

INFORMATICA GENERALE II

Ingegneria delle Telecomunicazioni

Università di Trento

A.A. 2003/2004

II Bimestre

Marco Roveri

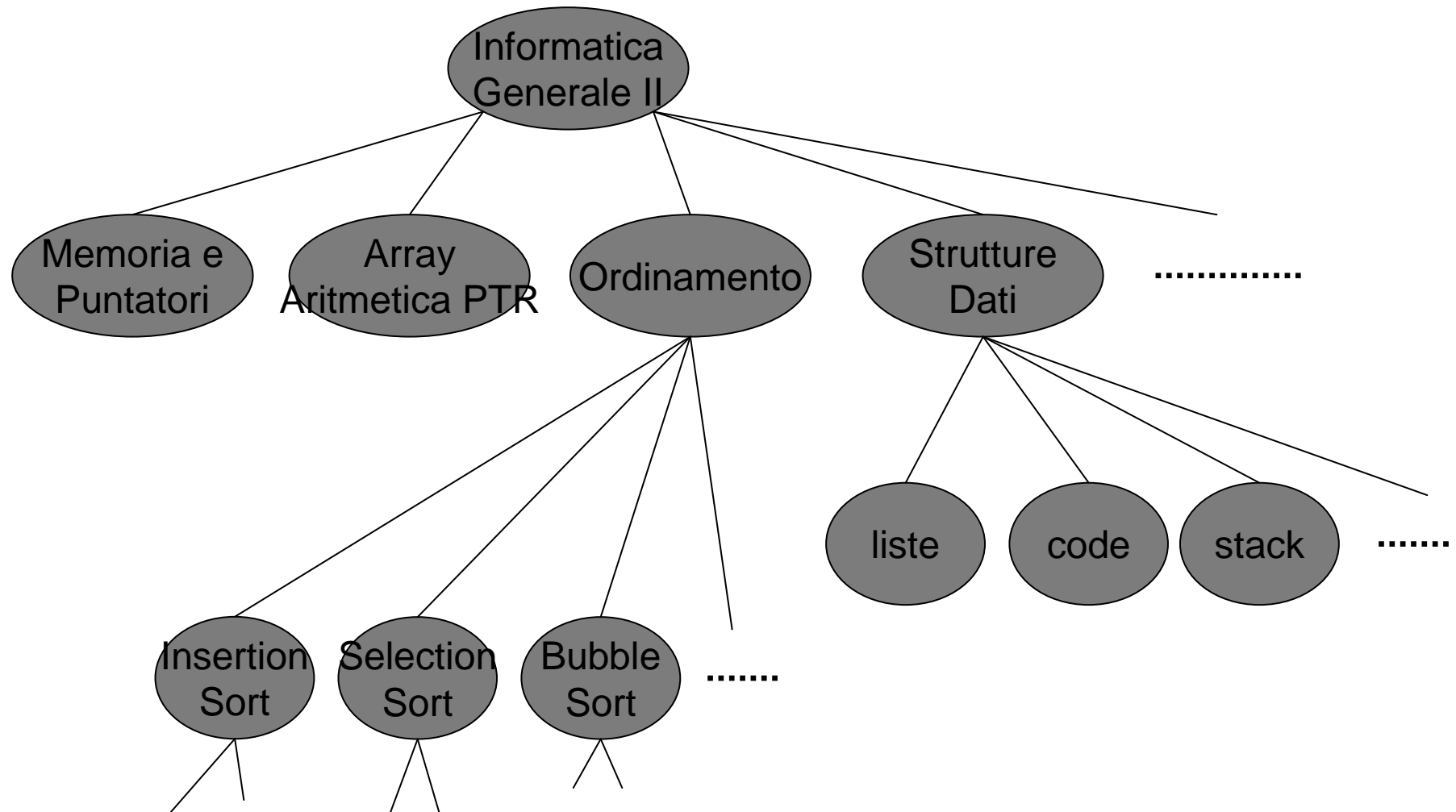
roveri@irst.itc.it

Alberi

Alberi

- Gli alberi sono una struttura matematica che gioca un ruolo molto importante nella progettazione e nell'analisi di algoritmi:
 - Gli alberi sono spesso utilizzati per descrivere proprietà dinamiche degli algoritmi.
 - Spesso utilizziamo strutture dati che rappresentano implementazioni concrete di alberi.
- Questo tipo di ADT lo incontriamo nella vita di tutti i giorni:
 - L'albero genealogico della propria famiglia (da cui deriva la maggior parte della terminologia impiegata nella teoria degli alberi).
 - Nei tornei sportivi
 - Per rappresentare l'organigramma di aziende.
 - Per rappresentare l'analisi sintattica dei linguaggi di programmazione.
 - Il file system di un sistema operativo.
 - gerarchie
 -

Alberi

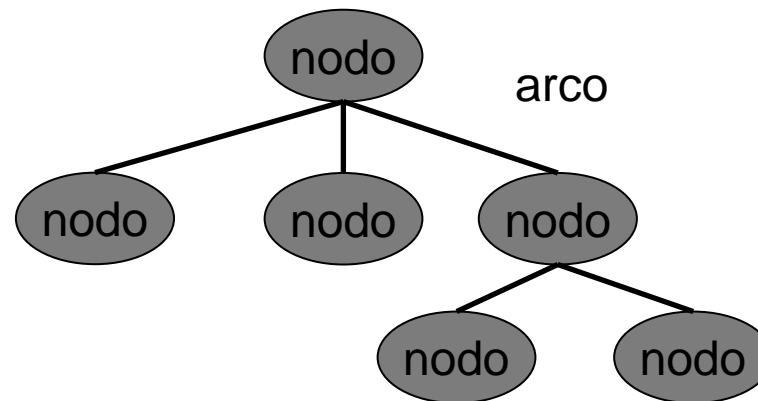


Tipi di Alberi

- Esistono diversi tipi di alberi, ed è importante distinguere tra modello astratto e modello concreto (ovvero tra modello matematico e implementazione).
- In ordine di generalità decrescente distinguiamo:
 - Alberi.
 - Alberi con radice.
 - Alberi ordinati.
 - Alberi M-ari e binari.

