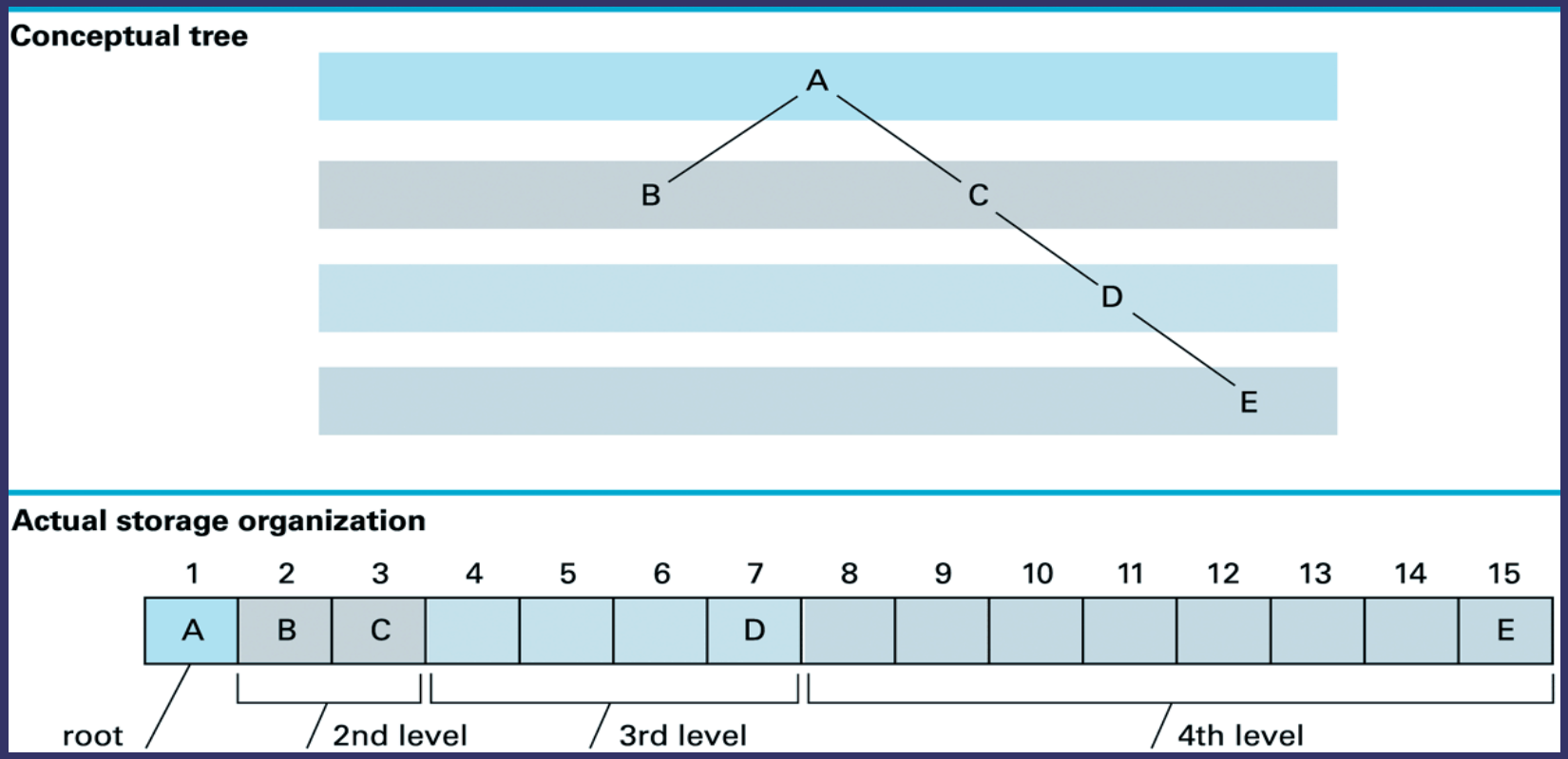
Fratello di D: 4 + 1 = 5 → E

Fratello di E : 5 – 1 = 4 → D

Quando è conveniente una implementazione con memoria contigua e quando con i puntatori?

- Dipende dalla struttura dell'albero
- **Albero bilanciato**: per ogni nodo dell'albero i suoi sottoalberi hanno lo stesso numero di discendenti
- Per un albero bilanciato tutta le celle di memoria contigue sono piene

Esempio


Conceptual tree
A
B
C
D
E
Actual storage organization
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
A B C D E
root
2nd level
3rd level
4th level

Implementazione di alberi binari mediante puntatori

```
typedef  int Tatomo;

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

Implementazione di alberi binari

```
int costruisci(albero S, albero D,Tatomo 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, Tatomo 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;
}
```

ESEMPIO RICERCA

Esempio di ricerca:

```
typedef struct T{
        int key;
      char nome[31];
}Tatomo

albero ricerca(albero A, int chiave)
{
        albero aux;
        if (null(A) || radice(A).key == chiave)  return A
        aux = ricerca (sin(A), chiave);
        if (!aux) return ricerca(dest(A), chiave);
        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 in formazioni 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

