Alberi

Gli alberi sono delle strutture dati **gerarchiche** i quali livelli sono dati da un rapporto *genitoriale*. Considerata la radice (il punto più ***in alto***), da essa discenderanno tutti i figli (*nipoti, pronipoti...*).

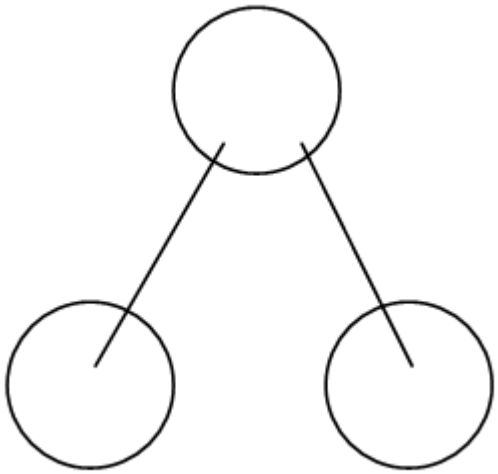
Essi possono essere visti come dei veri e propri alberi.

Il nodo

L'elemento costitutivo dell'albero è il nodo, le cui proprietà sono:

1. **l'arietà**, ovvero il numero di figli *possibili*;
2. **i figli**, ovvero un *elenco di nodi*;
3. **i parent**, ovvero un nodo *genitore*.

> un albero si definisce in base *al numero di figli che può avere* (quindi in base alla sua ***arietà***), infatti è definito **n-ario**.
>
> > Nel nostro caso parliamo di *alberi **binari***. (quindi con al più 2 figli, uno destro e uno sinistro).



I due nodi condividono lo stesso genitore e la stessa **profondità**, ovvero il ***livello del nodo***. (La radice - *o root* - ha profondità 0, i figli della radice hanno profondità 1...) Il numero di livelli che abbiamo in un albero è chiamato "**altezza** dell'albero". I nodi che non hanno figli si chiamano **foglie**.

> Il numero di nodi in un albero è **sempre** pari (o minore) a
>
> $$n = 2^{a}ltezzaAlbero - 1$$
>
> L'altezza dell'albero è data da
>
> $$ciel(log_2 n) = h$$
>
> > il "*ciel*" si usa per **arrotondare per eccesso**
>
> Un albero binario è completo se e solo se sono presenti tot nodi:
>
> $$2^h - 1$$
>
> > *(per h si intende l'altezza dell'albero)*
>
> Un modo alternativo per calcolare i nodi è:

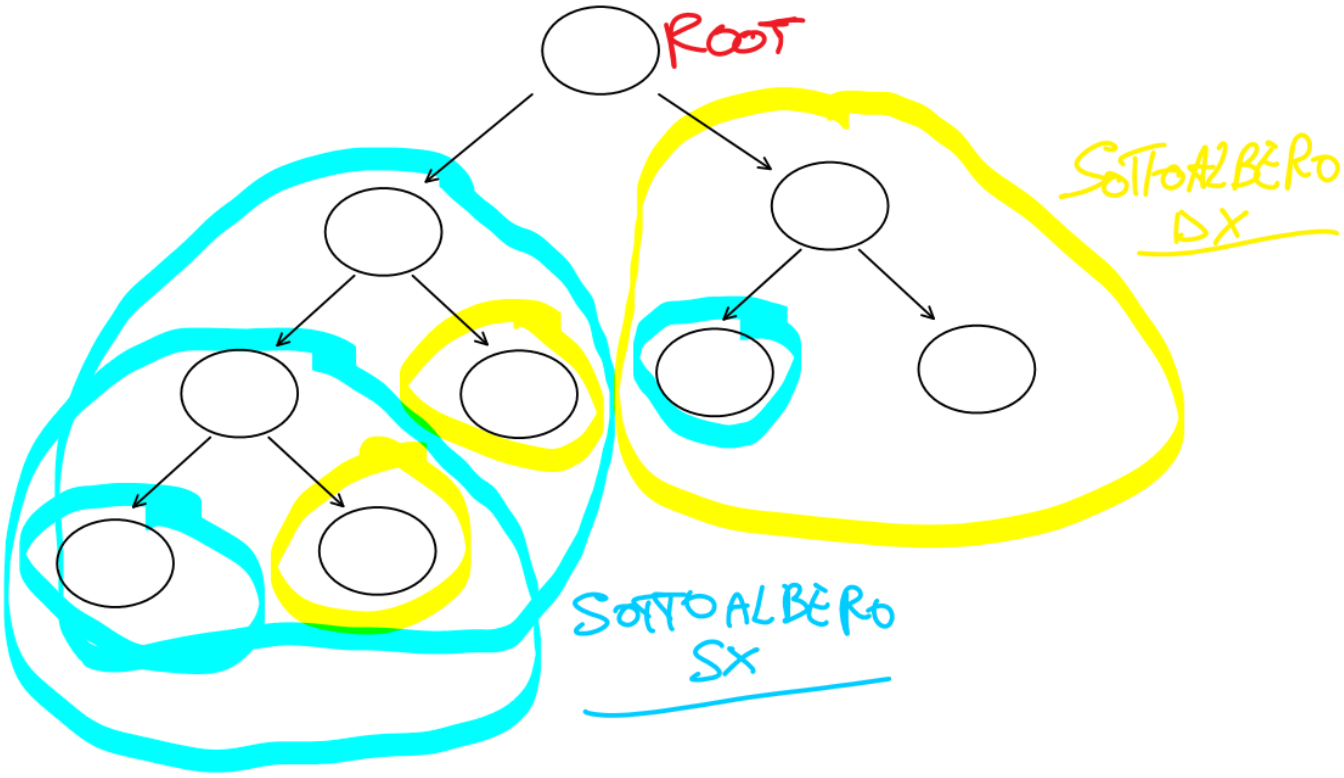
$$\sum_{n=0}^{k} 2^n$$

(*per k si intende il livello*)

Albero bilanciato

Un albero si dice *bilanciato* se per ogni nodo la differenza dell'altezza fra sottoalbero sinistro e sottoalbero destro è al più pari ad uno.

> Il sottoalbero è dato dai figli, dai nipoti, dai pronipoti...



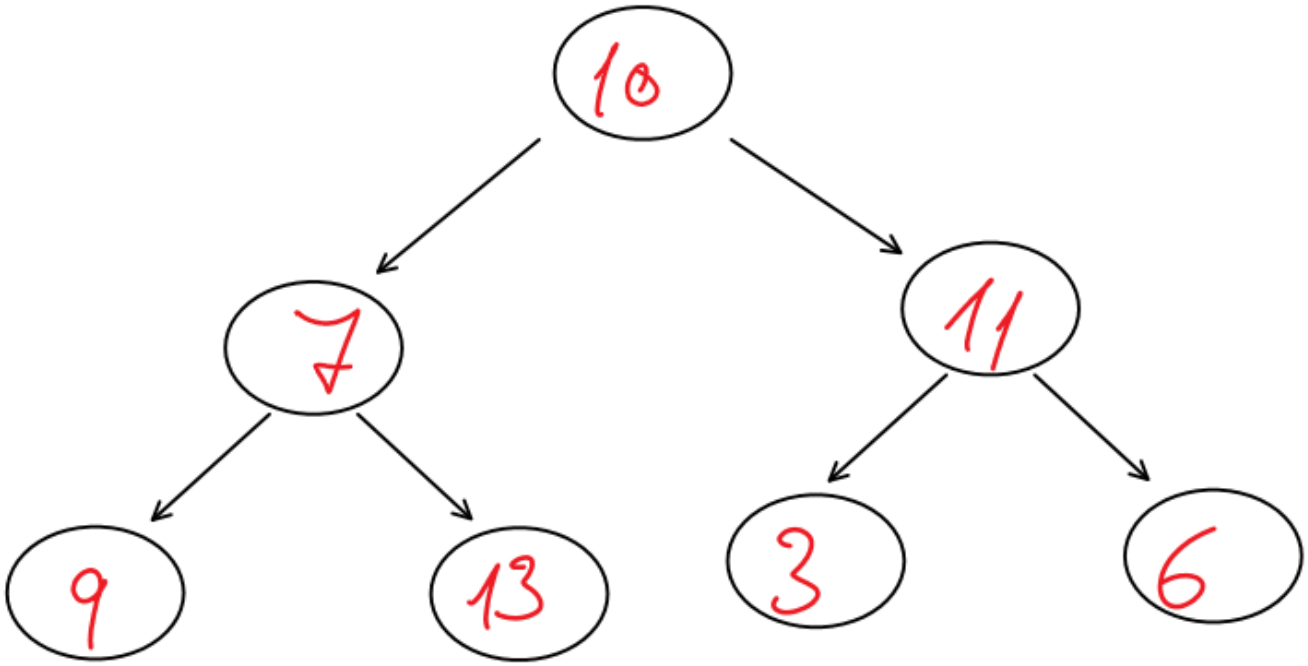
L'albero soprastante è bilanciato, ma **non completo**. Il sottoalbero a sinistra ha altezza pari a 3, mentre il sottoalbero a destra ha altezza pari a 2.

