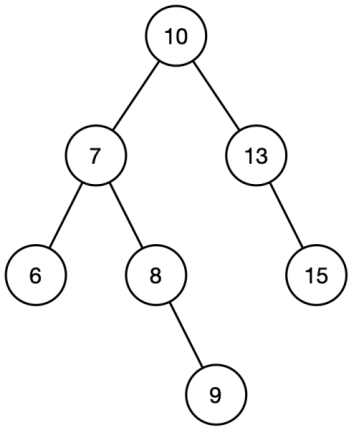
▼ 6.0 - Alberi

▼ 6.1 - Alberi binari di ricerca

> Gli **alberi binari di ricerca (BST - Binary Search Tree)** sono alberi binari radicati in cui ogni nodo presenta una chiave confrontabile e dati associati alla chiave. La peculiarità di tale albero è il fatto di avere, per ogni nodo v, nel sottoalbero sinistro chiavi $\leq v.key$ e nel sottoalbero destro chiavi $\geq v.key$. Tale proprietà consente di effettuare una **ricerca binaria** sull'albero, in modo da diminuire i costi computazionali delle operazioni basilari.



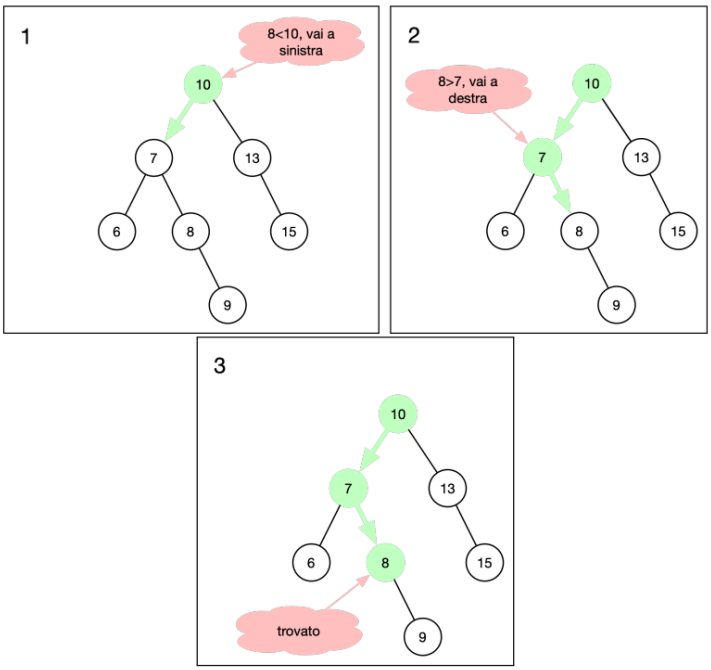
Esempio di albero binario di ricerca.

Le operazioni basilari di un BST sono le seguenti:

- **search(BST T, Key k)**: ritorna il nodo con chiave k in T, null se k non appare in T.
- **max(BST T)**: ritorna il nodo con chiave massima in T.
- **min(BST T)**: ritorna il nodo con chiave minima in T.
- **predecessor(BST T)**: ritorna il nodo avente la più grande chiave $\leq T.key$ se $T.key$ non è la chiave minima in T, altrimenti ritorna null.
- **successor(BST T)**: ritorna il nodo avente la più piccola chiave $\geq T.key$ se $T.key$ non è la chiave massima in T, altrimenti ritorna null.
- **insert(BST T, Key k, Data d)**: inserisce un nodo con chiave k e dati d in T.
- **delete(BST T, Key k)**: rimuove il nodo con chiave k in T.

Search

L'idea è quella di utilizzare la ricerca binaria.



Esempio di search in BST.

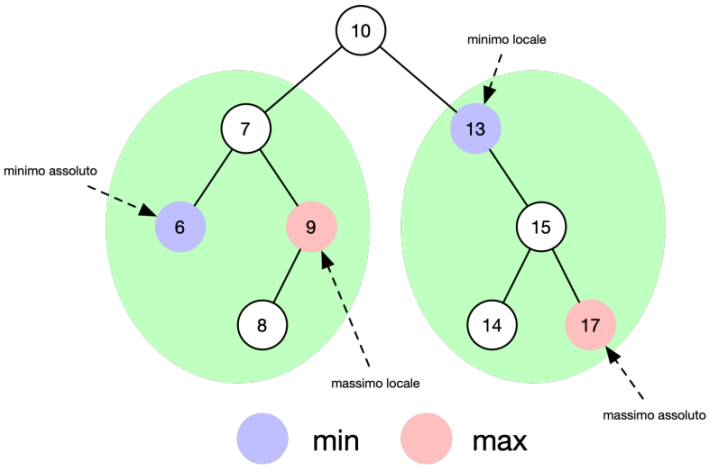
Lo **pseudocodice** di search è il seguente:

```
Node search(BST T, Key k) {
  tmp = T.root
  while (tmp != NIL) {
    if (k == tmp.key)
      return tmp
    else if (k < tmp.key)
      tmp = tmp.left
    else
      tmp = tmp.right
  return NIL
}
```

Il **costo computazionale** di search è il seguente:

- Caso **ottimo**, quando $k = T.key$: $O(1)$.
- Caso **pessimo**, quando k si trova nel livello h (h: altezza dell'albero): $O(h)$.

Max e min



Esempio di massimo/minimo locale/assoluto.

Dato un albero binario T, il nodo massimo è il nodo **più a destra** in T, mentre il nodo minimo è quello **più a sinistra** in T.

Lo **pseudocodice** di max e min è il seguente:

```
Node max(BST T) {
  while (T != null && T.right != null)
    T = T.right
  return T
}

Node min(BST T) {
  while (T != null && T.left != null)
    T = T.left
  return T
}
```

Il **costo computazionale** di max e min è il seguente:

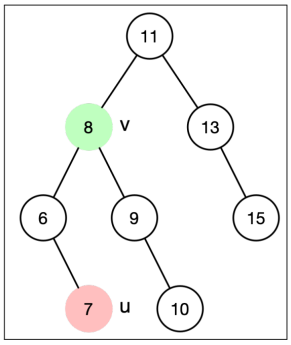
- Caso **ottimo**, quando, nel caso di max, T non ha il figlio destro, oppure, nel caso di min, T non ha il figlio sinistro: $O(1)$.
- Caso **pessimo**, quando il massimo/minimo si trova nel livello h (h: altezza dell'albero): $O(h)$.

