Binary Search Tree (Alberi binari di ricerca)

E' una struttura dati ricorsiva definita induttivamente nel seguente modo:

a) $\emptyset \in BRT(A)$ (l'albero vuoto fa parte dell'insieme)

b)

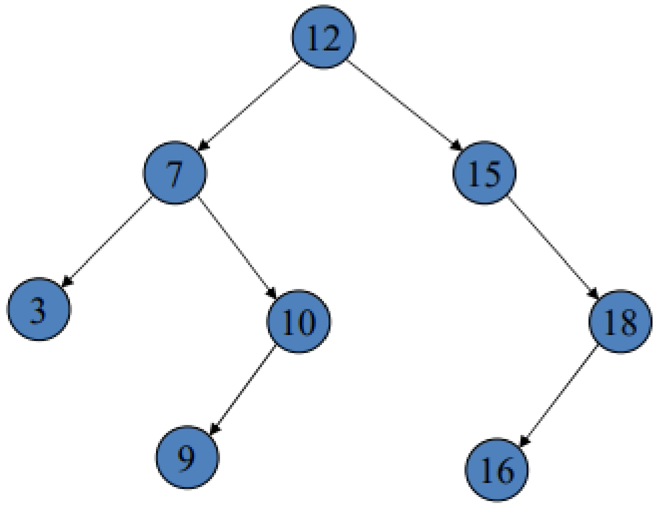
$$a \in A \ \wedge \ l \in BRT(A) \ \wedge \ r \in BRT(A) \ \wedge \ \forall c \in keys(l).c < a \ \wedge \ \forall c \in keys(r).a < c$$

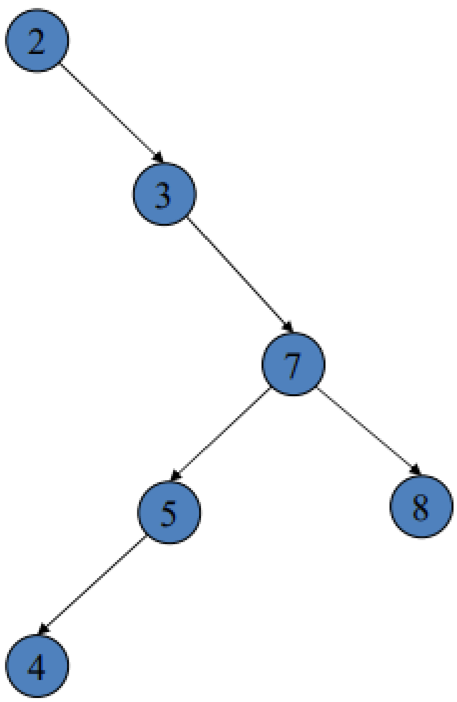
$$\Downarrow$$

$$\{a, l, r\} \in BRT(A)$$

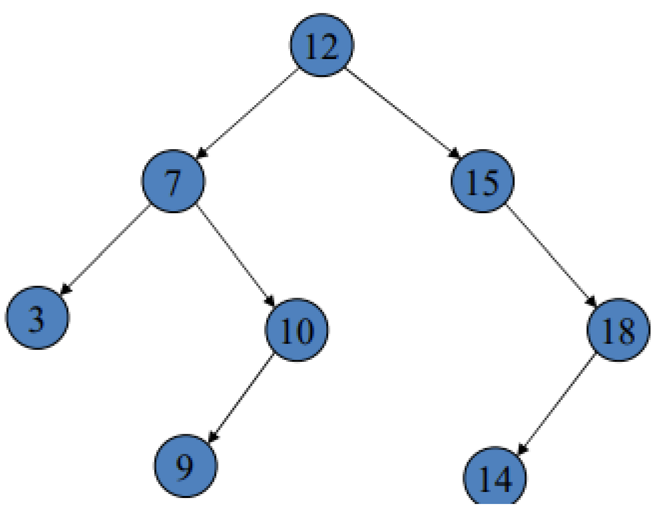
In parole povere, in un albero con radice **a** e sottoalberi **l**(sinistro) e **r**(destro), ogni chiave in **l** è minore di **a** e ogni chiave in **r** è maggiore di **a**

Esempi BST





Mentre invece il seguente albero **NON E'** un **BST:**



Applicando la definizione ricorsiva di BST,scopriamo che il nodo 14, *appartenente al sottoalbero destro di 15, è minore di 15 stesso, violando la definizione di BST*

Ricerca

In maniera dicotomica, confronto l'elemento con i due sottoalberi, e mi sposto in base al risultato.
Il tempo computazionale è proporzionale all'altezza **h** dell'albero, dunque $O(h)$.
Nel caso peggiore, tuttavia, l'altezza h potrebbe essere uguale al numero di nodi, e quindi la ricerca diventerebbe *lineare*

Ricorsiva

```
Ric-Search(x, T)
    ▷ pre: x chiave, T binario di ricerca
    ▷ post: il nodo S ∈ T con S.key = x se esiste, nil altrimenti
if T = nil then return nil
else
    if x = T.key then return T
    else
        if x < T.key then return Search(x, T.left)
        else      ▷ x > T.key
            return Search(x, T.right)
        end if
    end if
end if
```