Alberi binari di ricerca (*o BST*)

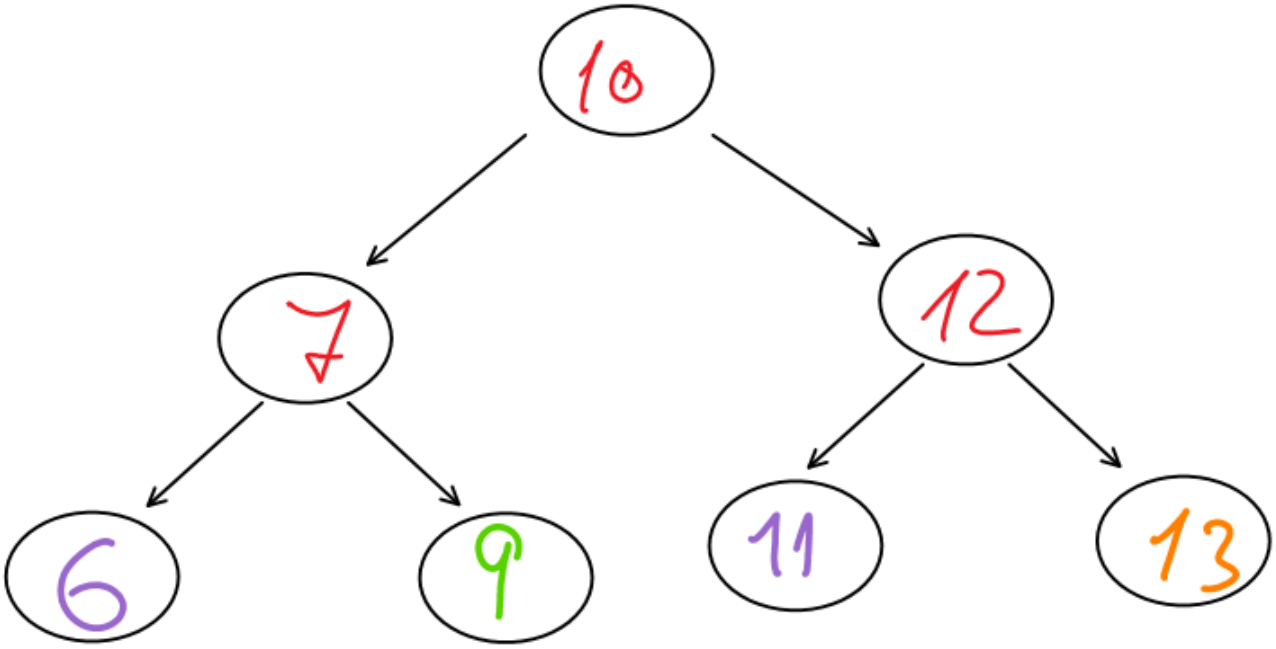
Cosa sono

Un albero binario di ricerca è un albero binario **i cui valori nel sottoalbero di *sinistra* sono minori del genitore**, mentre **i valori nel sottoalbero di *destra* sono maggiori del genitore**.

*questo **non** è un BST*



ma se opportunamente ordinato, esso può diventarlo



Implementazione del nodo

Per implementare il nodo istanzio una classe template in cui dichiaro come attributi *private* il valore (`template`) e i due nodi (*i figli*), implemento come metodi *public* il costruttore, il setter e il getter dei figli e del valore e l'overload di ostream. **Devo inoltre dichiarare in maniera *private* che esiste una classe `template` friend di quella attuale**.

Codice

In primo luogo controlliamo se è già stato dichiarato il nodo altrimenti lo dichiariamo. Includiamo le librerie necessarie.

```C++
#ifndef BST_NODE_H
#define BST_NODE_H
#include<iostream>
using namespace std;
```

Successivamente istanzio la classe `template` **BSTNode**, dichiarando come attributi la chiave (template), i due nodi, il puntatore al genitore e la classe template `BST` che sarà `friend`.

```C++
template <typename T>
class BSTNode{
    private:
    T key;
    BSTNode<T>* left;
    BSTNode<T>* right;
    BSTNode<T>* parent;
```

```
    template <typename U>
    friend class BST;
```

Implementiamo poi i metodi *public*:

1. costruttore, a cui è passato come parametro il valore della chiave e che - appunto - inizializza la chiave, inizializza inoltre il nodo sinistro e quello destro a `nullptr`.

```C++
public:
BSTNode(T key) : key(key){
    left = nullptr;
    right = nullptr;
}
```

2. **setter** di Left, Right e Parent, a cui è passato come parametro il puntatore ad un nodo

```C++
void setLeft(BSTNode<T>* left){
    this->left = left;
}

void setRight(BSTNode<T>* right){
    this->right = right;
}

void setParent (BSTNode<T>* parent){
    this->parent = parent;
}
```

3. **getter** di Left, Parent e Right, che ritorna il puntatore ad un nodo

```C++
BSTNode<T>* getLeft(){
    return this->left;
}

BSTNode<T>* getRight(){
    return this->right;
}

BSTNode<T>* getParent(){
    return this->parent;
}
```

4. **setter** della chiave, a cui è passato come parametro un valore template

```C++
void setKey(T key){
    this->key=key;
}
```

5. **getter** della chiave che restituisce un valore template

```C++
T getKey(){
    return this->key;
}
```

6. l'overload di ostream

```C++
friend ostream& operator<<(ostream& out, BSTNode<T>& node)
{
    out << "BSTNode =" << &node << " key=" << node.key;
        out << "left = " << node.left << " right = " <<
node.right;
    return out;
}
```

7. il **distruttore** *questo va implementato negli alberi perché se si elimina un nodo si eliminano grazie al distruttore anche i suoi sottoalberi*

```C++
    ~BSTNode(){
        delete left;
        delete right;
    }
```

Implementazione della classe albero

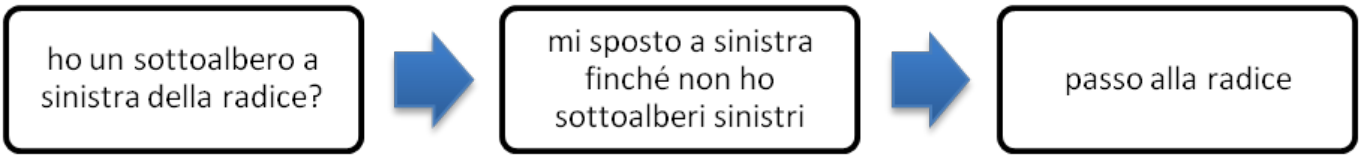
Per implementare l'albero istanzio una classe template in cui dichiaro come attributo *private* la radice, implemento come metodi *public* il costruttore, `isEmpty`, `insert` (uno a cui passo solo il valore da inserire e uno a cui passo sia il valore che il puntatore al nodo, questo perché il secondo sarà la **procedura ricorsiva**), i tre tipi di `visit`.

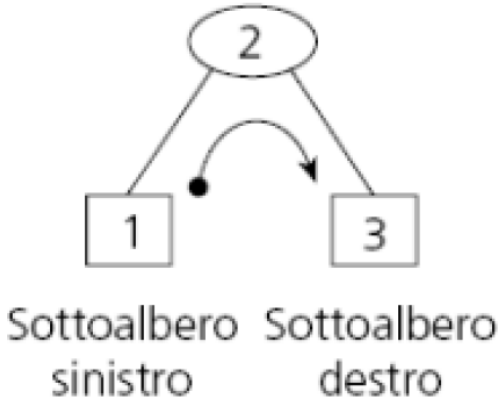
Procedimento insert

Tipi di visite (*modi di stampare*)

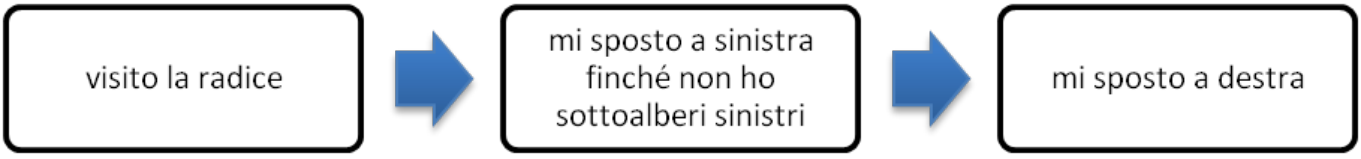
In totale esistono tre tipi di visite:

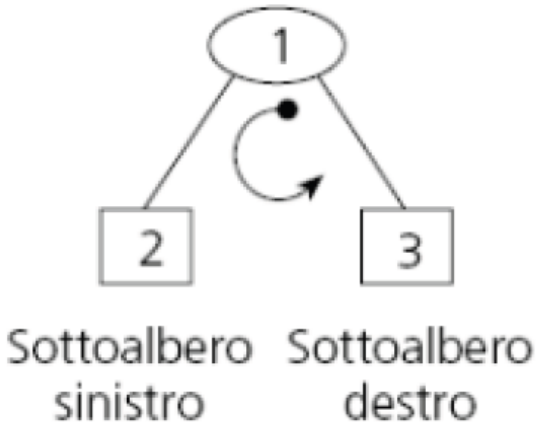
1. visita **inorder**: prima visito il sottoalbero sinistro, poi la radice e infine il sottoalbero destro.



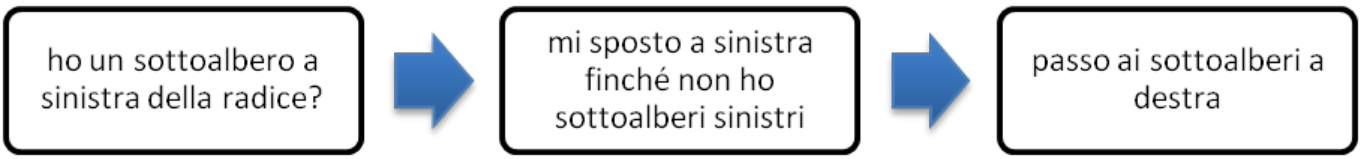


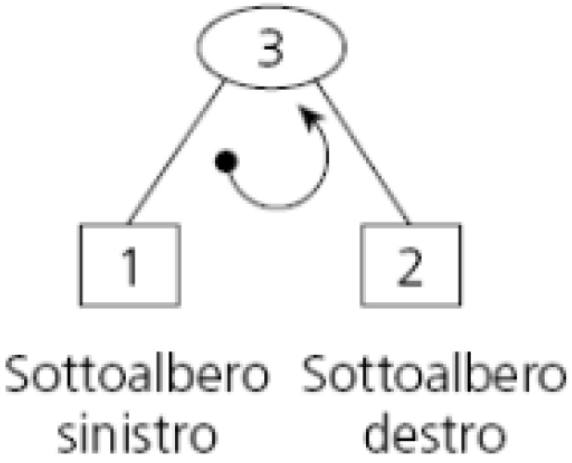
2. visita **preorder** : prima visito la radice, poi il sottoalbero sinistro e infine il sottoalbero destro





3. visita **postorder**: prima visito il sottoalbero sinistro, poi il sottoalbero destro e infine la radice

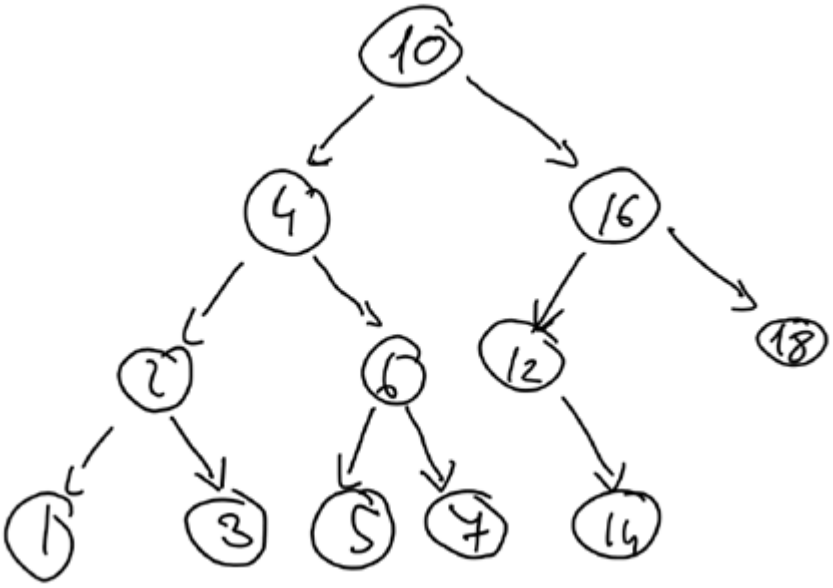




> un altro modo per visitare un albero è fare la "**stampa per livello**". Esso permette di stampare i nodi che si trovano allo **stesso livello** dell'albero.