Predecessor

Per calcolare il predecessore di un nodo all'interno di un albero binario bisogna distinguere due casi:

- **Caso 1: il nodo ha un figlio sinistro**.

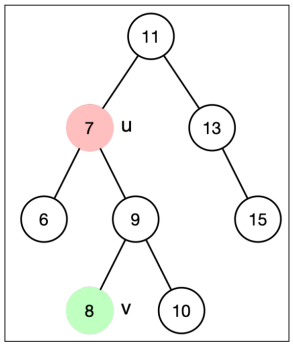
  Il predecessore è il nodo con chiave massima nel sottoalbero sinistro del nodo dato in input.



Esempio di predecessore nel caso 1 (v: nodo dato in input, u: predecessore).

- **Caso 2: il nodo non ha un figlio sinistro**.

  Il predecessore è il primo antenato tale che il nodo dato in input stia nel suo sottoalbero destro.



Esempio di predecessore nel caso 2 (v: nodo dato in input, u: predecessore).

Lo **pseudocodice** di predecessor è il seguente:

```
Node predecessor(Node T) {
  if (T == null)
    return NIL
  else if (T.left != null)
    // case 1
    return max(T.left)
  else {
    // case 2
    P = T.parent
    while (P != null && T == P.left) {
      T = P
      P = P.parent
    }
    return P
  }
}
```

Il **costo computazionale** di predecessor è il seguente:

- Caso **ottimo**, quando nel caso 1 il figlio sinistro è il massimo del sottoalbero che ha tale nodo come radice, oppure nel caso 2 quando il nodo corrente è un figlio destro: $O(1)$.
- Caso **pessimo**, quando nel caso 1 il nodo corrente è la radice e il predecessore si trova nel livello h (h: altezza dell'albero), oppure nel caso 2 quando il nodo corrente si trova nel livello h e l'unico sottoalbero destro a cui appartiene è quello della radice: $O(h)$.

Successor

L'idea per trovare il successore di un nodo è simmetrica a quella del predecessore, dunque lo **pseudocodice** è il seguente:

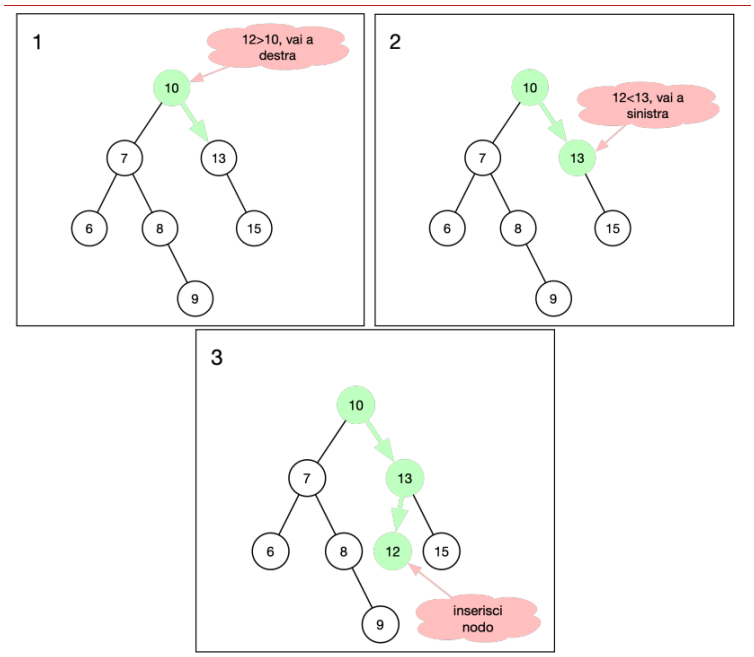
```
Node successor(Node T) {
  if (T == null)
    return NIL
  else if (T.right != null)
    // case 1
    return min(T.right)
  else {
    // case 2
    P = T.parent
    while (P != null && T == P.right) {
      T = P
      P = P.parent
    }
    return P
  }
}
```

Il **costo computazionale** di successor è il seguente:

- Caso **ottimo**, quando nel caso 1 il figlio destro è il minimo del sottoalbero che ha tale nodo come radice, oppure nel caso 2 quando il nodo corrente è un figlio sinistro: $O(1)$.
- Caso **pessimo**, quando nel caso 1 il nodo corrente è la radice e il successore si trova nel livello h (h: altezza dell'albero), oppure nel caso 2 quando il nodo corrente si trova nel livello h e l'unico sottoalbero sinistro a cui appartiene è quello della radice: $O(h)$.

Insert

L'idea è quella di cercare la posizione in cui inserire il nuovo nodo tramite ricerca binaria.



Esempio di insert in BST.

Lo **pseudocodice** di insert è il seguente:

```
void insert(BST T, Key k, Data d) {
  N = new Node(k, d)
```

```
  P = null
  S = T.root
  // search parent node
  while (S != null) {
    P = S
    if (k < S.key) S = S.left
    else S = S.right
  }
  // insert node
  if (P == null)
    T.root = N
  else {
    N.parent = P
    if (k < P.key) P.left = N
    else P.right = N
  }
}
```

Il **costo computazionale d**i insert è il seguente:

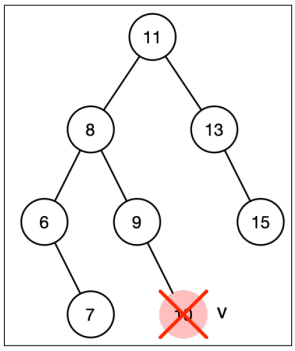
- Caso **ottimo**, quando il nuovo nodo deve essere inserito come figlio della radice o l'albero era vuoto: $O(1)$.
- Caso **pessimo**, quando il nuovo nodo deve essere inserito nel livello h (h: altezza dell'albero): $O(h)$.

