

Unità didattica: alberi binari di ricerca (search binary trees)

[5-T]

Titolo: Definizioni ed algoritmi di gestione

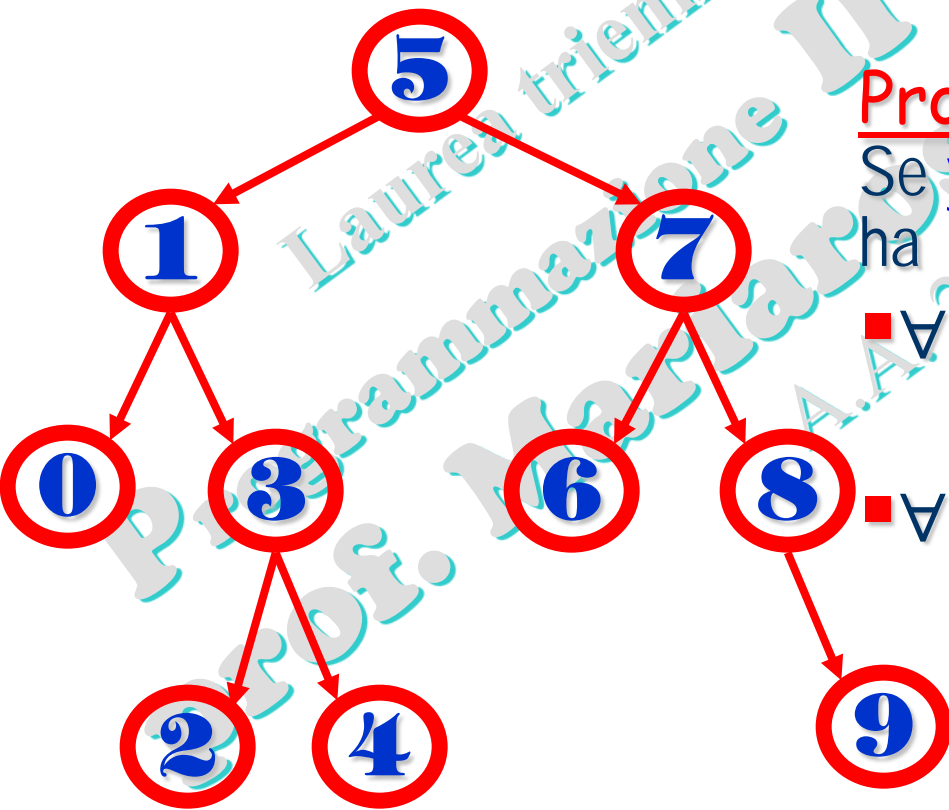
Argomenti trattati:

- ✓ Definizione e proprietà di un albero binario di ricerca
- ✓ Algoritmo per la costruzione e visita di un albero binario di ricerca
- ✓ Algoritmo ricorsivo di ricerca binaria su un albero binario di ricerca

Prerequisiti richiesti: alberi binari

Alberi binari ordinati secondo una chiave (o ABR - Alberi Binari di Ricerca - Binary Search Trees - BST)

