UNIVERSITÀ degli STUDI di CATANIA

# Scaplegoat tree

Giulia Meo

May 2021

## Indice

# 1 Introduzione alla struttura dati

## 1.1 Che cose un albero binario di ricerca?

Un albero binario di ricerca è un particolare tipo di struttura dati che tramite un' organizzazione dei nodi gerarchica permette di effettuare in maniera efficiente diverse operazioni su insiemi dinamici di dati. La struttura è composta da nodi che sono caratterizzati da: una chiave dell'elemento che rappresentano, un puntatore al nodo situato alla loro destra, un puntatore al nodo sinistro ed un puntatore al nodo padre.
Rispetto ai diversi nodi della struttura si evidenziano due nodi caratteristici: la **root** che è il primo nodo dell'albero ed è inoltre l'unico nodo il cui padre è NIL e le **foglie** che costituiscono i nodi finali dell'albero ed hanno come figlio destro e sinistro dei puntatori a NIL.

La struttura dell'albero viene caratterizzata dalle seguenti proprietà:

- Il sottoalbero destro e il sottoalbero sinistro devono essere entrambi due alberi binari di ricerca;
- Il sottoalbero sinistro di un nodo x contiene soltanto i nodi con chiavi minori della chiave del nodo x;
- Il sottoalbero destro di un nodo x contiene soltanto i nodi con chiavi maggiori della chiave del nodo x;

### 1.1.1 Visite in un albero binario di ricerca:

Con il termine "Visitare" intendiamo l'operazione che esamina sequenzialmente tutti i nodi della struttura.
Si possono effettuare tre tipologie di visite nei nodi dell'albero:

- Inorder = visita in ordine simmetrico nella quale si prosegue lungo un cammino di nodi nel sottoalbero sinistro, quindi si esamina la radice e infine si visita il sottoalbero destro;
- Preorder = visita in ordine anticipato nella quale si esamina prima la radice e quindi si visitano il sottoalbero sinistro e quello destro;
- Postorder = visita in ordine posticipato nella quale si prosegue lungo un cammino di nodi prima il sottoalbero sinistro poi quello destro e infine si esamina la radice.

### 1.1.2 Altezza e profondità di un albero binario di ricerca:

In un albero binario, la **profondità** di un nodo è espressa dalla lunghezza del cammino dalla radice al nodo stesso cioè il numero di archi che intercorrono tra la radice e il nodo. La profondità massima di un nodo all'interno di un albero è detta **altezza** dell'albero e viene calcolata a partire dalla root considerando che essa ha profondità pari a 0.

### 1.1.3 Bilanciamento di un albero binario di ricerca:

L'ordine di inserimento dei nodi all'interno dell'albero risulta rilevante per la costruzione dello stesso, il diverso inserimento dei nodi porta l'albero ad assumere diverse forme avendo un impatto sulle performance della struttura dati. Il **bilanciamento** si presenta quando tutti i nodi sono distribuiti equamente all'interno dell'albero e assicura lo svolgimento delle varie operazioni con una complessità pari a O (log n ). Tale complessità viene calcolata in base all'altezza dell'albero e nel caso in cui l'albero fosse sbilanciato potrebbe nel caso peggiore essere pari a O(n). E quindi importante mantenere la struttura dati il più bilanciata possibile, esistono varie strutture permettono un corretto bilanciamento, tra queste le principali sono gli alberi rosso neri, gli AVL tree e gli scaplegoat-tree che analizziamo in questo documento.

## 1.2 Che cose lo scaple goat tree?

Un albero del capro espiatorio è un albero binario di ricerca autobilanciato, inventato da Arne Andersson e da Igal Galperin e Ronald L. Rivest. Fornisce caso peggiore O(log n) come tempo di ricerca e O (log n) ammortizzato per l'inserimento e la cancellazione. A differenza della maggior parte degli altri alberi di ricerca binaria autobilanciati gli alberi del capro espiatorio non hanno alcun sovraccarico di memoria aggiuntivo per i nodi sviluppando le caratteristiche di un albero di ricerca binario standard in cui un nodo memorizza solo una chiave e due puntatori al nodi figli. Ciò rende gli alberi del capro espiatorio più facili da implementare e di minori dimensioni.
Invece delle piccole operazioni di ribilanciamento incrementale utilizzate da molti degli altri algoritmi di altre strutture come gli alberi rossoneri e gli AVL tree, gli alberi del capro espiatorio scelgono raramente ma in modo costoso un nodo da rendere "**capro espiatorio**" e partendo da esso ricostruiscono completamente la sottostruttura radicata nel nodo, ripristinando il bilanciamento all'interno dell'albero.
Si dice inoltre che un albero di ricerca binario è bilanciato in peso se metà dei nodi si trova a sinistra della radice e metà a destra. Un nodo di bilanciamento del peso $\alpha$ è definito come **fattore di bilanciamento** e detemina la conformazione dell'albero rispetto al criterio di bilanciamento.
Per soddisfare il criterio di equilibrio del peso rilassato si ha:

- $size(left) <= \alpha * size(node)$
- $size(right) <= \alpha * size(node)$

La scelta di $\alpha$ è decisiva per la struttura infatti anche un albero degenere come una lista concatenata potrebbe soddisfare questa condizione se $\alpha$ = 1, mentre scegliendo un $\alpha$ = 0,5 si avrebbe una corrispondenza con alberi binari quasi completi.

