

Laboratorio 22

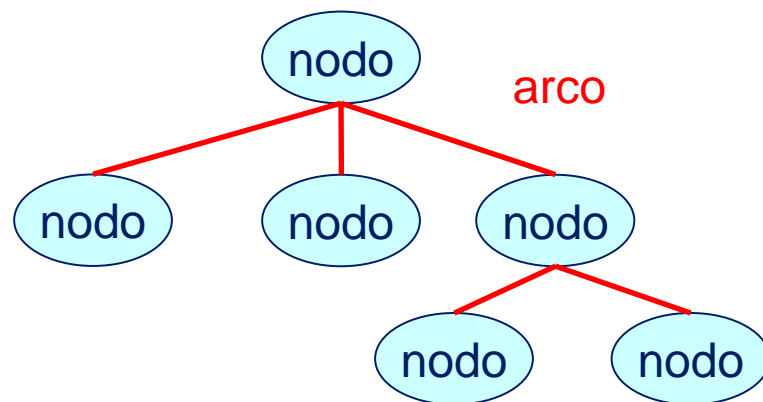
DISI – aa 2023/24

Pierluigi Roberti
Carmelo ferrante

Alberi binari di ricerca

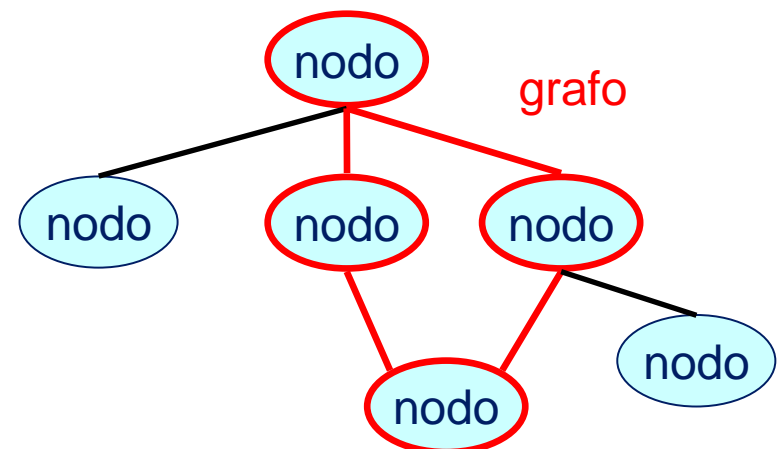
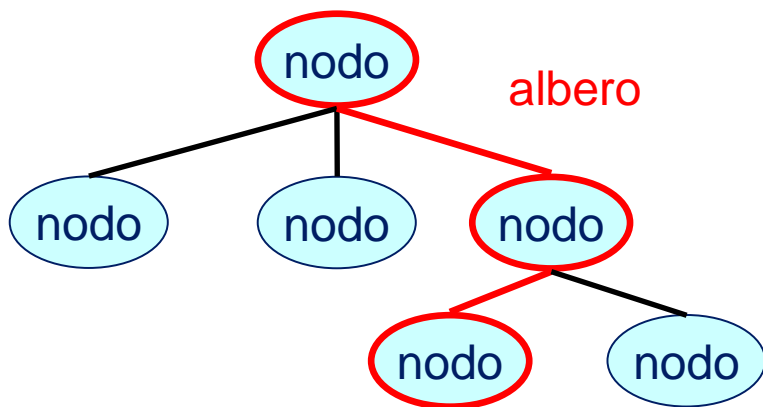
Albero

- **Definizione**: Un albero è un insieme non vuoto di *vertici* ed *archi* che soddisfa alcune proprietà:
 - Un *vertice* (o *nodo*) è un oggetto semplice che può essere dotato di un nome, e di una informazione associata (denominata spesso *chiave* o *key*).
 - Un *arco* è una connessione tra due nodi.



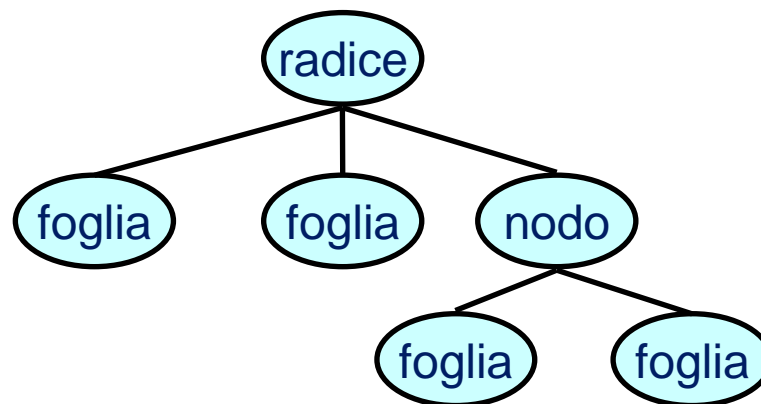
Cammino in un albero

- **Definizione:** un *cammino* nell'albero è una sequenza di vertici distinti, in cui i vertici successivi sono connessi da un arco dell'albero.
- La proprietà che definisce un albero è quella per cui esiste *esattamente* un cammino che connette ogni coppia di nodi.
 - Se tra una coppia di nodi esiste più di un cammino, parliamo di grafi (trattati nella prossima lezione).
- Un insieme di alberi disgiunto si chiama *foresta*.



Alberi: definizioni

- Un albero *con radice* è un albero in cui un particolare nodo viene identificato come la *radice* o *root* dell'albero (sono le strutture classicamente utilizzate in informatica).
- In un albero con radice, ogni nodo è la radice di un *sottoalbero* formato dal nodo medesimo e da tutti i nodi ad esso sottostanti.
- Esiste un solo cammino tra la radice e ognuno degli altri nodi dell'albero.