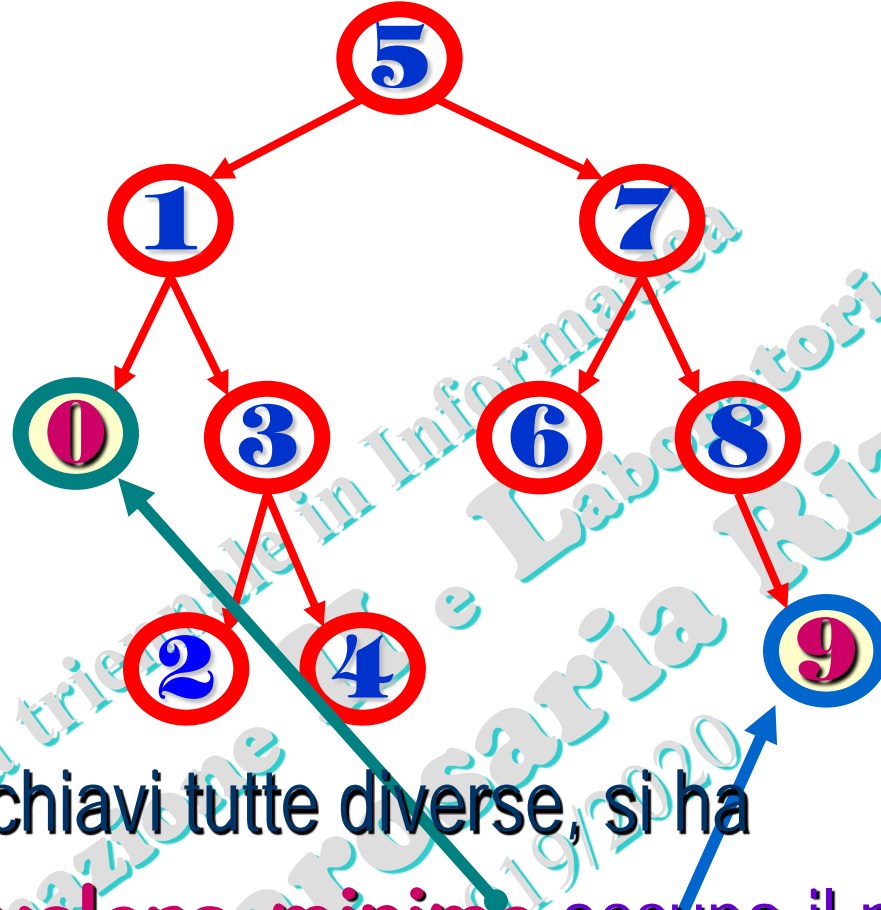
- È un particolare albero binario in cui l'**inorder visit** (visita in **ordine simmetrico**) consente di accedere alle informazioni in ordine crescente di chiave (**key(nodo)**);
- È un particolare albero binario dove la **ricerca binaria** risulta particolarmente semplice;



Proprietà di un BST:

Se y è un qualsiasi nodo dell'albero si ha

- \forall nodo x del sottoalbero sinistro di y
 $key(x) \leq key(y)$
- \forall nodo z del sottoalbero destro di y
 $key(y) \leq key(z)$

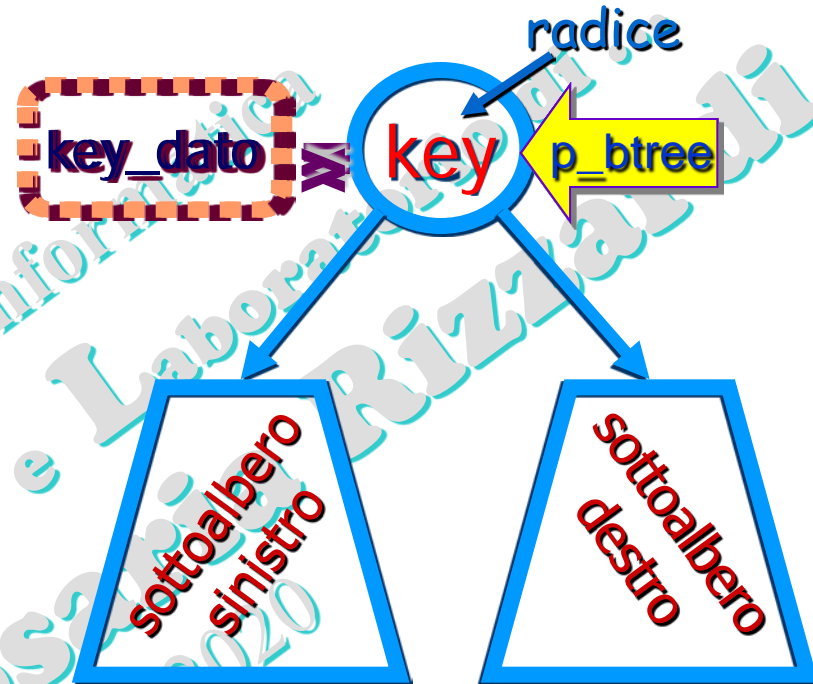


In un **ABR** con chiavi tutte diverse, si ha

- ✓ la chiave di **valore minimo** occupa il nodo foglia del sottoalbero più a sinistra (cioè la prima foglia che si incontra con la visita inorder);
- ✓ la chiave di **valore massimo** occupa il nodo foglia del sottoalbero più a destra (cioè l'ultima foglia che si incontra con la visita inorder);