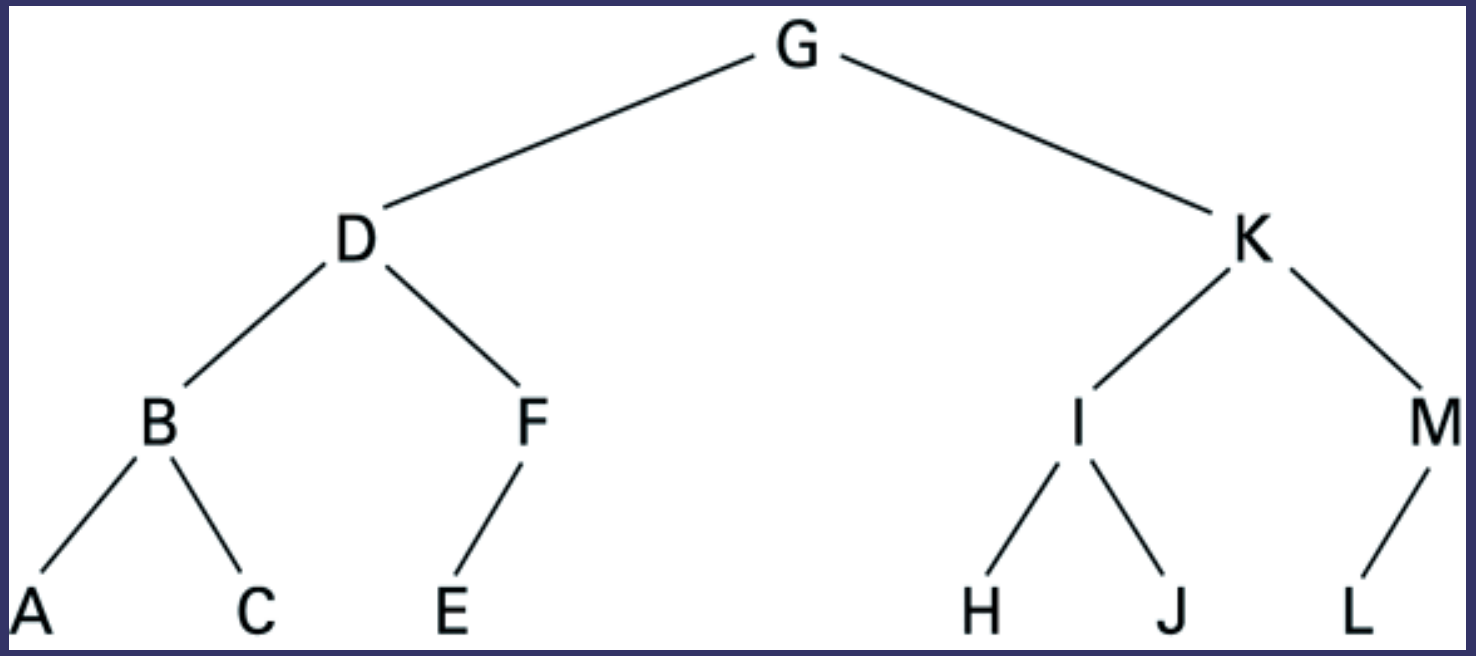
Esempio

depth 0 →

depth 1 →

depth 2 →

depth 3 →



A, B, C, D, E, F, G, H, I, J, K, L, M

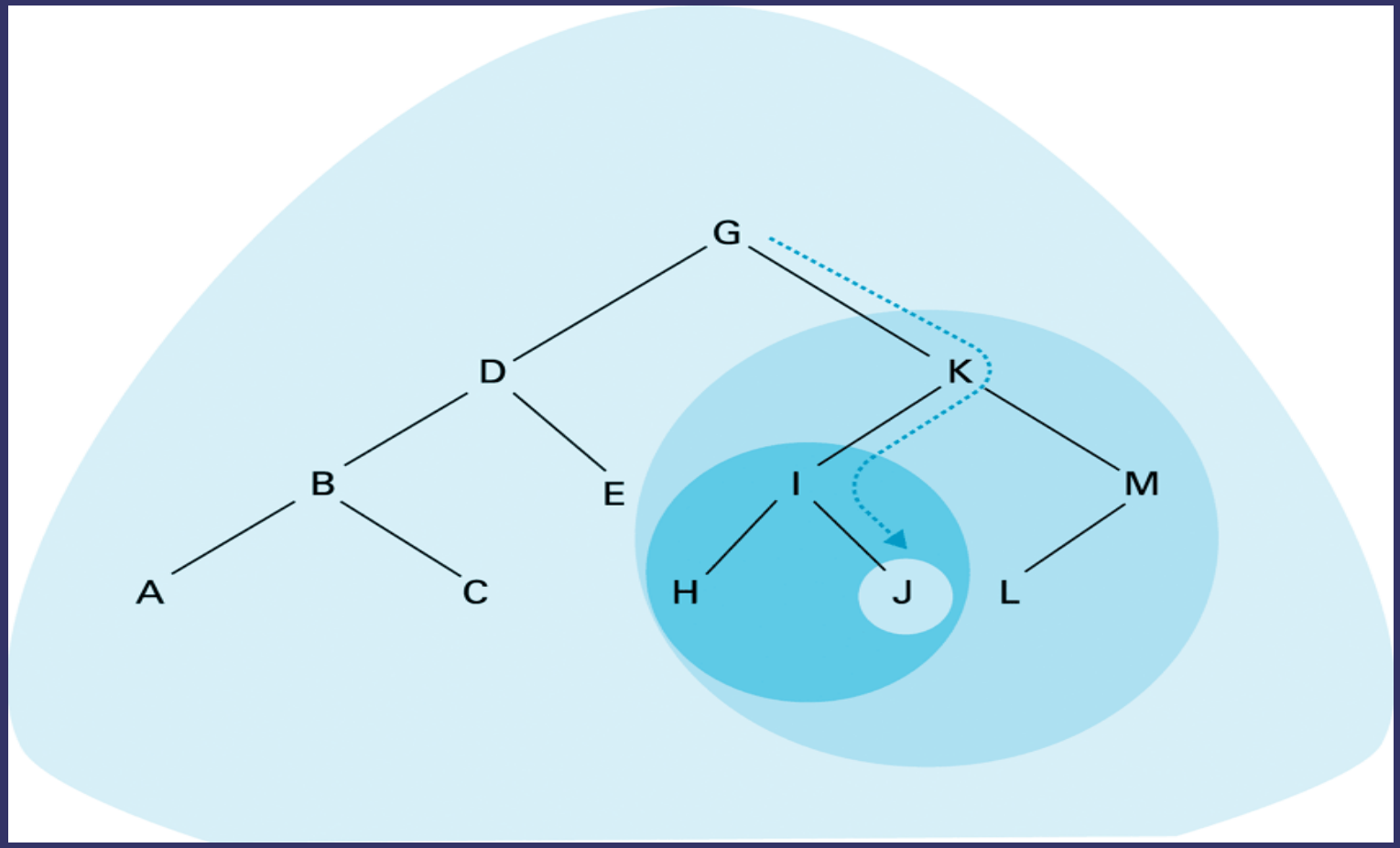
level: 3 2 3 1 3 2 0 3 2 3 1 3 2

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

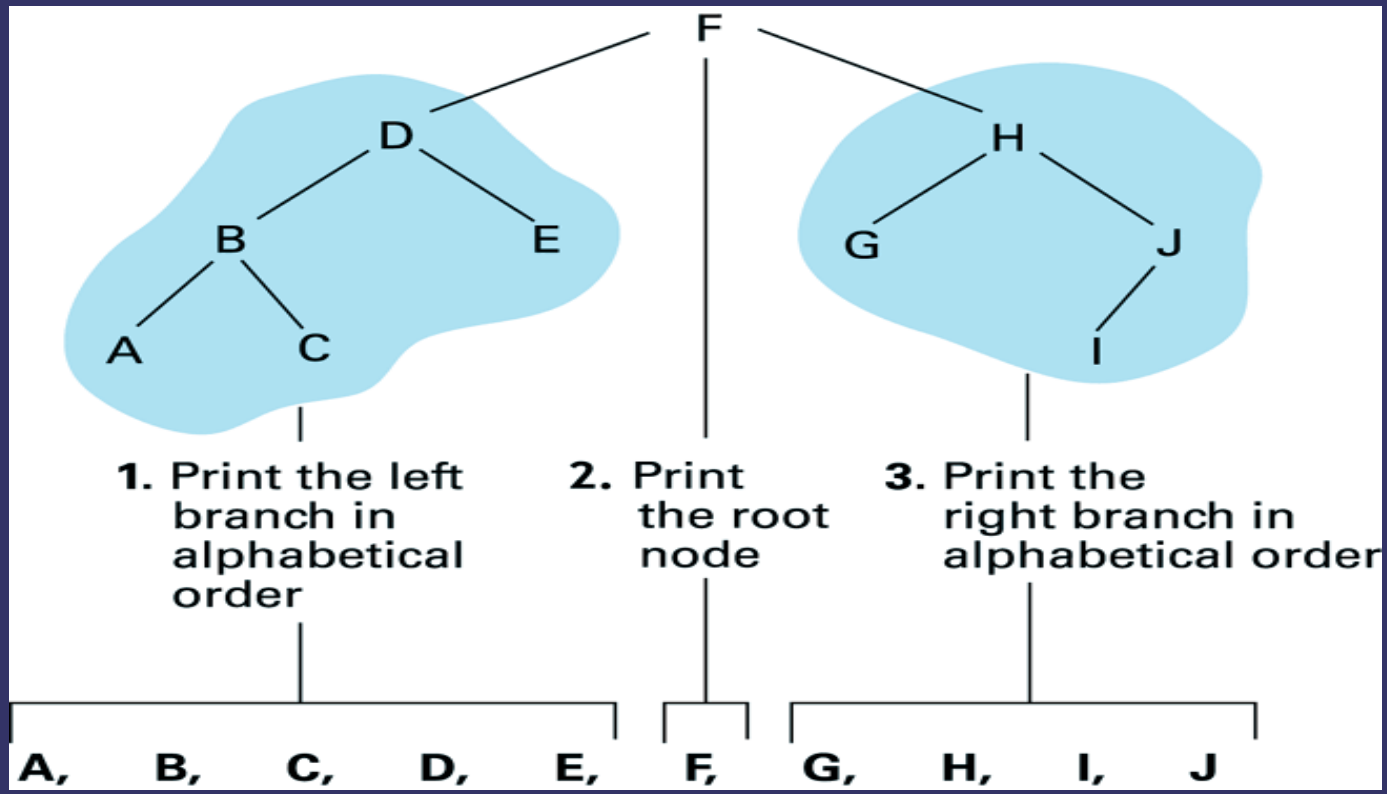
Esempio di ricerca

```
typedef struct T{
     int key;
char nome[31];
}Tatomo

albero ricerca(albero A, int chiave){
if (null(A) || radice(A).key == chiave)  return A;
if (radice(A).key > chiave)
     return ricerca (sin(A), chiave);
return  ricerca(dest(A), chiave);
}
```

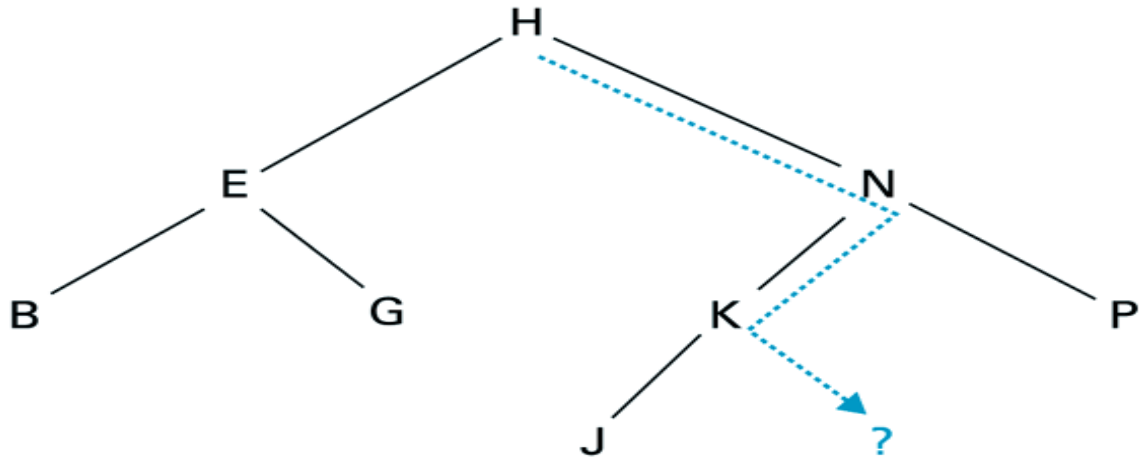
Visita