Iterativa

$\text{IT-SEARCH}(x, T)$

▷ pre: x chiave, T binario di ricerca

▷ post: il nodo $S \in T$ con $S.key = x$ se esiste, nil altrimenti

while $T \neq nil$ **and** $x \neq T.key$ **do**

  if $x < T.key$ **then**

    $T \leftarrow T.left$

  else

    $T \leftarrow T.right$

  end if

end while

return T

Stampa delle etichette

Per stampare le etichette in ordine, bisogna:

1. Stampare tutte le etichette in ordine del sottoalbero sinistro
2. Stampare l'etichetta *radice*
3. Stampare tutte le etichette in ordine del sottoalbero destro

Applicando una definizione ricorsiva, otteniamo il seguente algoritmo:

$\text{PRINT-INORDER}(T)$

▷ pre: T binario di ricerca

▷ post: stampate le chiavi in T in ordine

if $T = nil$ **then**

  return

end if

$\text{PRINT-INORDER}(T.left)$

print $T.key$

$\text{PRINT-INORDER}(T.right)$

Ricerca del minimo(e del massimo)

Per trovare l'elemento minimo in un **BST**, basta semplicemente spostarsi sempre nel sottoalbero sinistro mentre attraversiamo l'albero.

In modo analogo per trovare l'elemento massimo, ci spostiamo nel sottoalbero destro ogni volta.

$\textsc{Tree-Min}(T)$

$\quad\triangleright$ pre: T binario di ricerca non vuoto

$\quad\triangleright$ post: il nodo $S \in T$ con $S.key$ minimo

$S \leftarrow T$

while $S.left \neq nil$ **do**

$\quad S \leftarrow S.left$

end while

return S

Successore di un nodo

Il successore di un nodo *N* in un albero *T* è quel nodo la cui etichetta è la minore tra quelle più grandi del nodo *N*.

$\textsc{Tree-Succ}(N)$

$\quad\triangleright$ pre: N nodo di un albero bin. di ricerca

$\quad\triangleright$ post: il successore di N se esiste, nil altrimenti

if $N.right \neq nil$ **then**

$\quad$ **return** $\textsc{Tree-Min}(N.right)$

else $\quad\triangleright$ il successore è l'avo più vicino con etichetta maggiore

$\quad P \leftarrow N.parent$

$\quad$ **while** $P \neq nil$ **and** $N = P.right$ **do**

$\quad\quad N \leftarrow P$

$\quad\quad P \leftarrow N.parent$

$\quad$ **end while**

$\quad$ **return** P

end if

Inserimento di un nodo

L'inserimento in un **BST** avviene sempre al livello delle foglie sostituendo un sottoalbero vuoto in modo che rimanga un **BST**

```
TREE-INSERT(N, T)
    ▷ pre: N nuovo nodo con N.left = N.right = nil, T è un albero binario di ricerca
    ▷ post: N è un nodo di T, T è un albero binario di ricerca
P ← nil
S ← T
while S ≠ nil do        ▷ inv: se P ≠ nil allora P è il padre di S
    P ← S
    if N.key = S.key then
        return
    else
        if N.key < S.key then
            S ← S.left
        else
            S ← S.right
        end if
    end if
end while
N.parent ← P
if P = nil then
    T ← N
else
    if N.key < P.key then
        P.left ← N
    else
        P.right ← N
    end if
end if
```

Cancellazione di un nodo

Possiamo suddividere il problema in 3 casistiche:

1. Il nodo da eliminare **non ha** figli:
   Basta settare a nil il riferimento che punta a suo padre

2. Il nodo da eliminare **ha** esattamente **un** figlio:
   Basta agganciare il sottoalbero esistente al padre del nodo che

vogliamo eliminare.

$\text{1-D}\textsc{elete}(Z, T)$

$\triangleright$ pre: Z nodo di T con esattamente un figlio

$\triangleright$ post: Z non è più un nodo di T

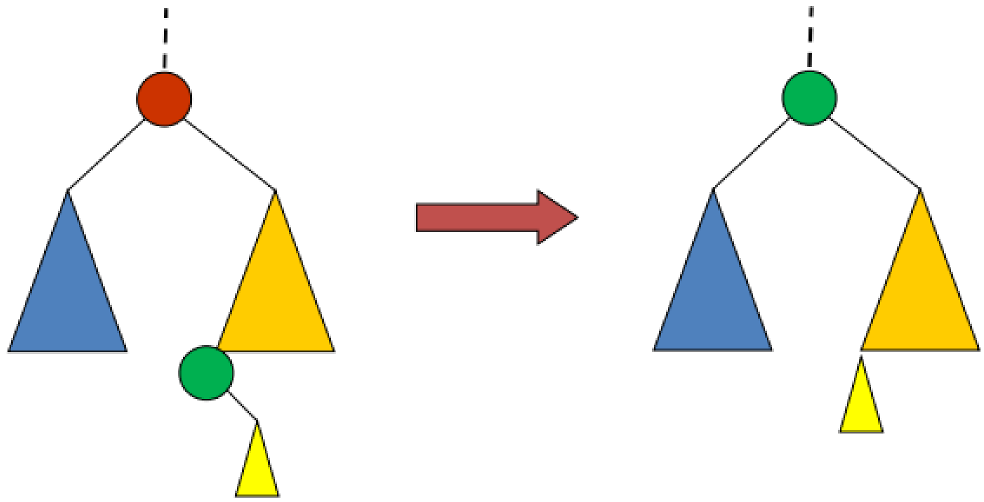
```
if Z = T then
    if Z.left ≠ nil then
        T ← Z.left
    else
        T ← Z.right
    end if
    Z.parent ← nil
else
    if Z.left ≠ nil then
        Z.left.parent ← Z.parent
        S ← Z.left
    else
        Z.right.parent ← Z.parent
        S ← Z.right
    end if
    if Z.parent.right = Z then
        Z.parent.right ← S
    else
        Z.parent.left ← S
    end if
end if
```