Algoritmo ricorsivo di ricerca binaria

come tutti gli algoritmi della classe "divide et impera" anche la ricerca binaria si esprime molto semplicemente in notazione ricorsiva



```
char function b_search_ric(KeyType key_dato, b_tree* p_btree)
if ( p_btree == NULL )      return 0; /* cioè falso - non trovato! */
else
    if key_dato == p_btree->key return 1; /* cioè vero - trovato! */
    else
        if key_dato < p_btree->key /* continua la ricerca nel sottoalb. sx */
            return b_search_ric(key_dato, p_btree->pt_sx)
        else /* continua la ricerca nel sottoalb. dx */
            return b_search_ric(key_dato, p_btree->pt_dx)
```

La costruzione di un ABR può avvenire con un algoritmo ricorsivo tipo ricerca binaria

Input: N (numero nodi), n[] (campo chiave dei nodi)

Output: root (puntatore a BST)

```
typedef struct BSTree
{
    char key;
    struct BSTree *sx, *dx;
} BST;
```

inserisce nodo radice

```
main()
{
    ...
    BST *root=(BST*)malloc(sizeof(BST));
    pt->key = n[0];
    pt->dx = pt->sx = NULL;
    buildBST(N-1,n+1,root,root);
    ...
}
```

```
void buildBST(int N, char n[], BST *root, BST *pt)
{
    BST *NEW;
    if ( N <= 0 ) return;
    if (n[0] > pt->key)
        if (pt->dx != NULL)
            buildBST(N,n,root,pt->dx);
        else
        {
            NEW = (BST *)malloc(sizeof(BST));
            NEW->dx = NEW->sx = NULL;
            NEW->key = n[0];
            pt->dx = NEW;
            buildBST(N-1,n+1,root,root);
        }
    else /* a[0] <= pt->key */
        if (pt->sx != NULL)
            buildBST(N,n,root,pt->sx);
        else
        {
            NEW = (BST *)malloc(sizeof(BST));
            NEW->dx = NEW->sx = NULL;
            NEW->key = n[0];
            pt->sx = NEW;
            buildBST(N-1,n+1,root,root);
        }
}
```

caso base

inserisce nodo
come figlio dx

inserisce nodo
come figlio sx

ordine di arrivo

Esempio

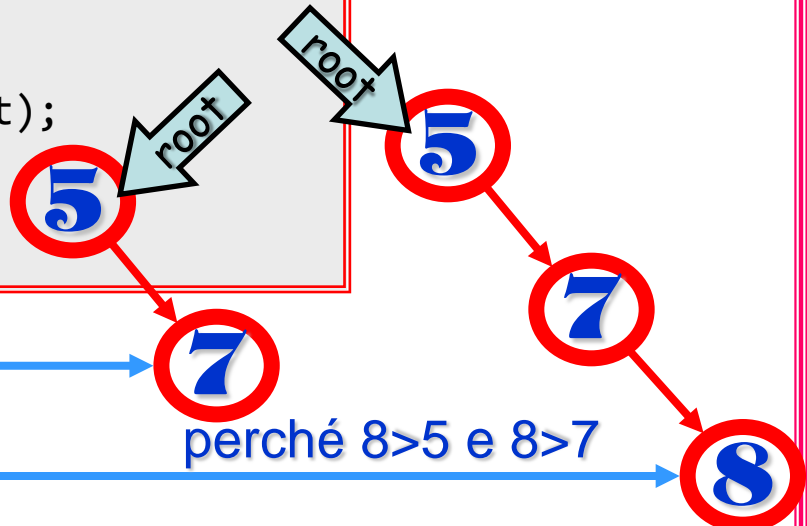
- 5
- 7
- 8
- 1
- 6
- 0
- 3
- 4
- 9
- 2

1°
2°
3°

```
main()
{
    ...
    BST *root=(BST*)malloc(sizeof(BST));
    pt->key = n[0];
    pt->dx = pt->sx = NULL;
    buildBST(N-1,n+1,root,root);
    ...
}
```