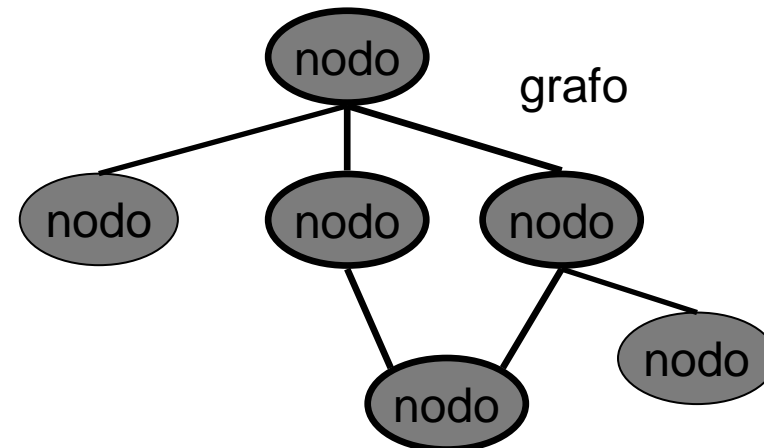
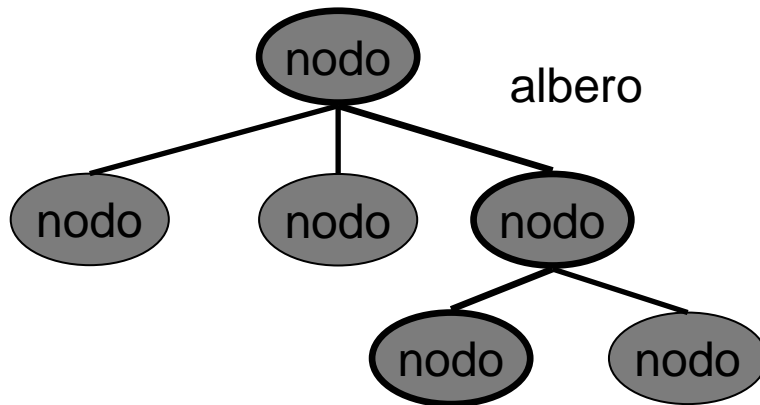
Albero

- 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.



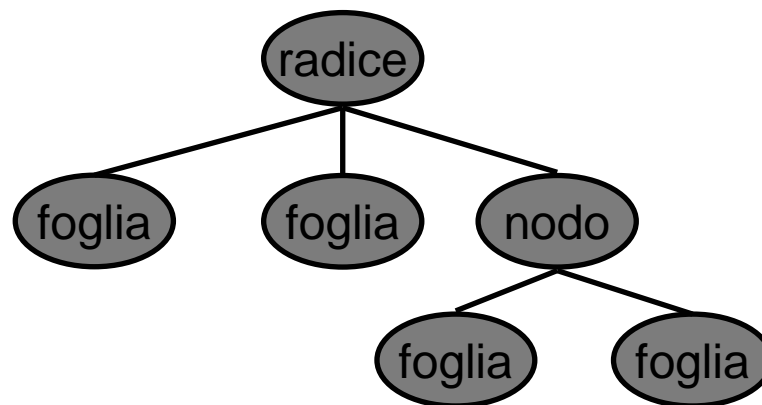
Cammino in un albero

- 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 nelle prossime lezioni).
- Un insieme di alberi disgiunto si chiama *foresta*.



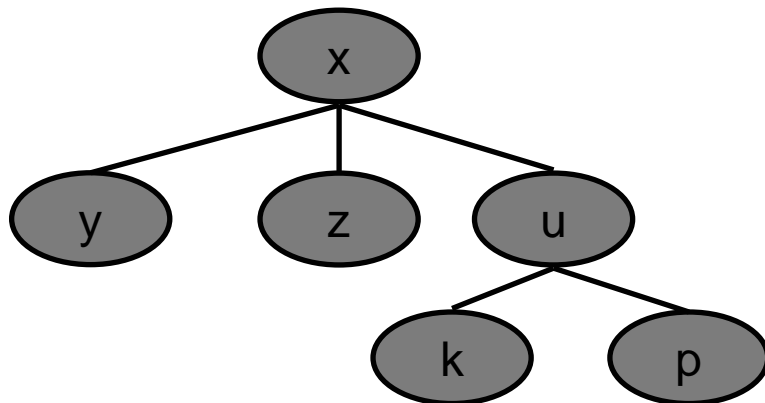
Alberi: definizioni

- 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

- Diciamo che un nodo n_1 è *sotto* n_2 (n_2 è *sopra* n_1) se n_2 è nel cammino tra n_1 e la radice.
- Ogni nodo tranne la radice ha un solo nodo sopra di se, detto nodo *padre*.
- I nodi appena sotto un dato nodo sono detti nodi *figli*.
- I nodi senza figli vengono detti *foglie*.



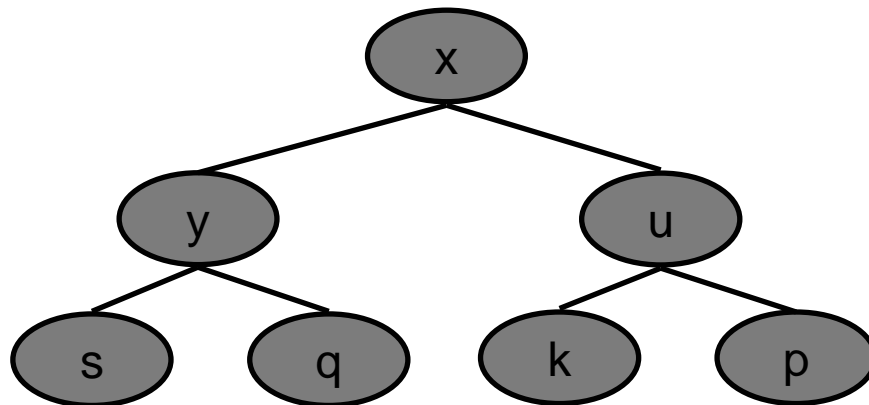
- x è sopra y, z, u
- k e p sono sotto u
- x è padre di y, z, u
- y, z, u sono figli di x
- y, z, k, p sono foglie

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 esattamente N .

