

Relazione laboratorio Algoritmi e Strutture Dati

11 agosto 2020

Brugnera Matteo 137370@spes.uniud.it 137370

Rasera Giovanni 143395@spes.uniud.it 143395



**UNIVERSITÀ
DEGLI STUDI
DI UDINE**

Indice

1	Alberi binari di ricerca semplici	2
1.1	Introduzione	2
2	Alberi binari di ricerca di tipo AVL	3
2.1	Introduzione	3
3	Alberi binari di ricerca di tipo Red-Black	4
3.1	Introduzione	4
4	Algoritmo per il calcolo dei tempi	5

1 Alberi binari di ricerca semplici

1.1 Introduzione

La prima struttura dati che viene presa in considerazione è quella dei *BST*. Per la loro rappresentazione è stata creata una classe che crea i singoli nodi, ognuno dei quali è composto da quattro attributi:

- **key:** elemento che rappresenta la chiave numerica del nodo;
- **val:** elemento che rappresenta il valore del nodo sotto forma di stringa;
- **left e right:** entrambi rappresentano i figli del nodo, rispettivamente quello sinistro e destro, e vengono inizializzati a *NILL*.

```
//Attributi
int key;
string val;
node *left;
node* right; }

//Costruttore del nodo
node(int key, string val) {
    this->key = key;
    this->val = val;
    this->left = nullptr;
    this->right = nullptr;
}
```

Ogni nodo viene poi inizializzato grazie alla funzione **create** dove, una volta passati come parametri la chiave e il valore, mi crea il nodo. Successivamente possiamo trovare delle funzione di supporto che servono a rendere la struttura più versatile e facile da utilizzare:

1. **la funzione insert:** inserisce un nuovo nodo nell'albero di ricerca passatogli come argomento, assumendo che esso non sia già contenuto al suo interno;
2. **la funzione find:** cerca all'interno dell'albero il nodo con chiave numerica *k* (passata come argomento) e restituisce (qualora esso esista) il valore di tipo stringa ad esso legato;
3. **la funzione clear:** rimuove ricorsivamente tutti i nodi dall'albero, che diventerà quindi vuoto, liberando così la memoria;
4. **la funzione min:** cerca il nodo con il valore minimo all'interno dell'albero;
5. **la funzione remove:** elimina il nodo cercato sistemando poi i suoi figli, qualora essi esistano;

2 Alberi binari di ricerca di tipo AVL

2.1 Introduzione

La seconda struttura dati che analizziamo è quella degli alberi creati da *Adelson-Velsky and Landis (AVL tree)*.

Oltre a soddisfare le proprietà di un albero di ricerca semplice, l'AVL ha una caratteristica in più: *per ogni nodo x all'interno dell'albero, le altezze dei sottoalberi di sinistra e di destra differiscono al massimo di 1*.

Tale proprietà viene garantita *eseguendo opportune rotazioni sui nodi sbilanciati*, partendo dal nodo sbilanciato più profondo e procedendo risalendo l'albero lungo il cammino di accesso a quel nodo.

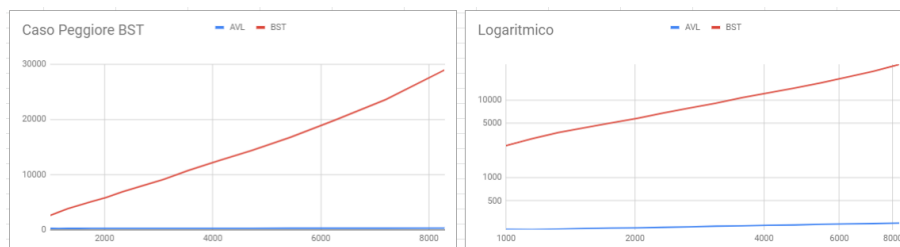
Per la loro rappresentazione è stata creata una classe che crea i singoli nodi, ognuno dei quali è composto da sei attributi:

- **key:** elemento che rappresenta la chiave numerica del nodo;
- **val:** elemento che rappresenta il valore del nodo sotto forma di stringa;
- **left, right e father:** rappresentano i figli del nodo, rispettivamente quello sinistro e destro, e il padre del nodo. Vengono inizializzati a *NILL*.
- **height:** rappresenta l'altezza del nodo ed è inizializzato a 1;

Possiamo trovare una serie di funzioni che servono a rendere la struttura funzionante:

1. **la funzione get height:** ritorna l'altezza dell'albero;
2. **la funzione destra e sinistra:** eseguono la rotazione dell'albero verso destra o sinistra;
3. **la funzione valore di bilanciamento:** ritorna 0 se la root è null oppure sono perfettamente bilanciati, 1 se c'è più peso a sinistra oppure -1 se c'è più peso a destra;
4. **tutte le funzioni viste precedentemente sui BST;**

Di seguito vediamo un confronto tra BST e l'AVL. Il caso peggiore nei BST lo si ha quando vengono inseriti i numeri tra 1 ed n uno dopo l'altro, andando a creare un albero sbilanciato verso di un lato, formando quindi un unico "path". L'AVL non è soggetto invece a questo problema, come dimostrato nel seguente grafico:



3 Alberi binari di ricerca di tipo Red-Black

3.1 Introduzione

L'ultima struttura dati che analizzeremo è quella dei Red-Black Tree.

Come prima cosa viene creata sia la tipologia di colore che un nodo può avere (ovvero *nero* o *rosso*), che la tipologia del nodo stesso (ovvero *root*, *left* o *right*).

Proprio come per le altre due strutture, anche qui viene creata la stessa classe

```
enum color {
    RED, BLACK, DOUBLE_BLACK
};
typedef enum color RBcolor;
string strColor(RBcolor c) {
    switch (c){
        case RBcolor::BLACK:
            return "black";
        case RBcolor::RED:
            return "red";
        case RBcolor::DOUBLE_BLACK:
            return "DOUBLE_BLACK";
        default:
            break;
    }
    return "";
}
```

per la rappresentazione del nodo, con però l'aggiunta di attributi in più come *il nodo padre* e *il colore del nodo*.

Troviamo inoltre la presenza degli stessi metodi dei BST con l'aggiunta di:

- **rotateR** e **rotateL**: rispettivamente ruotano a destra e sinistra l'albero grazie ad un perno passato come argomento e serve per ri-bilanciare l'albero;
- **getColor**: mi dice il colore del nodo passato come argomento;
- **uncle**: mi ritorna lo zio del nodo preso in considerazione;
- **grandfather**: mi ritorna lo zio del nodo preso in considerazione;
- **checkBST**: ritorna true qualora l'albero abbia tutti i puntatori giusti;
- **check**: ritorna true qualora l'RBT abbia tutti i puntatori giusti, colori giusti e black-height corretta;

4 Algoritmo per il calcolo dei tempi

Viene eseguita una stima dei tempi medi e **ammortizzati** per l'esecuzione di **n** operazioni, sia di inserimento che di ricerca, nelle tre strutture dati viste precedentemente.

Viene scelto inizialmente un parametro **n** che corrisponde al numero di operazioni che vogliamo che vengano fatte su un albero inizialmente vuoto. Le operazioni che vengono eseguite sono:

1. **generazione in modo pseudo-casuale del valore k;**
2. **ricerca di un nodo con chiave k;**
3. **qualora esso non esista, si crea un nuovo nodo con chiave k e lo si inserisce all'interno della struttura;**
4. **gestione della memoria:** ad ogni iterazione la memoria viene cancellata grazie ai metodi *clear* presenti dentro al codice degli alberi.

Infine abbiamo che il tempo ammortizzato di tutte le operazioni eseguite è il risultato dato da il *tempo totale impiegato per l'esecuzione delle operazioni diviso il parametro n*.

Abbiamo ottenuto quindi un tempo ammortizzato che ha un errore relativo che non supera mai l'1%