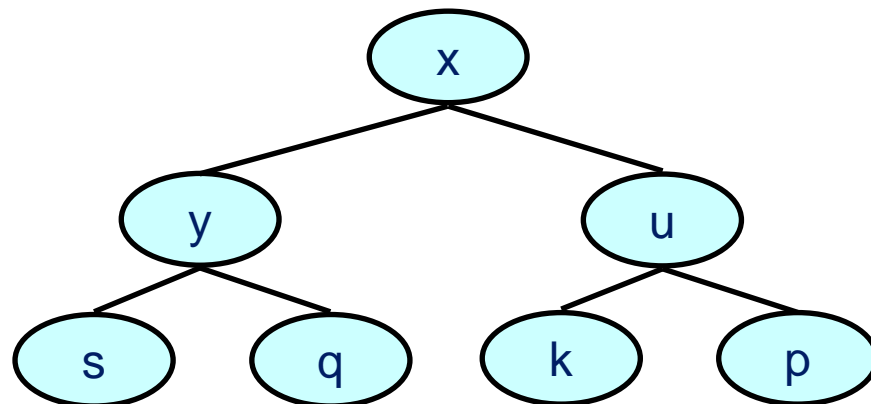


Alberi: definizioni

- Un albero è *ordinato* se è un albero con radice e se è specificato l'ordine dei figli di ciascun nodo.
- Se ogni nodo *deve* avere un numero specifico N di figli, parliamo di albero *N-ario*.
 - Un caso particolare è l'albero binario, dove $N = 2$.
- La distinzione tra alberi ordinati ed alberi N-ari riguarda il numero di figli di ciascun nodo:
 - In un albero ordinato i nodi possono avere un numero arbitrario di figli.
 - In un albero N-ario il numero di figli è al massimo N .

Alberi binari

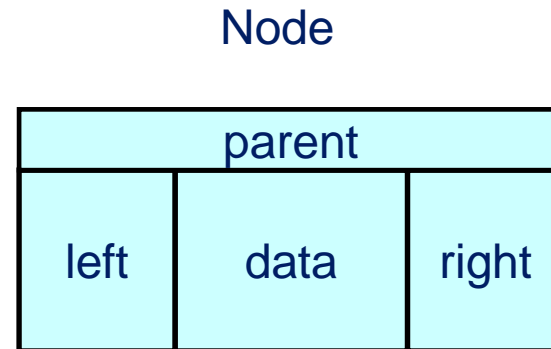
- Un albero binario è un albero ordinato formato da due tipi di nodi:
 - Nodi terminali foglie.
 - Nodi interni con due figli. Dato che i due figli sono ordinati parleremo di:
 - *Figlio di sinistra*
 - *Figlio di destra*



- y figlio sinistro di x
- u figlio destro di x
- k figlio sinistro di u
- p figlio destro di u

Rappresentazione di alberi binari

```
struct Node {  
    int data;  
    Node * parent;  
    Node * left;  
    Node * right;  
    Node(int d) {  
        data = d;  
        parent = NULL;  
        left = right = NULL;  
    }  
} Node;
```



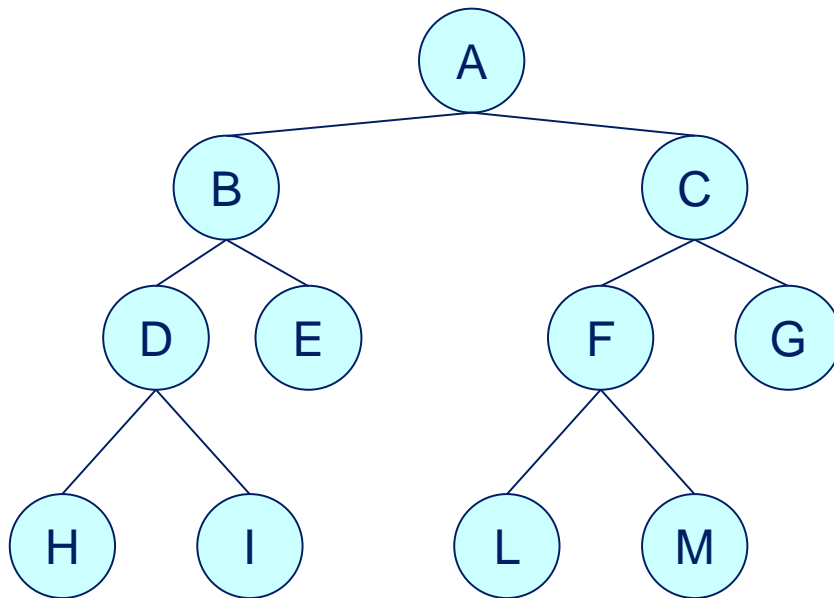
Puntatore a parent
sarà utile per realizzare
alcuni algoritmi in modo efficiente

Algoritmi di attraversamento di alberi

■ Abbiamo diversi *modi* per attraversare un albero:

- Pre-ordine (*preorder*): visitiamo prima il nodo e poi i sottoalberi di sinistra e di destra.
- In-ordine (*inorder*): visitiamo prima il sottoalbero di sinistra, poi il nodo, ed infine il sottoalbero di destra.
- Post-ordine (*postorder*): visitiamo prima i sottoalberi di sinistra e di destra, poi il nodo.