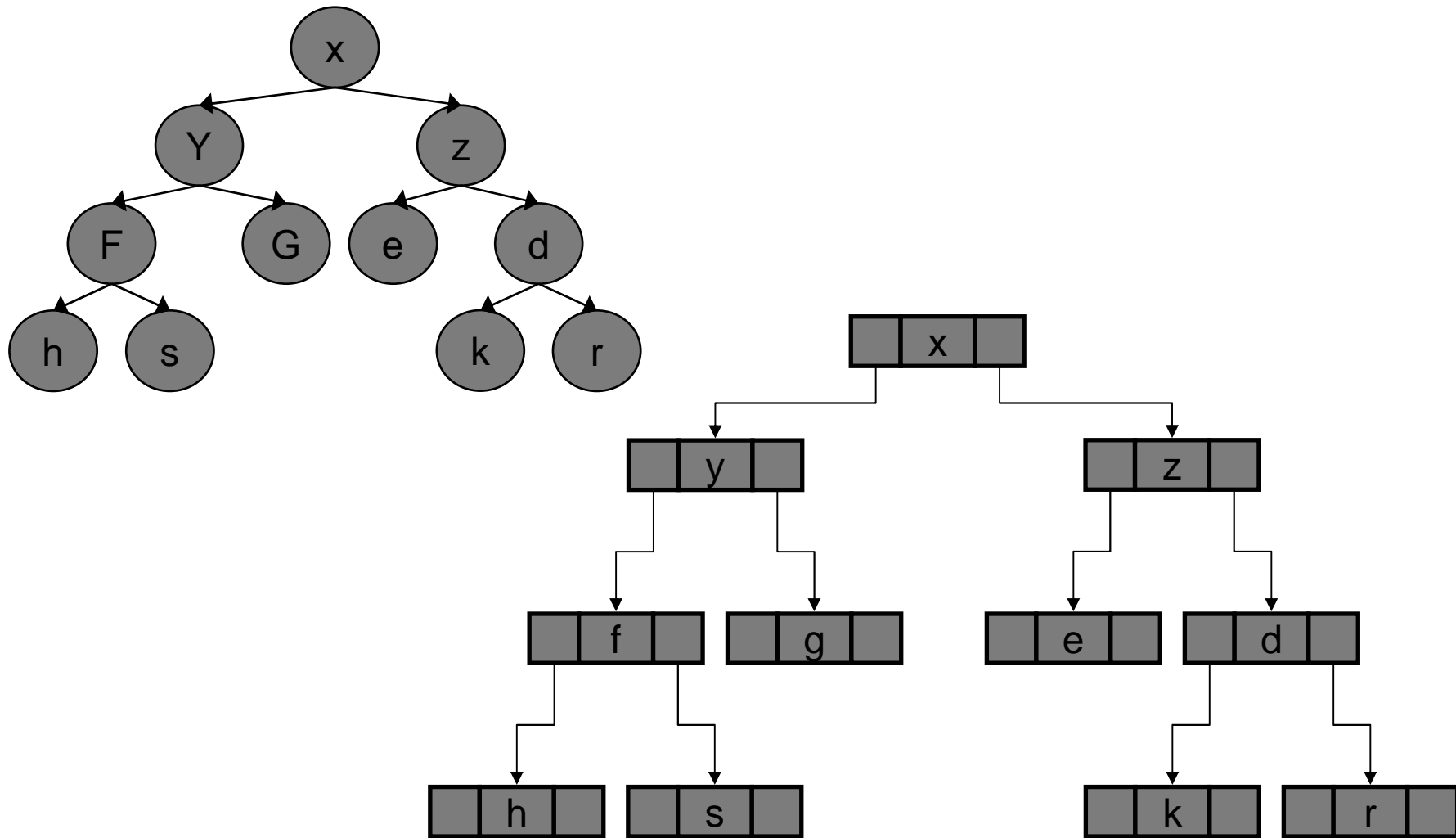
Alberi binari

- 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*



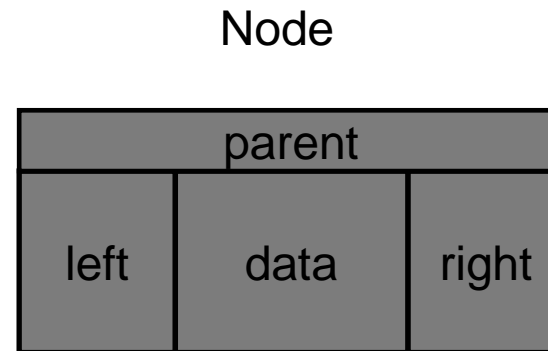
- y figlio sinistro di x
- u figlio destro di x
- k figlio sinistro di u
- p figlio destro di u

Rappresentazione di alberi binari



Rappresentazione di alberi binari

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



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

Algoritmi di attraversamento di alberi

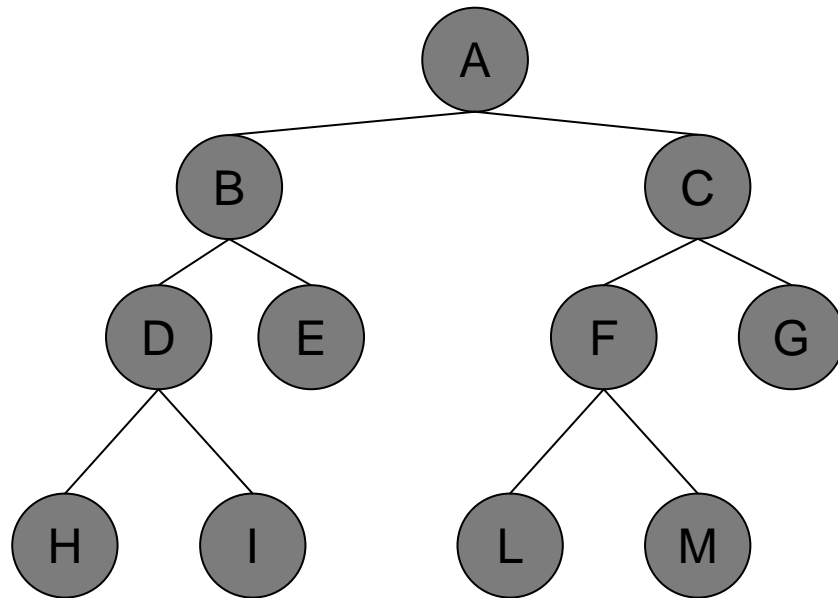
- Problema: dato un puntatore ad un nodo di un albero, scandire in modo sistematico ogni nodo dell'albero una ed una sola volta.
- Soluzione:

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

Algoritmi di attraversamento di alberi

- Abbiamo diversi *modi* per attraversare un albero:
 - Preordine (*preorder*): visitiamo prima il nodo e poi i sottoalberi di sinistra e di destra.
 - Inordine (*inorder*): visitiamo prima il sottoalbero di sinistra, poi il nodo, ed infine il sottoalbero di destra.
 - Postordine (*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

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


Algoritmi ricorsivi di attraversamento di alberi binari

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

Algoritmi non ricorsivi di attraversamento di alberi binari

- Gli algoritmi ricorsivi sono molto “intuitivi” per lavorare su strutture ricorsive.
- Però a volte sono poco efficienti a causa dell'eccessivo uso dello stack che necessitano (variabili locali, punto di ritorno, ...).
- Per sopperire a questi problemi, si possono pensare algoritmi iterativi che possono fare un uso esplicito di uno stack (memorizzo solo quello che mi serve).
- Nel seguito vedremo come scrivere una versione iterativa dell'algoritmo “traverse” ricorsivo che abbiamo visto prima.