### 1.2.1 Notazioni:

Ciascun nodo della struttura è identificato da:

- key[x] = la chiave immagazzinata nel nodo
- left[x] = il puntatore al figlio sinistro del nodo
- right[x] = il puntatore al figlio destro del nodo
- parent[x] = il puntatore al padre del nodo

A livello generale nella struttura dati vengono immagazzinati i seguenti campi:

- root[T] = un puntatore alla root dell'albero;
- nodeNumber[T] = il numero di nodi contenuti all'interno dell'albero, viene calcolata con size(root[T]);
- maxSize = il massimo valore di size[T] dall'ultima volta che l'albero è stato completamente ricostruito, questo valore è necessario per svolgere le operazioni di cancellazione;

Definiamo inoltre le notazioni necessarie per la definizione e l'utilizzo della struttura dati:

- size (x) = la dimensione del sottoalbero radicato in x, dove per dimensione intendiamo il numero di nodi che fanno parte dell'albero che ha come radice il nodo x;
- brother (x) = l'altro figlio del padre del nodo x;
- height (x) = l'altezza del nodo x, cioè la lunghezza del percorso piu lungo dal nodo stesso sino ad una foglia, dove l'altezza dell'albero è definita dall'altezza della root;
- depth (x) = la profondità del nodo x, il numero di archi che si devono attraversare per giungere dalla root sino al nodo, dove la root ha quindi profondità 0;

### 1.2.2 Proprietà dello scaple goat tree:

Lo scapegoat-tree possiede diverse propietà che devono essere sempre rispettare per garantirne il bilanciamento. Vi sono infatti dei vincoli sull'altezza rispetto al numero dei nodi dell'albero infatti se l'albero ha un'altezza che è attualmente troppo alta rispetto le sue dimensioni, significa che esso è sbilanciato.
Definiamo come **nodeNumber** il numero di nodi contenuti all'interno dell'albero e come **MaxNode** il limite superiore per il numero di elementi, identificheremo questi valori con **n** e **q** per facilitarne la leggibilità.
Adoperiamo n e q come dei contatori che vengono incrementati quando dei nuovi elementi vengono aggiunti all'albero, inoltre il valore di n viene decrementato di 1 quando un elemento è rimosso dall'albero poiche il numero di nodi è decrementato. Ed effettuamo inoltre dei controlli sul valore di q per gestire attentamente le cancellazioni ed evitare che l'albero diventi sbilanciato a seguito di eccessive chiamate alla procedura di cancellazione dei nodi.
Quando n diventa minore di q/2 a seguito di diverse cancellazioni, l'albero subisce una ricostruzione totale per mantenere il bilanciamento.
Una **ricostruzione totale** è un operazione in cui viene ricostruito un nuovo albero totalmente bilanciato partendo dai nodi della struttura e distribuendoli da zero scegliendo una root appropiata e distribuendo i figli in modo bilanciato.

Dalla premessa possiamo quindi dedurre che:

$$q/2 < n < q$$

Diciamo inoltre che il nostro albero è bilanciato in altezza se:

$$h\alpha(n) = \log_{1/\alpha} n$$

Inoltre la massima altezza permissibile all'interno della struttura dati è definita da

$$\log_{1/\alpha} q \leq log_{1/\alpha} 2n < log_{1/\alpha} n + 2$$

con:

$$1/2 < \alpha < 1$$

Nel nostro caso in cui abbiamo scelto per la struttura dati $\alpha$ = 2/3, procediamo quindi sostituendo alpha con 2/3 all'interno della formula ottenendo:

$$\log_{3/2} q \leq log_{3/2} 2n < log_{3/2} n + 2$$

L'altezza massima dello scaplegoat-tree attualmende nel nostro caso di studi corrisponde a: $\log_{3/2} q$.
All'interno della struttura, quando essa risulta sbilanciata è possibile sempre identificare un **nodo del capro espiatorio** tramite la seguente formula:

$$\frac{\mathrm{size}(w.child)}{\mathrm{size}(w)} > \frac{2}{3}$$

## 1.3 Operazioni all'interno dello scaplegoat tree:

Nella struttura possono essere svolte le seguenti operazioni:

### 1.3.1 Operazione di inserimento:

L'inserimento di un nodo in un albero di capro espiatorio avviene nella fase iniziale in modo simmetrico ad un normale albero di ricerca binario. Successivamente si esegue un passaggio aggiuntivo in cui vengono incrementate le due variabili nodeNumber e maxNode. Inolte si calcola la profondità del nodo e si controlla se essa è maggiore di $log3/2n$ dove n è il numero di nodi nell'albero, nel caso in cui la profondita superi quel valore sarà necessario cercare lo scaple goat node e rendere l'albero bilanciato. Per ripristinare il bilanciamento ci spostiamo all'interno dell'albero a partire dal nodo appena inserito finche non andiamo incontro ad un nodo w per cui $size(w) > (2/3) * size(w.parent)$ tale nodo sarà considerato l'attuale scapegoat node. Da qui ricostruiamo il sottoalbero che ha come radice il padre di w.

### 1.3.2 Operazione di cancellazione:

Per l'operazione di cancellazione procediamo inizialmente eliminando il nodo come in uno standard BST, decremendiamo quindi nodeNumber. Successivamente procediamo con un confronto ta nodeNumber e maxSize ed in caso di necessità ricostruiamo l'albero da zero.