Delete

Per rimuovere un nodo all'interno di un BST occorre distinguere tre casi:

- **Caso 1, il nodo da rimuovere è una foglia**.

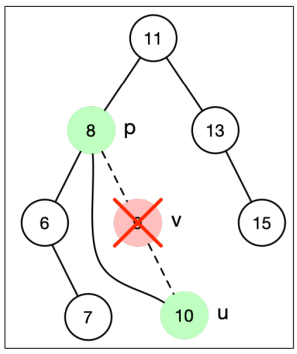
  Viene semplicemente rimosso il nodo da rimuovere.



Esempio di delete in un BST nel caso 1.

- **Caso 2, il nodo da rimuovere ha un solo figlio**.

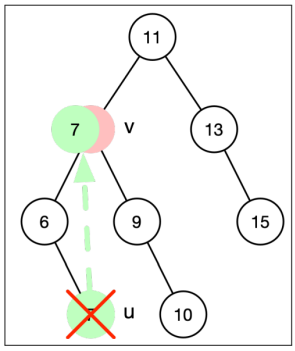
  Viene rimosso il nodo da rimuovere e l'unico figlio di tale nodo diventa figlio del padre del nodo da rimuovere.



Esempio di delete in un BST nel caso 2.

- **Caso 3, il nodo da rimuovere ha due figli**.

  Viene copiato il predecessore/successore del nodo da rimuovere al posto di quest'ultimo e viene rimosso tale predecessore/successore.



Esempio di delete in un BST nel caso 3.

Lo **pseudocodice** di delete è il seguente:

```
void delete(BST T, Key k) {
  v = search(T, k)
  if (v != null) {
    if (v.left == null || v.right == null)
      // case 1 or 2
      deleteNode(T, v)
    else {
      // case 3
      u = predecessor(v)
      v.key = u.key
      v.data = u.data
      deleteNode(T, u)
    }
  }
}

void deleteNode(BST T, Node v) {
  p = v.parent
  if (p != null) {
    if (v.left == null && v.right == null) {
      // case 1
      if (p.left == v)
        p.left = null
      else p.right = null
    } else if (v.right != null) {
      // case 2
      if (p.left == v)
        p.left = v.right
      else p.right = v.right
```

```
    } else if (v.left != null) {
      // case 2
      if (p.left == v)
        p.left = v.left
      else p.right = v.left
    }
  } else if (v.right != null)
    // case 2
    T.root = v.right
  else
    // case 1 or 2
    T.root = v.left
}
```

Il **costo computazionale** di delete è il seguente:

Notiamo che la funzione deleteNode ha sempre un costo $O(1)$.

- Caso **ottimo**, quando la ricerca e il predecessor si trovano nel caso ottimo: $O(1)$.
- Caso **pessimo**, quando la ricerca e il predecessor si trovano nel caso pessimo: $O(h)$(h: altezza dell'albero).

Caso medio

Notiamo che per tutte le funzioni basilari degli alberi BST il costo computazionale dipende dell'altezza h dell'albero. Tale altezza, per un albero BST con n nodi, può variare da un caso pessimo in cui $h = \Theta(n)$, quando l'albero binario è una lista, e un caso ottimo in cui $h = \Theta(\log n)$, quando l'albero binario è perfetto. È possibile però dimostrare che un BST costruito da n inserimenti casuali ha un altezza media $\overline{h} = O(\log n)$. Quindi il costo di tutte le funzioni basilari nel caso medio è $O(\log n)$.

Dizionario con albero binario di ricerca

Visto che un dizionario ha come operazioni basilari quelle di search, insert e delete, le quali sono già state analizzate per il caso degli alberi BST, possiamo pensare di poter implementare un dizionario tramite quest'ultima struttura dati. In questo modo otteniamo i seguenti costi computazionali confrontati agli altri tipi di implementazione.

	SEARCH		INSERT		DELETE	
	Medio	Pessimo	Medio	Pessimo	Medio	Pessimo
Array non ordinati	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$\Theta(n)$	$\Theta(n)$
Array ordinati	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Lista concatenata	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Albero Binario di Ricerca	$O(\overline{h})$	$O(h)$	$O(\overline{h})$	$O(h)$	$O(\overline{h})$	$O(h)$

Riassunto dei costi per le operazioni basilari delle diverse implementazioni di un dizionario.

▼ 6.2 - Alberi AVL

Abbiamo visto che l'utilizzo di alberi binari di ricerca consentono di effettuare operazioni di ricerca, inserimento e rimozione con costi $O(h)$, dove h è l'altezza dell'albero. Abbiamo inoltre visto che l'altezza di un albero è $\Omega(\log n)$ e $O(n)$. Il nostro obiettivo, per mantenere le operazioni basilari di un albero con un costo $O(\log n)$, è quello di creare una struttura dati ad albero bilanciato che non si sbilanci a seguito di operazioni di inserimento e rimozione di nodi.

Una struttura dati di questo tipo è un **albero AVL**, ovvero un albero binario di ricerca **perfettamente bilanciato**. Tale albero mantiene il suo bilanciamento in quanto viene automaticamente bilanciato in seguito ad inserimenti e rimozioni che causano sbilanciamento.