Algoritmi di attraversamento di alberi



- Preorder:
A B D H I E C F L M G
- Inorder:
H D I B E A L F M C G
- Postorder:
H I D E B L M F G C A

Algoritmi ricorsivi di attraversamento di alberi binari

- Visit è una funzione che stampa il contenuto del nodo

```
void preorder(Node * h, void visit(Node *)) {  
    if (h == NULL) return;  
    visit(h);  
    preorder(h->left, visit);  
    preorder(h->right, visit);  
}
```

Algoritmi ricorsivi di attraversamento di alberi binari

```
void inorder(Node * h, void visit(Node *)) {  
    if (h == NULL) return;  
    inorder(h->left, visit);  
    visit(h);  
    inorder(h->right, visit);  
}
```

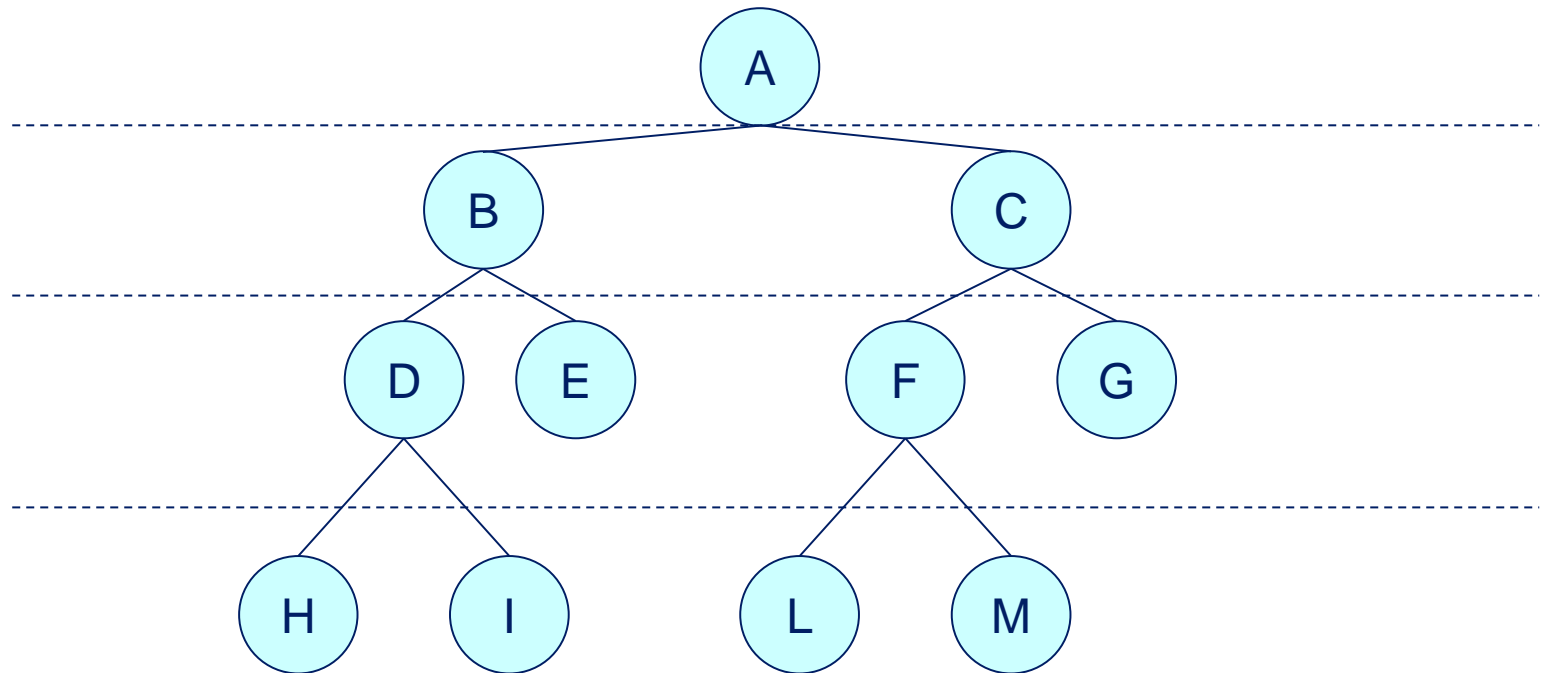
Algoritmi ricorsivi di attraversamento di alberi binari

```
void postorder(Node * h, void visit(Node *)) {  
    if (h == NULL) return;  
    postorder(h->left, visit);  
    postorder(h->right, visit);  
    visit(h);  
}
```

Ulteriori tecniche di attraversamento di un albero

- Un modo alternativo per visitare un albero consiste nel visitare i nodi secondo l'ordine in cui essi appaiono sulla carta scandendo gli elementi dalla cima al fondo e da sinistra verso destra.
- Questo metodo prende il nome di **level-order**, in quanto i nodi di ciascun livello sono visitati uno dopo l'altro.
- La caratteristica di questo algoritmo consiste nel fatto che *non* corrisponde ad una implementazione legata alla struttura ricorsiva dell'albero.

Attraversamento *level-order*



Attraversamento *level-order*

A B C D E F G H I L M

Esercizio 1

creazione albero di strutture

- Creare una struttura denominata **Tdato** con
 - Campi: **int eta; char nome[LMAX];**
 - Costruttori: default e specifico
 - Metodo: stampa => [nome-eta]
- Creare una struttura per un nodo denominata **Tnodo** con
 - Campi: **Tdata dato; Tnodo* dx; Tnodo* sx;**
 - Costruttori: default e specifico
 - Metodo: stampa => richiamare stampa Tdata
- Creare i seguenti alias:
typedef Tdato Dato;
typedef Tnodo Nodo;
typedef Tnodo* Tree;
- Nel main creare una variabile t1 di tipo puntatore a Tnodo (Tree) ed inizializzarla opportunamente
Tree t1;