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

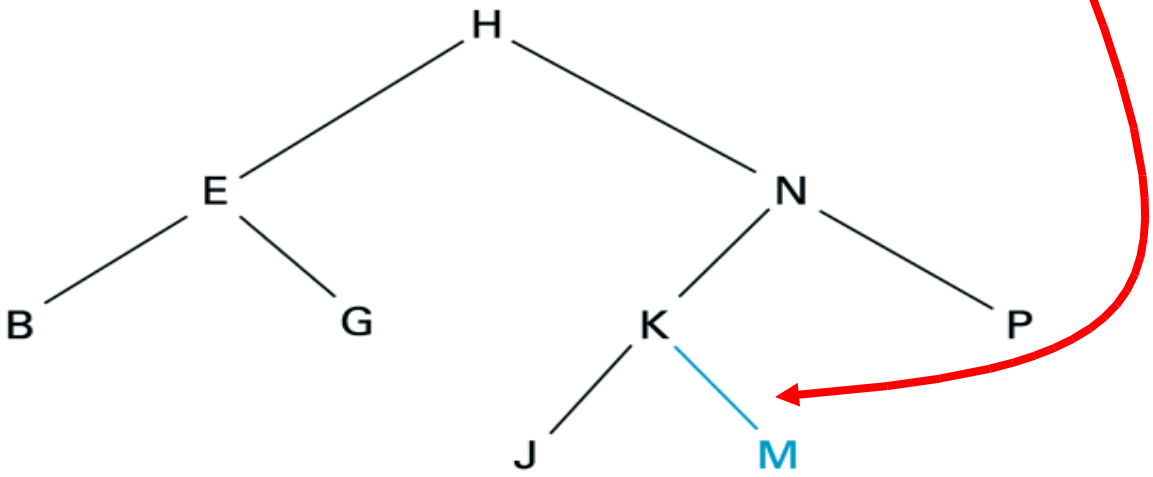


INSERIMENTO IN ALBERI BINARI

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



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



INSERIMENTO IN ALBERI BINARI
- versione ricorsiva -

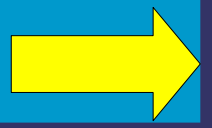
```
/* la funzione inserisce il dato nell'albero A e ritorna true
se l'elemento è stato inserito con successo */

int insord (albero *A, Tatomo dato)
{   Tatomo rad;
if (null(*A)) return(costruisci(NULL,NULL,dato,A));
rad =radice(*A);
if (rad == dato) return 0;
if (rad > dato) return insord(&(sin(*A),dato)
return insord(&dest(*A),dato);
}
```