Nozioni preliminari

Fattore di bilanciamento

Il **fattore di bilanciamento** $\beta(v)$ di un nodo v è dato dalla differenza tra l'altezza del suo sottoalbero sinistro e destro:

$$\beta(v) = altezza(v.left) - altezza(v.right)$$

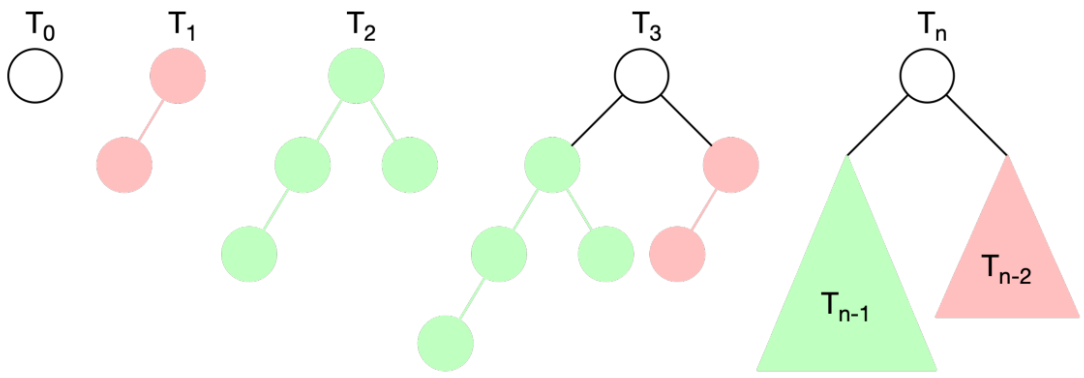
Bilanciamento in altezza

Un albero si dice **bilanciato in altezza** se per ogni suo nodo v le altezze dei suoi sottoalberi sinistro e destro differiscono al più di 1:

$$|\beta(v)| = 1$$

Altezza di un albero AVL

Siccome un albero AVL è un albero bilanciato in altezza, possiamo calcolare la sua altezza massima calcolando l'altezza dell'albero bilanciato con sbilanciamento massimo. È possibile costruire un tale albero utilizzando l'algoritmo di Fibonacci, e ottenendo così un albero, detto albero di Fibonacci, il quale per ogni nodo v non foglia $|\beta(v)| = 1$.



Costruzione dell'albero di Fibonacci.

Per calcolare il numero di nodi di un albero di Fibonacci utilizziamo il seguente teorema:

Il **numero di nodi di un albero di Fibonacci** con altezza h è equivalente a:

$$n_h = F_{h+3} - 1$$

(Dimostrazione a slide 9 del pacco di slide “Alberi AVL”).

Ricordando che $F_n = \Theta(\phi^n)$, dove $\phi \approx 1.618$, otteniamo dunque che $n_h = F_{h+3} - 1 = \Theta(\phi^{h+3}) = \Theta(\phi^h)$ e possiamo quindi concludere che $h = \Theta(\log n_h)$.

Poichè l’albero di Fibonacci è l’albero bilanciato con altezza massima possiamo concludere che l’**altezza di un albero AVL** è $\Theta(\log n)$.

Mantenere il bilanciamento

Per **mantenere il bilanciamento** su un albero AVL occorre modificare il codice delle funzioni insert e delete degli alberi binari, mentre la funzione search rimane invariata.

Tale obiettivo si raggiunge innanzitutto tenendo traccia delle altezze dei sottoalberi sinistro e destro di ogni nodo, al fine di calcolare poi il fattore di bilanciamento β. Tali operazioni possono essere effettuate utilizzando i seguenti pseudocodici, entrambi con costo $O(1)$:

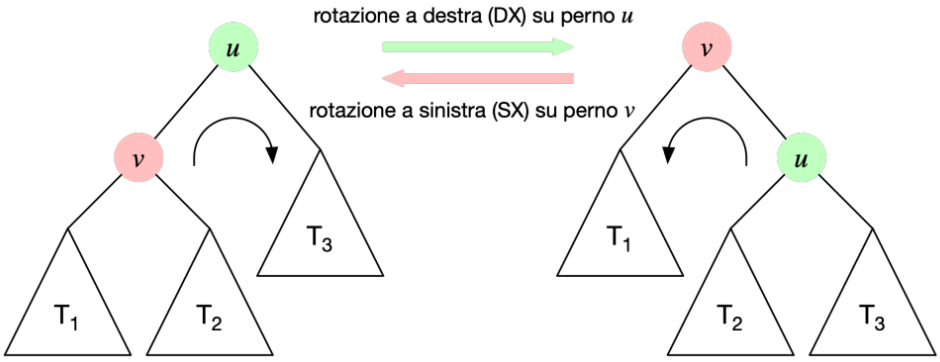
```
void updateHeight(Node T) {
  if (T != null) {
    lh = -1
    rh = −1
    if (T.left != null)
      lh = T.left.height
    if (T.right != null)
      rh = T.right.height
    T.height = max(lh, rh) + 1
  }
}

int β(Node T) {
  lh = -1
  rh = −1
  if (T.left != null)
    lh = T.left.height
  if (T.right != null)
    h = T.right.height
  return lh − rh
}
```

Per mantenere il bilanciamento degli alberi AVL occorre inoltre, a seguito di uno sbilanciamento causato da un’aggiunta/rimozione di un nodo dall’albero, effettuare delle rotazioni, introduciamo dunque tale concetto.

Rotazioni

Una **rotazione semplice**, la quale può essere effettuata sia a destra che a sinistra, è un’operazione fondamentale per ribilanciare un albero. Visualizziamo tale operazione nella seguente immagine:



Con tali rotazioni la proprietà di ordine di un BST viene preservata in quanto nelle rotazioni a destra $v.key \leq u.key \leq T_3$ e nelle rotazioni a sinistra $u.key \geq v.key \geq T_1$.

È possibile effettuare una rotazione semplice utilizzando una funzione con costo $O(1)$.

Tipologie di sbilanciamenti

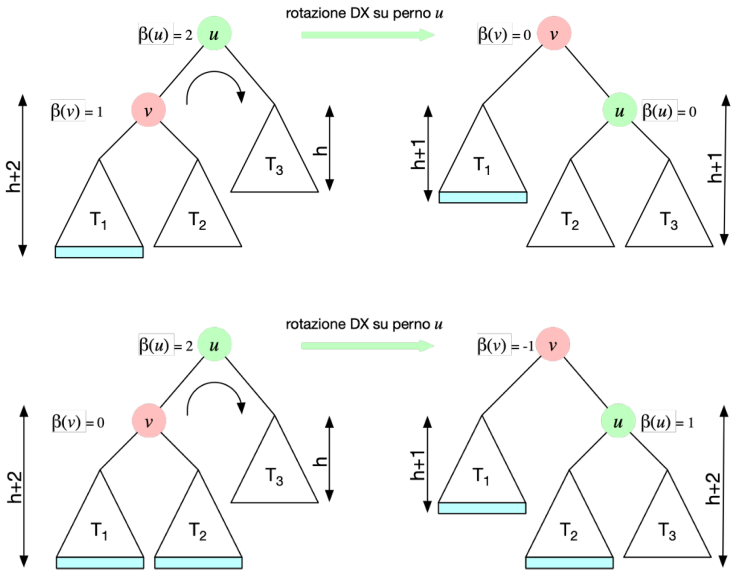
Assumendo di avere un albero AVL bilanciato, in cui un suo sottoalbero radicato in un nodo u diventa sbilanciato a seguito di una operazione di inserimento o rimozione, è possibile ricadere in 4 casi di sbilanciamento:

- **Sbilanciamento SS**: $\beta(u) = 2$ e $\beta(u.left) \geq 0$.
- **Sbilanciamento DD**: $\beta(u) = -2$ e $\beta(u.right) \leq 0$.
- **Sbilanciamento SD**: $\beta(u) = 2$ e $\beta(u.left) = -1$.
- **Sbilanciamento DS**: $\beta(u) = -2$ e $\beta(u.right) = 1$.

Ribilanciamenti

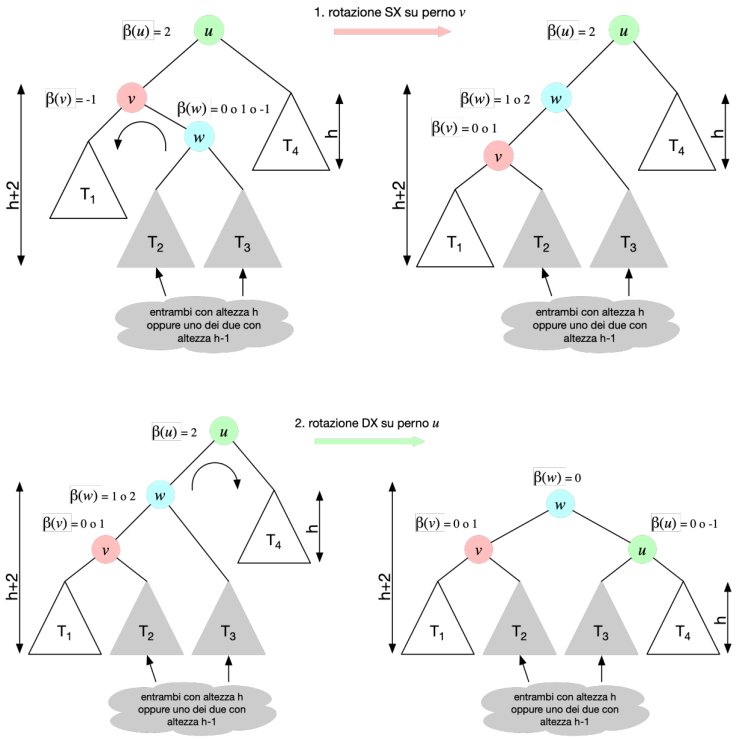
Per ciascuno dei 4 casi di sbilanciamento occorre effettuare una particolare rotazione al fine di riottenere un albero bilanciato.

- **Sbilanciamento SS** $\implies$ Rotazione DX su u.



- **Sbilanciamento DD** $\implies$ Rotazione SX su u.

- **Sbilanciamento SD** $\implies$ Rotazione SX su $u.left$ e rotazione DX su u.



- **Sbilanciamento DS** $\implies$ Rotazione DX su $u.right$ e rotazione SX su u.

Visto che in tutti i casi, per ribilanciare l'albero, vengono utilizzare delle rotazioni le quali funzioni hanno costo costante $O(1)$, anche il ribilanciamento di un albero AVL ha un costo costante $O(1)$.

Inserimento e rimozione in un albero AVL

A questo punto è possibile analizzare il costo complessivo dell'inserimento e della rimozione di un nodo all'interno di un albero AVL.

In generale, il costo dell'inserimento e della rimozione di un nodo all'interno di un albero è $O(h)$, e visto che l'altezza di un albero AVL è $\log n$ concludiamo che tali operazioni in un albero AVL hanno un costo pessimo $O(\log n)$. A questa operazione dobbiamo però aggiungere i costi dell'aggiornamento delle altezze dei sottoalberi, nel caso pessimo $O(\log n)$, e il costo della procedura di ribilanciamento nel caso in cui si presentano dei nodi critici (il fattore di bilanciamento del loro sottoalbero diventa ± 2), che nel caso della rimozione possono essere molteplici in un unico percorso radice-foglio, dunque il costo pessimo di ribilanciamento è $O(\log n)$.

Concludiamo dunque che il **costo computazionale** dell'inserimento e della rimozione di un nodo in un albero AVL è:

- Caso **pessimo**: $O(\log n)$.

Dizionario con albero binario di ricerca

Riassumiamo il costo delle operazioni basilari di un dizionario implementato tramite albero AVL confrontati con le altre tipologie di implementazione.

	SEARCH		INSERT		DELETE	
	Medio	Pessimo	Medio	Pessimo	Medio	Pessimo
Array non ordinati	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$\Theta(n)$	$\Theta(n)$
Array ordinati	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Lista concatenata	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Albero Binario di Ricerca	$O(\bar{h})$	$O(h)$	$O(\bar{h})$	$O(h)$	$O(\bar{h})$	$O(h)$
Albero AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

Riassunto dei costi per le operazioni basilari delle diverse implementazioni di un dizionario.

▼ 6.3 - Algoritmi di decisione su alberi

Game tree

Un **game tree** è un albero che rappresenta tutte le possibili partite in un gioco a turni. In tale albero i nodi rappresentano una situazione di gioco, gli archi tutte le mosse giocabili a partire da un nodo e le foglie gli stati finali di gioco (vittoria, sconfitta e patta).

I game tree vengono utilizzati per realizzare algoritmi di decisione per giochi a turni, i quali riescono a scegliere le scelte migliori da effettuare in ogni possibile situazione di gioco.