Esercizio 1

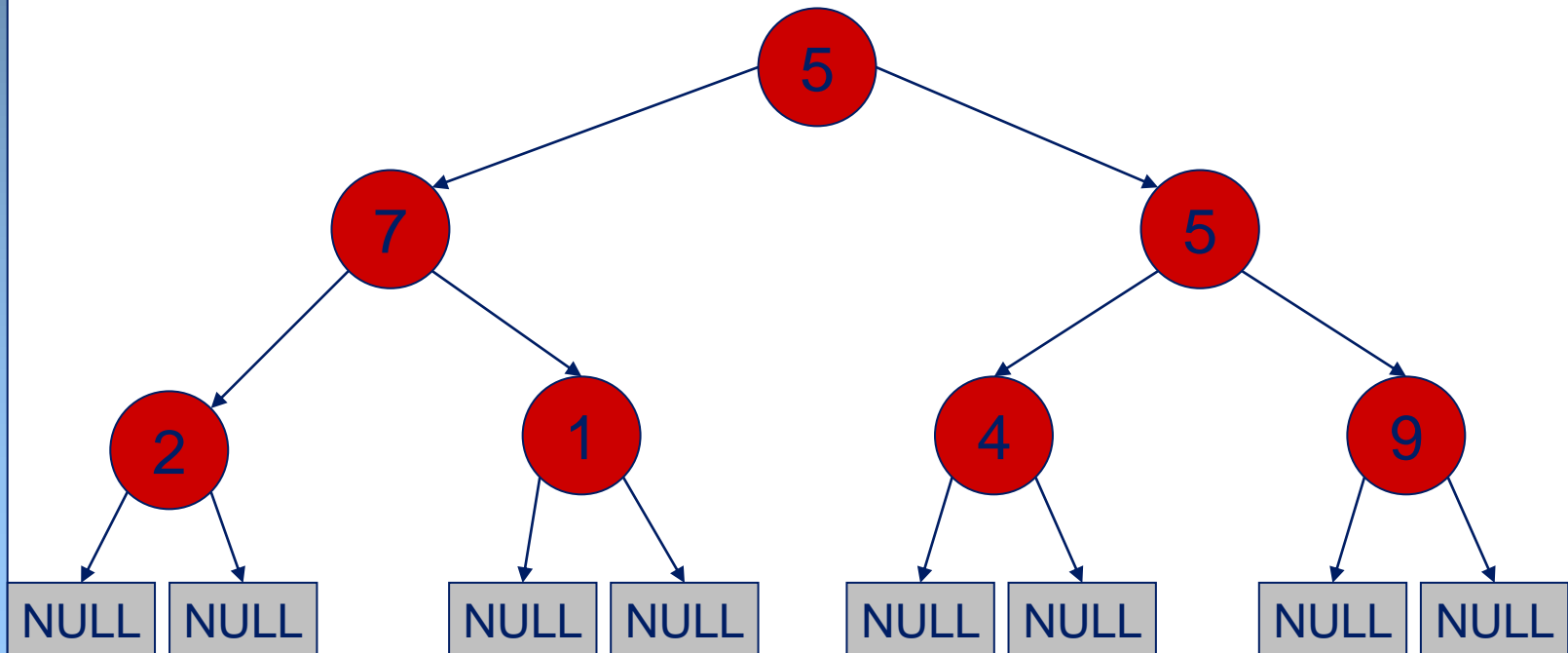
creazione albero di strutture

- Implementare le seguenti funzioni:

Tree costruisci(Dato d, Tree sx, Tree dx) ;

void inordine(Tree radice) ;

- Per creare il seguente albero e stamparne il contenuto



Esercizio 1

creazione albero di strutture

```
Tree costruisci(Dato d, Tree sx, Tree dx){
    Tree radice;
    radice = new Nodo();
    radice->dato = d;
    radice->sx = sx;
    radice->dx = dx;
    return radice;
    // return new Nodo(d, sx, dx);
}

void inordine(Tree radice){
    if (!(radice==NULL)) {
        inordine(radice->sx);
        radice->stampa();
        inordine(radice->dx);
    }
}
```

Esercizio 1

creazione albero di strutture

```
void stampa_nodo(Tree n ){
    n->stampa();
}

void inordine_gen(Tree radice, void f(Tree) ){
    if (!(radice==NULL)) {
        inordine(radice->sx);
        f(radice);
        inordine(radice->dx);
    }
}

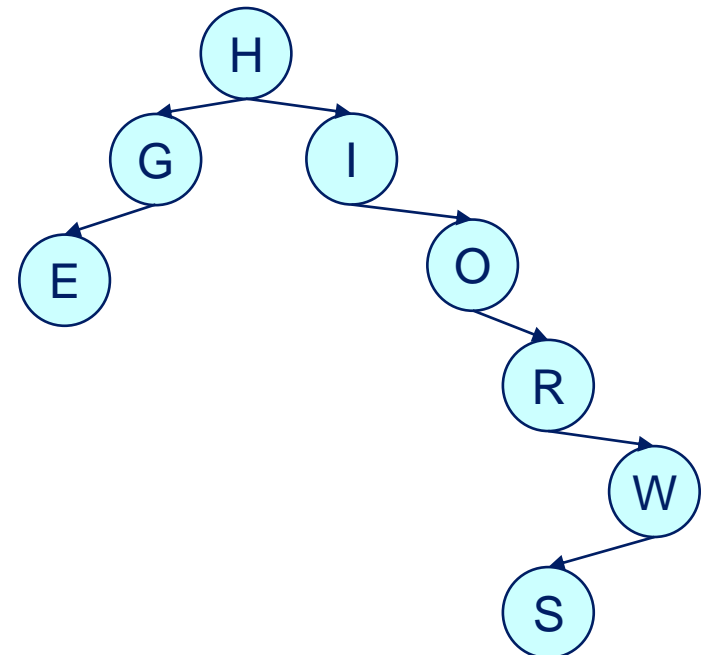
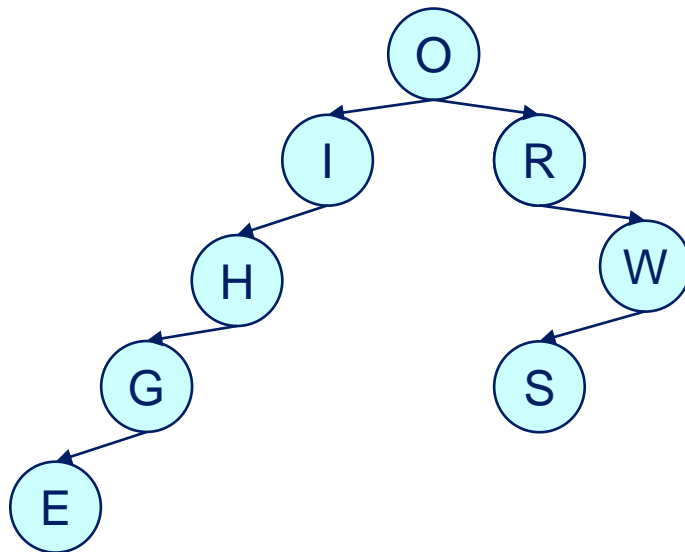
// esempio invocazione
Tree root=NULL;
// ... popolare albero...
inordine_gen(root, stampa);
```