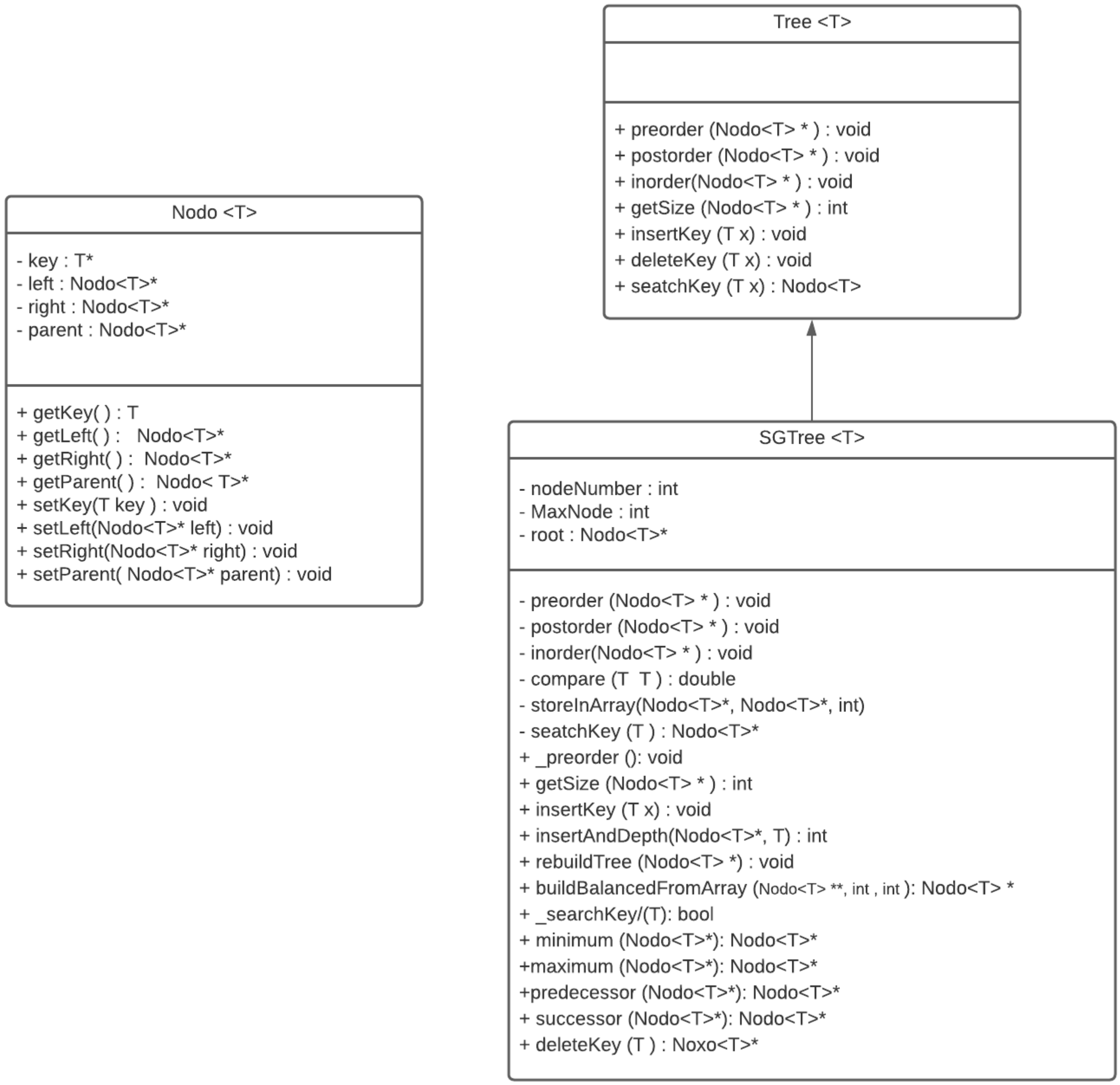
### 1.3.3 Operazione di ricostruzione:

Durante la ricostruzione, convertiamo l'albero in un BST bilanciato il piu possibile. Procediamo con una visita inorder memorizzando i dati all'interno di un array e successivamente costruiamo un nuovo BST partendo dall'array dividendolo ricorsivamente in due metà.

### 1.3.4 Operazione di ricerca:

L'operazione di ricerca è la medesima di un classico BST, procediamo dal nodo root spostandoci a destra e sinistra tra i nodi dell'albero in base alla nostra chiave procedendo alla ricerca del nodo da cercato.

# 2 Spiegazione dei metodi con relativo pseudocodice e costi computazionali



## 2.1 Metodi relativi all'inserimento

**Algorithm 1:** insertKey(T x)

```
Result: Aggiunta di un nuovo nodo all'interno della struttura contente la chiave ricevuta in input
Nodo newNode = new Nodo (key);
MaxNode++ ;
depth = InsertAndDepth(newNode, key) ;
if depth > log32(n) then
    p = node.parent ;
    while 3*size(p) <= 2*size(p.parent) do
        p = p.parent ;
        rebuildTree(p.parent) ;
    end
end
return depth >= 0 ;
```

**Descrizione metodo insertKey:**
Metodo che consente l'inserimento di nuovi elementi all'interno dell'albero. Viene instanziato un nuovo nodo contenente la chiave e successivamente viene richiamato il metodo InsertAndDepth che svolge l'operazione di inserimento restituendo il valore della profondità in cui si trova il nuovo nodo. Successivamente viene eseguito un controllo per assicurarsi che l'albero non sia diventato sbilanciato a seguito del nuovo inserimento. Viene quindi verificato che depth non sia maggiore del log3/2(MaxNode). In caso affermativo l'inserimento termina ed in caso contratio si procede alla ricerca dello Scaplegoat-node partendo dal padre del nodo appena inserito e continuando a salire all'interno della struttura sino a che:
$3 * size(p) <= 2 * size(p.Parent())$
Successivamente da quel nodo si procede con la ricostruzione dell'albero tramite il metodo rebuildTree.