Algoritmi non ricorsivi di attraversamento di alberi binari

```
void traverse(Node * l, void visit(Node *)) {  
    ts = new tree_stack();  
    tree_push(ts, l);  
    while( ! is_empty(ts) ) {  
        l = tree_pop(ts);  
        visit(l);  
        if ( l->left != NULL) tree_push(ts, l->left);  
        if ( l->right != NULL) tree_push(ts, l->right);  
    }  
}
```

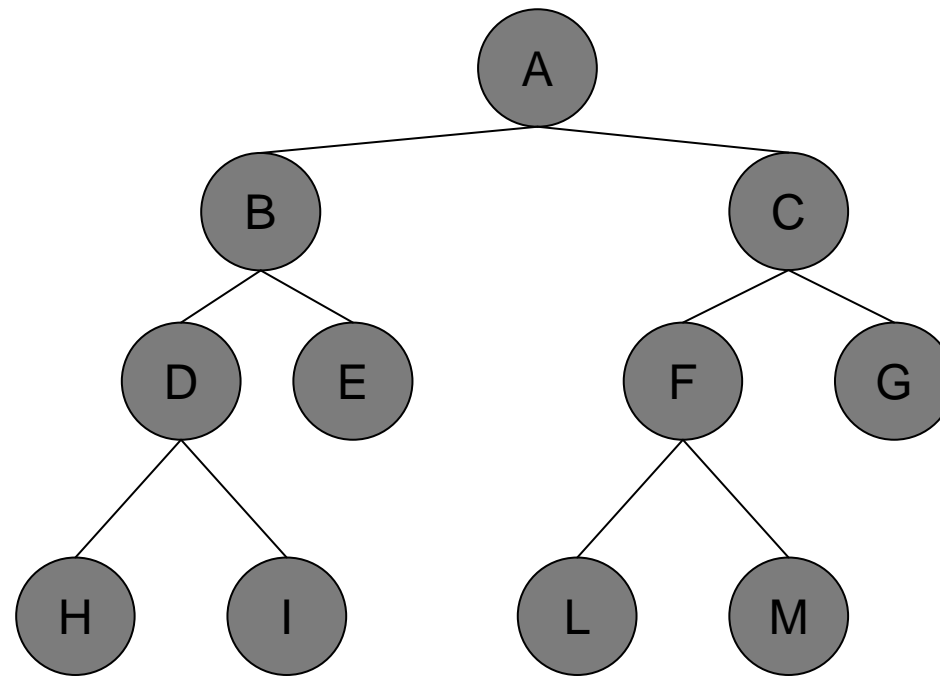
Considerazioni sull'algorithmo “traverse” iterativo

- Lo schema proposto è uno schema concettuale. Gli algoritmi preorder, inorder, postorder iterativi si possono ottenere dallo schema proposto via semplificazioni.
 - Nell'attraversamento preorder non c'è bisogno di effettuare la push sui nodi che visitiamo.
- **Esercizio:**
 - Provare a scrivere gli algoritmi preorder, postorder, inorder non ricorsivi a partire dallo schema concettuale proposto.
 - Provarli su un albero di prova e verificare che i risultati prodotti dalla versione ricorsiva ed iterativa coincidono.

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 quest' 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

Attraversamento *level-order*

- Domanda: è possibile ottenere l'algoritmo *level-order* dall'algoritmo "traverse" iterativo precedente?
- Risposta:
 - Si sostituendo lo stack con una coda.

Attraversamento *level-order*

```
void level-order(Node * l, void visit(Node *)) {  
    ts = new tree_queue();  
    tree_put(ts, l);  
    while( ! is_empty(ts) ) {  
        l = tree_get(ts);  
        visit(l);  
        if ( l->left != NULL) tree_put(ts, l->left);  
        if ( l->right != NULL) tree_put(ts, l->right);  
    }  
}
```

Algoritmi di utilità su alberi

- Molti dei problemi che operano sugli alberi ammettono soluzioni:
 - ricorsive che ne sfruttano la struttura ricorsiva.
 - del tipo *divide et impera* che generalizzano gli algoritmi di attraversamento.
 - Analizziamo un albero partendo dalla radice, per poi esaminare (ricorsivamente) i suoi sottoalberi.
 - Possiamo eseguire calcoli, prima dopo o fra le chiamate ricorsive.

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.
 - Stampare un albero.
 - Disegnare un albero.
- Nel seguito vedremo algoritmi atti a risolvere questi problemi.