INSERIMENTO IN ALBERI BINARI - versione iterativa -

```
/* stessa insord ma implementata in modo iterativo*/
int insord_it (albero *A, Tatomo 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);
      }
```



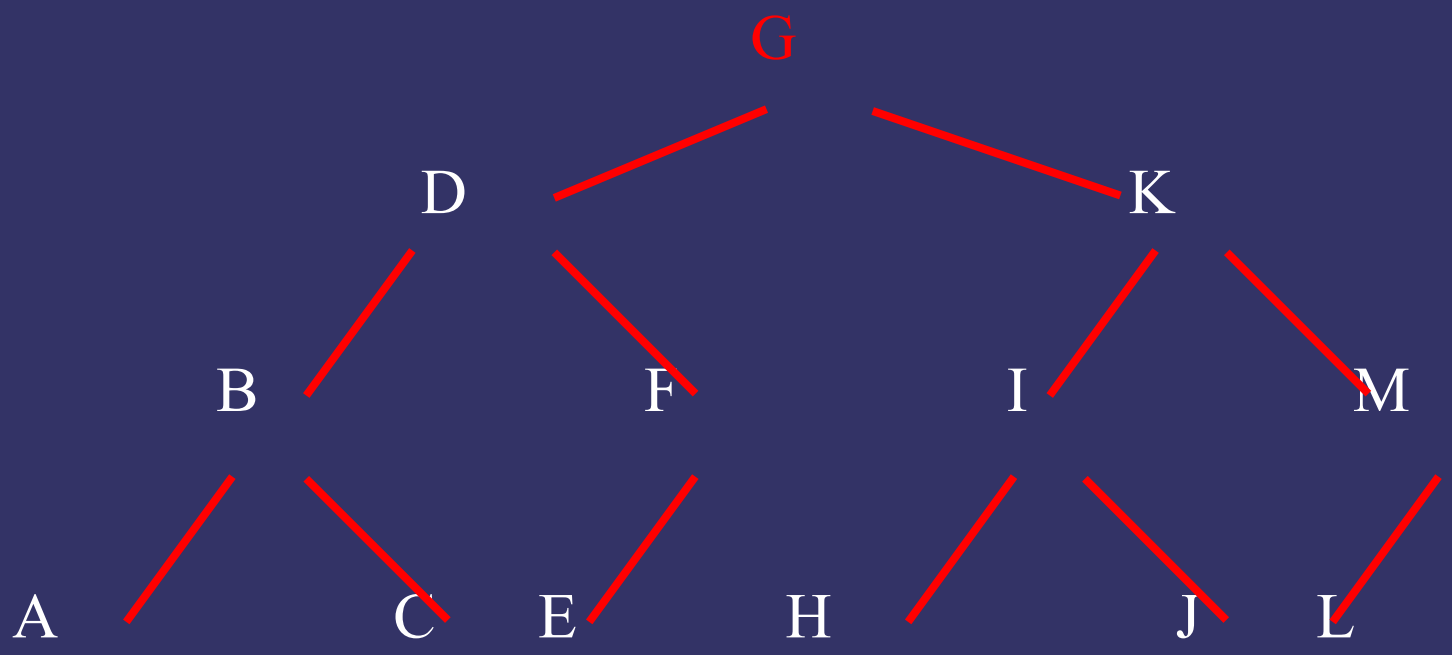
INSERIMENTO IN ALBERI BINARI
- versione iterativa -

```
        flag = costruisci(NULL, NULL, dato, &P);
        if (!flag) return 0;
        if (radice(Prec) > dato)  Prec -> sx = P;
        else Prec -> dx = P;
        return 1;
}
```