Sintassi nel caso in cui
parametro di una funzione
è una funzione

Alberi binari di ricerca

■ **Definizione:** Un *albero binario di ricerca* (*binary search tree (BST)*) è un albero binario dove la *chiave* (dato) memorizzata in ciascun nodo è:

- maggiore o uguale alla chiave di tutti i nodi del sottoalbero sinistro di quel nodo;
- minore o uguale alle chiavi di tutti i nodi del sottoalbero destro di quel nodo.



Alberi binari di ricerca

- Su un albero binario di ricerca di solito si definiscono alcune operazioni base:
 - *Inserimento* di un nuovo dato nell'albero.
 - *Cancellazione* di un dato dal BST.
 - *Ricerca* di un dato nel BST.
 - *Ordinamento* dei dati del BST.
 - *Ricerca del minimo/massimo* in un BST.
 - *Ricerca del successore/predecessore* in un BST.
 - *Unione* di due BST.

Minimo e Massimo in un BST

- L'elemento in un BST la cui chiave sia minima (massima) può essere determinato seguendo il nodo *left* (*right*, rispettivamente).

```
Node * Tree_Min(Node * t) {  
    // we assume t != NULL  
    while(t->left != NULL)  
        t = t->left;  
    return t;  
}
```

```
Node * Tree_Max(Node * t) {  
    // we assume t != NULL  
    while(t->right != NULL)  
        t = t->right;  
    return t;  
}
```

Ricerca in un BST

- L'operazione più comune su un BST è la ricerca di una chiave memorizzata nell'albero.
 - La procedura analizza la radice tracciando un percorso nell'albero.
 - Per ogni nodo X dell'albero incontrato, confronta la chiave del nodo X con la chiave cercata.
 - Se le due chiavi sono uguali, abbiamo terminato.
 - Se la chiave è minore della chiave di X , continuo la ricerca sul nodo sinistro.
 - Se la chiave è maggiore della chiave di X , continuo la ricerca sul nodo destro.
 - La complessità asintotica di questo algoritmo è $O(h)$ dove h è l'altezza del BST.

Ricerca in un BST

```
Node * Tree_Search(Node * x, int k) {  
    if ((x == NULL) || (x->data == k)) return x;  
    if (k < x->data)  
        return Tree_Search(x->left, k);  
    else  
        return Tree_Search(x->right, k);  
}
```

Ricerca in un BST

```
Node * Tree_Search(Node * x, int k) {  
    while ((x != NULL) && (x->data != k)) {  
        if (k < x->data)  
            x = x->left;  
        else  
            x = x->right;  
    }  
    return x;  
}
```


Esercizio 2

creazione albero **ordinato** di strutture

- Implementare le seguenti funzioni:

Tree ins_ord (Tree radice, Dato x)

bool ricerca (Tree radice, Dato x)

bool ricercaBin (Tree radice, Dato x)

Maggiore
a destra
Minore o uguale
a sinistra

- Per creare il un albero ordinato a partire da questa sequenza

5 7 3 2 1 4 9

- Stamparne il contenuto (funzione inordine)
- Ricercare con i 2 approcci (ricerca e ricercaBin) il valore **9** all'interno dell'albero

Esercizio 2

creazione albero ordinato di strutture

- Utile definire metodi per il confronto in struttura Tdato

bool eq (Tdato x)

// eq: equal – controlla se un Tdato è uguale a x

bool gt (Tdato x)

// gt: greater than – controlla se un Tdato è più grande di x

bool lt (Tdato x)

// lt: lesser than – controlla se un Tdato è più piccolo di x

Esercizio 2

creazione albero ordinato di strutture

Tree ins_ord (Tree n, Dato x)

// Ricorsione

// Inserire Dato x in albero partendo da Nodo n

Se Nodo n NULL

 Creare nuovo Nodo con Dato x

 Restituire indirizzo nuovo Nodo

Se x maggiore di dato di n

 Inserire Dato x in sotto-albero di destra

 Aggiornare Nodo n corrente

 Restituire indirizzo Nodo n

Altrimenti

 Inserire Dato x in sotto-albero di sinistra

 Aggiornare Nodo n corrente

 Restituire indirizzo Nodo n

Esercizio 2

creazione albero ordinato di strutture

bool ricerca(Tree n, Dato x)