3. Il nodo da eliminare **ha due** figli:

il nodo da eliminare è il nodo rosso

il minimo del suo sottoalbero destro (nodo verde) ha un figlio al massimo

si può eliminare il nodo verde e copiare la sua etichetta nel nodo rosso

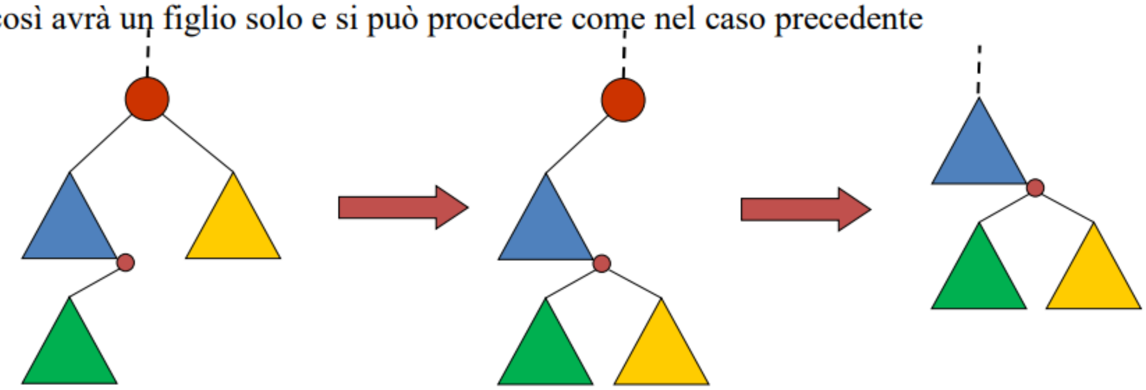


oppure

il nodo da eliminare è il nodo rosso grande

il suo sottoalbero destro (triangolo giallo) può essere agganciato come sottoalbero destro al massimo del suo sottoalbero sinistro (nodo rosso piccolo)

così avrà un figlio solo e si può procedere come nel caso precedente



Algoritmo che riassume tutte e tre le casistiche:

```
TREE-DELETE(Z, T)
    ▷ pre: Z nodo di T
    ▷ post: Z non è più un nodo di T
if Z.left = nil ∧ Z.right = nil then        ▷ Z è una foglia
    if Z = T then
        T ← nil
    else
        if Z.parent.left = Z then        ▷ Z è figlio sinistro
            Z.parent.left ← nil
        else        ▷ Z è figlio destro
            Z.parent.right ← nil
        end if
    end if
else
    if Z.left = nil ∨ Z.right = nil then
        1-DELETE(Z, T)
    else        ▷ Z ha due figli e dunque si può cercare il minimo in Z.right
        Y ← TREE-MIN(Z.right)
        Z.key ← Y.key
        TREE-DELETE(Y, T)
    end if
end if
```

Salvare un BST in una lista

In modo simile alla stampa di etichette di un **BST**, possiamo facilmente trasformare un **BST** in una lista.

```
TOLIST-INORDER(T)
    ▷ pre: T binario di ricerca
    ▷ post: ritorna la lista ordinata delle chiavi in T
if T = nil then
    return nil
else
    L ← TOLIST-INORDER(T.left)
    R ← TOLIST-INORDER(T.right)
    R ← LISTINSERT(T.key, R)
    return APPEND(L, R)
end if
```

Si crea due liste L e R contenenti, rispettivamente, gli elementi del

sottoalbero sinistro e destro.
Questi vengono poi fusi attraverso la funzione append.
Per via di Append, ha una complessità quadratica.

Se visitiamo le etichette in modo decrescente, possiamo ottenere però un algortimo lineare.

$\textsc{ToList-Inorder}(T, L)$
    ▷ pre: T binario di ricerca
    ▷ post: ritorna la lista ordinata delle chiavi in T concatenata con L
if $T = nil$ **then**
    return L
else
    $L \leftarrow \textsc{ToList-Inorder}(T.right, L)$
    $L \leftarrow \textsc{ListInsert}(T.key, L)$
    return $\textsc{ToList-Inorder}(T.left, L)$
end if