**Complessita computazionale:**
L'inserimento ha un tempo di esecuzione ammortizzato pari a O (log n).

**Algorithm 2:** InsertAndDepth (newNode,key)

```
Result: inserimento di un nodo secondo una standard BSTinsert e restituzione della profondità del nodo
depth = 0;
Nodo tmp = root;
Nodo parent = NULL;
nodeNumber++;
while tmp != NULL do
    parent = tmp;
    if key <= tmp.Key then
        tmp = tmp.Left;
    else
        tmp = tmp.Right;
    end
    depth++;
end
if parent == NULL then
    root = newNode;
    return depth;
end
if key <= parent.Key then
    parent.Left = newNode;
else
    parent.Right = newNode;
end
x.Parent = parent;
return depth;
```

**Descrizione metodo InsertAndDepth:**

La procedura si occupa di inserire un nodo nell'albero seguendo i medesimi passaggi svolti da un classico albero binario di ricerca. Partendo dalla root ed effettuando gli opportuni confronti il metodo si sposta a destra o a sinistra fra i nodi dell'albero per cercare la giusta posizione da assegnare alla nuova chiave.

Inoltre durante l'inserimento incrementa in valore di una variabile per misurare la profondita del nuovo nodo e restituirla in output alla funzione chiamante.

**Complessita computazionale:**

Il metodo scorre l'array sino a trovare la posizione corretta del nuovo nodo inserendolo come una nuova foglia. La complessità tenendo conto che lo scaplegoat-tree mantiene i suoi nodi bilanciati è nel caso peggiore O(log n).

**Algorithm 3:** rebuildTree(Nodo* node)

```
Result: Ricostruzione dell'albero partendo dal nodo ricevuto come parametro
int numNode = size(node);
Nodo parentNode = node.parent;
Nodo arrayNode = new Nodo [numNode];
storeInArray(node, arrayNode, 0);
if parentNode == NULL then
    root = buildBalancedFromArray(arrayNode, 0, numNode);
    root.Parent = NULL;
else
    if parentNode.right == node then
        parentNode.right = buildBalancedFromArray(arrayNode, 0, numNode) ;
        Nodo tp = parentNode;
        parentNode = parentNode.right;
        parentNode.Parent = parentNode ;
    else
        parentNode.left = buildBalancedFromArray(arrayNode, 0, numNode);
        Nodo tp = parentNode;
        parentNode = parentNode.left;
        parentNode.parent = parentNode;
    end
end
```

**Descrizione metodo rebuildTree:** Funzione che permette di ricostruire parte della struttura o l'intero albero a partire dal nodo in input. La funzione richiama il metodo storeInArray() per salvare in un vettore il risultato della visita inorder effettuata sull'albero partendo dal nodo scelto come scaplegoat-node. Successivamente il metodo converte l'array nel BST piu bilanciato possibile utilizzando la procedura builBalancedFromArray().
Il metodo esegue inoltre tre differenti controlli:

- Nel caso in cui il genitore è NULL, vuol dire che il nodo in questione è la root dell'albero per cui viene risettato il valore della root durante la chiamata buildBalancedFromArray portandono al nuovo valore risultante;
- Se il nodo è un figlio destro, settiamo come nuovo figlio destro il risultato della chiamata buildBalancedFromArray e sistemiamo le relazioni di parentela;
- Se il nodo è un figlio sinistro, settiamo come nuovo figlio sinistro il risultato della chiamata buildBalancedFromArray e sistemiamo le relazioni di parentela.

**Complessita computazionale:** La complessità computazionale è nel caso peggiore O(n).

**Algorithm 4:** storeInArray(Nodo node, Nodo array[], i)

```
Result: Visita inorder effettuata sull'albero e immagazzinata all'interno di un array
if node == NULL then
    return i;
end
i = storeInArray(node.left, array, i);
arr[i++] = node;
return storeInArray(node.right, array, i);
```

**Descrizione metodo storeInArray:**
Questa funzione salva all'interno di un array il risultato di una visita inorder effettuata sull'albero che ha come radice il nodo ricevuto in input. E un metodo ricorsivo che si sposta all'interno dell'albero seguendo i passi di una visita inorder.

**Complessita computazionale:**
La complessità di questo medoto corrisponde a quella della visita inorder che è pari ad O(n).

**Algorithm 5:** buildBalancedFromArray(Nodo a, int i, int n)

```
Result: Un albero binario con i nodi bilanciati
if nodeNumber== 0 then
    return NULL;
end
median = nodeNumber / 2;
array[i+median].Left = buildBalancedFromArray(array, i, median) ;
if array[i+median].Left!= NULL) then
    Nodo a1 = array[i+media].Left;
    a1.Parent = array[i+median];
end
array[i+median].Right = buildBalancedFromArray(array, i+median+1, nodeNumber-median-1);
if array[i+median].Right != NULL then
    Nodo a1 = array[i+median].Right;
    a1.Parent = array[i+median];
end
return array[i+median];
```

**Descrizione metodo buildBalancedFromArray:**

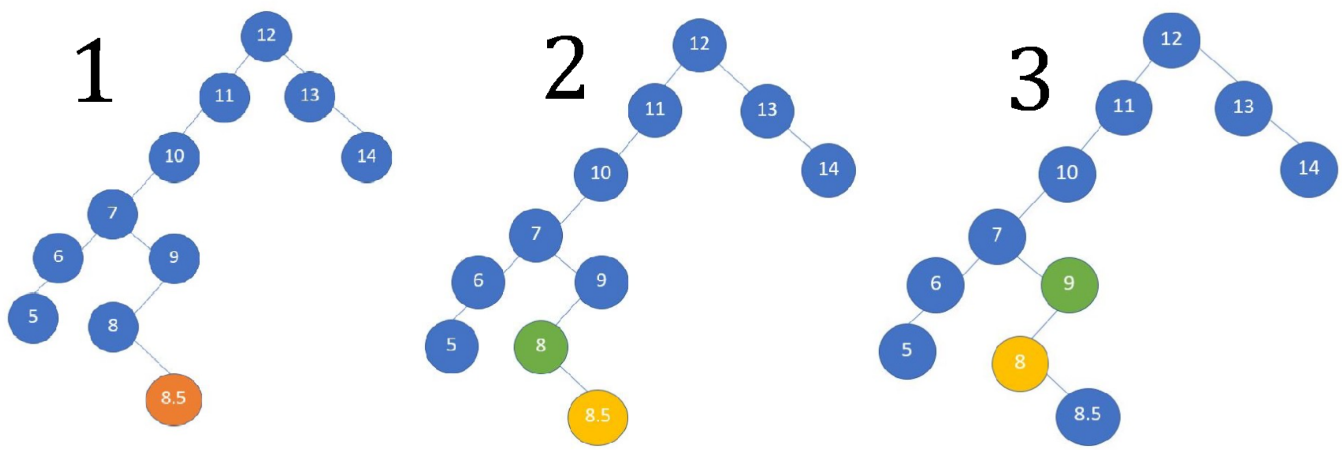
Funzione che ricostruisce l'albero bilanciando il piu possibile i suoi nodi. Il metodo riceve in input un vettore contenente i nodi ordinati secondo una visita inorder, il nodo da cui far partire la ricostruzione e la dimensione del vettore. Nel caso in cui il vettore sia vuoto la funzione restituisce NULL. Altrimenti seleziona il nodo che si trova a metà dell'array e lo rende la nuova root dell'albero o del sottoalbero, successivamente esegue diverse chiamate ricorsive per selezionare i figli sinistri e successivamente i figli destri del nuovo sottoalbero.

