RELAZIONE

All'attenzione di: prof. Daniele Francesco Santamaria, titolo Splay Trees

Preparato da: Riccardo Raciti Matricola: 1000001108

Anno accademico 2020/2021

1) Introduzione dell’algoritmo

Splay Trees algoritmo introdotto da Daniel Sleator e Robert Tarjan nel 1985, sono degli alberi “auto-regolanti”. Gli alberi Splay mantengono il bilanciamento senza alcuna esplicita condizione di bilanciamento. Il costo ammortizzato di ciascuna operazione in un albero di n nodi è O(lg n).
La proprietà fondamentale di questo algoritmo è che gli elementi ai quali si è acceduto più recentemente tenderanno a trovarsi più prossimi alla radice. Per cui gli alberi splay sono preferiti per l'implementazione di cache, in cui le informazioni non sono accedute uniformemente (accesso casuale), ma una parte degli elementi vengono acceduti più frequentemente di altri (accesso localizzato). Ad ogni accesso l'algoritmo splay sposta tale nodo alla radice, e di conseguenza gli elementi acceduti più frequentemente si trovano sempre più prossimi alla radice dell'albero, rendendoli più velocemente accessibili e migliorando sensibilmente i tempi di accesso globali alla cache nelle operazioni di ricerca e cancellazione. La ricerca in tale algoritmo risulta più efficiente di un normale BST e talvolta anche di un albero bilanciato.
Gli Splay Trees implementano M operazioni consecutive di

- Ricerca;
- Inserimento;
- Cancellazione.

In tempo O(M lg N) dove N è il numero di inserimenti, ognuna delle M operazioni viene eseguita in tempo ammortizzato O(lg n), cioè del tipo O(a(n)).

2) Metodi

Le funzioni assumono due parametri (x,T), dove x sarà il valore della chiave mentre T la root, T sarà sempre la root per mantenere traccia dell'ordinamento interno dopo ogni funzione o visita della struttura.
La funzione che garantisce la proprietà fondamentale dello Splay Trees è la funzione Splay, che usufruisce delle funzioni di rotazione.

SPLAY(x,T) :

- Si ricerca x su T mediante ricerca binaria
- Partendo dal nodo dove la ricerca si è fermata e procedendo verso la radice si applicano le rotazioni Right e Left, che chiameremo R_rotate e L_rotate;
- Ritorneremo il nodo T.

```
SPLAY(x, T)
        head
        head.right = head.left = NULL
        T1 = T2 = &head
        while sempre vero
                if(x <= T.key)   //si controlla la posizione di x
                        if(!T.left) //se non esiste figlio sinistro usciamo dal ciclo
                                break;
                        if(x<T.left.key)
                                T  =  R_rotate(T) //settiamo la nuova root
                                if(!T.left) //se non esiste figlio sinistro della root
                                        break
                T2.left = root
```

```
        T2 = T2.left
        T = T.left
        T2.left = NIL
        else if (x  >  T.key) //si controlla la posizione di x
                //effettuiamo i controlli sul figlio destro
                if(!T.right)
                        break
                if(x  >  T.right.key) then
                        T  = L_rotate(T) //settiamo la root tramite una left
                                                //rotate
                        if(T.right) then
                                break
                T1.right = root
                T1 = T1.right
                T = T.right
                T1.right = NIL
        else
                break
//si collegano i sotto alberi destri e sinistri alla root
T1.right = T.left
T2.left = T.right
T.left = head.right
T.right = head.left
return T
```

Funzione R_Rotate(T):

- Si ruota verso destra l'albero considerando come fulcro il nodo T

```
R_Rotate(T)
        y = T.left
        T.left = y.right
        y.right = x
        return y
```

Funzione L_Rotate(T):

- Si ruota verso sinistra l'albero considerando come fulcro il nodo T

```
L_Rotate(T)
        y = T.right
        T.right= y.left
        y.left = x
        return y
```

Funzione Search(x,T):

- Si chiama Splay(x,T).

```
Search(x,T)
        return Splay(x,T)
```

Funzione INSERT(x,T):