Calcolo numero nodi di un albero

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

```
int count( Node * l ) {  
    if ( l == NULL ) return 0;  
    return count(l->left) + count(l->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 pari ad 1 più il livello del padre, intendendo che la radice ha livello 0. L'altezza di un albero è pari al massimo tra i livelli di tutti i suoi nodi.

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

Stampa di un albero

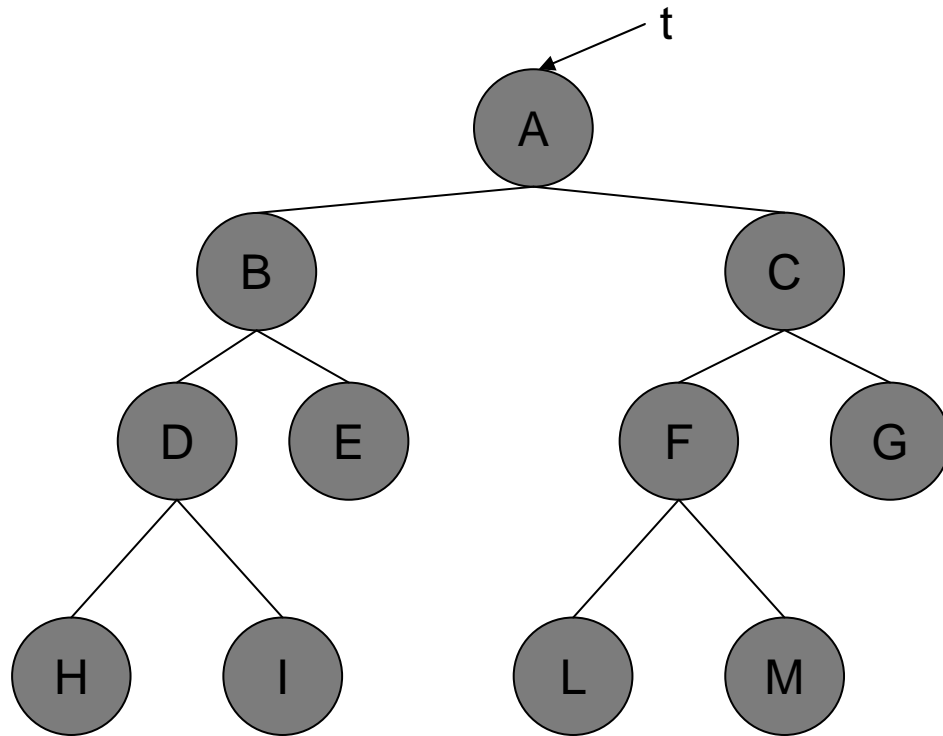
- Una funzione molto utile quando scriviamo programmi che elaborano alberi è quella che stampa o disegna l'albero.
- L'approccio standard consiste nel tenere traccia dell'altezza di un nodo per stamparlo.

Stampa di un albero

```
void printnode(char x, int h) {  
    for( int i = 0; i < h; i++) cout << "  ";  
    cout << x << endl;  
}
```

```
void draw(Node * l, int h) {  
    if (l == NULL) { printnode('*', h); return;}  
    draw(l->right, h +1);  
    printnode(l->data, h);  
    draw(l->left, h+1);  
}
```

Stampa di un albero



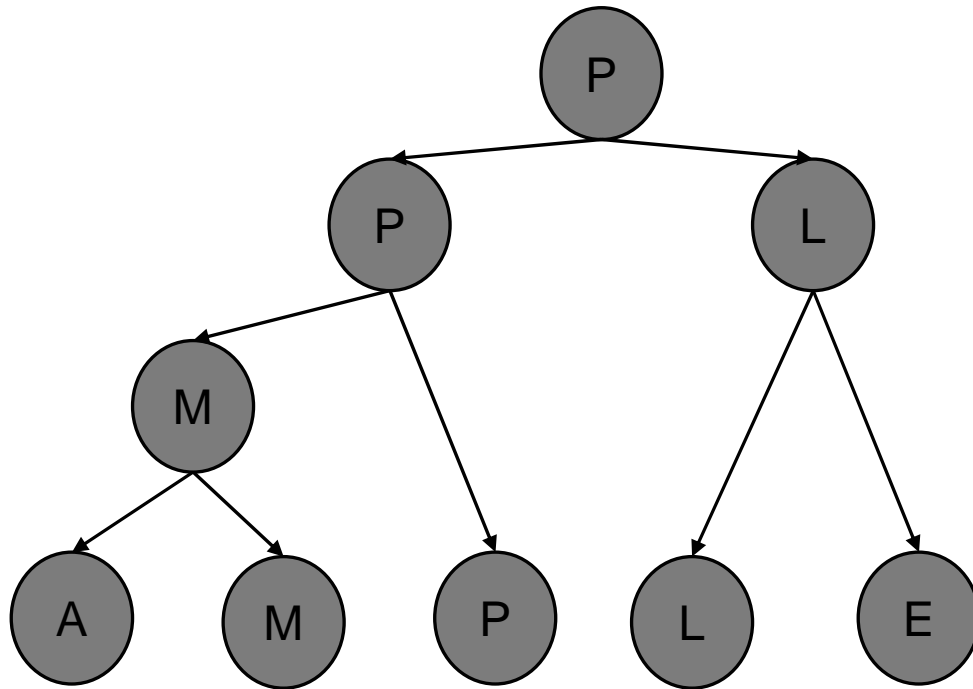
```
draw(t, 0);
```

```
      *
    G  *
  C    *
      *
    M  *
      *
    F  *
      *
    L  *
  A    *
      *
    E  *
      *
  B    *
      *
    I  *
      *
    D  *
      *
    H  *
```


Esempio uso alberi: costruzione di un torneo

- Supponiamo di voler costruire un *torneo*: ovvero un albero binario in cui il contenuto di ogni nodo è una copia del maggiore dei suoi figli. In particolare il nodo alla radice è il *massimo* del torneo.
- I dati sulle foglie sono I dati a disposizione nel problema, mentre il resto dell'albero è una struttura dati che ci consente di determinare il massimo in modo efficiente.
 - I dati di partenza sono memorizzati in un array $a[l], \dots, a[r]$.
 - Ricorsivamente dividiamo l'array in due parti $a[l], \dots, a[m]$ e $a[m+1], \dots, a[r]$ e costruiamo i tornei per queste due parti.
 - Costruiamo il torneo per l'intero array, impostando i link di un nuovo nodo in modo che facciano riferimento ai due tornei calcolati ricorsivamente, copianod poi in tal enodo il maggiore tra i due dati che troviamo nelle radici di questi due tornei.

Esempio uso alberi: costruzione di un torneo



A	M	P	L	E
---	---	---	---	---

Esempio uso alberi: costruzione di un torneo