**Complessita computazionale:**

Procedura ricorsiva che nel caso peggiore ha una complessità pari ad O(n), consideriamo che andiamo ottendo il mediano con un O(1) ed eseguiamo la procedura n volte rispetto agli n nodi che compongono l'albero.

## 2.2 Esempio di inserimento all'interno dell'albero

L'esempio in figura presenta uno scaplegoat-tree in cui sono stati già inseriti i nodi già presenti nella truttura tramite l'operazione insertkey(), tali nodi non hanno portato ad uno sbilanciamento dell'albero per cui l'inserimento è stato processato in modo analogo ad un normale BST.




**Passo 1)** Il nodo appena inserito è il nodo arancione con chiave 8.5, tramite la procedura insertKey() esso ha trovato la sua posizione all'interno dell'albero come figlio destro del nodo 8.
Attualmente i dati a nostra disposizione sono:

- La profondità del nuovo nodo che è uguale a 6;
- Il maxNode al momento dell'inserimento pari ad 11 ;

A questo spunto Calcolando il log3/2 di 11 otteniamo il valore 5,9 che è inferiore alla profondità del nuovo nodo equivalente a 6, ci accorgiamo quindi che il nostro albero risulta sbilanciato e che dovremo procedere risalendo l'albero a partire dal nodo arancione fino allo scaplegoat-node.

**Passo 2)** Ci spostiamo quindi all'interno dell'albero nel nodo verde con chiave 8 e controlliamo se esso è o meno lo scaplegoat-node. Quinidi procediamo applicando la formula:

$$\frac{\mathtt{size(w.child)}}{\mathtt{size(w)}} > \frac{2}{3}$$

Dove w è il nodo contenente la chiave 8 e w.child è il nodo figlio contentente la chiave 8.5. Calcoliamo quindi:

- Size(w) = 2;
- Size(w.child) = 1;

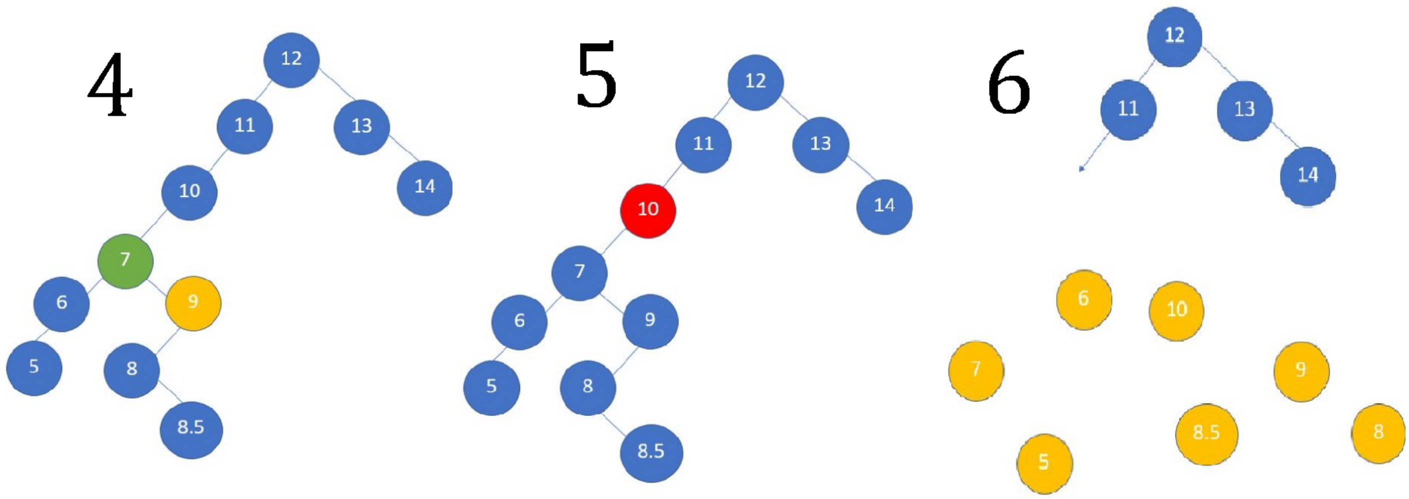
Ed il ristultato finale è 1/2 che risulta inferiore a 2/3 per cui il nodo attualmente analizzato non è uno scaplegoat-node. La nostra ricerca dovrà quindi proseguire all'interno dell'albero spostandoci ancora verso la root.