Il problema solitamente, in algoritmi di tale genere, è quello di capire come effettuare le scelte di gioco senza dover generare e visitare l'intero albero, il quale è tipicamente troppo ampio.

Algoritmo MiniMax

> L'algoritmo **MiniMax** è un algoritmo di decisione che consiste nel cercare di minimizzare la massima perdita possibile.

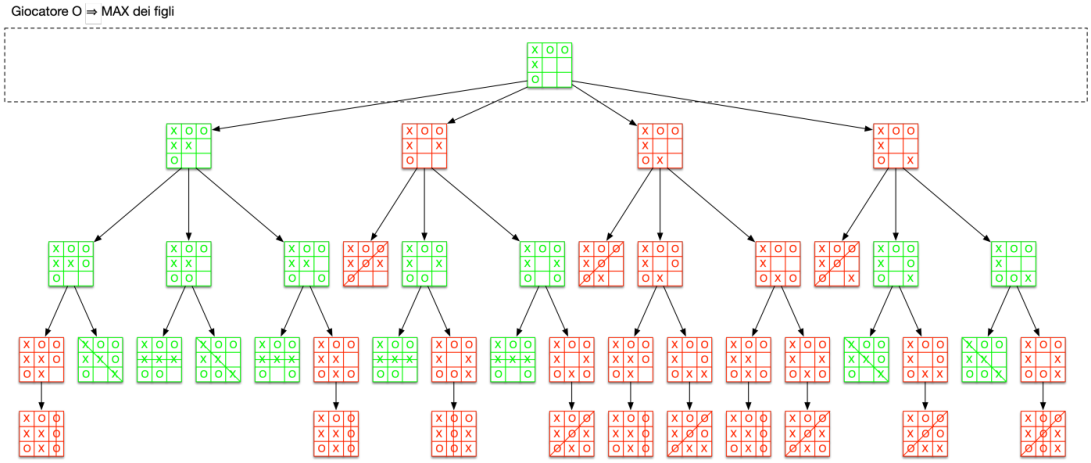
Funzionamento

Per attuare tale algoritmo occorre etichettare tramite un valore le diverse configurazioni finali di gioco, presenti nelle foglie dell'albero:

- +1: vittoria.
- 0: patta.
- -1: sconfitta.

A questo punto occorre propagare tale valore dalle foglie alla radice al fine di comprendere quale scelta sia la migliore da intraprendere. Per propagare i valori si utilizza dunque la tecnica della massimizzazione per i nodi scelti dal proprio giocatore, ovvero gli viene assegnato il valore massimo tra quelli assegnati ai figli, e della minimizzazione per i nodi scelti dal giocatore avversario, ovvero gli viene assegnato il valore minimo tra quelli assegnati ai figli.

Capiamo il tutto con un esempio nel gioco tic tac toe nel quale il proprio giocatore utilizza il simbolo x e viene utilizzato il colore rosso per il valore -1 e il colore verde per il valore +1:



Notiamo che alle foglie sono stati assegnati i valori +1/-1 in base alla situazione di vittoria/sconfitta. A questo punto, livello dopo livello, occorre massimizzare quando l'arco è dato dal turno del proprio giocatore e minimizzare quando l'arco è dato dal turno del giocatore avversario, fino ad arrivare alla radice. Tramite questo algoritmo dunque si suppone che il giocatore avversario faccia la migliore scelta possibile per minimizzare la sua perdita. Una volta arrivato alla radice il giocatore ha la possibilità di fare la scelta migliore al fine di minimizzare la massima perdita possibile, dunque nel gioco del tic tac toe di vincere o pareggiare, nei casi in cui la vittoria non è possibile.

Algoritmo

Lo **pseudocodice** di MiniMax è il seguente:

```
int miniMax(Node T, bool playerA) {
  if (isLeaf(T)) eval = evaluate(T)
  else if (playerA == true) {
    for (c ∈ children(T))
      eval = max(eval, miniMax(c, false))
  } else {
    for (c ∈ children(T))
      eval = min(eval, miniMax(c, true))
  }
  return eval
}
```

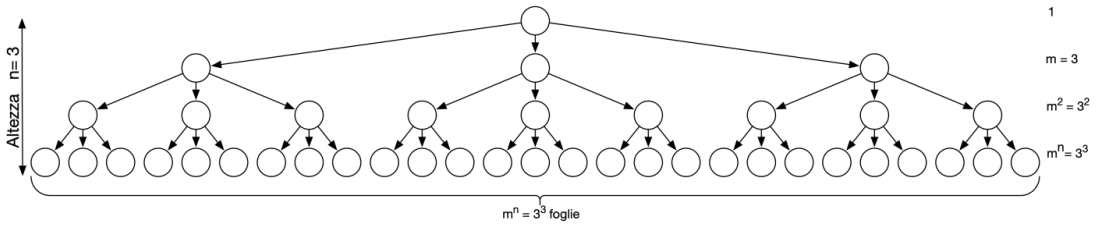
Il **costo computazione** di MiniMax è il seguente:

- **Tempo** (viene visitato tutto l'albero): $\Theta(n)$.
- **Memoria**: $O(h)$.

Grandezza di un game tree

Per stimare in maniera esatta il costo computazionale di MiniMax occorre calcolare il numero esatto di nodi di un game tree. Purtroppo però tale operazione è molto complessa, dunque possiamo solamente trovare un upper bound al game tree effettivo ammettendo configurazioni di gioco illegali, ovvero configurazioni che non sono possibili in una partita reale.

Per fare ciò occorre supporre che un giocatore ha a disposizione al massimo m mosse possibili per turno e il gioco termina in massimo n turni. Ad esempio nel gioco forza 4, $m = 7$, ossia le colonne della matrice nelle quali è possibile effettuare una mossa, e $n = 6 \times 7$, ossia il prodotto tra le righe e le colonne della matrice, il numero dei buchi della matrice.