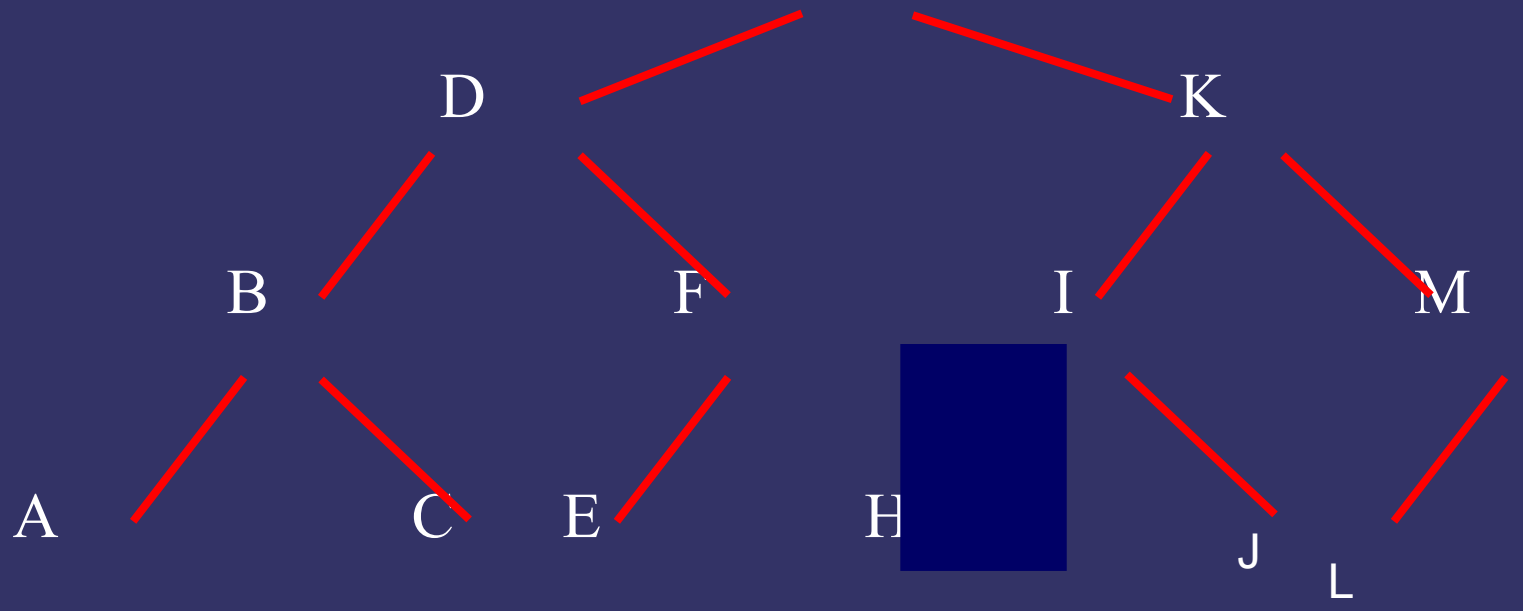
CANCELLAZIONE INTERO ALBERO

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

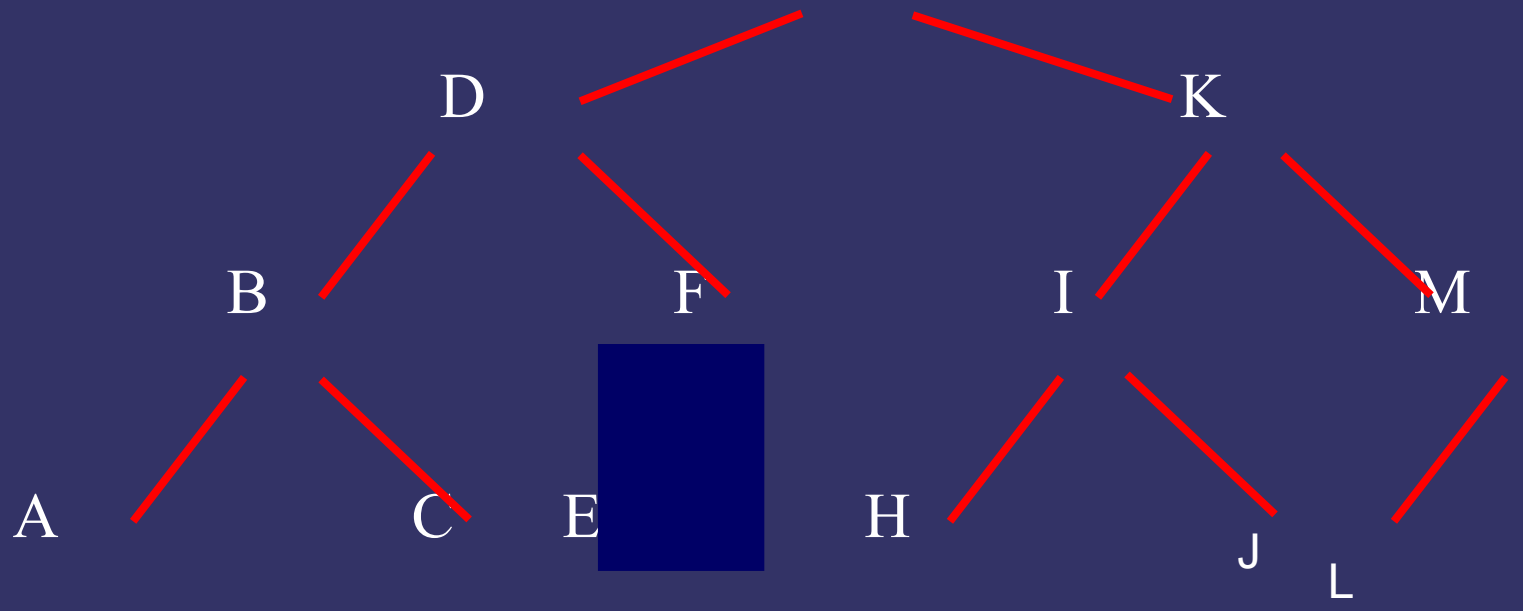
Cancella un nodo



Cancella un nodo



Cancella un nodo



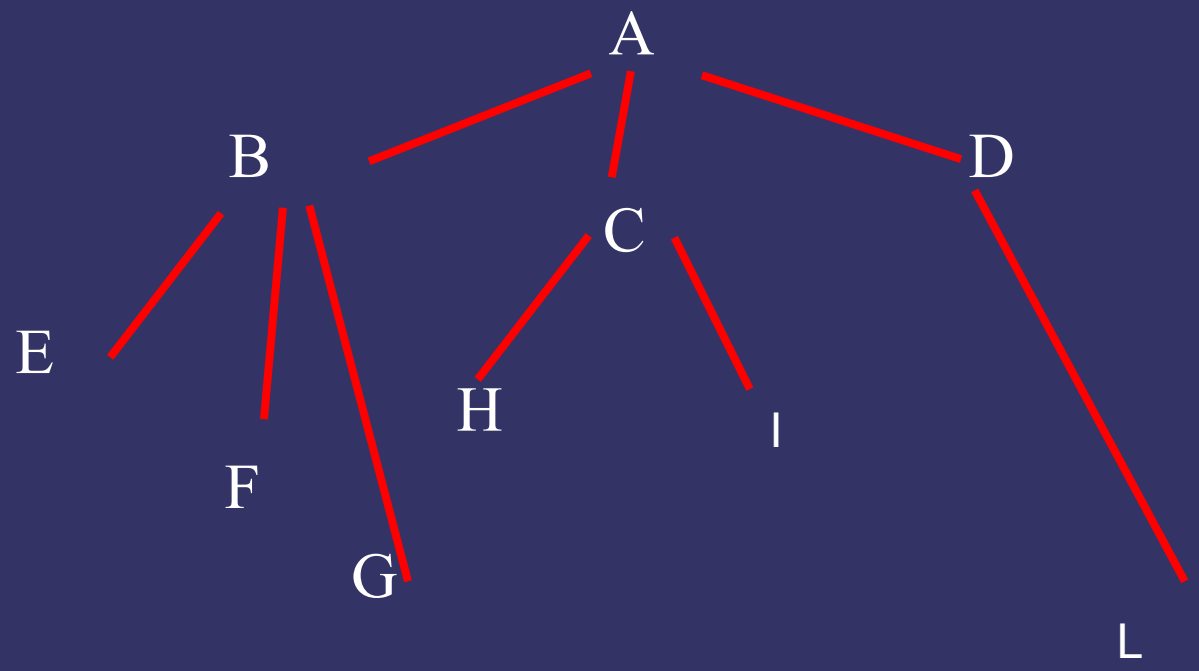
CANCELLAZIONE DI UN ELEMENTO IN UN ALBERO BINARIO DI RICERCA

```
void cancella (albero *A, Tatomo dato)
{    Tatomo rad;
albero Q;
if (!null(*A)){
      rad = radice(*A)
      if (rad > dato) cancella(&(sin(*A));
      else {
            if  (rad < dato) cancella(&des(*A));
            else {
                  Q = *A;
                  if (null(des(Q)) *A = (*A)->sn;
                  else if (null(sin(Q))) *A = (*A)->dx;
                        else rim(Q, &sin(Q));
}}}}
```

CANCELLAZIONE DI UN ELEMENTO IN UN ALBERO BINARIO DI RICERCA

```
void rim(Alb B, Alb *C)
{
    if (!null(dest(*C)))
        rim(B,&dest(*C));
    else
    {
        B->dato = radice(*C);
        free( *C);
    }
}
```

Albero generale



Alberi generali

- Definito in modo ricorsivo, un albero è:
  - vuoto
  - un elemento (radice) associato a una foresta di (sotto)alberi, cioè una lista di sottoalberi.
- ADT albero genreale