// Ricorsione

// Ricerca Dato x in albero partendo da Nodo n

Se Nodo n NULL

Restituire FALSE

Se x uguale a dato di n

Restituire TRUE

dato_in_sx = Ricerca Dato x in sotto-albero di sinistra

dato_in_dx = Ricerca Dato x in sotto-albero di destra

Se dato presente in uno dei due sotto-alberi

Restituire TRUE

Altrimenti

Restituire FALSE

Esercizio 2

creazione albero ordinato di strutture

bool ricercaBin(Tree n, Dato x)

// Ricorsione

// Ricerca Dato x in albero partendo da Nodo n

Se Nodo n NULL

Restituire FALSE

Se x uguale a dato di n

Restituire TRUE

Se x maggiore di dato di n

Ricerca Dato x in sotto-albero di destra

Restituire risultato della ricerca

Altrimenti

Ricerca Dato x in sotto-albero di sinistra

Restituire risultato della ricerca

Algoritmi di utilità su alberi

- Spesso capita di dover calcolare i valori di alcuni parametri strutturali di un albero, partendo dal nodo radice.
 - Calcolare il numero di nodi dell'albero.
 - Calcolare l'altezza di un albero.
 - Cammino in un albero.
 - Stampare un albero.
 - Disegnare un albero.

Calcolo numero nodi di un albero

- Il problema del calcolo del numero di nodi di un albero è molto semplice:

```
int contaNodi( Node * h ) {  
    if (h == NULL) return 0;  
    return contaNodi(h->left) + contaNodi(h->right) + 1;  
}
```

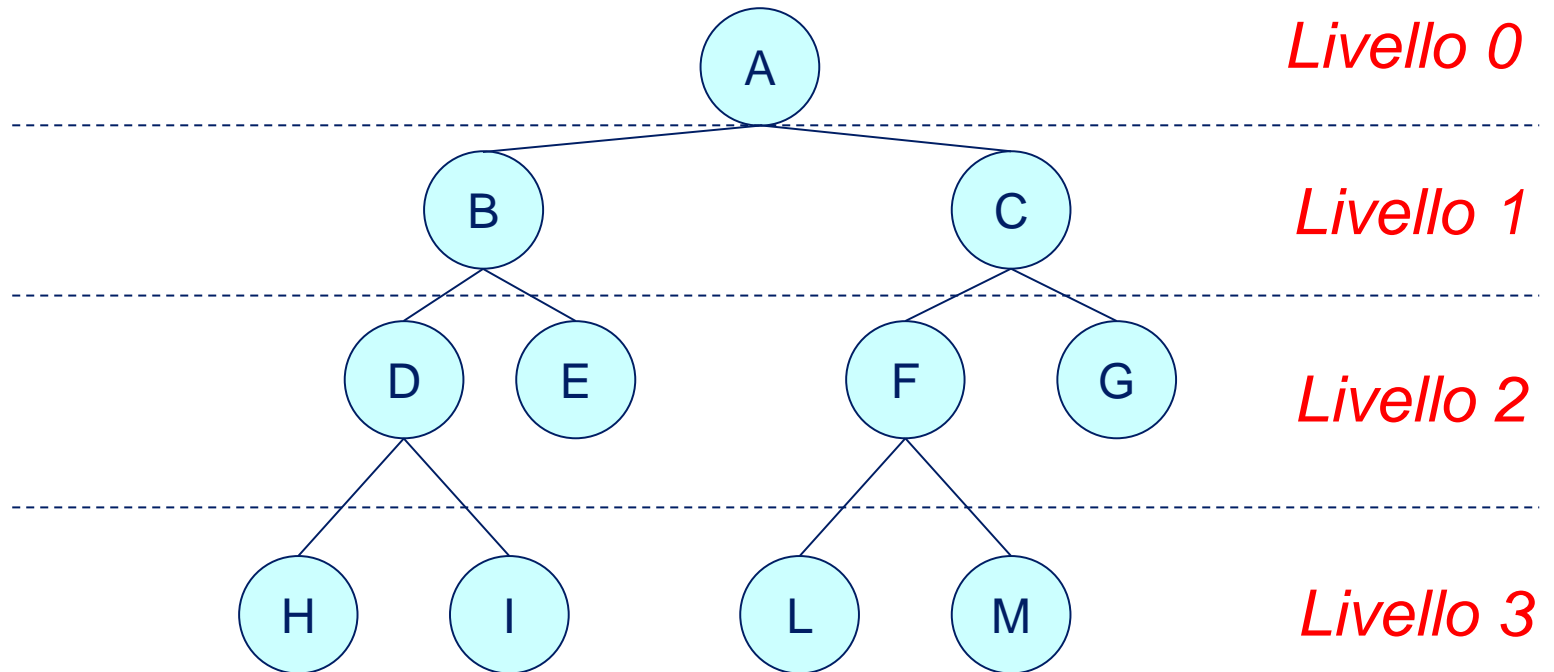
- Questo semplice algoritmo non dipende dall'ordine delle chiamate ricorsive.
 - Scambiando left con right il risultato non cambia.