```
Node * torneo( int a[], int l, int r) {  
    int m = (l + r) / 2;  
    Node * x = new Node(a[m]);  
    if (l == r) return x;  
    x->left = torneo(a, l, m);  
    x->right = torneo(a, m+1, r);  
    int u = x->left->data;  
    int v = x->right->data;  
    if (u > v)  
        x->data = u;  
    else  
        x->data = v;  
    return x;  
}
```

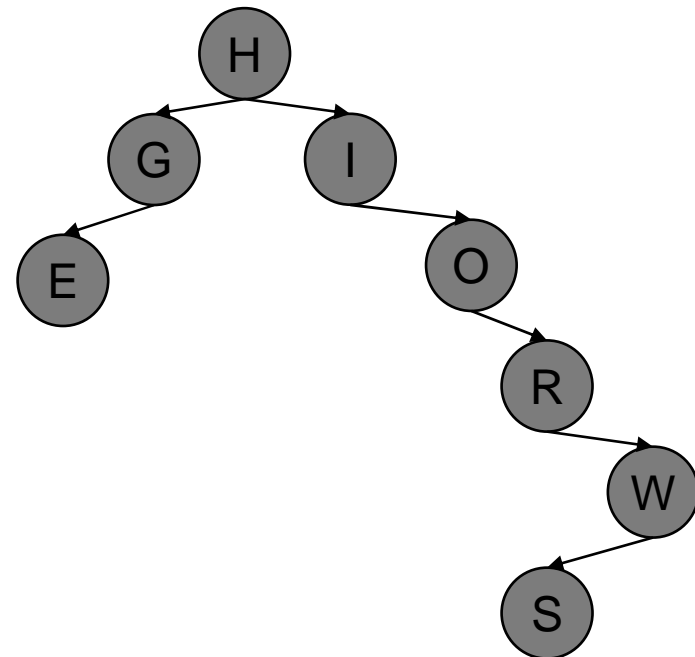
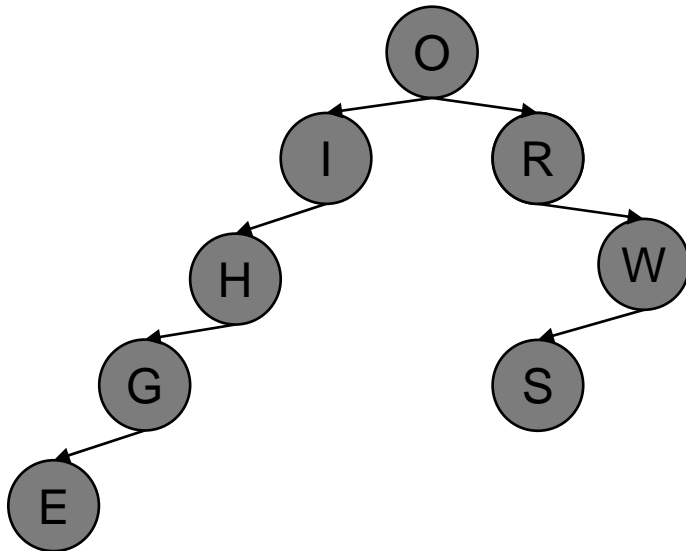
Esercizio

- 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.

Alberi binari di ricerca BST

Alberi binari di ricerca

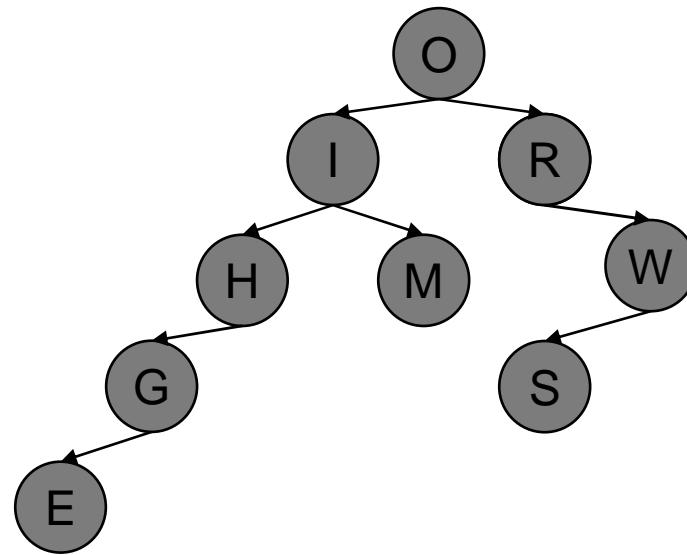
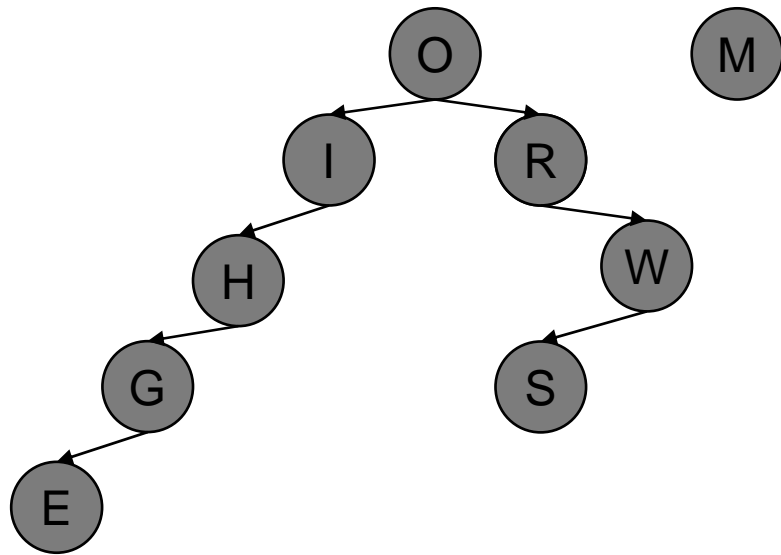
- Definizione: Un *albero binario di ricerca* (*binary search tree* (*BST*)) è un albero binario dove la *chiave* (dato) memorizzata in ciascun nodo è maggiore o uguale alla chiave di tutti i nodi del sottoalbero sinistro di quel nodo, e minore o uguale alle chiavi di tutti i nodi del sottoalbero destro 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.
- Nel seguito vedremo queste operazioni nel dettaglio, analizzandone le prestazioni.

Inserimento in un BST



Inserimento in un BST

- Versione ricorsiva secondo la definizione di BST.

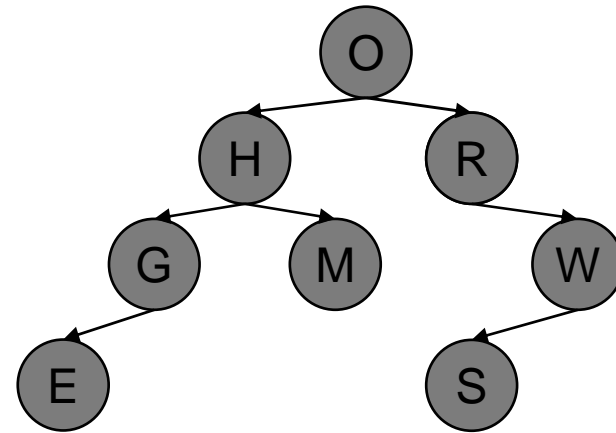
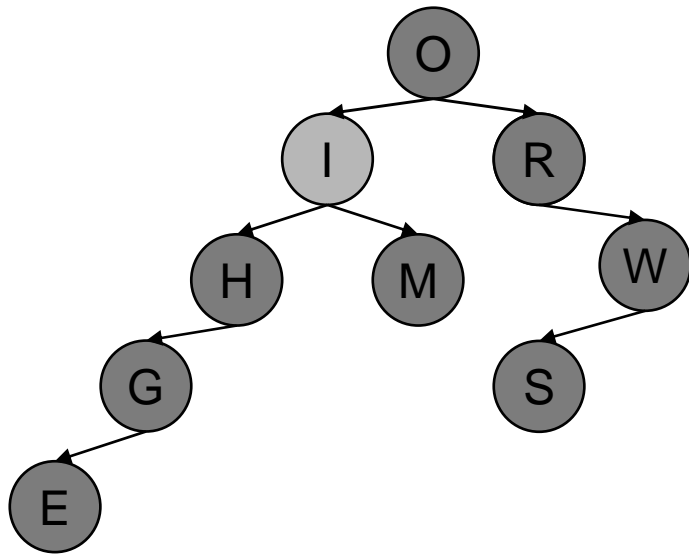
```
void insert(Node *& s, char x) {  
    if (s == NULL) {  
        s = new Node(x);  
        return;  
    }  
    if (x < s->data)  
        insert(s->left, x);  
    else  
        insert(s->right, x);  
}
```

Inserimento in un BST

- Versione iterativa