Esempio

Considerato il seguente BST:



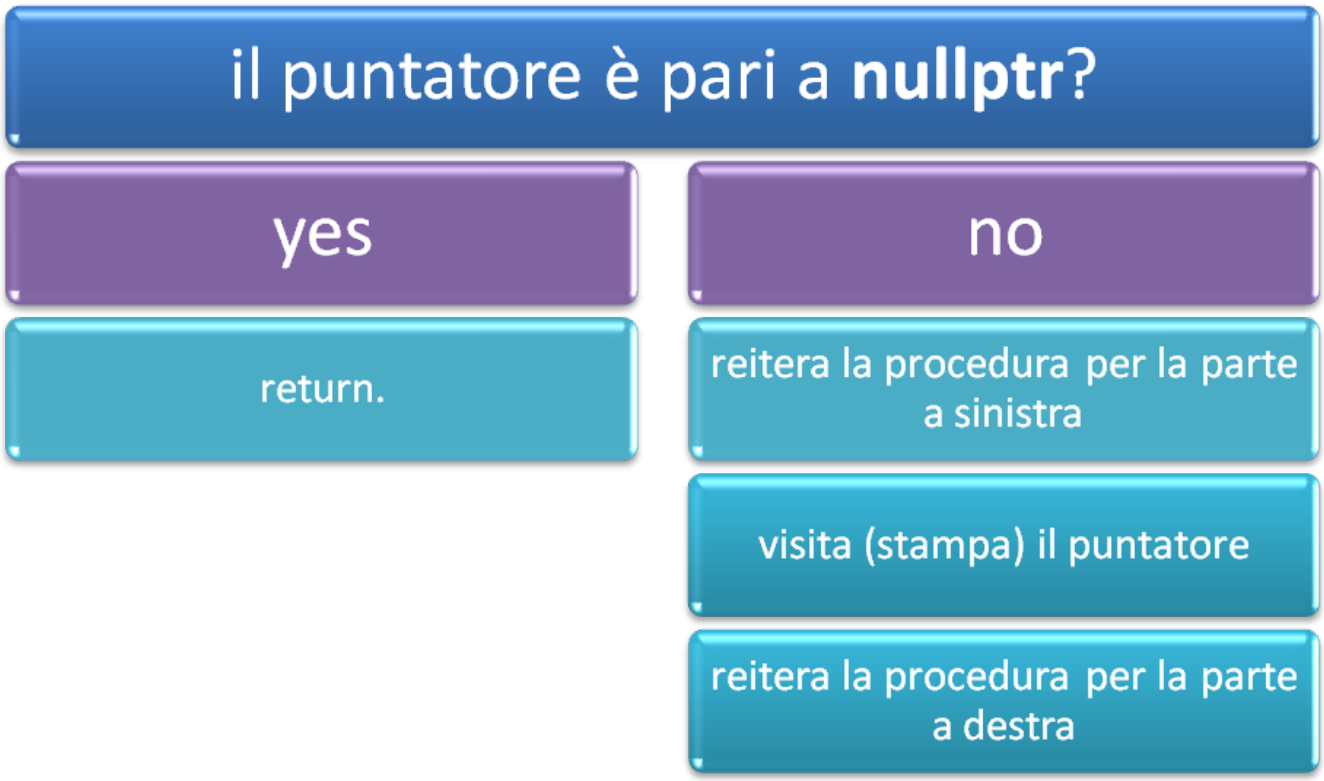
Per le diverse visite avremo stampati i valori nel seguente ordine:

1. visita inorder:
   `1 - 2 - 3 - 4 - 5 - 6 - 7 - 10 - 12 - 14 - 16 - 18`

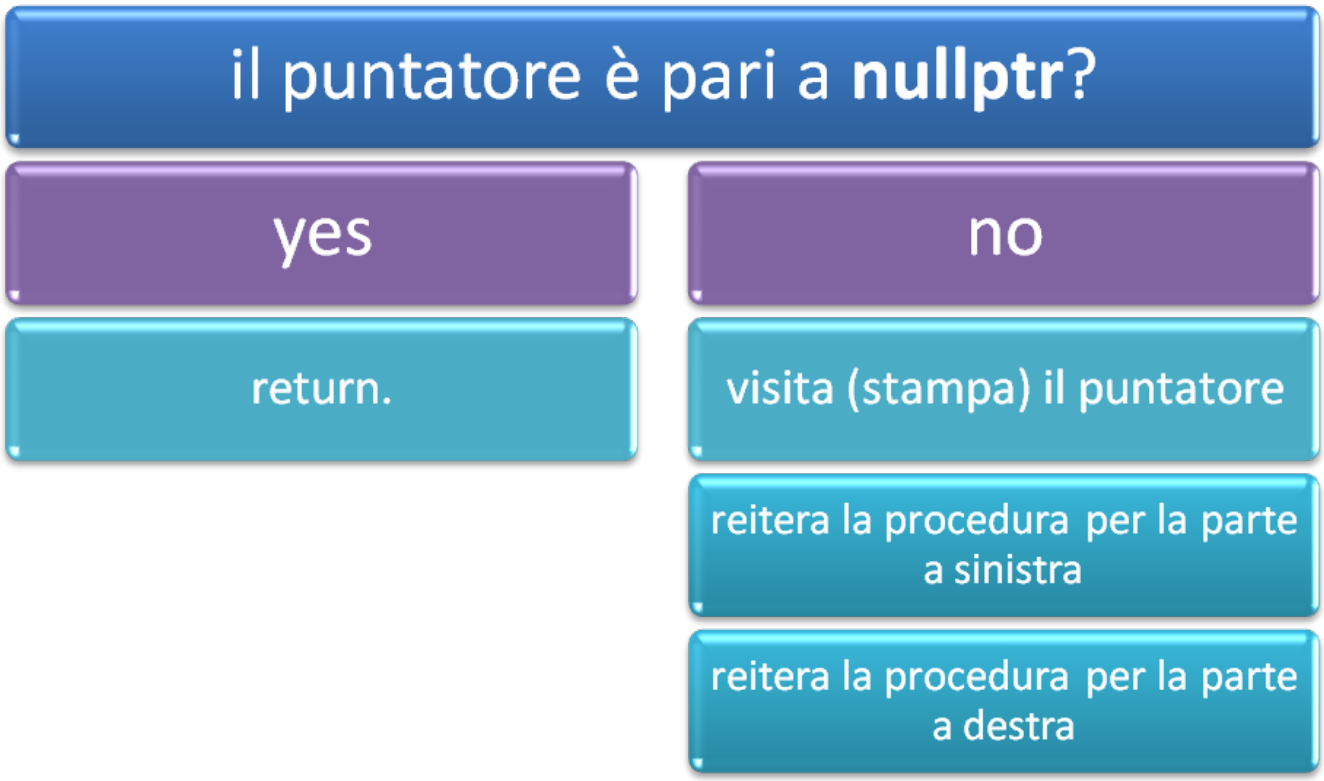
2. visita preorder:
   `10 - 4 - 2 - 1 - 3 - 6 - 5 - 7 - 16 - 12 - 14 - 18`
3. visita postorder:
   `1 - 3 - 2 - 5 - 7 - 6 - 4 - 14 - 12 - 18 - 16 - 10`

> possiamo notare che visitando **inorder** saranno stampati i numeri in ordine *crescente*.