Il numero $P(m, n)$ di **partite possibili** è dato dal numero di foglie, quindi contando che alcune di queste sono illegali:

$$P(m, n) \leq m^n = O(m^n)$$

Il numero totale $T(m, n)$ di **nodi del game tree** è limitato da:

$$T(m, n) \leq 1 + m + m^2 + \cdots + m^n = \sum_{k=0}^{n} m^k = \frac{m^{n+1} - 1}{m - 1} = O(m^n)$$

Grandezza di un game tree su un gioco con n oggetti

Se un gioco, come tic tac toe, ha n **oggetti a disposizione tra i quali scegliere** e i quali, dopo la scelta, non sono più disponibili, è possibile migliorare l'accuratezza degli upper bound scelti in precedenza.

In questo caso il numero $P(n)$ di partite disponibili partendo da una scelta di n oggetti diventa:

$$P(n) \leq n! = O(n!)$$

Il numero totale $T(n)$ di **nodi del game tree** diventa:

$$T(n) \leq 1 + n + n(n-1) + n(n-1)(n-2) + \cdots + n! = \cdots = O(n!)$$

Costo computazionale di MiniMax

A seguito dell'analisi della grandezza di un game tree possiamo elencare con maggiore esattezza i **costi computazionali** della funzione MiniMax.

Per un gioco con m mosse ed un massimo di n turni abbiamo il seguente costo computazionale:

- Costo **pessimo** in termini di **tempo**: $O(m^n)$.
- Costo **pessimo** in termini di **memoria**: $O(h)$.

Per un gioco con n oggetti consumabili abbiamo il seguente costo computazionale:

- Costo **pessimo** in termini di **tempo**: $O(n!)$.
- Costo **pessimo** in termini di **memoria**: $O(n)$.

Algoritmo AlphaBeta

Ottimizzazione per MiniMax

È possibile pensare ad una **piccola ottimizzazione** di MiniMax per aumentarne le prestazioni.

Notiamo infatti che nel caso in cui un nodo ha almeno un figlio con il valore **massimo/minimo assoluto**, tale valore viene trasmesso al padre nel caso in cui occorre massimizzare/minimizzare. In

base a tale concetto possiamo dunque interrompere la visita sui sottoalberi del nodo attuale nel momento in cui uno dei figli abbia come valore il massimo/minimo assoluto.

Aggiungiamo dunque tale controllo nello pseudocodice:

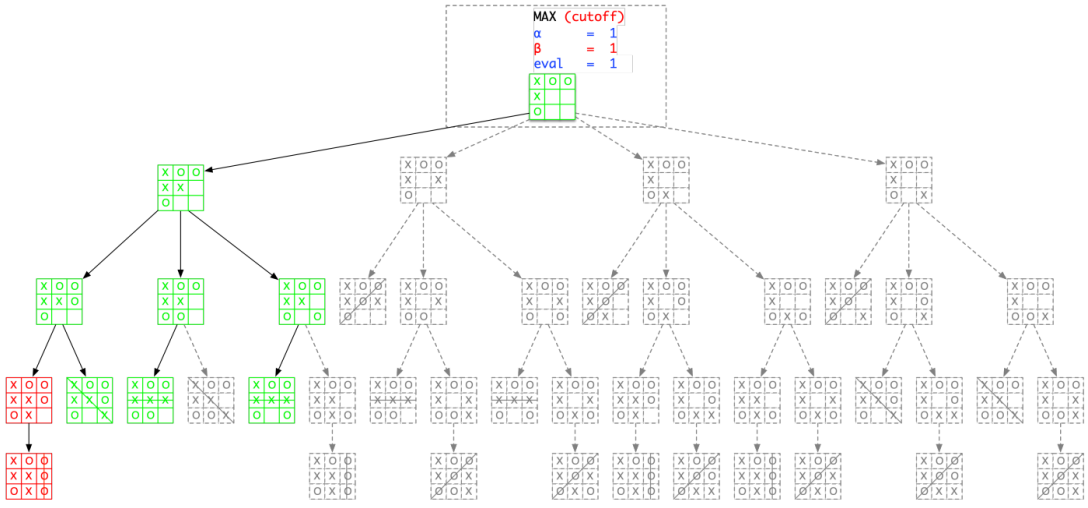
```
int MiniMax(Node T, bool playerA) {
  if (isLeaf(T)) eval = evaluate(T)
  else if (playerA == true) {
    for (c ∈ children(T)) {
      eval = max(eval, MiniMax(c, false))
      if (eval == maxval) break
    }
  } else {
    for (c ∈ children(T)) {
      eval = min(eval, MiniMax(c, true))
      if (eval == minval) break
    }
  }
  return eval
}
```

Generalizzazione dell'ottimizzazione

Tale ottimizzazione può essere **generalizzata** costruendo un nuovo algoritmo di decisione, chiamato **AlphaBeta**, il quale partendo dall'algoritmo MiniMax utilizza due valori, α e β, al fine di visitare meno nodi.

In tale algoritmo α rappresenta il **punteggio minimo** ottenibile dal nostro giocatore, β invece rappresenta il **punteggio massimo** ottenibile dal giocatore avversario. Tali valori permettono di rappresentare il range $[\alpha, \beta]$ di possibile risultati per i figli di tale nodo. Se ad un certo punto, per un certo nodo, $\beta \leq \alpha$ allora non ha più senso continuare ad analizzare gli altri suoi figli e questi vengono saltati.

Visualizziamo tale algoritmo in esecuzione sul game tree di una partita di tic tac toe:



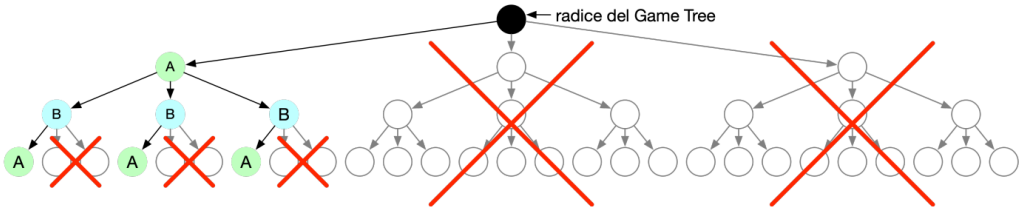
Visualizzazione grafica di ottimizzazione di MiniMax tramite l'algoritmo AlphaBeta.