inserisce nodo radice

```
void buildBST(int N, char n[], BST *root, BST *pt)
{
    BST *NEW;
    if ( N <= 0 ) return;
    if (n[0] > pt->key)
        if (pt->dx != NULL)
            buildBST(N,n,root,pt->dx);
        else
        {
            NEW = (BST *)malloc(sizeof(BST));
            NEW->dx = NEW->sx = NULL;
            NEW->key = n[0];
            pt->dx = NEW;
            buildBST(N-1,n+1,root,root);
        }
    ...
}
```



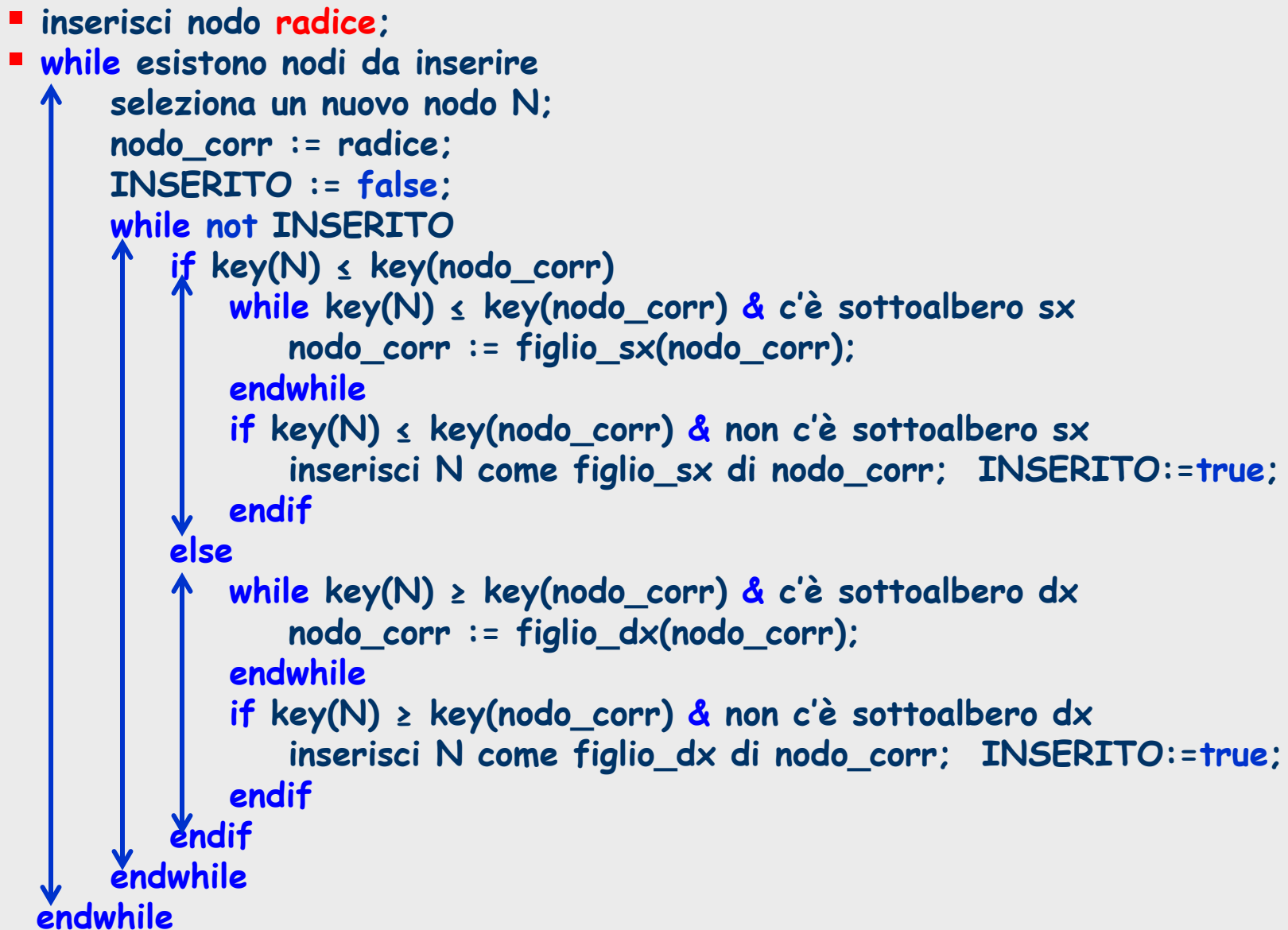
perché 7>5

perché 8>5 e 8>7

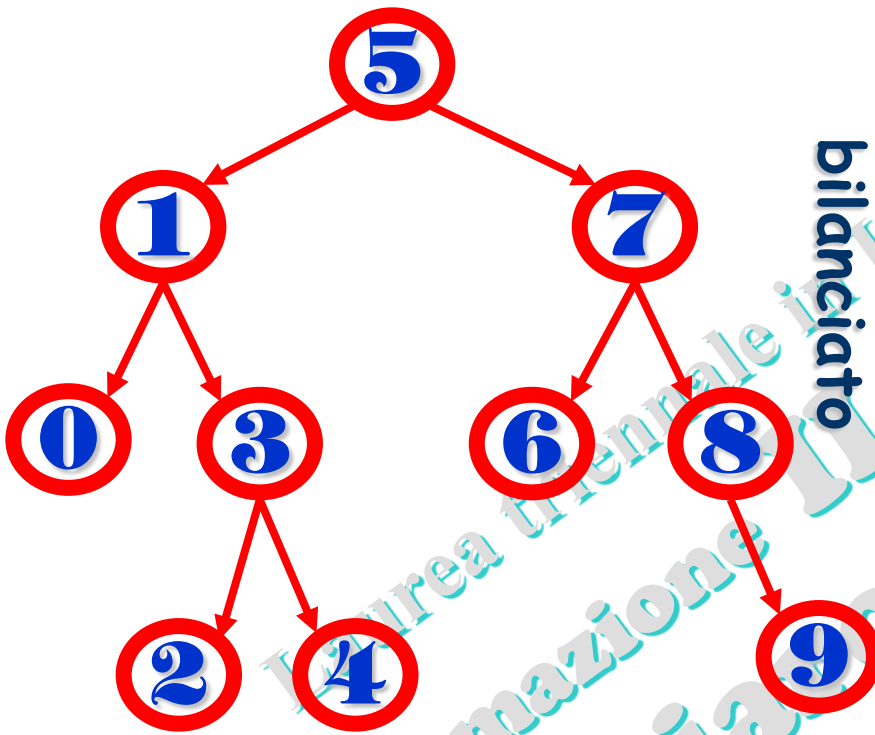
prof.

... cioè la costruzione di un ABR può avvenire con un algoritmo iterativo tipo ricerca binaria

```
■ inserisci nodo radice;  
■ while esistono nodi da inserire  
    seleziona un nuovo nodo N;  
    nodo_corr := radice;  
    INSERITO := false;  
    while not INSERITO  
        if key(N) ≤ key(nodo_corr)  
            while key(N) ≤ key(nodo_corr) & c'è sottoalbero sx  
                nodo_corr := figlio_sx(nodo_corr);  
            endwhile  
            if key(N) ≤ key(nodo_corr) & non c'è sottoalbero sx  
                inserisci N come figlio_sx di nodo_corr; INSERITO:=true;  
            endif  
        else  
            while key(N) ≥ key(nodo_corr) & c'è sottoalbero dx  
                nodo_corr := figlio_dx(nodo_corr);  
            endwhile  
            if key(N) ≥ key(nodo_corr) & non c'è sottoalbero dx  
                inserisci N come figlio_dx di nodo_corr; INSERITO:=true;  
            endif  
        endif  
    endwhile  
endwhile
```