Calcolo dell'altezza di un albero

- **Definizione:** Il *livello* di un albero è definito ricorsivamente sulla struttura dell'albero nel modo seguente:
 - la radice ha livello 0;
 - ogni altro nodo ha un livello pari al livello del padre più 1.
- **Definizione:** L'altezza di un albero è pari al massimo tra i livelli di tutti i suoi nodi.

```
int height (Node * h) {  
    if (h == NULL) { return -1; }  
    int u = height(h->left); // ordine non importante  
    int v = height(h->right);  
    if (u > v) return u+1;  
    return v + 1;  
}
```


Calcolo dell'altezza di un albero



Altezza dell'albero = 3

Calcolo dell'altezza di un albero

- **Definizione:** Il *livello* di un albero è definito ricorsivamente sulla struttura dell'albero nel modo seguente:
 - la radice ha livello 0;
 - ogni altro nodo ha un livello pari al livello del padre più 1.
- **Definizione:** L'altezza di un albero è pari al massimo tra i livelli di tutti i suoi nodi.

```
int height (Node * h) {  
    if (h == NULL) return -1;  
    int r = MAX(height(h->left), height(h->right));  
    return r+1;  
}
```

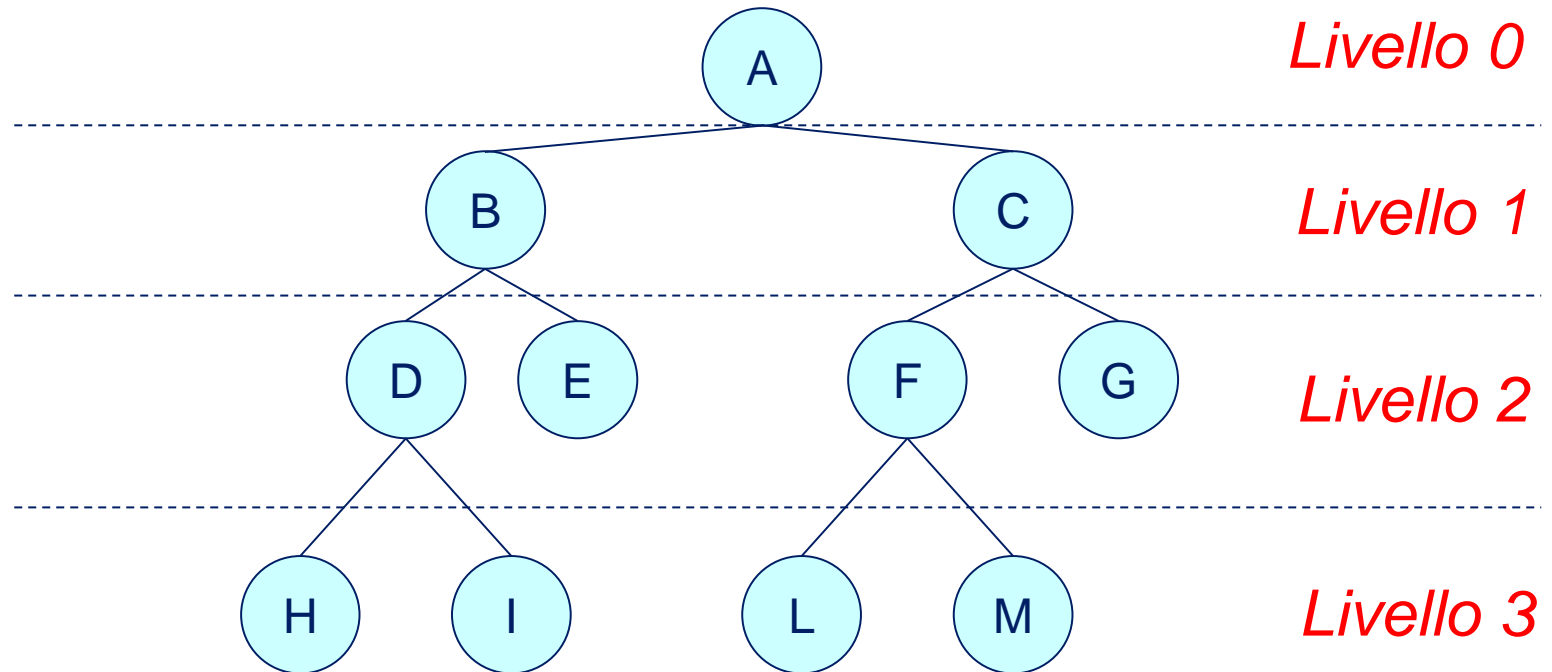
```
int MAX(int a, int b) {  
    if (a > b) return a;  
    return b;  
}
```

Lunghezza del cammino di un albero

- **Definizione**: La *lunghezza del cammino* di un albero è la somma dei livelli di tutti i nodi dell'albero.
- Scrivere un programma che sfruttando la definizione di cui sopra calcoli la lunghezza del cammino di un albero:
 - La lunghezza del cammino di un nodo nullo è 0.
 - La lunghezza di un cammino di un nodo di livello h è pari a h addizionata alla lunghezza del cammino del sottoalbero di sinistra (l) e alla lunghezza del cammino del sottoalbero di destra (r).

$$lp(n, h) = \begin{cases} 0 & \text{se } n = NULL \\ h + lp(n.left, h + 1) + lp(n.right, h + 1) & \text{se } n \neq NULL \end{cases}$$

Lunghezza del cammino di un albero



$$\begin{aligned}\text{Cammino} &= 1 + 1 + 2 + 2 + 2 + 2 + 3 + 3 + 3 + 3 \\ &= 2 + 8 + 12 = 22\end{aligned}$$

Lunghezza del cammino di un albero

```
int path_length(Node * n) {  
    return path_length_recur(n, 0);  
}  
  
int path_length_recur(Node * n, int h) {  
    if (n == NULL) { return 0;}  
    int ll = path_length_recur(n->left, h+1);  
    int lr = path_length_recur(n->right, h+1);  
    return h + ll + lr;  
}
```

Esercizio 3

analisi albero ordinato di strutture

- Implementare le seguenti funzioni:

void preordine(Tree radice)

void postordine(Tree radice)

int altezza(Tree radice)

int contaNodi(Tree radice)

int contaFoglie(Tree radice)

int lungCammino(Tree radice, int h)

- Invocare correttamente dal main tali funzioni stampando l'eventuale dato ritornato in modo appropriato