**Passo 3)**Analizziamo l'attuale nodo verde che contiene la chiave 9, per cui:

- Size(w) = 3;

- Size(w.child) = 2;

Applicando la formula otteniamo 2/3 che non risulta maggio a 2/3 pwr cui il nodo non è uno scaplegoat-node. La nostra ricerca dovrà quindi proseguire salendo all'interno dell'albero.




**Passo 4)** Il nuovo nodo sospetto è il nodo con chiave 7. Per questo nodo:

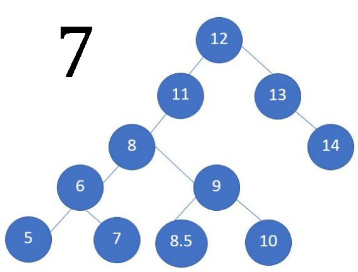
- Size(w) = 6;
- Size(w.child) = 3;

Il ristultato è 1/2 che risulta ancora inferiore a 2/3 per cui il nodo non è uno scaplegoat-node. La nostra ricerca dovrà quindi continuare all'interno dell'albero.

**Passo 5)** L'attuale nodo esaminato è il nodo con chiave 10. Per questo nodo:

- Size(w) = 7;
- Size(w.child) = 6;

Il ristultato ottenuto è 6/7 che risulta maggiore di 2/3, per cui esso è il nostro scaplegoat-node.
c **Passo 6)** Trovato lo scaplegoat-node viene operata un' operazione di ricostruzione parziale che porterà l'albero nella forma finale totalmente bilanciata, osservabile nell'immagine 7.



## 2.3 Metodi relativi alla ricerca

**Algorithm 6:** searchKey(T key)

**Result:** Viene restituito se riscontrato il nodo contenente la chiave, altrimenti viene restituito l'ultimo nodo appartenente al percorso

```
tmp = root ;
while tmp != NULL AND compare(key, tmp.getKey()) != 0 do
    if compare(key, tmp.getKey() ¡ 0 then
        tmp = tmp.left;
    else
        tmp = tmp.right;
    end
end
return tmp;
```

**Descrizione metodo searchKey:**
Il metodo presa in input una chiave controlla partendo dalla root e spostandosi all'interno dell'albero tramite una serie di confronti, se la chiave è presente oppure assente rispetto alla struttura.

**Complessita computazionale:**
Ogni operazione di ricerca svolta in uno Scapegoat tree ha nel caso peggiore una complessità di O(log n).

## 2.4 Metodi relativi alla cancellazione

**Algorithm 7:** successor(Nodo node))

```
Result: Viene restituito se presente il successore del nodo all'interno della struttura
if node == NULL then
    return node;
end
if node.Right != NULL then
    return minimum(node.Right);
end
Nodo tmp = node.Parent;
while tmp != NULL  node == tmp.Right do
    node = tmp;
    tmp = tmp.Parent;
end
return tmp;
```

**Descrizione metodo successor:**
Il metodo ricevuto in input un nodo controlla se esso appartiene o meno alla struttura e in caso positivo cerca il successore cercando il minimo dei nodi sulla destra se presente un figlio destro o altrimenti partendo dal genitore del nodo e proseguendo alla ricerca del successore.

**Complessita computazionale:**
La complessità computazionale nel caso peggiore di questo medoto corriisponde ad O(log n) poiche tale valore rappresenta l'altezza massima della struttura.

**Algorithm 8:** minimum(Nodo node)

```
Result: il piu piccolo dei nodi a partire dall'input
if node == NULL then
    return node;
end
Nodo tmp = node;
while tmp.Left do
    tmp = tmp.Left;
end
return tmp;
```

**Descrizione metodo minimum:**
Il metodo partendo dal nodo ricevuto in input cerca il piu piccolo nodo appartenente all'albero che ha come root il nodo ricevuto in input.

**Complessita computazionale:**
La complessità nel caso peggiore è pari ad O(log n) che corrisponde all'altezza massima raggiungibile dall'albero.

**Algorithm 9:** deleteKey(T key)

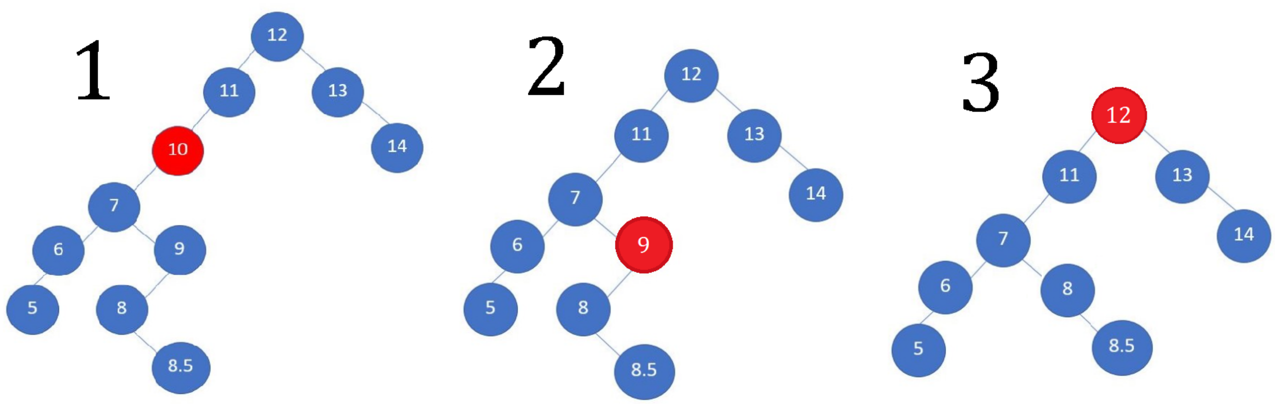
```
Result: Elimina se presente il nodo relativo alla chiave
Nodo node = searchKey(key);
if node == NULL then
    return NULL;
end
Nodo parent = node.Parent;
if node.Left == NULL then
    if parent != NULL then
        if node == parent.Left then
            parent.Left = node.Right ;
        else
            parent.Right = node.Right ;
        end
    else
        root = node.Right ;
    end
    if node.Right then
        node.Right.setParent = parent ;
    end
    return node ;
end
if node.Right == NULL then
    if parent != NULL then
        if node == parent.Left then
            parent.Left = node.Left ;
        else
            parent.Right = node.Left ;
        end
    else
        root = node.Left ;
    end
    node.Left.Parent = parent ;
    return node ;
end
Nodo successive = successor(node); deleteKey(successive.Key); node.Key = successive.Key ;
nodeNumber– ;
if nodeNumber ¡ MaxNode/2 then
    MaxNode = nodeNumber ;
    Nodo array = new Nodo [nodeNumber] ;
    root = buildBalancedFromArray(array, 0, nodeNumber) ;
    root.Parent = NULL ;
end
return successive;
```