- Si verifica l'esistenza della root, se essa non dovesse esistere il nodo inserito sarà la root, poiché sarebbe il primo elemento ad essere inserito;
- Si chiama la funzione Splay(x,T) e si associa il valore a T;
- Si inserisce il nodo x;
- Si ritorna il nodo root.

```
INS(x,T)
        G = NIL
        if(!G) then
                G = _NewNode(x) //creazione di un nuovo Nodo
        else
                G.key = x
        if(!T) //se la root non esiste
                T = G //si pone la root uguale al nuovo nodo
                G = NIL
                return T
        T = Splay(x,T) //si riordina l'albero per eliminare le possibili violazione
                        //introdotte con l'inserimento
        if(x < T.key) //si cerca la corretta posizione per la chiave
                G.left = T.left
                G.right = T
                T.left = NIL
                T = G
        else if(x > T.key) //si cerca la corretta posizione per la chiave
                G.right = T.right
                G.left = T
```

```
            T.right = NIL
            T = G
        else
            return T
        G = NIL
        return T
```

Funzione DELETE(x,T):

- Si cancella il nodo contenente la chiave x;
- Se esiste il figlio sinistro di T si ridefinisce la root utilizzando la funzione Splay;
- Se non esiste la root sarà il figlio destro di T.

```
Delete(x,T)
        G       //nodo
        if(!T) //se l'albero è vuoto torniamo nulla
            return NULL
        T = Splay(x,T)
        if(x != T.key) //se l'albero ha un solo elemento
            return T
        else
            if(!T.left) //ricerchiamo la chiave e la "scolleghiamo"
                G = T
                T  = T.right
            else
                G = T
                T = Slay(x,T.left)
```

```
                T.right = G.right
    return T
```

3) Analisi dei costi e dei cenni teorici

Quando si accede a un nodo x, viene eseguita un'operazione di splay su x per spostarlo alla radice. Per eseguire un'operazione di splay, eseguiamo una sequenza di passaggi di splay, ognuno dei quali avvicina x alla radice. Eseguendo un'operazione di splay sul nodo di interesse dopo ogni accesso, i nodi recentemente acceduti vengono mantenuti vicino alla radice e l'albero rimane approssimativamente bilanciato in modo da raggiungere i limiti di tempo ammortizzati desiderati.

Ogni passaggio particolare dipende da tre fattori:

- Se x è il figlio sinistro o destro del suo nodo genitore, p.
- Se p è la radice o meno, e in caso contrario.
- Se p è il figlio sinistro o destro del suo genitore, g (il nonno di x).

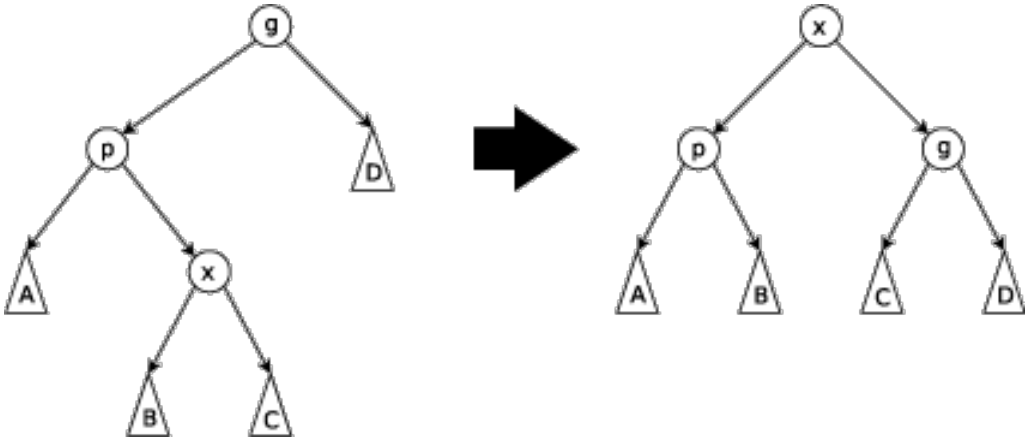
È importante ricordare di impostare gg (il bisnonno di x) in modo tale che punti ad x dopo ogni operazione di splay. Se gg è nullo, allora x ovviamente è la radice e deve essere aggiornato come tale.

Esistono tre tipi di passaggi di apertura, ciascuno dei quali ha due varianti simmetriche: mancino e destrorso. Per semplicità, viene mostrato solo uno di questi due per ogni tipo. (Nei seguenti diagrammi, i cerchi indicano i nodi di interesse e i triangoli indicano i sotto alberi di dimensioni arbitrarie).