  Il tipo Abero è un ADT <S,F,C dove

  S = {albero, foresta, atomo, boolean}

  albero è il dominio di interesse

  foresta è una lista di alberi

  atomo è il dominio degli elementi che formano l'albero

Alberi

F = {costruiuci, radice, ritornaforesta, null, primoalbero, restoforesta}

costruisci : atomo x foresta $\rightarrow$albero

radice : albero $\rightarrow$ atomo

ritornaforesta : albero $\rightarrow$ foresta

primoalbero : foresta $\rightarrow$albero

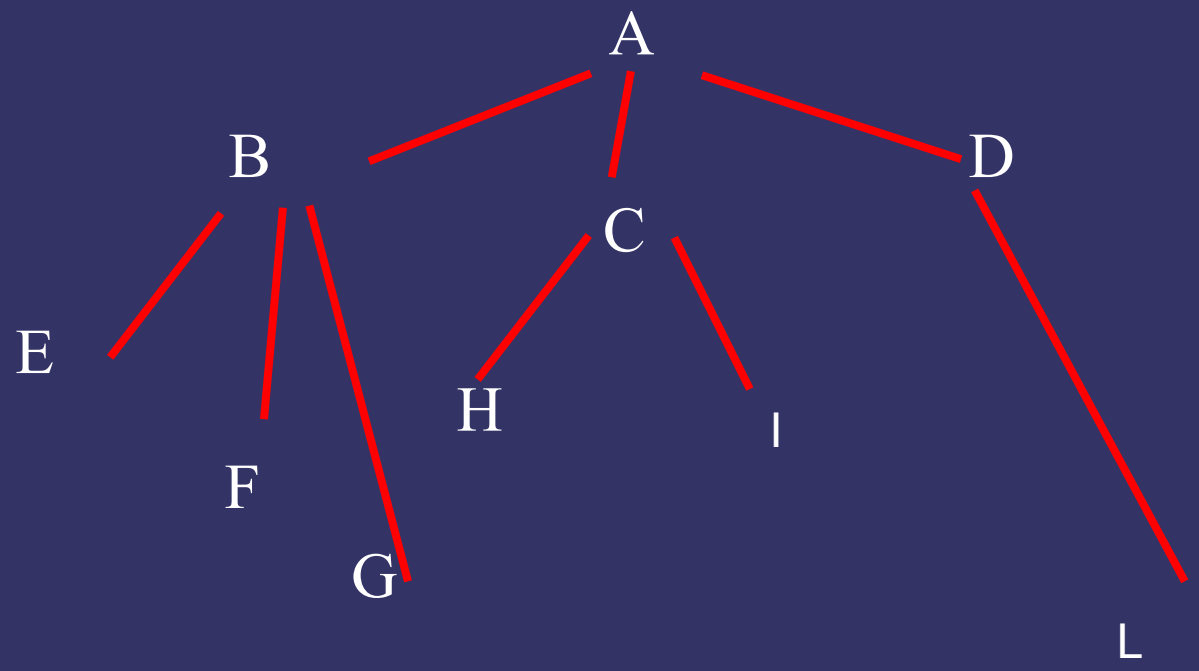
restoforesta : foresta $\rightarrow$foresta

nullA : albero $\rightarrow$ boolean

nullF : foresta $\rightarrow$ boolean

C = albero è vuoto , è la costante che denota l'albero privo di elementi

Albero generale

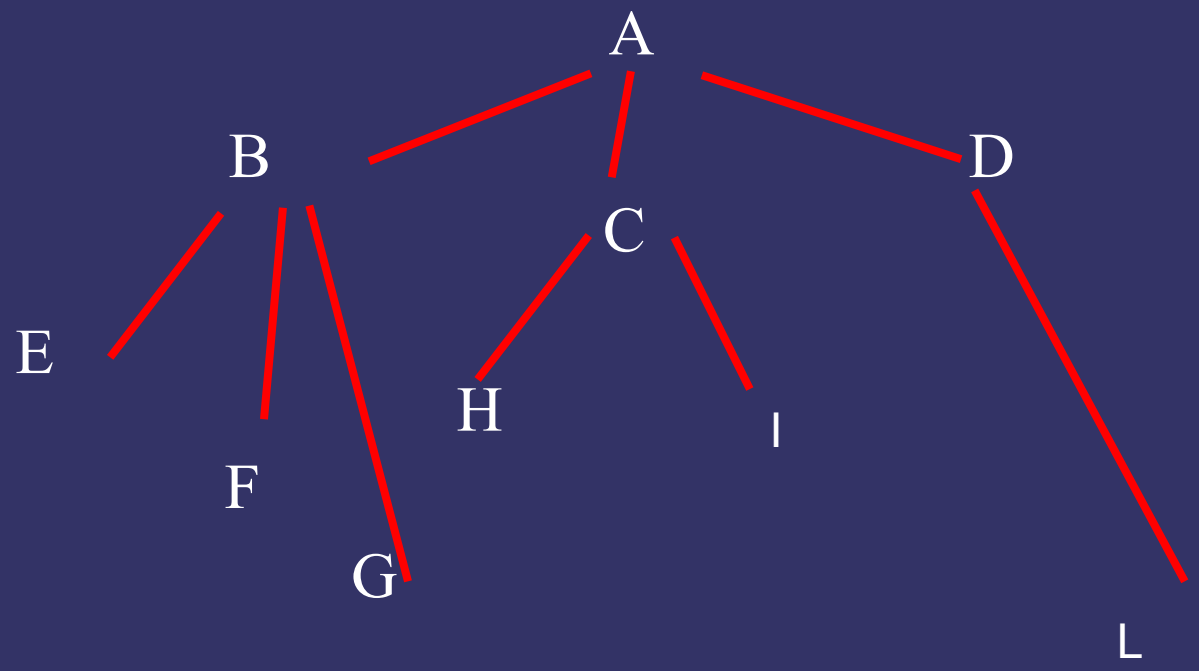


Visita in ordine anticipato

```
void visita_anticipata(albero A){
      if (!nullA(A)) {
            stampa (radice(A));
            visitaforesta(ritornaforesta(A);
      }
}

void visitaforesta(foresta F){
      while (!nullF(f)) {
            visita_anticitata(primoalbero(F));
            F = restoforesta(F);
         }
}
```