Lo **pseudocodice** è il seguente:

```
int AlphaBeta(Node T, bool playerA, int α, int β) {
  if (isLeaf(T)) eval = evaluate(T)
  else if (playerA == true) {
    for (c ∈ children(T)) {
```

```
      eval = max(eval, AlphaBeta(c, false, α, β))
      α = max(eval, α)
      if (β <= α) break
    }
  } else {
    for (c ∈ children(T)) {
      eval = min(eval, AlphaBeta(c, true, α, β))
      β = min(eval, β)
      if (β <= α) break
    }
  }
  return eval
}
```

Notiamo che nel caso in cui ad ogni turno del proprio giocatore viene analizzata come prima mossa una mossa che lo porta alla vittoria, si evita di visitare la restanti mosse.



Caso ottimo di visita del game tree tramite algoritmo AlphaBeta.

In tal caso il numero $T(m,n)$ dei nodi dell'albero che vengono visitati è limitato da:

$$T(m,n) \leq 1+1+m+m+m^2+m^2+\cdots+m^{n/2}+m^{n/2}$$
$$= \sum_{k=0}^{n/2} 2m^k = 2\frac{m^{(n/2)+1}-1}{m-1} = O(m^{n/2}) = O(\sqrt{m^n})$$

Otteniamo dunque uno **speed-up quadratico** rispetto all'algoritmo MiniMax.

Il **costo computazionale** è il seguente:

- Caso **ottimo**: $O(\sqrt{m^n})$.
- Caso **pessimo**, quando viene visitato tutto l'albero: $O(m^n)$.

Il caso ottimo si riesce ad ottenere sempre solo nel caso in cui si trova una **strategia ottima** che permette sempre di valutare come prima mossa una mossa vincente, tale strategia renderebbe però inutile l'algoritmo AlphaBeta. Il costo computazionale medio invece è difficile da calcolare, comunque sappiamo che il numero di nodi valutati in tal caso è minore dell'algoritmo MiniMax.

Ricerca limitata in profondità

Visto che anche nel caso ottimo dell'algoritmo AlphaBeta l'albero potrebbe avere una dimensione molto grande per una visita in tempi ragionevoli, spesso è utile limitare la ricerca ad un **livello massimo di profondità** e valutare le configurazioni non-finali tramite euristiche nel seguente modo (X: vittoria, 0: patta, $-X$: sconfitta):

- $0 < eval \leq X$: configurazione favorevole.
- $-X \leq eval < 0$: configurazione sfavorevole.
- 0: totale incertezza o patta.

In questo modo non si riesce ad avere la certezza di minimizzare la perdita ma vengono scelte le configurazioni più favorevoli al fine di ottenere la **maggiore possibilità di vittoria possibile**.

Lo **pseudocodice** è il seguente:

```
int AlphaBeta(Node T, bool playerA, int α, int β, int depth) {
  if (depth == 0 || isLeaf(T))
    eval = evaluate(T, depth)
  else if (playerA == true) {
    for (c ∈ children(T)) {
      eval = max(eval, AlphaBeta(c, false, α, β, depth − 1)
      α = max(eval, α)
      if (β <= α) break
    }
  } else {
    for (c ∈ children(T)) {
      eval = min(eval, AlphaBeta(c, true, α, β, depth − 1))
      β = min(eval, β)
      if (β ≤ α) break
  }

  T.label = eval
  return eval
}
```

Algoritmo IterativeDeepening

Nel caso in cui si hanno dei **limiti di tempo per la scelta** di una mossa è molto difficile stimare il livello massimo di profondità visitabile in modo completo tramite algoritmo AlphaBeta con limite di profondità, in quanto quest'ultimo effettua una visita in profondità e non in ampiezza. Una stima troppo ottimistica del livello di profondità massima causerebbe una visita completa solo per poche mosse, mentre una stima troppo pessimista causerebbe una valutazione poco accurata delle varie mosse a disposizione.

Una soluzione a tale problema è quella di effettuare una **visita in ampiezza** la quale va a tentativi visitando ogni volta l'albero partendo dalla radice e aggiungendo di volta in volta un nuovo livello alla profondità che si va a visitare. Nel momento in cui si raggiunge il limite di tempo prestabilito viene utilizzato l'eval ottenuto nell'ultima visita completa effettuata.

Lo **pseudocodice** è il seguente:

```
int IterativeDeepening(Node T, bool playerA, int depth) {
  α = MinAlpha
  β = MaxBeta
  eval = 0
  for (d = 0, ... , depth) {
    if (timeIsRunningOut()) break
    eval = AlphaBeta(T, playerA, α, β, d)
  }
  return eval
}
```

Il **costo computazionale** in termini di **memoria** è il seguente:

- Caso **pessimo**: $O(d)$.

Per calcolare il **costo computazionale** in termini di **tempo** osserviamo che la radice viene visitata $d+1$ volte, i suoi figli d volte ecc, dunque effettuando alcune equivalenze matematiche otteniamo che:

- Caso **pessimo**: $O(m^d)$.

Notiamo dunque che a parità di profondità la ricerca in ampiezza IterativeDeepening ha lo **stesso costo asintotico** della ricerca in profondità AlphaBeta, in quanto il numero di nodi al livello d domina l'intero costo. In termini pratici è facile però notare che le costanti moltiplicative nascoste dalla notazione asintotica di IterativeDeepening lo rendono **concretamente più lento** di AlphaBeta, in quanto gli stessi nodi vengono visitati più volte. Tale svantaggio in termini di tempo viene ripagato dalla possibilità di avere un maggiore controllo sulla visita del Game Tree quando questa deve avvenire entro certi limiti di tempo.

Gestione delle ripetizioni

Infine analizziamo un'ulteriore ottimizzazione notando che molti stati di gioco possono comparire più volte nell'albero in quanto uno stesso stato può essere ottenuto con differenti sequenze di mosse. Per velocizzare la ricerca è dunque utile riconoscere le configurazioni già valutate al fine di non ripetere il lavoro.