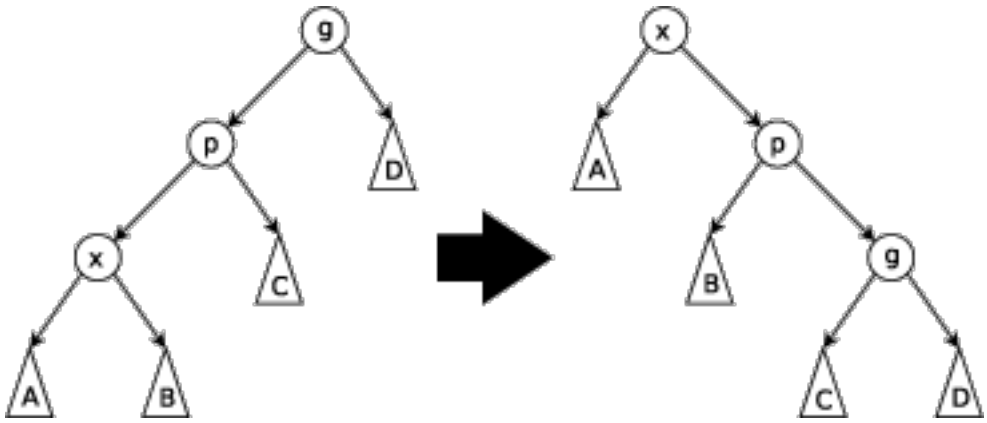
I tre tipi di passaggi di apertura sono:

1. ZIG : questo passaggio viene eseguito quando p è la radice. L'albero viene ruotato sul bordo tra x e p.

2. ZIG-ZAG: questo passaggio viene eseguito quando p non è la radice e x è un figlio destro, p può essere un figlio sinistro o destro. L'albero viene ruotato sul bordo tra p e x, quindi ruotato sul bordo risultante tra x e g.



3. ZIG-ZIG: questo passaggio viene eseguito quando p non è la radice e x e p sono entrambi figli di destra o entrambi figli di sinistra. Nel caso in cui x e p sono entrambi figli di sinistra l'albero viene ruotato sul bordo che unisce p con il suo genitore g, quindi ruota sul bordo che unisce x con p. Si noti che i passaggi a zig-zig sono l'unica cosa che differenzia gli alberi splay dal metodo di rotazione a radice introdotto da Allen e Munro prima dell'introduzione degli alberi splay.



Una semplice analisi ammortizzata degli alberi splay può essere eseguita utilizzando il metodo del potenziale.
Si definisce:

- S(v) =dif numero nodi del sotto albero con radice v;
- size (r) = il numero di nodi nel sottoalbero radicato nel nodo r (incluso r).
- rank (r) = log2 (size (r)).
- Φ = la somma dei ranghi di tutti i nodi dell'albero.
  Φ tenderà ad essere alto per alberi poco bilanciati e basso per alberi ben bilanciati.

Per applicare il metodo del potenziale, si calcola prima ΔΦ: la variazione del potenziale causata da un'operazione di Splay.
Si controlla ogni caso separatamente. Indicando con rank′ la funzione rank dopo l'operazione Splay.
x, p, g saranno i nodi interessati dall'operazione di rotazione (come negli esempi soprastanti).
Zig tale che

$$\Delta\Phi = = \text{rank}'(p) - \text{rank}(p) + \text{rank}'(x) - \text{rank}(x)$$

[poiché solo p e x cambiano i ranghi]

$$= \text{rank}'(p) - \text{rank}(x)$$ *[da rank′(x)=rank(p)]*

$$\leq \text{rank}'(x) - \text{rank}(x)$$ *[da rank′(p)<rank′(x)]*

Zig-zig tale che

$$\Delta\Phi = \text{rank}'(g) - \text{rank}(g) + \text{rank}'(p) - \text{rank}(p) + \text{rank}'(x) - \text{rank}(x)$$

[p rank′(x)=rank(g)]

$$\leq \text{rank}'(g) + \text{rank}'(x) - 2\,\text{rank}(x)$$ *[poiché rank(x)<rank(p) e rank′(x)>rank′(p)]*

$$\leq 3(\text{rank}'(x) - \text{rank}(x)) - 2$$ *[a causa della concavità della funzione logaritmica]*

Zig-zag tale che

$$\Delta\Phi = \text{rank}'(g) - \text{rank}(g) + \text{rank}'(p) - \text{rank}(p) + \text{rank}'(x) - \text{rank}(x)$$
$$\leq \text{rank}'(g) + \text{rank}'(x) - 2\,\text{rank}(x) \quad \textit{[poiché rank(x)=rank(g) e rank(x)<rank(p)]}$$
$$\leq 3(\text{rank}'(x) - \text{rank}(x)) - 2 \quad \textit{[a causa della concavità della funzione logaritmica]}$$