```
void Insert(Node * & s, char x) {  
    Node v = new Node(x);  
    Node * y = NULL; Node * x = s;  
    while(x != NULL) {  
        y = x;  
        if (v->data < x->data) x = x->left;  
        else x = x->right;  
    }  
    if (y == NULL) s = v;  
    else if (v->data < y->data) y->left = v;  
    else y->right = v;  
}
```

Cancellazione in un BST



Cancellazione in un BST

- Versione ricorsiva

```
void remove(Node * & h, char v) {  
    if (h == NULL) return;  
    char w = h->data;  
    if (v < w) remove(h->left, v);  
    if (v > w) remove(h->right, v);  
    if (v == w) {  
        Node * t = h;  
        h = joinLR(h->left, h->right);  
        delete t;  
    }  
}
```

Cancellazione in un BST

```
// Unisce sottolaberi sinistro e destro
Node * joinLR(Node * a, Node * b) {
    if (b == NULL) return a;
    partR(b, '0'); b->left = a;
    return b;
}
```

```
// pone il k-esimo nodo più piccolo del BST alla radice
void partR(Node * & h, char k) {
    char t = (h->left == NULL) ? '0' : h->left->data;
    if (t > k) { partR(h->left, k); rotR(h); }
    if (t < k) { partR(h->right, k-t-1); rotL(h); }
}
```

Cancellazione in un BST

```
// rotazione a destra
void rotR(Node * & h) {
    Node * x = h->left;
    h->left = x->right;
    x->right = h;
    h = x;
}
```

```
// rotazione a sinistra
void rotL(Node * & h) {
    Node * x = h->right;
    h->right = x->left;
    x->left = h;
    h = x;
}
```

Cancellazione in un BST

- La versione ricorsiva vista, prima può essere resa iterativa sfruttando il puntatore al nodo padre.

```
Node * Tree_Delete(Node * &r, Node * z) {
    Node * y;
    if ((z->left == NULL) || (z->right == NULL)) y = z;
    else y = Tree_Successor(z);
    if (y->left != NULL) x = y->left;
    else x = y->right;
    if (x != NULL) x->parent = y->parent;
    if (y->parent == NULL)
        r = x;
    else if (y == y->parent->left)
        y->parent->left = x;
    else
        y->parent->right = x;
    if (y != z) z->data = y->data;
    return y;
}
```

A differenza del caso precedente qui passiamo direttamente il nodo che vogliamo eliminare.

Unione di due BST

- Altra operazione importante per gli alberi binari è l'unione di due BST.
 - Se uno dei due sottoalberi è vuoto, il risultato dell'unione è l'altro sottoalbero.
 - Altrimenti, combiniamo i due BST eleggendo (arbitrariamente) la radice del primo BST a radice del nuovo BST, e inserendo alla radice del secondo BST quella radice, e combinando quindi (ricorsivamente) la coppia dei due sottoalberi di sinistra e di destra.

Unione di due BST

```
Node * join(Node * a, Node * b) {  
    if (b == NULL) return a;  
    if (a == NULL) return b;  
    insert(b, a->data);  
    b->left = join(a->left, b->left);  
    b->right = join(a->right, b->right);  
    delete a;  
    return b;  
}
```

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;  
}
```

Successore di un nodo in un BST

- Spesso è importante determinare il successore nell'ordinamento determinato da una visita inorder dell'albero.
- Se tutte le chiavi sono distinte, il successore di un nodo X, è il nodo con la più piccola chiave minore della chiave di X.

```
Node * Tree_Successor(Node * x) {  
    if (x->right != NULL)  
        return Tree_Min(x->right);  
    Node * y = x->parent;  
    while( (y != NULL) && (x == y->right) ) {  
        x = y;  
        y = y->parent;  
    }  
    return y;  
}
```

Notare che in questa realizzazione, sfruttiamo il fatto che nel nodo memorizziamo anche il puntatore al padre (i.e., `y->parent`).