**Descrizione metodo deleteKey:**
Il metodo deleteKey, opera una cancellazione standard come in un BST, con l'aggiunta nel caso l'albero diventi particolarmente sbilanciato di un operazione di ricostruzione totale della struttura.

**Complessita computazionale:**

La cancellazione nel caso pegiore ha una complessità di O(n) nel caso in cui debba essere ricorstruito l'intero albero, ma il tempo ammortizzato è di O(log n).

## 2.5 Esempio di cancellazione all'interno dell'albero




**Passo 1)** Procediamo con l'eliminazione del nodo 10, dopo l'eliminazione i valori dell'upper bound ed il numero di nodi corrispondono a:

- maxNode = 11;
- nodeNumber = 10;

Il numero di nodeNumber risulta ancora maggiore rispetto a maxNode/2 che equilave a 6.

**Passo 2)** Procediamo con l'eliminazione del nodo 9, dopo l'eliminazione i valori dell'upper bound ed il numero di nodi corrispondono a:

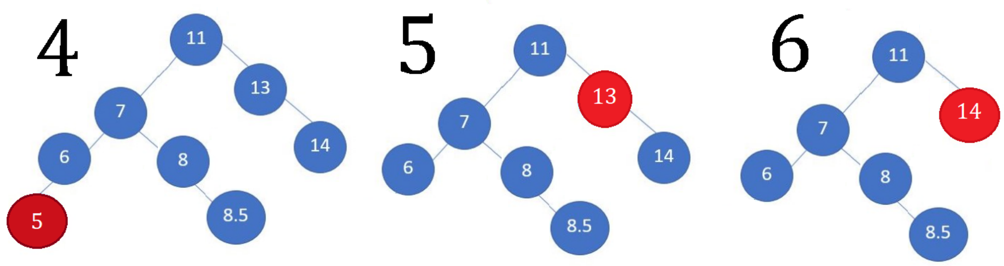
- maxNode = 11;
- nodeNumber = 9;

Il numero di nodeNumber risulta ancora maggiore rispetto a maxNode/2 che equilave a 6.

**Passo 3)** Procediamo con l'eliminazione del nodo 12, dopo l'eliminazione i valori dell'upper bound ed il numero di nodi corrispondono a:

- maxNode = 11;
- nodeNumber = 8;

Il numero di nodeNumber risulta ancora maggiore rispetto a maxNode/2 che equilave a 6.




**Passo 4)** Procediamo con l'eliminazione del nodo 5, dopo l'eliminazione i valori dell'upper bound ed il numero di nodi corrispondono a:

- maxNode = 11;
- nodeNumber = 7;

Il numero di nodeNumber risulta ancora maggiore rispetto a maxNode/2 che equilave a 6.

**Passo 5)** Procediamo con l'eliminazione del nodo 13, dopo l'eliminazione i valori dell'upper bound ed il numero di nodi corrispondono a:

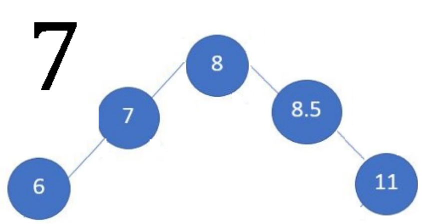
- maxNode = 11;
- nodeNumber = 6;

Il numero di nodeNumber risulta ancora maggiore rispetto a maxNode/2 che equilave a 6.

**Passo 6)** Procediamo con l'eliminazione del nodo 14, dopo l'eliminazione i valori dell'upper bound ed il numero di nodi corrispondono a:

- maxNode = 11;
- nodeNumber = 10;

Il numero di nodeNumber risulta adesso minore di maxNode/2, quindi oltre all'eliminazione standard del nodo procediamo con una ricostruzione totale della struttura.




**Passo 7)** Abbiamo ricostruito totalmente la struttura in modo bilanciato e riportato l'upper-bound al valore attuale:

- maxNode = 5;
- nodeNumber = 5;

Il ristultato è 1/2 che risulta ancora inferiore a 2/3 per cui il nodo non è uno scaplegoat-node. La nostra ricerca dovrà quindi continuare all'interno dell'albero.

## 2.6 Altri metodi:

**Algorithm 10:** log32(int n)

```
Result: Funzione per restituire il valore di log32(n)
double const log23 = 2.4663034623764317 ;
return (float)(log23 * log(n)) ;
```

**Descrizione metodo log32:**
Il metodo calcola è restituisce il logaritmo in base 2/3 del numero preso in imput;

**Complessita computazionale:**
La complessità computazionale è pari ad O(1),

**Algorithm 11:** getSize(Nodo node)

```
Result: Conta il numero di nodi appartenenti al sottoalbero radicato nel nodo
if node == NULL then
    return 0 ;
end
return 1 + getSize(node.getLeft()) + getSize(node.getRight()) ;
```

**Descrizione getSize:**
Il metodo calcola in modo ricorsivo la dimensione del’albero che ha come radice il nodo ricevuto in input.