Albero generale



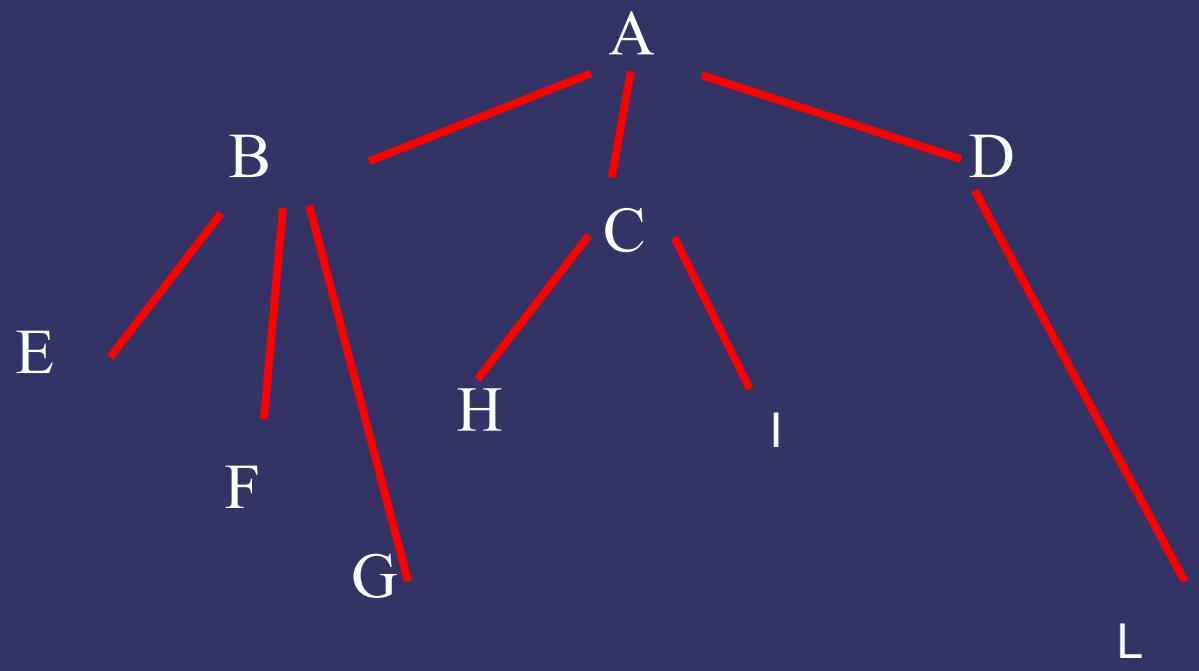
A B E F G C H I D L

Visita in post_ordine

```
void visita_postordine(albero A){
    if (!nullA(A)) {
        visitaforesta(ritornaforesta(A);
        stampa (radice(A));
    }
}

void visitaforesta(foresta F){
    while (!nullF(f)) {
        visita_postordine(primoalbero(F));
        F = restoforesta(F)
    }
}
```

Albero generale



E F G B H I C L D A

Alberi generali

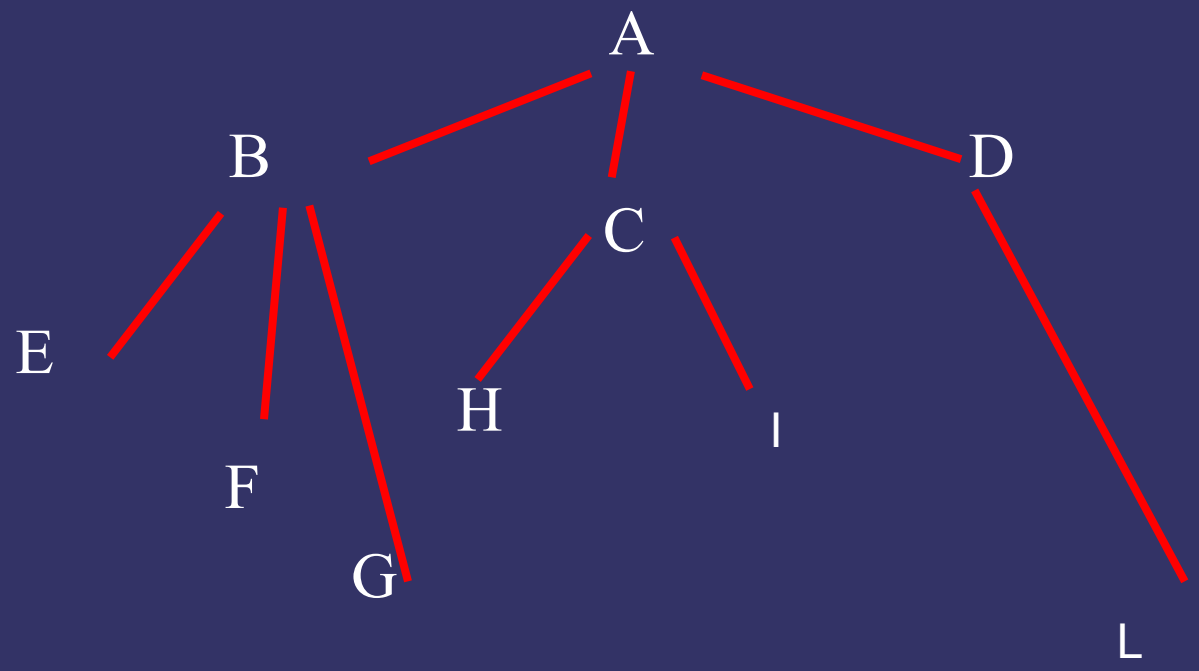
- In C si può definire un albero generale nel modo seguente:

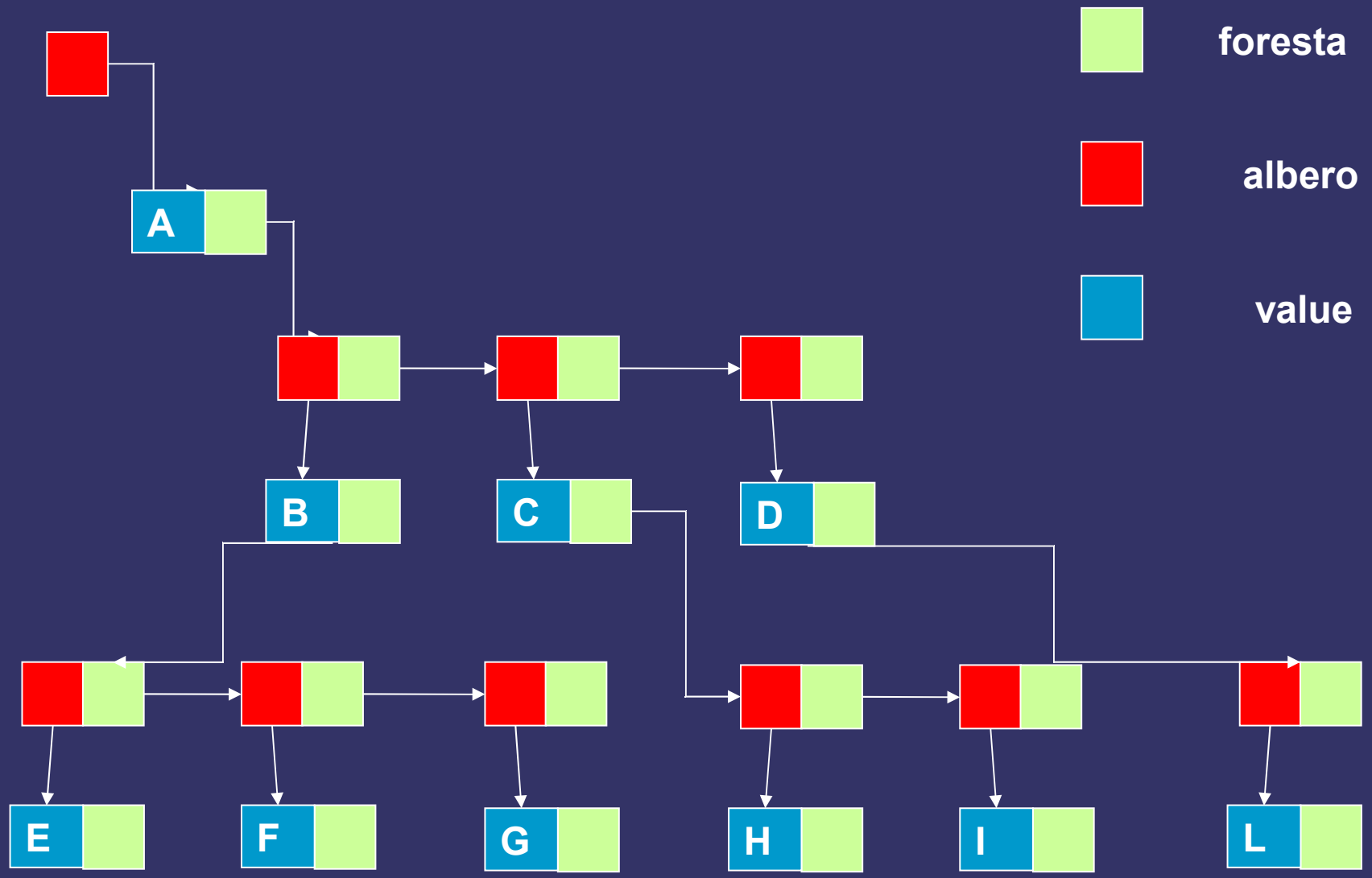
```
typedef  int Tatomo;

typedef struct nodoA *albero;
typedef struct nodoF *foresta;

struct nodoA{
    TAtomo value;
    foresta next;
};
struct nodoF{
    albero alb;
    foresta next;
};
```