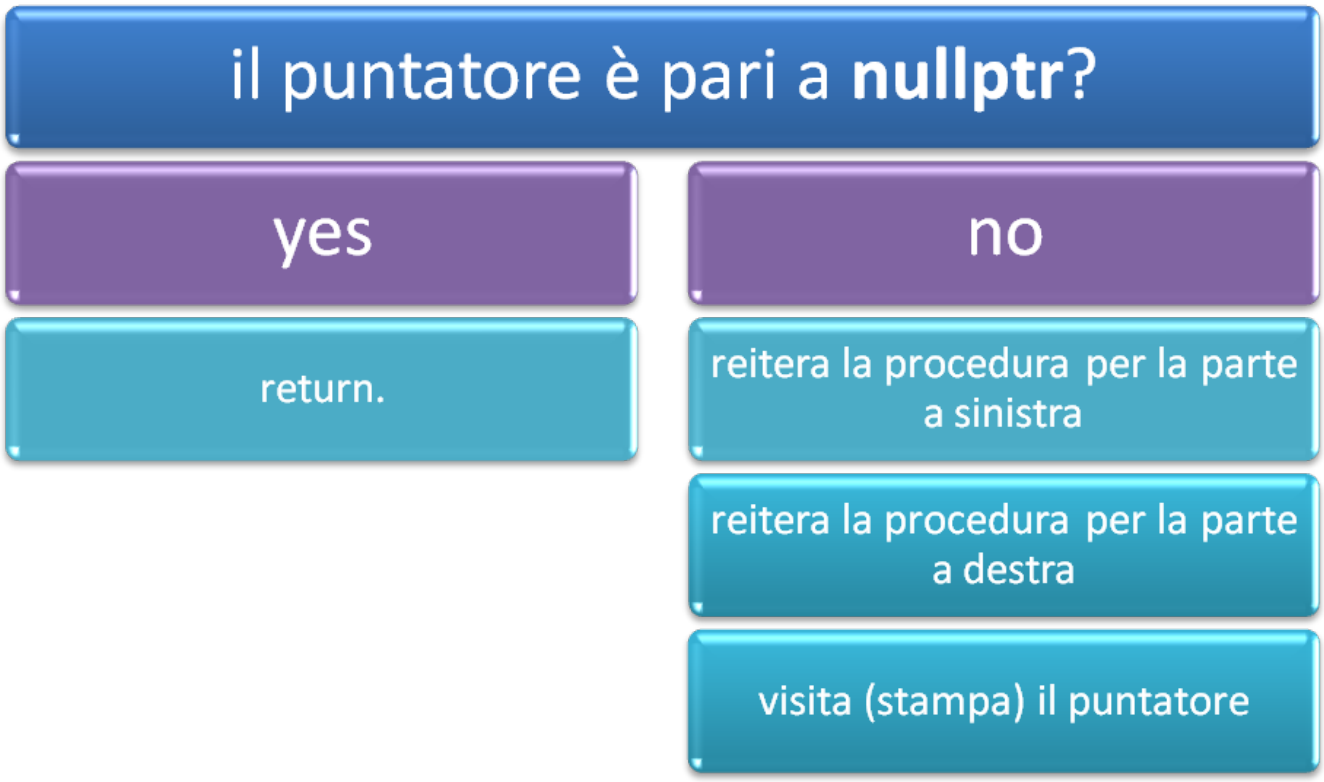
Procedimento visita inorder



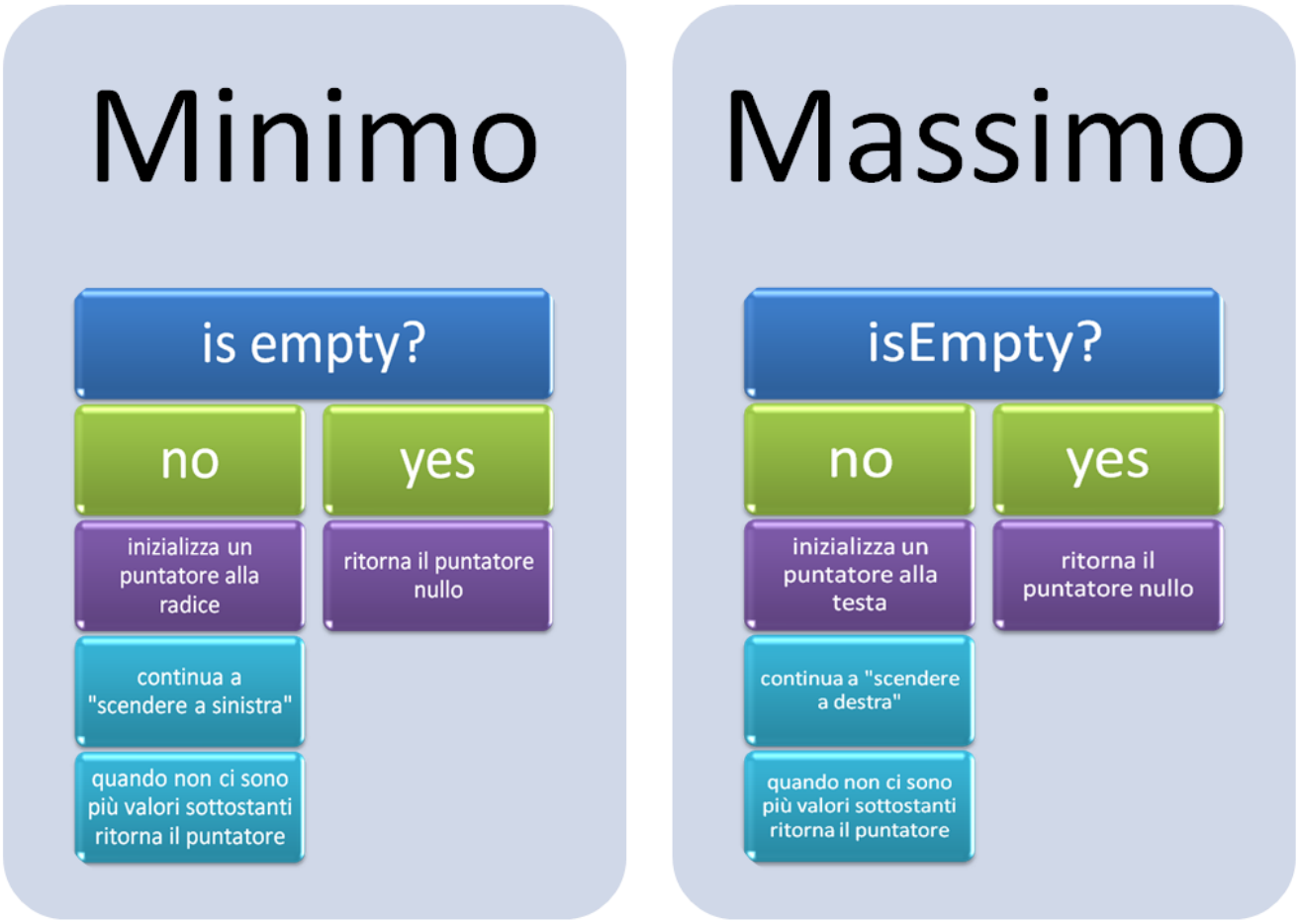
Procedimento visita preorder



Procedimento visita postorder



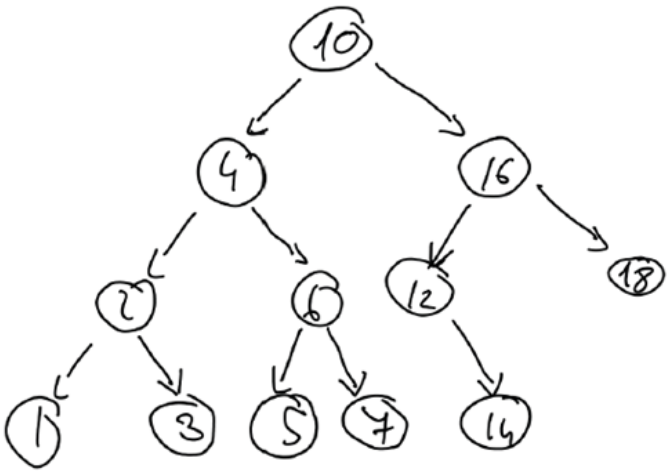
Ricerca del minimo e del massimo



Predecessore e successore

Il predecessore è il massimo di un sottoalbero sinistro. Il successore è il minimo del sottoalbero destro

Esempio



Se considero la radice

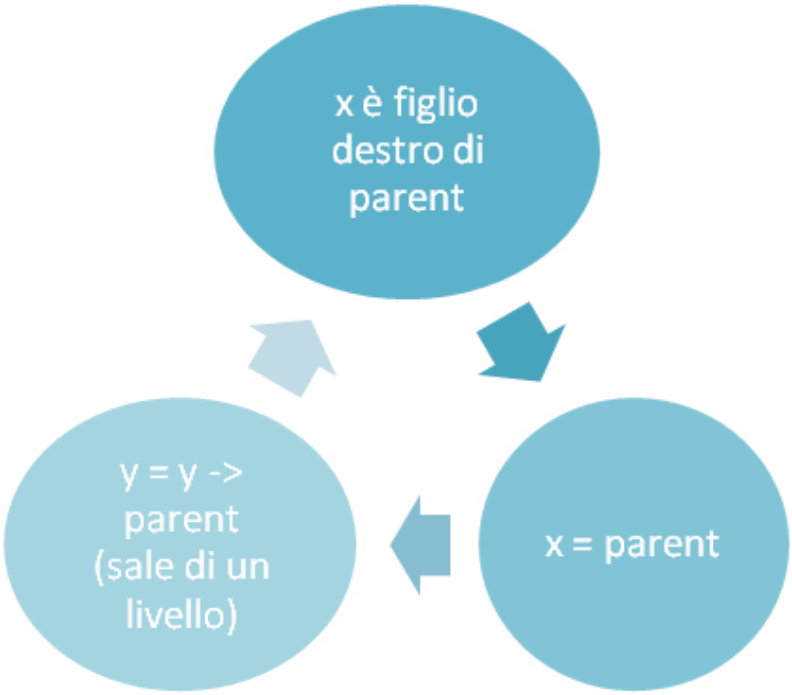
> il successore è 12
> il predecessore è 7

Se considero una figlio alla cui destra non ho nulla, devo risalire finché non individuo un numero che possa essere considerato successore (*ovvero che sia maggiore*).
In egual modo se considero un figlio alla cui sinistra non ho nulla, devo risalire finché non individuo un numero che possa essere considerato successore (*ovvero che sia minore*).

> > con una visita `inorder` posso conoscere in maniera immediata il predecessore e il successore: **il predecessore sarà il primo elemento, il successore sarà l'ultimo elemento.**
> > (*ovviamente il predecessore del minimo e il successore del massimo sarà* `NIL`).