**Complessita computazionale:**
La complessità nel caso peggiore è di O(n) nel caso in cui si sia ricevuta in input la root dell’intero albero.

# 3 Dimostrazioni oppure citazioni a libri che presentano le dimostrazioni

## 3.1 Tempo di esecuzione ammortizzato

In molte situazioni pratiche, ci troviamo a eseguire un algoritmo ripetutamente nel tempo. In contesti di questo tipo, è spesso utile analizzare le prestazioni di una operazione algoritmica in media su una sequenza di esecuzioni, piuttosto che su una singola esecuzione nel caso peggiore. Infatti, alcune operazioni potrebbero richiedere episodicamente molto tempo per una configurazione dei dati svantaggiosa, per poi essere piu veloci in seguito. Un'analisi puntuale nel caso peggiore darebbe risultati molto elevati, mentre un'analisi su una sequenza di esecuzioni rivelerebbe invece un buon comportamento in media. Questo tipo di analisi viene chiamata analisi ammortizzata e spesso caratterizza in modo piu preciso le prestazioni di una operazione algoritmica che viene effettuata ripetutamente nel tempo.

## 3.2 Teorema di Pat Morin's

Durante una chiamata al metodo inserisci(x) all'interno dello scaplegoat tree il costo per trovare il nodo del capro espiatorio w e ricostruire il sottoablero che ha w come root è o(size w). Il tempo di esecuzione dell'operazione di inserimento con ricostruzione è O (size (w)). Senza la ricostruzione, l'inserimento richiederà tempo O (log n), perché è una ricerca binaria standard per determinare la posizione dell'inserimento nell'albero e il tempo costante per eseguire l'inserimento. Il processo di verifica della presenza di un capro espiatorio non è banale in quanto implica molte chiamate alla funzione size, che comporta il conteggio dei nodi nella sottostruttura. Il motivo per cui la dimensione non è disponibile in un tempo costante e dato dal fatto che nella struttura non memorizziamo tali informazioni extra in nessun nodo. Nonostante ciò le operazioni ricostruzione dell'albero abbiano una complessità nel caso peggiore pari ad O(n) si presentano raramente all'interno della struttura, per cui non influiscono pesantemente sul calcolo finale della complessità.

## 3.3 Ricerca nello scaple goat tree

Ogni operazione di ricerca implementata nello scapelgoatTree ha una complessita nel caso peggiore di O(log n). Questo poiche lo scapeloat tree risulta vagamente alpha-bilanciato.

$$h(T) \le h_\alpha(n),\ h_\alpha(n) = \left\lfloor \log_{\left(\frac{1}{\alpha}\right)} n \right\rfloor$$

Considerando che:

- h(x) è l'altezza dell'albero x,
- T è l'albero in questione
- n è dato dalla size(T)
- alfa è un valore fisso in questo caso 2/3

Un operazione di ricerca possiede nel caso peggiore complessita di O(h $\alpha$(n)) = O(log n).

# 4 Approcci alle possibili implementazioni

Esistono due principali implementazioni degli alberi binari di ricerca:

**Rappresentazione dinamica tramite nodi linkati**
Tale approccio è quello scelto in questo progetto e la sua struttua è osservabile nell' UML sovrastante in cui distinguamo:

- La classe *nodo* che rappresenta un nodo dell'albero e che contenete una chiave, un figlio destro, un figlio sinistro ed un genitore;
- La classe *albero* che è un interfaccia di un classico albero da cui derivare le varie tipologie di albero a seconda delle nostre necessità;
- La classe *SGtree* che tramite l'istanziazione di nodi permette di costruire l'intera struttura dell'albero ed eseguire i diversi metodi esamitani in questo documento.

**Rappresentazione in un array sequenziale**
Un albero binario può essere implementato in un array, tale implementazione è osservabile negli heap binari in cui

- La radice dell'albero è memorizzata all'indice 1 dell'array;
- Il figlio destro di un nodo considerando che il padre è situato nella locazione i si trova a 2*i+1;
- il nodo figlio si troca a 2*i.

Tale struttura però è funzionale in un albero binario completo in cui tutti i livelli contengono il massimo di numeri possibili ad eccezzione dell'ultimo livello che può essere non pieno ma riempito sempre da sinistra verso destra. Nella nostra attuale struttura poiche non è possibile nonostante essa sia bilanciata garantire che l'albero sia completo si incorrerebbe in diversi problemi nello stabilire la gerarchia padre figlio dei nodi. Una possibile soluzione sarebbe quella di avere un array della stessa dimensione del numero massimo di nodi che l'albero puo contenere rispetto all' altezza e lasciare a NULL i nodi che non hanno un valore all'interno dell'albero. Tale rappresentazione rischia di lasciare numerosi campi vuoti all'interno della struttura. Inoltre con un array avremmo un numero limitato di nodi per cui sarebbe preferibile sfruttare una lista linkata per poter accedere ad un numero non finito di nodi.

Personalmente ho implementato lo scaplegoat-tree seguendo il paradigma di programmazione ad oggetti, definendo quindi due classi, una per i nodi e l'altra per l'intera sctuttura composta dai diversi nodi. La separazione di queste classi ha permesso di separare in modo chiaro i vari elementi che compongono la struttura permettendomi di instanziare i diversi nodi dell'albero e poter accedere nei diversi metodi al loro contenuto. Tale implentezione è la stessa vista durante le lezioni del corso ed è risultata la piu funzionale rispetto alle necessità richieste dall'a struttura.

Per a definizione di alcune procedure e per lo studio teorico ho preso spunto da: https://doc.lagout.org/Others/Data