Albero generale




foresta
albero
value
A
B
C
D
E
F
G
H
I
L

Implementazione

```
int costruisci(Tatomo x, Foresta f, albero *PA) {
    *PA = (albero)malloc(sizaof(struct nodoA))
    if (*PA == NULL) return 0 albero
    *PA ->value = x;
    *PA ->next = f;
    return 1
}

Tatomo  radice(albero A){return A->value)}

foresta   ritornafoesta(albero A){return A->next}
```

Implementazione

albero primoalbero (foresta f) {return: f ->alb}

foresta restoforesta (foresta f) {return: f ->next}

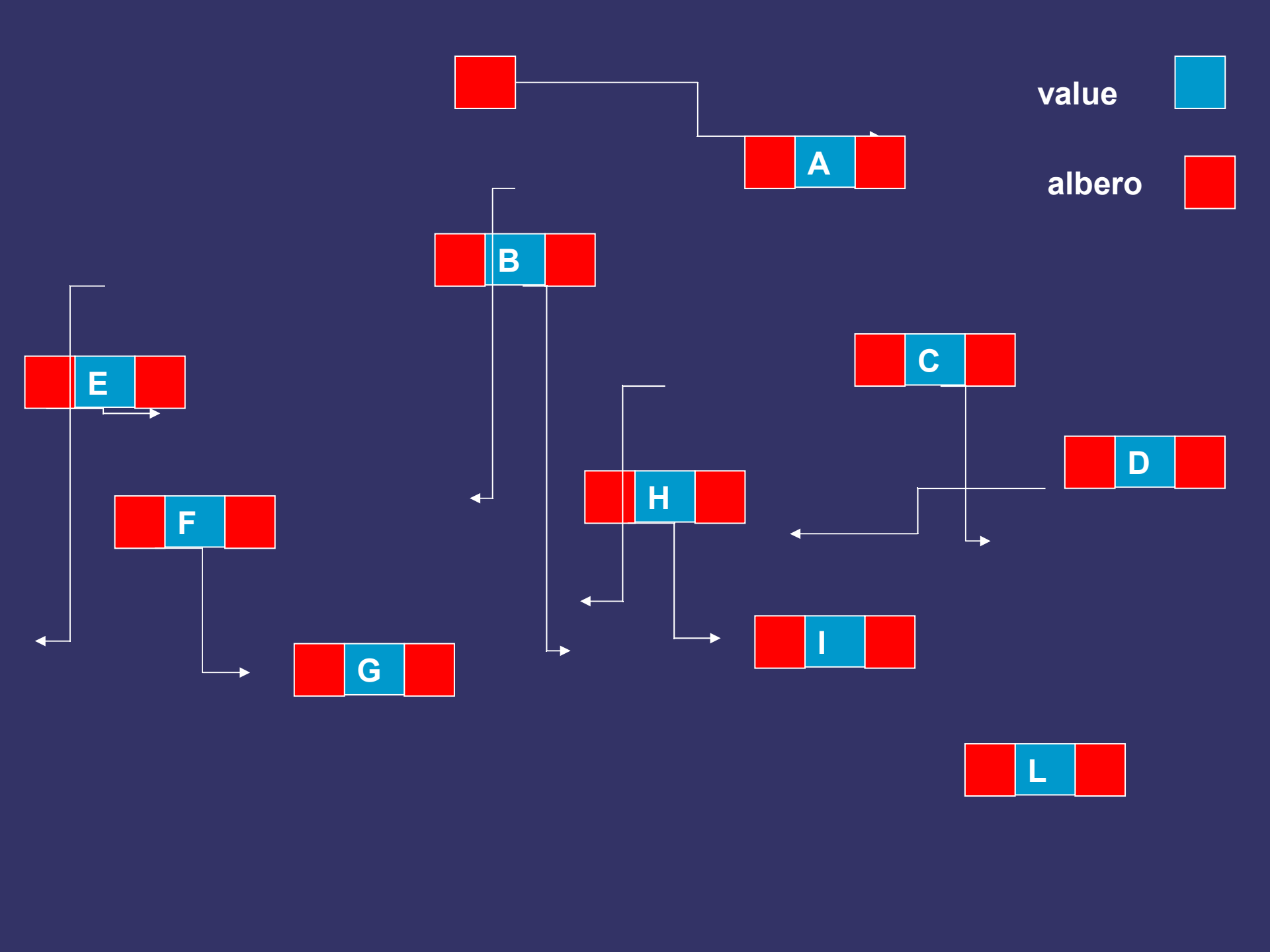
int nullA (albero A) {return A == NULL}

int nullF (foresta F) {return F == NULL}

Alberi generali

- Ogni albero genrerale può essere trasformato in un albero binario equivalente che ha come sottoalbero sinistro il primo figlio e come sottoalbero destro il fratello:

```
typedef  int Tatomo;
struct nodoA{
   TAtomo value;
   struct nodo *figlio, *fratello;
}albero;
```


value
albero
A
B
C
D
E
F
G
H
I
L