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;  
}
```

Performances degli algoritmi analizzati sui BST

- Mediamente le operazioni sui BST sono proporzionali all'altezza dell'albero.
- L'algoritmo di ricerca sull'albero effettua in media h confronti nel caso che l'albero sia abbastanza "bilanciato".
- Nel caso in cui l'albero binario degeneri in una lista, si ha il caso peggiore, infatti occorre scorrere tutta la lista per trovare l'elemento.

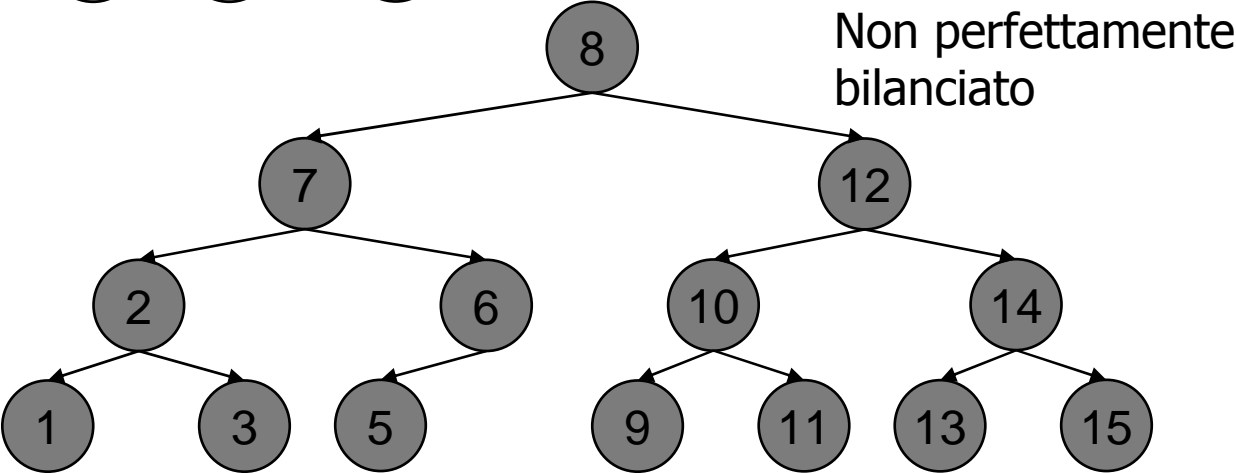
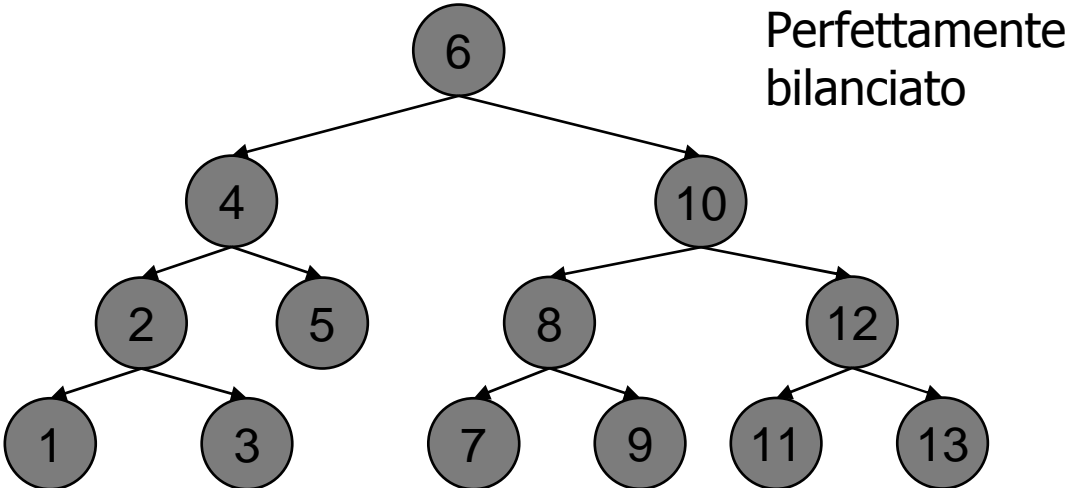
Esercizi

- Che cosa succede se effettuo una visita inorder di un BST?
- Add some exercise.

Alberi binari bilanciati

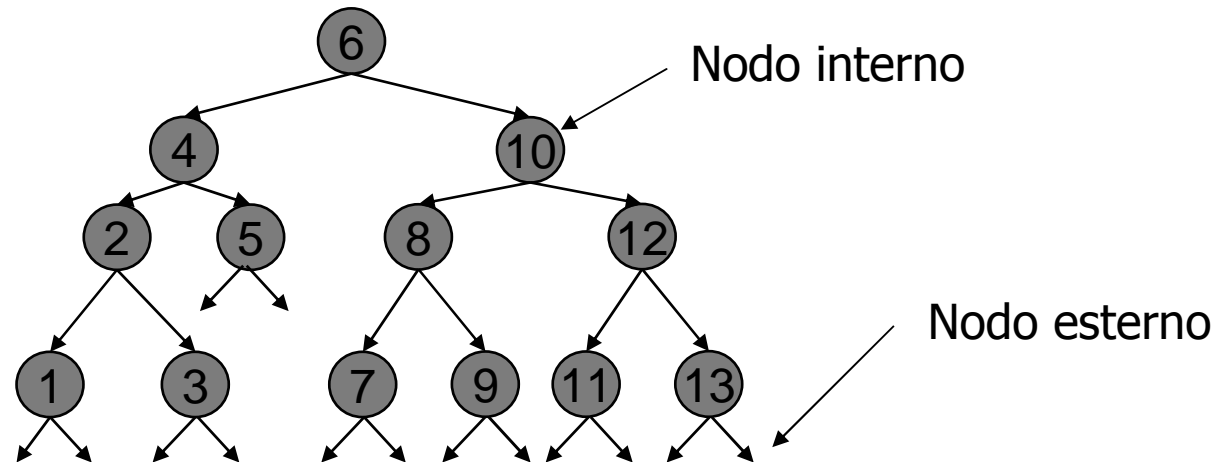
- Gli algoritmi visti operanti su alberi binari si comportano in modo soddisfacente in una stragrande maggioranza di casi, anche se in presenza dei casi peggiori forniscono prestazioni non sempre soddisfacenti.
- Sfortunatamente, il caso peggiore ha buona probabilità di verificarsi nella pratica (file già ordinati in senso inverso, alternanze di chiavi grandi e piccole,...).
- La situazione ideale è quella in cui l'albero BST è perfettamente bilanciato.

Esempio albero BST perfettamente bilanciato



Albero BST bilanciato

- Definizione: un nodo di un albero con almeno un figlio è detto *interno*. Altrimenti il nodo è detto *esterno*.



- Definizione: un albero binario di ricerca di N nodi è bilanciato se ha altezza non maggiore di $\lfloor \log(N) \rfloor$ e $N+1$ nodi esterni.

Bilanciamento di un BST

- Una soluzione al problema di migliorare le performances degli algoritmi sugli alberi, consiste nel periodicamente *bilanciare* l'albero.
- Un algoritmo lineare per bilanciare un BST consiste nel partizionare un BST mettendo la mediana alla radice, e ripetendo il procedimento per i sottoalberi.

```
void Balance(Node * & h) {  
    if ((h == NULL) || (h->data == 1)) return;  
    partR(h, h->data/2);  
    Balance(h->left);  
    Balance(h->right);  
} // assumiamo 1 valore minimo.
```