Definizione formale del successore

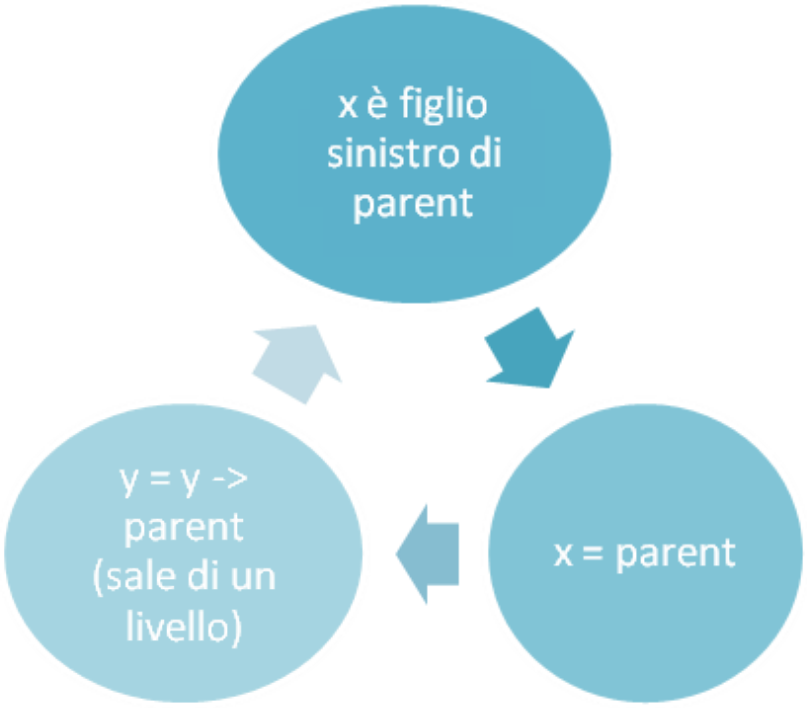
- Nel primo caso il successore sarà il minimo di questo sottoalbero destro
- Nel secondo caso il successore sarà **l'antenato più prossimo di x il cui figlio sinistro è anche un antenato di x**. Cioè a partire dal nodo in considerazione devo *risalire l'albero fino a quando il nodo che ho sopra è un figlio destro*. Appena il **nodo genitore è un figlio sinistro**, ho trovato il successore che stavo cercando, cioè il genitore dell'ultimo nodo che prendo in considerazione.



> procedo in maniera ricorsiva finché `x` non sarà più un figlio destro di parent, allora il successore sarà proprio y (cioè parent(x)

Definizione formale del predecessore

- Nel primo caso il successore sarà il massimo di questo sottoalbero sinistro
- Nel secondo caso il successore sarà **l'antenato più prossimo di x il cui figlio destro è anche un antenato di x**. Cioè a partire dal nodo in considerazione devo *risalire l'albero fino a quando il nodo che ho sopra è un figlio sinistro*. Appena il **nodo genitore è un figlio destro**, ho trovato il successore che stavo cercando, cioè il genitore dell'ultimo nodo che prendo in considerazione.



> Procedo in maniera ricorsiva finché `x` non sarà più un figlio sinistro di parent, allora il successore sarà proprio y (cioè parent(x)

Metodo di ricerca

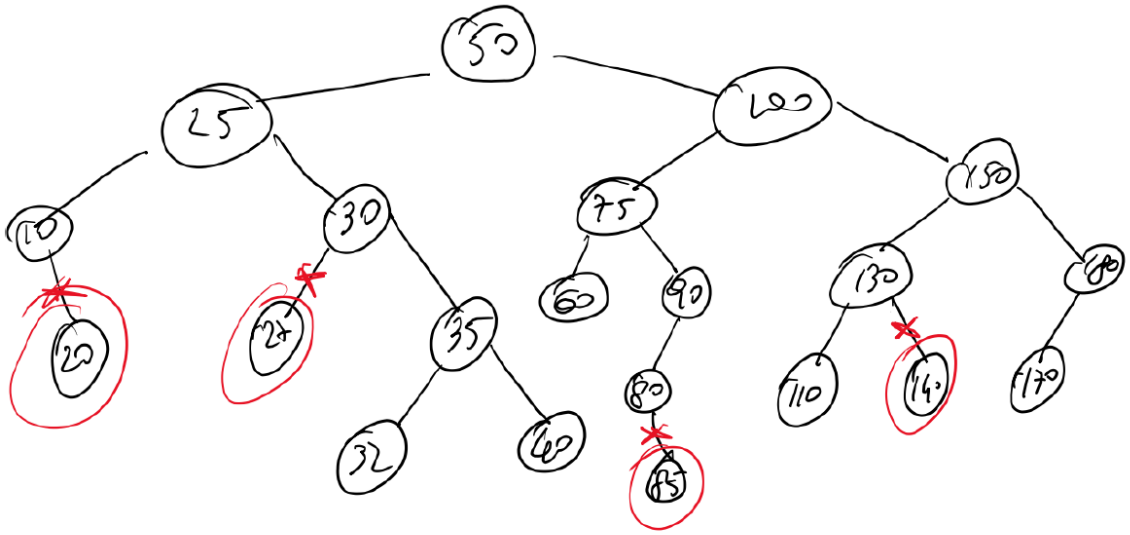
Per testare nel main il caso in cui il nodo non presenta sottoalberi a destra o a sinistra (*in riferimento al predecessore e al successore*), occorre fare la **ricerca**, prendendo così in riferimento una foglia. Per farlo si utilizza la ***ricerca dicotomica***.

Cancellazione

Per cancellazione intendiamo la rimozione di un nodo da un albero. Il modo in cui avviene la cancellazione dipende dal numero di figli che il nodo da eliminare ha. *Questo perché i figli **non** devono essere rimossi se non **specificatamente** indicato*.

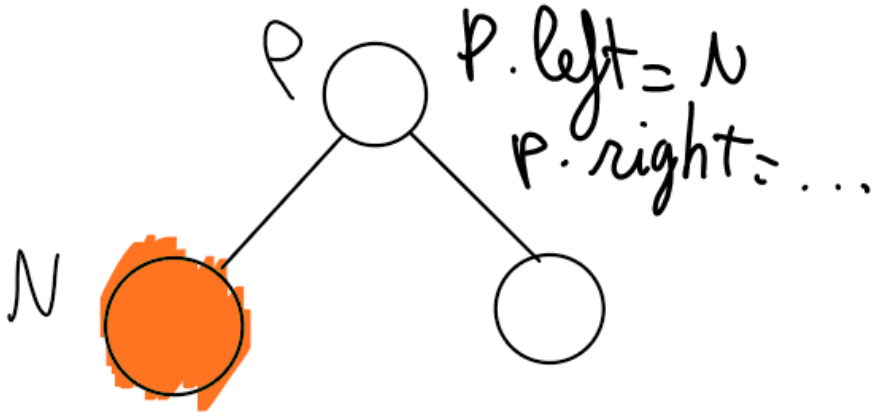
1) Il nodo è una foglia

Se il nodo è una foglia devo semplicemente rimuovere il riferimento con un `ptr` temporaneo e devo valutare se sto eliminando il figlio sinistro o quello destro.



I nodi cerchiati rappresentano le **foglie** da rimuovere.