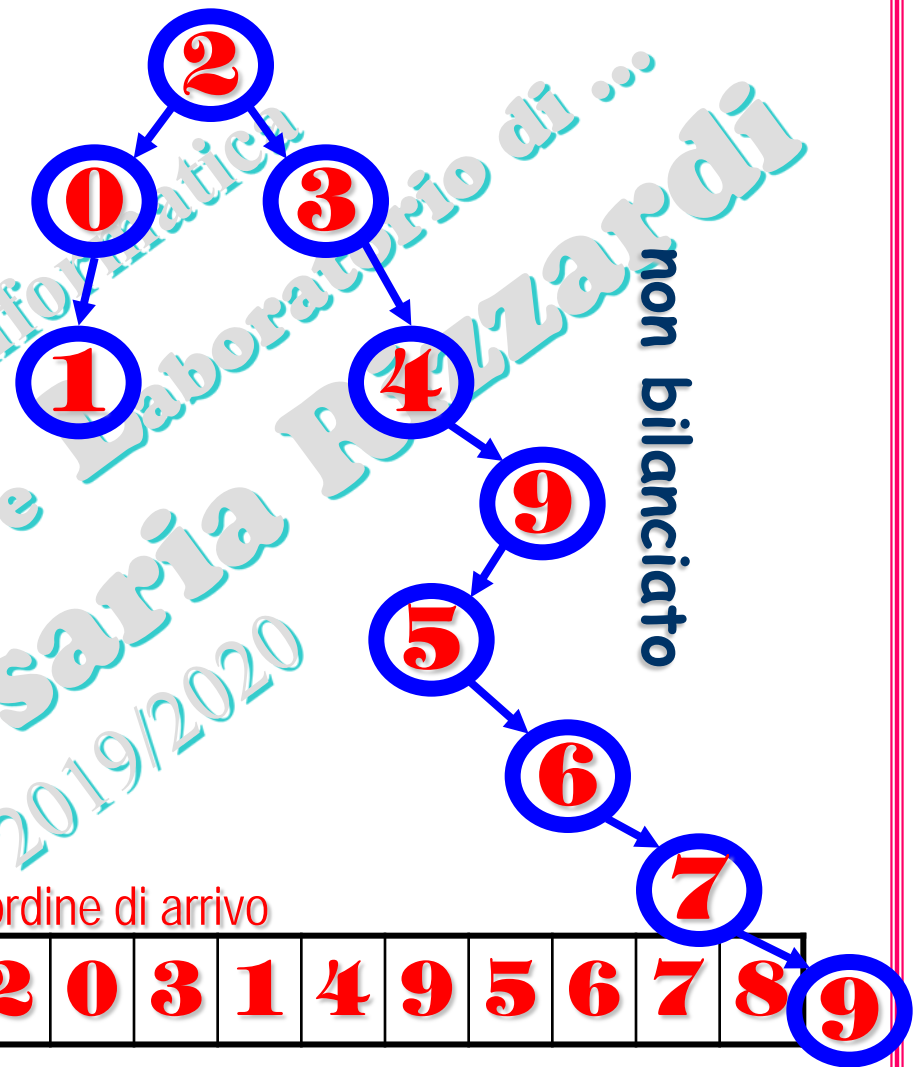
Il **bilanciamento** dell'albero binario di ricerca dipende dall'ordine (casuale) di arrivo dei nodi (*)



bilanciato

ordine di arrivo

5	7	8	1	6	0	3	4	9	2
---	---	---	---	---	---	---	---	---	---



non bilanciato

ordine di arrivo

2	0	3	1	4	9	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---	---

(*) L'operazione di **rotazione** di nodi negli **alberi Red-Black** consente il bilanciamento di un albero binario

Esercizi

1

Scrivere *function* C iterativa per la costruzione di un albero binario di ricerca implementato mediante array.

[liv. 2]

2

Scrivere *function* C iterativa per la costruzione di un albero binario di ricerca implementato mediante liste multiple.

[liv. 3]