Il costo ammortizzato di qualsiasi operazione è ΔΦ più il costo effettivo. Il costo effettivo di qualsiasi operazione zig-zig o zig-zag è 2 poiché ci sono due rotazioni da fare.
Quindi:
costo ammortizzato = costo + $\Delta\Phi \leq 3$ (rank′ (x) − rank (x)) che è O (log n).
L'operazione Zig aggiunge un costo ammortizzato di 1, ma vi è al massimo una di queste operazioni.
Quindi ora si sa che il tempo totale ammortizzato per una sequenza di m operazioni è:
$T_{ammortizzato}(m)$ = O (m \ log n)
Per passare dal tempo ammortizzato al tempo effettivo, si deve aggiungere la diminuzione del potenziale dallo stato iniziale, prima che venga eseguita qualsiasi operazione (Φi), allo stato finale, dopo che tutte le operazioni sono state completate (Φf).

$$\Phi_i - \Phi_f = \sum_x \text{rank}_i(x) - \text{rank}_f(x) = O(n \log n)$$

dove l'ultima disuguaglianza deriva dal fatto che per ogni nodo x, il rango minimo è 0 e il rango massimo è log (n).
Fatte tali premesse si può limitare il tempo effettivo:

$$T_{\text{actual}}(m) = O(m \log n + n \log n)$$

Analisi ponderata

L'analisi precedente può essere generalizzata nel modo seguente.

Si assegni ad ciascun nodo r un peso w (r).

Definendo:

- dimensione (r) = la somma dei pesi dei nodi nel sottoalbero radicato nel nodo r (compreso r).
- rango (r) e Φ esattamente come sopra.

Si applica la stessa analisi precedente ed il costo ammortizzato di un'operazione di splaying è uguale ad:

$$\text{rank}(root) - \text{rank}(x) = O(\log W - \log w(x)) = O\left(\log \frac{W}{w(x)}\right)$$

dove W è la somma di tutti i pesi.

La diminuzione dal potenziale iniziale a quello finale è delimitata da:

$$\Phi_i - \Phi_f \leq \sum_{x \in tree} \log \frac{W}{w(x)}$$

poiché la dimensione massima di ogni singolo nodo è W e la minima è w (x).

Quindi il tempo effettivo è limitato da:

$$O\left(\sum_{x \in sequence} \log \frac{W}{w(x)} + \sum_{x \in tree} \log \frac{W}{w(x)}\right)$$

4) Implementazioni possibili

Per la progettazione di un albero non sono molte le possibili scelte progettuali. Se non è necessario effettuare frequentemente operazioni di inserimento e cancellazioni o non è affatto necessario effettuarle e non si vuole usare troppa memoria è possibile implementare un albero di ricerca binario su un array ordinato con la restrizione che il numero degli elementi sia 2n -1 con $n \in N$.
La radice occuperà la prima posizione dell'array e i figli di questa radice sono a loro volta nell'array e occupano come posizione (2i) e (2i+1) dove i è la posizione della radice, del padre e dei due figli.
Si fa notare che questa implementazione è ottimale se l'albero è completo cioè se tutti gli elementi che costituiscono l'albero hanno esattamente due figli, tranne ovviamente le foglie, altrimenti è necessario un flag booleano, in realtà un array di accompagnamento, che ci indica se la posizione è valida o meno.
I vantaggi dell'utilizzo di questa implementazione sono la semplicità di accesso e di gestione degli elementi della lista, al contrario le operazioni di inserimento e in generale la dimensione considerevole di un array di un grande albero giocano a sfavore di questa implementazione che risulta essere di conseguenza sconveniente da usare.
Un'altra implementazione che prevede l'uso di array è quella della rappresentazione collegata con array, in cui, in una tabella a tre colonne si hanno, rispettivamente per riga, in quella centrale il valore, in quella sinistra l'indirizzo del figlio sinistro e in quella destra l'indirizzo di quello destro. È necessario aggiungere una variabile inizio per indicare il punto in cui dobbiamo iniziare l'analisi, al contrario, se un indirizzo è a zero è da considerarsi NULL, stesso risultato si può ottenere prendendo in considerazione anziché un array un semplice nodo strutturato a tre campi, etichetta, figlio sinistro, figlio destro e con un puntatore al primo nodo, e di