Procedimento

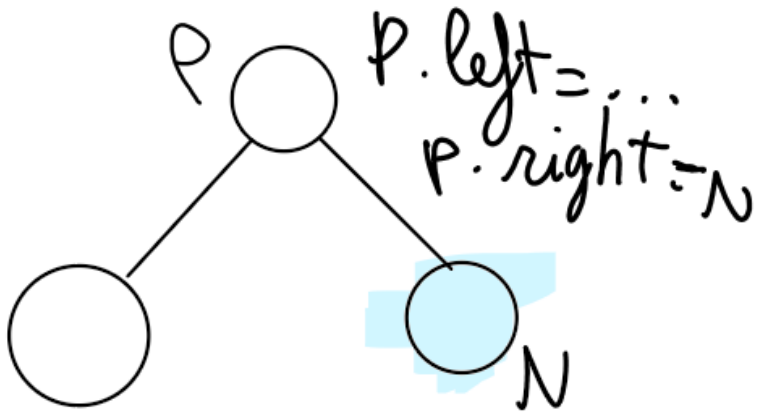


N.left = NIL

N.right = NIL

N.Parent = P

node -> parent->left = NIL



N.left = NIL

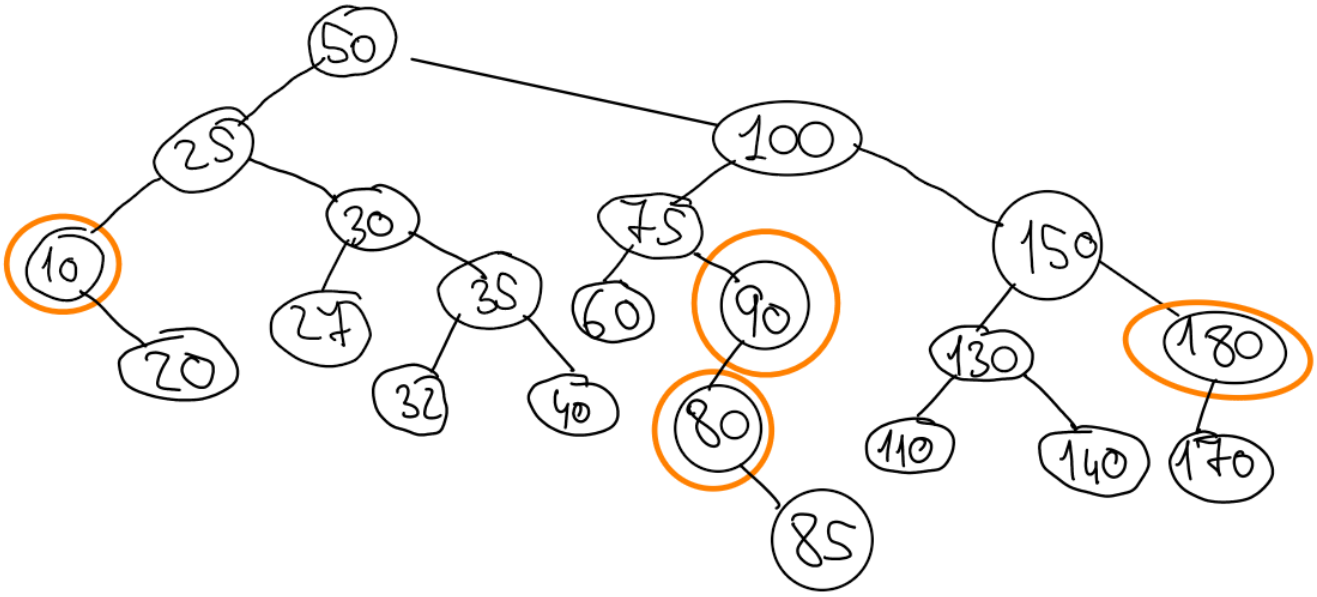
N.right = NIL

N.Parent = P

node -> parent->right = NIL

2) Il nodo ha un figlio

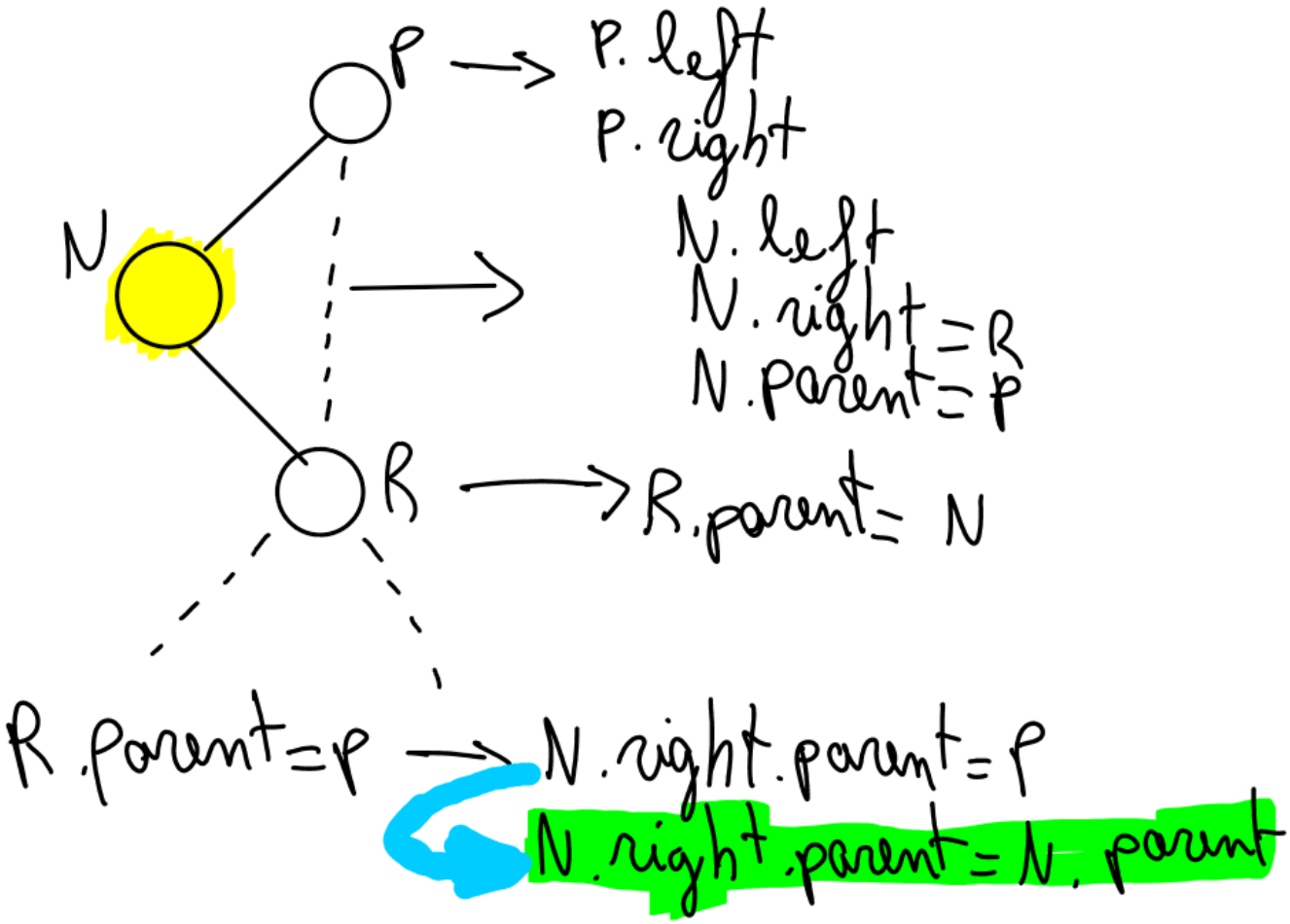
Se il nodo ha un figlio, tale figlio prenderà "il suo posto" *diventando il figlio del suo genitore*. Se - ad esempio - è figlio sinistro, suo figlio diverrà figlio sinistro del "nonno".



I nodi cerchiati rappresentano i **genitori** da rimuovere.

Procedimento


il nodo da eliminare ha un figlio
sinistro
destro
pongo il parent del figlio pari al parent del nodo stesso
pongo il parent del figlio pari al parent del nodo stesso
il nodo da eliminare è figlio
il nodo da eliminare è figlio
sinistro
destro
sinistro
destro
unisco la sinistra del parent con la sua sinistra
unisco la destra del parent con la sua destra
unisco la sinistra del parent con la sua sinistra
unisco la destra del parent con la sua destra
ritorno il nodo
ritorno il nodo
ritorno il nodo
ritorno il nodo



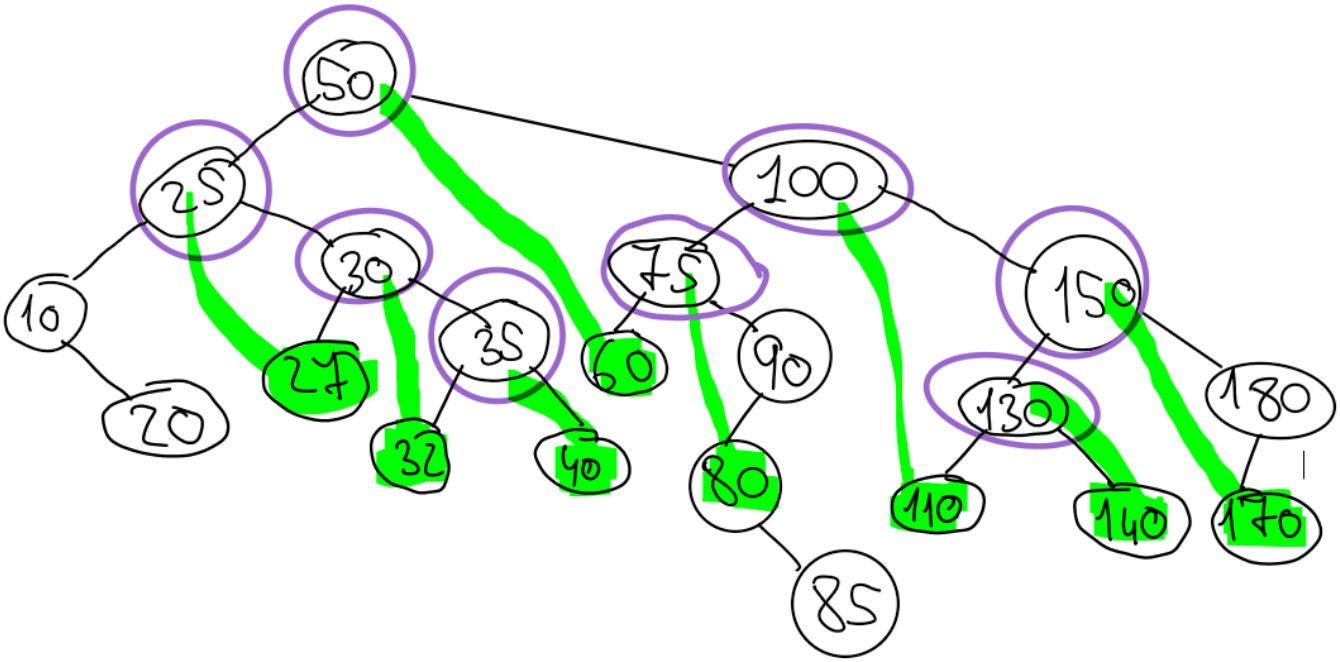
3) Il nodo ha due figli

Concettualmente non voglio rimuovere il nodo, ma il valore all'interno (*la chiave* e non *l'indirizzo del nodo*). Inoltre è importante mantenere la struttura, rispettando quindi la ***proprietà del BST***.

> Nel terzo caso quindi **si parla di sostituire e riorganizzare** in funzione della proprietà (per cui a sinistra sono presenti i valori minori e a destra i valori maggiori)

Per far ciò sostituiamo la chiave con il **successore** del nodo da eliminare. (*In questo caso si parla del primo caso in quanto è ovviamente presente il sottoalbero destro*). E poi si elimina il successore (tornando al primo o al secondo caso)

> Il successore ha **al più un figlio**, non esistendo elementi minori a destra, altrimenti esisterebbe un nodo alla sua sinistra che sarebbe però il nuovo successore.



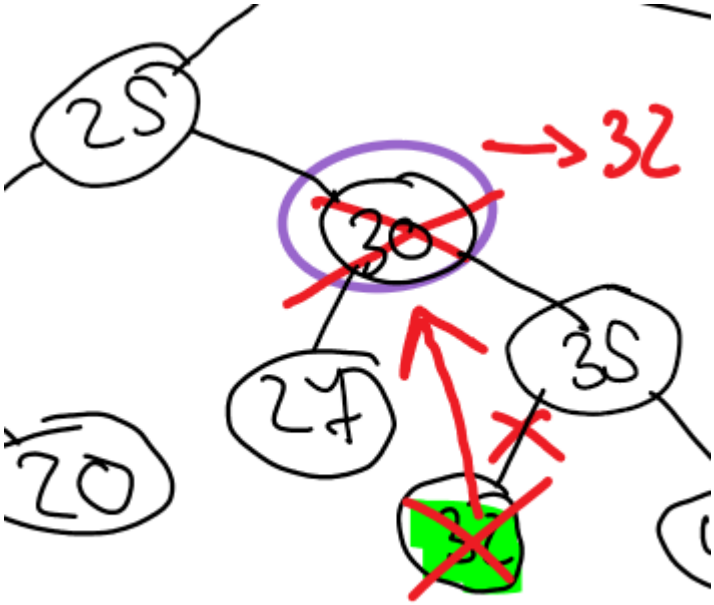
I nodi cerchiati rappresentano i **genitori** da rimuovere, collegati ai loro successori.