Esercizio 3

analisi albero ordinato di strutture

```
int altezza(Tree n)
```

```
// Ricorsione
```

```
// Calcola altezza di albero binario partendo da Nodo n
```

```
    Se Nodo n NULL
```

```
        Restituire -1
```

```
    alt_sx = calcolare altezza sotto-albero sinistra
```

```
    alt_dx = calcolare altezza sotto-albero destra
```

```
    Restituire 1 + massimo tra alt_dx e alt_sx
```

Esercizio 3

analisi albero ordinato di strutture

```
int contaNodi(Tree n)
```

```
// Ricorsione
```

```
// Calcolare numero nodi in albero binario partendo da Nodo n
```

```
    Se Nodo n NULL
```

```
        Restituire 0
```

```
    n_sx = calcolare numero nodi in sotto-albero sinistra
```

```
    n_dx = calcolare numero nodi in sotto-albero destra
```

```
    Restituire 1 + n_dx + n_sx
```


Esercizio 3

analisi albero ordinato di strutture

```
int contaFoglie(Tree n)
```

```
// Ricorsione
```

```
// Foglia → nessun nodo figlio → campi sx e dx NULL
```

```
// Calcolare numero foglie in albero binario partendo da Nodo n
```

```
    Se Nodo n NULL
```

```
        Restituire 0
```

```
    Se Nodo n non ha figli
```

```
        Restituire 1
```

```
    f_sx = calcolare numero foglie in sotto-albero sinistra
```

```
    f_dx = calcolare numero foglie in sotto-albero destra
```

```
    Restituire f_dx + f_sx
```

Esercizio 3

analisi albero ordinato di strutture

```
int lungCammino(Tree n, int lvl)
```

```
// Ricorsione
```

```
// Calcolare lunghezza cammino partendo da Nodo n con livello lvl
```

```
    Se Nodo n NULL
```

```
        Restituire 0
```

```
    c_sx = calcolare cammino sotto-albero sinistra (livello +1)
```

```
    c_dx = calcolare cammino sotto-albero destra (livello +1)
```

```
    Restituire lvl + c_dx + c_sx
```

```
    0 se n = NULL
```

```
C(n, h) =
```

```
    h + C(sx, h+1) + C(dx, h+1) se n != NULL
```

Esempi aggiuntivi - funzioni su alberi

Ricerca in albero

Conteggio delle iterazioni

```
bool ricerca (Tree radice, Dato x){  
    if( radice == NULL )  
        return false;  
    if( radice->d.eq(x) )  
        return true;  
    return ( ricerca(radice->sx, x) || ricerca(radice->dx, x) );  
}
```

```
bool ricerca_per_valutazione (Tree radice, Dato x, int n){  
    cout << n << endl;  
    if( radice==NULL )  
        return false;  
    if( radice->d.eq(x) )  
        return true;  
    return ( ricerca_per_valutazione(radice->sx, x, n+1) ||  
            ricerca_per_valutazione(radice->dx, x, n+1) );  
}
```

```
int n=0;  
ricerca_per_valutazione(root, Tdato(3), n);
```

Ricerca in albero

Conteggio delle iterazioni

```
bool ricerca (Tree radice, Dato x){  
    if( radice == NULL )  
        return false;  
    if( radice->d.eq(x) )  
        return true;  
    return ( ricerca(radice->sx, x) || ricerca(radice->dx, x) );  
}
```

```
bool ricerca_per_valutazione (Tree radice, Dato x, int *n){  
    cout << *n << endl;  
    if( radice==NULL )  
        return false;  
    if( radice->d.eq(x) )  
        return true;  
    return ( ricerca_per_valutazione(radice->sx, x, &++(*n)) ||  
            ricerca_per_valutazione(radice->dx, x, &++(*n)) );  
}
```

```
int n=0;  
ricerca_per_valutazione(root, Tdato(3), &n);
```

Ricerca binaria in albero

Conteggio delle iterazioni

```
bool ricercaBin (Tree radice, Dato x){  
    if( radice == NULL ) return false;  
    if( radice->d.eq(x) ) return true;  
    if( x.gt(radice->d) ) return ricercaBin(radice->dx, x);  
    else return ricercaBin(radice->sx, x);  
}
```

```
bool ricercaBin_per_valutazione (Tree radice, Dato x, int n){  
    cout << n << endl;  
    if( radice == NULL ) return false;  
    if( radice->d.eq(x) ) return true;  
    if( x.gt(radice->d) ) return ricercaBin_per_valutazione(radice->dx, x, n+1);  
    else return ricercaBin_per_valutazione(radice->sx, x, n+1);  
}
```

```
int n=0;  
ricercaBin_per_valutazione(root, Tdato(3), n);
```

Ricerca binaria in albero

Conteggio delle iterazioni

```
bool ricercaBin (Tree radice, Dato x){  
    if( radice == NULL ) return false;  
    if( radice->d.eq(x) ) return true;  
    if( x.gt(radice->d) ) return ricercaBin(radice->dx, x);  
    else return ricercaBin(radice->sx, x);  
}
```

```
bool ricercaBin_per_valutazione (Tree radice, Dato x, int* n){  
    cout << *n << endl;  
    if( radice == NULL ) return false;  
    if( radice->d.eq(x) ) return true;  
    if( x.gt(radice->d) ) return ricercaBin_per_valutazione(radice->dx, x, &++(*n));  
    else return ricercaBin_per_valutazione(radice->sx, x, &++(*n));  
}
```

```
int n=0;  
ricercaBin_per_valutazione(root, Tdato(3), &n);
```

Rappresentazione grafica di albero

Sorgente in file soluzione

- Albero ordinato a partire da questa sequenza

5 7 3 2 1 4 9



Funzioni in file sorgente
delle soluzioni