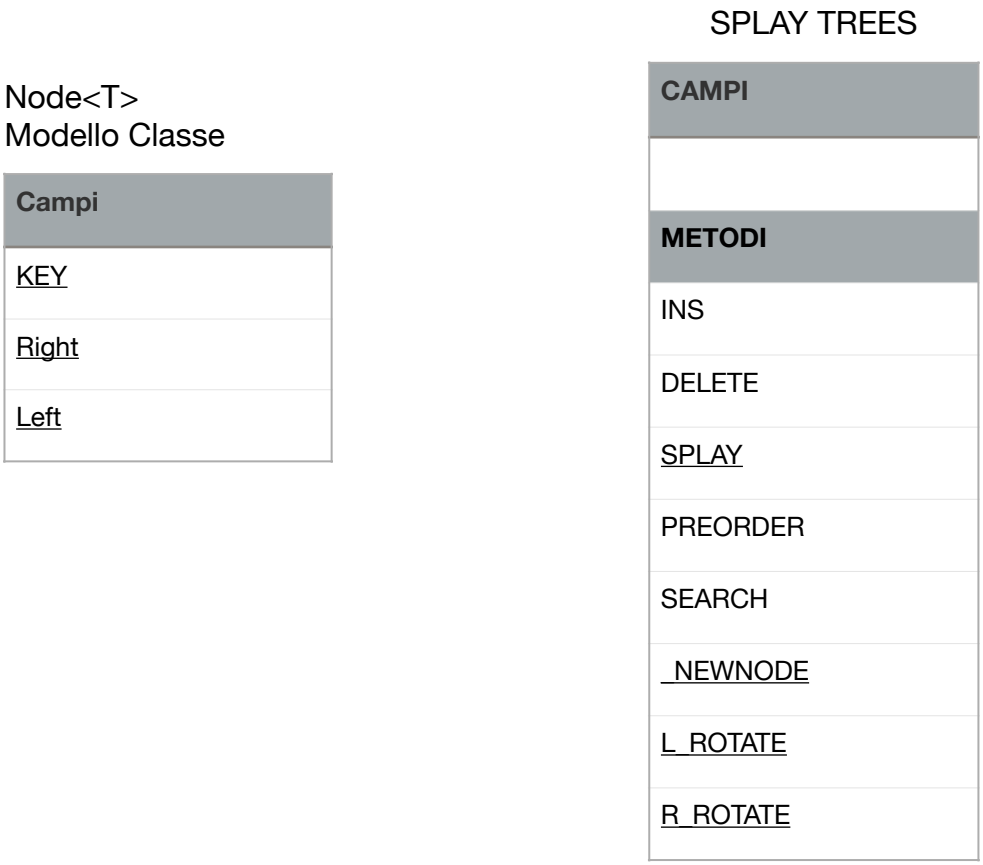
fatto si ci ricollega all'immagine di accompagnamento alle due tabelle precedenti.
Oppure la struttura si può implementare tramite l'implementazione classica in cui ogni nodo dell'albero oltre al suo valore ha un puntatore al figlio destro ed uno al figlio sinistro, in questo modo è possibile, partendo dalla radice, discendere nell'albero fino ad arrivare alle foglie. Tutti i nodi sono uguali, l'unica differenza è che nessun nodo punterà alla radice (infatti la radice non è né un figlio destro, né un figlio sinistro) e le foglie non avendo figli non punteranno a nulla (nil, NULL value).
In seguito, all'implementazione dell'albero gli si aggiungono i metodi a noi necessari, nel nostro caso le funzioni di rotazione Right e Left, che chiameremo R_rotate e L_rotate e la funzione Slay che sarà quella che ci garantirà la proprietà dello Splay Tree.

5) Scelta implementativa

Per questo progetto si è scelto di procedere in modo diverso dal solito non basandosi su materiale e/o progetti già implementati ma prendendo il tutto come una sfida, per tanto si è deciso fin da subito di implementare diversamente la base dell'albero cioè i nodi. Non creando la solita classe nodo con i relativi attributi e funzioni ma creando una struttura Nodo (struct).
In questa struttura Nodo vi sono presenti come attributi: una variabile che identifica la chiave del nodo e due puntatori per linkare il figlio destro e sinistro. Fatto cioè si è scelto di implementare l'albero attraverso l'implementazione classica che prevede i link dei nodi tramite il loro valore in modo da avere alla sinistra della root solo nodi con valore della chiave minore o uguale al valore della root, mentre alla sua destra valori solo maggiori.

5) UML

Di seguito L'UML relativo all'algoritmo.



Gli elementi sottolineati sono quei metodi o campi di tipo private.