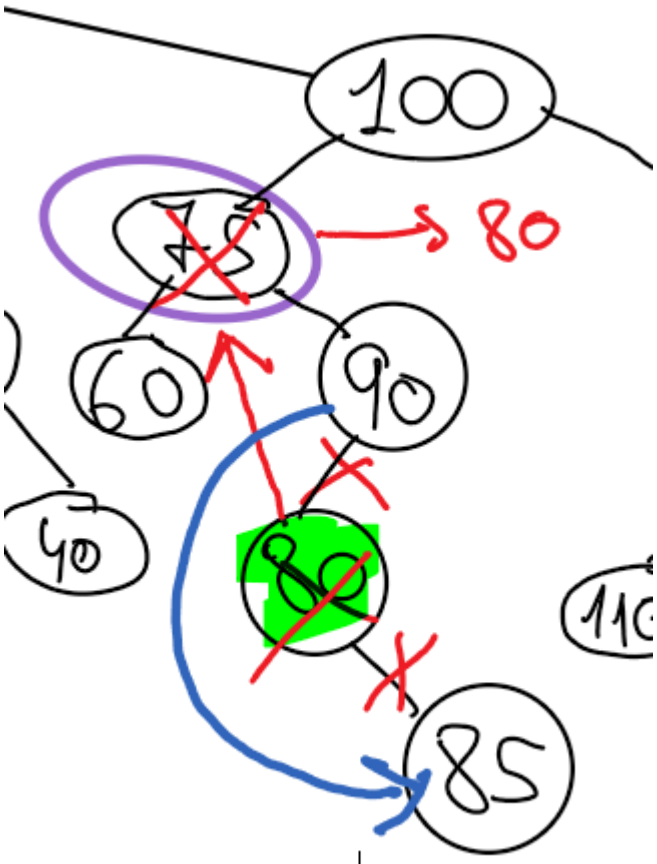
Procedimento

Il seguente è il procedimento di **tutta** l'eliminazione, il procedimento riguardante *solo il terzo punto* è quello in arancione.


isEmpty?
no
yes
cerco il nodo da eliminare, è stato trovato?
ritorna il puntatore nullo
no
yes
ritorna il puntatore nullo
inizializzo un puntatore richiamando la funzione delete
inizializzo un puntatore richiamando la funzione successore
controllo se è diverso dal puntatore nullo
faccio lo swap tra il valore key e il valore del successore
ritorno il puntatore implementato
pongo il puntatore inizializzato poc'anzi (toDelete) passando come parametro il successore

ritorno il puntatore implementato



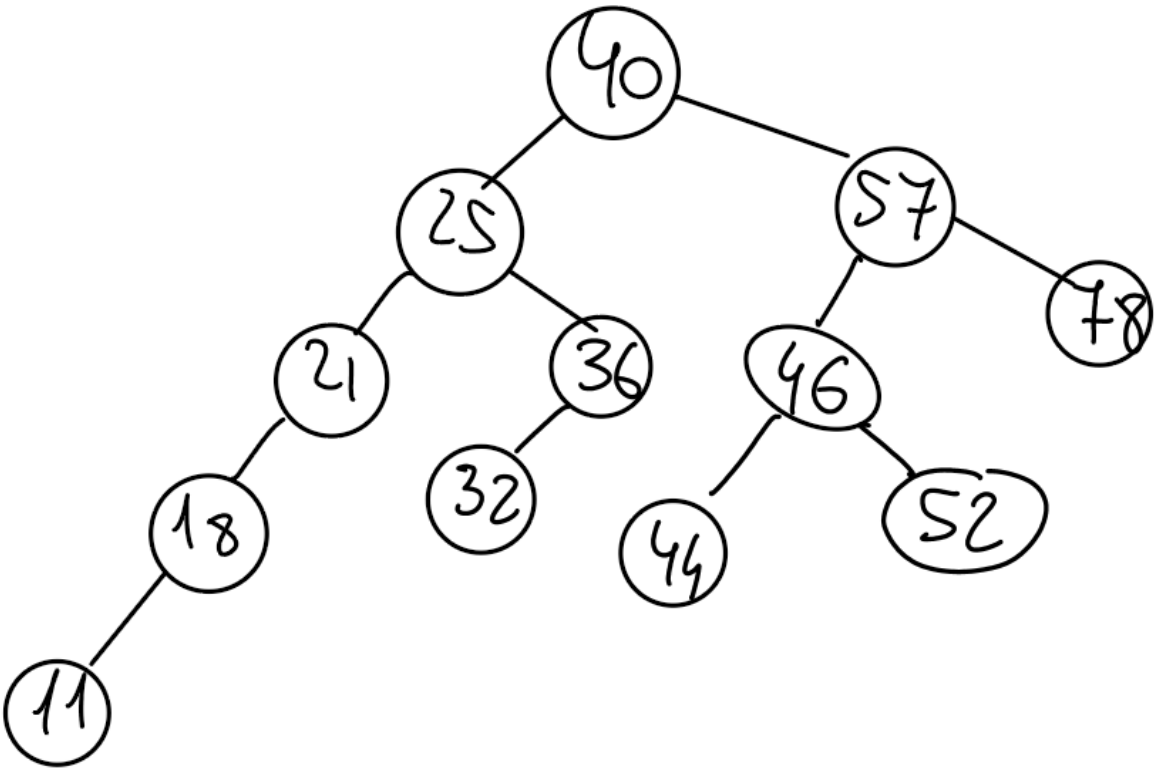


Simulazioni

Inseriamo ora i seguenti numeri:

`40 - 25 - 21 - 18 - 57 - 36 - 46 - 32 - 78 - 52 - 11 - 44`

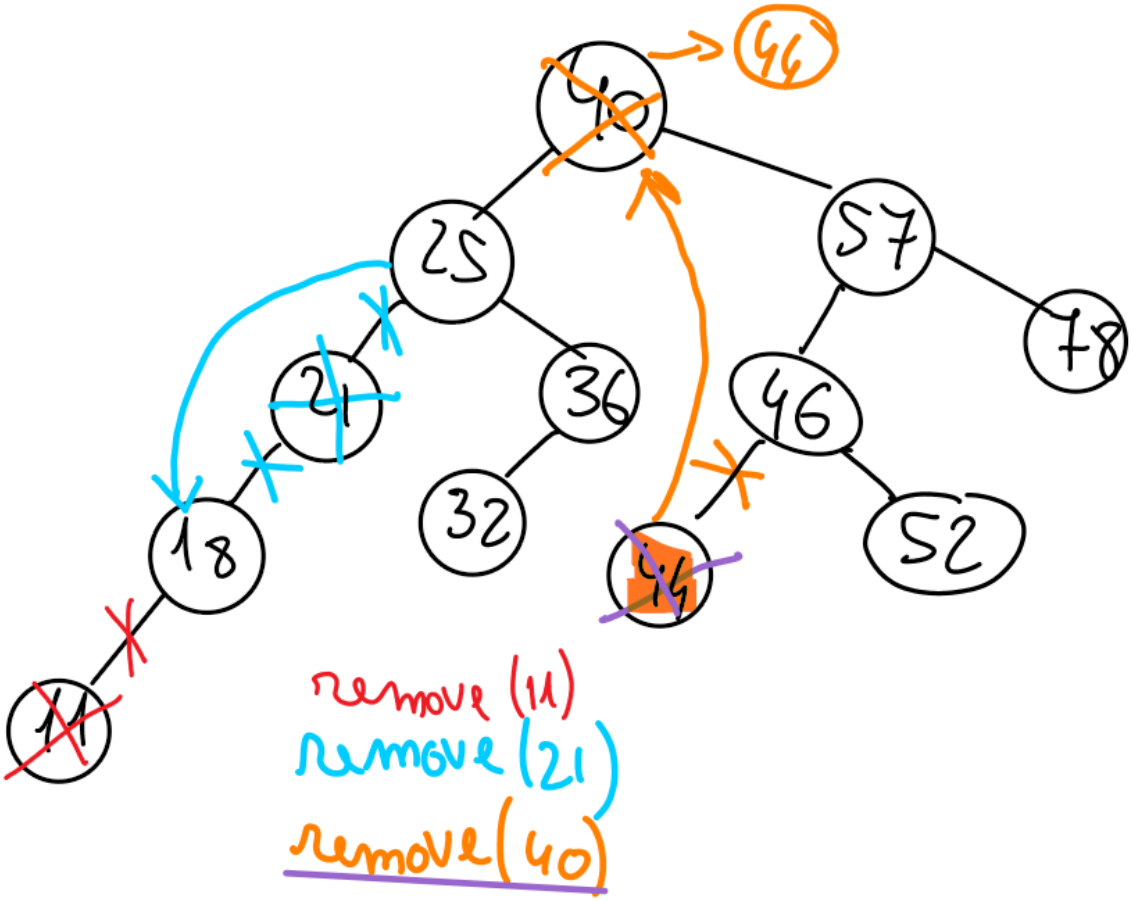
Ottenendo

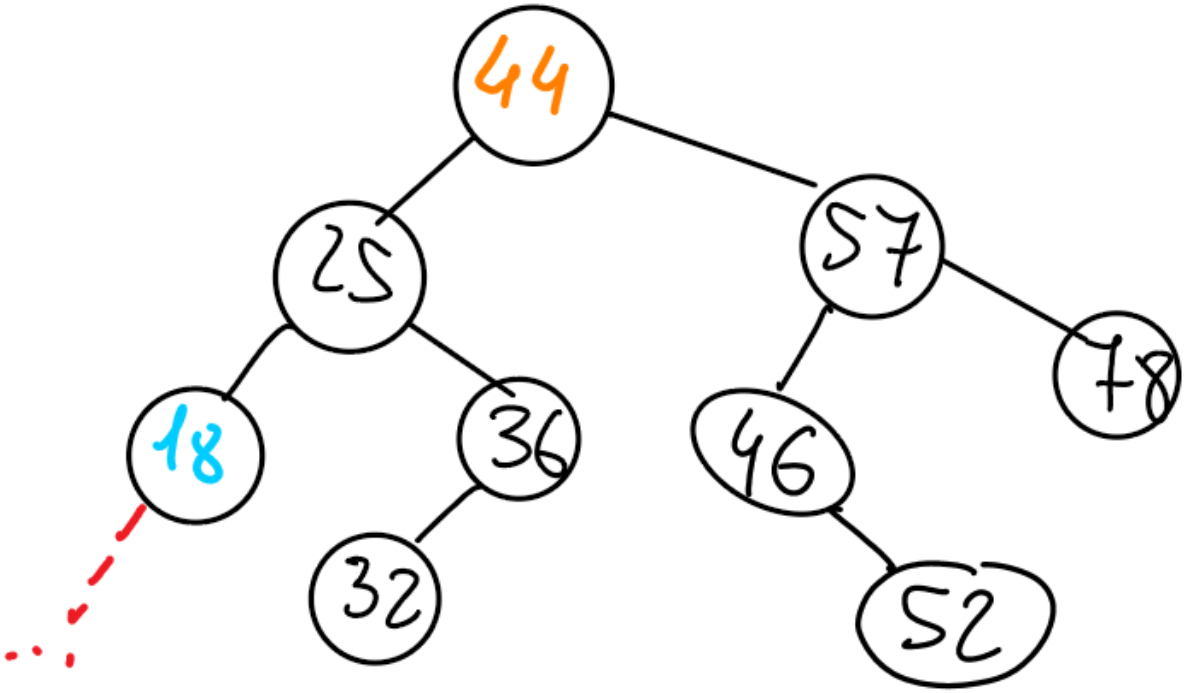


Eliminiamo poi i seguenti numeri:

`11 - 21 - 40`

Ottenendo


46
40
25
57
21
36
78
46
18
32
44
52
11
remove (11)
remove (21)
remove (40)


44
25
57
18
36
46
78
32
52

Complessità operazioni

La complessità delle operazioni di un BST è

$$O(log_n)$$

perché varia in base all'*altezza* dell'albero.

Codice

In primo luogo controlliamo se BST_H è stato definito e lo definiamo. Includiamo le librerie e l'header (`bst_node.h`) necessari.

```C++
#ifndef BST_H
#define BST_H
#include "bst_node.h"
#include<iostream>
using namespace std;
```

Successivamente istanziamo una classe `template` che abbia come attributo *private* il puntatore alla **radice**

```C++
template <typename T>
class BST{
    private:
    BST_Node<T>* root;
```

E implemento come metodi public:

1. il costruttore in cui inizilizzo la radice a `nullptr`

```C++
public:
BST(){
    root = nullptr;
}
```

2. `isEmpty` che ritorna il valore booleano "true" se la radice corrisponde a `nullptr`

```C++
bool isEmpty(){
    return root == nullptr;
}
```

3. `insert` a cui passo come parametro un valore template `key` e che se l'albero è vuoto pone la radice a key, altrimenti richiama un'altra funzione `insert` alla quale si passano come parametri *root* e *key*.

```C++
void insert(T key){
    if(this->isEmpty()){
        root = new BSTNode<T>(key);
        return;
    }
    insert (root, key);
}
```

4. `insert` a cui passo come parametri un valore template e un puntatore. Se il puntatore corrisponde e `nullptr` inserisco un nuovo nodo, altrimenti controllo se è minore o maggiore del valore puntato e passo nuovamente la funzione; devo fare questa ricorsione finché non trovo `nullptr` in left o right.

C++

```
void insert(BSTNode<T>* ptr, T key){
    if (ptr == nullptr){
        ptr = new BSTNode<T>(key);
        return;
    }
    //caso base. non ha un figlio sinistro
    if (ptr->left == null && key<=ptr->key){
        ptr->left = new BSTNode<T>(key);
        ptr->left->setParent(ptr);
        return;
    }
    //caso base. non ha un figlio destro
    if (ptr->right == nullptr && key>ptr->key){
        ptr->right = new BSTNode<T>(key);
        ptr->right->setParent(ptr);
        return;
    }
    else if (key<=ptr->key){
        insert(ptr->left, key);
    }

    else {
        insert(ptr->right, key);
    }
}
```

5. `visit` a cui passo come parametro il puntatore al nodo e che semplicemente stampa il nodo passato

```C++
void visit(BSTNode<T>* node){
    cout << *node << endl;
}
```

6. `inorder` a cui passo come parametro il puntatore a un nodo. Dopodiché controllo se il puntatore è pari a `nullptr` altrimenti reitero la funzione considerando il puntatore a sinistra, stampo il puntatore centrale e reitero la funzione considerando il puntatore a destra

```C++
void inorder(BSTNode<T>* ptr){
    if (ptr == nullptr)
        return;

    inorder(ptr->left);
    visit(ptr);
    inorder(ptr->right);
}
```

```C++
void inorder(){
    inorder (root);
    }
```

7. ricerca del **minimo** e del **massimo**. Controllo se l'albero è vuoto, in quel caso ritorno `nullptr`, altrimenti inizializzo un puntatore alla testa e "scendo" a sinistra (*per il minimo*) o a destra (*per il massimo*) e ritorno il puntatore.

```C++
BSTNode<T>* min(){
    if(isEmpty()){
        return nullptr;
     }
    BSTNode<T>* ptr = root;
    while (ptr->left) {
        ptr = ptr->left;
    }
    return ptr;
}

BSTNode<T>* max(){
if(isEmpty()){
        return nullptr;
    }
    BSTNode<T>* ptr = root;
    while (ptr->right){
        ptr = ptr->right;
    }
    return ptr;
}
```

8. ricerca del **successore** che restituisce un puntatore a nodo e a cui passo un puntatore a nodo; se l'albero è vuoto restituisce `nullptr`, altrimenti restituisce il minore del sottoalbero destro (se ne ha uno) o inizializza un nodo che punta al genitore del nodo da eliminare, scorro l'albero finché il nodo (*passato come parametro*) è diverso da `nullptr` e posso "scendere a destra"

```C++
BSTNode<T>* successor(BSTNode<T>* x){
    if (this->isEmpty()){
```

```
            return nullptr;
        }

        if (x->right){
            return this->min(x->right);
        }

        BSTNode<T>* y = x->parent;
        while(x!=nullptr && x=y->right){
            x=y;
            y = y->parent;
        }
        return y;
    }
```

9. ricerca del **predecessore** che restituisce un puntatore a nodo e a cui passo un puntatore a nodo; se l'albero è vuoto restituisce `nullptr`, altrimenti restituisce il maggiore del sottoalbero sinistro (se ne ha uno) o inizializza un nodo che punta al genitore del nodo da eliminare, scorro l'albero finché il nodo (*passato come parametro*) è diverso da `nullptr` e posso "scendere a sinistra"

C++

```
    BSTNode<T>* successor(BSTNode<T>* x){
        if (this->isEmpty()){
            return nullptr;
        }

        if (x->left){
            return this->min(x->left);
        }
```

```
        BSTNode<T>* y = x->parent;
        while(x!=nullptr && x=y->left){
            x=y;
            y = y->parent;
        }
        return y;
    }
```

10. **ricerca dicotomica**, a cui passo sia un puntatore a nodo che il valore da ricercare e che restituisce un puntatore a nodo. Per prima cosa controlla se l'albero è vuoto e ritorna `nullptr`, altrimenti controlla - in riferimento al puntatore passato - dove si trova il valore cercato.

C++

```
BSTNode<T>* search(T key){
    return search(root, key);
}

BSTNode<T>* search (BSTNode<T>* ptr, T key){
    if (ptr == nullptr){
        return nullptr;
    }

    if (ptr->key == key){
        return ptr;
    }

    else if (ptr->key >= key){
        return search(ptr->left, key);
    }
```

```
        else {
            return search(ptr->right, key);
        }
```

11. eliminazione di una foglia o di un nodo con **un figlio** che restituisce un puntatore a nodo e a cui si passa come parametro un puntatore a nodo. In primo luogo si controlla se il nodo è una foglia, quindi se a sinistra e a destra è presente `nullptr`, dopodiché elimino banalmente la foglia ponendo il nodo a sinistra (o a destra) del suo parent pari a `nullptr`. Dopodiché controllo se è presente solo un nodo a destra (quindi solo a destra **non** è presente `nullptr`) e collego il suo figlio al suo genitore e viceversa (il genitore al figlio), rimpiazzando se stesso col figlio. Infine faccio la stessa operazione specularmente

C++

```
    BSTNode<T>* remove(BSTNode<T>* node){
        if (node->left == nullptr && node->right ==
nullptr){
            if (node == node->parent->left)
                node->parent->left = nullptr;
            if (node == node->parent->right)
                node->parent->right = nullptr;

            return node;
        }

        if (node->left == nullptr && node->right !=
nullptr)
            node->right->parent = node->parent;

            if(node == node->parent->left)
                node->parent->left = node->right;
```

```
            else if(node == node->parent->right)
                node->parent->right = node->right;
            return node;
        }
    }

        if (node->left!=nullptr && node->right==nullptr){
            node->left->parent = node->parent;

            if (node == node->parent->left)
                node->parent->left = node->left;
            else if (node == node->parent->right)
                node->parent->right = node->left;
            }
            return node;
        }
    return nullptr;
    }
```

11. **Rimozione di un valore o di un nodo** a cui si passa come parametro il valore template key e che ritorna un puntatore a nodo. Per prima cosa si controlla se l'albero è vuoto e in quel caso ritorna `nullptr`. Altrimenti si cerca il valore da eliminare. Se questo è pari a `nullptr` (quindi non è stato trovato) ritorna `nullptr`. Altrimenti si inizializza un puntatore a nodo che richiami la funzione `remove(node)`. Se il valore non è stato eliminato (*quindi è diverso da* `nullptr`), si inizializza un puntatore a nodo che richiama la funzione `successor(node)` e si inizializza un valore template swap per *"swapare"* il valore nel nodo da eliminare e il valore nel **successore**; infine si elimina il successore (richiamando `remove(next)` appunto sul successore).

C++

```
    BSTNode<T>* remove(T key){
        if(this->isEmpty()){
            return nullptr;
            }

        BSTNode<T>* node = search(key);

        if (node == nullptr){
            return nullptr;
        }

        BSTNode<T>* toDelete = this->remove(node);
        if (toDelete != nullptr){
            return toDelete;
        }

        BSTNode<T>* next = this->successor(node);
        T swap = node->key;
        node->key = next->key;
        next->key = swap;

        BSTNode<T>* toDelete = this->remove(next);
        return toDelete;
}
```