ABR LIBRARY

Libreria per il progetto sugli alberi binari di ricerca sviluppata dal gruppo 11 composto da Francesco Borrelli, Alessandro Grieco e Camilla Zampella durante il corso di Laboratorio di Algoritmi e Strutture Dati dell' a.a. 2017/2018.

Lo svolgimento della traccia è suddiviso in capitoli, ognuno dei quali rappresenta una funzionalità della traccia.

1 SOMMARIO

2 STRUTTURA DATI TREE

```
struct Tree{
    int info; //THE DATA STORED INTO THE NODE
    int h; //BST HEIGHT
    struct Tree *right; //RIGHT CHILD
    struct Tree *left;  //LEFT CHILD
};

typedef struct Tree tree;
```

La struttura dati Tree rappresenta un albero binario di ricerca e contiene al suo interno i seguenti campi

- info : contiene la chiave del nodo
- h : altezza del sottoalbero radicato in quel nodo
- right : figlio destro
- left : figlio sinistro

È stata modificata la struttura "di base" di un ABR aggiungendo il campo che tiene traccia dell'altezza, in quanto per alcuni algoritmi è necessaria questa informazione (si veda la funzione di stampa o il bilanciamento) ed è più opportuno accedervi in tempo costante piuttosto che visitare un intero albero per questa informazione.

3 GESTIONE DEI DUPLICATI IN UN ABR

La prima funzionalità richiesta dalla traccia prevede la modifica degli algoritmi di gestione di un albero binario di ricerca al fine di poter gestire duplicati all'interno dell'albero.

3.1 STRATEGIA DI RISOLUZIONE DEL PROBLEMA

Dato che in un ABR il sottoalbero sinistro contiene i nodi più piccoli della radice e il sottoalbero destro quelli più grandi, l'inserimento del nodo già esistente avviene nel sottoalbero destro dell'albero che ha come radice il valore da inserire modificando così l'invariante di un ABR in modo tale da avere tutti i nodi maggiori o uguali alla radice nel sottoalbero destro.

3.2 DETTAGLI IMPLEMENTATIVI

Di seguito l'implementazione che si adatta alla strategia di risoluzione del problema.

3.2.1 Funzione di inserimento

```
tree *insertNode(tree *head, int val){
    if(head!=NULL){
        if(head->info>val)
            head->left=insertNode(head->left, val);
        else
            head->right=insertNode(head->right, val);
            //if val already exists it will be inserted in the right sub-tree

        updateH(head);
    }else{ //creation of a new node
        head = (tree *)malloc(sizeof(tree));
        head->info = val;
        head->left = NULL;
        head->right = NULL;
        head->h=0;
    }
    return head;
}
```

È una funzione ricorsiva che permette l'inserimento di un nuovo nodo all'interno di un ABR o di creare un nuovo albero nel caso in cui quello preso in esame sia vuoto. Viene ricercata la posizione del valore da inserire sfruttando l'ordinamento delle chiavi all'interno di un ABR e, non appena si arriva ad una foglia, costruisce un nuovo nodo attaccandolo all'albero. La funzione updateH permette l'aggiornamento dell'altezza dell'albero nella chiamata locale.

Elenco dei parametri:

1. head : riferimento all'albero nel quale inserire il nuovo valore
2. val : il valore da inserire

Ritorna il riferimento alla radice della chiamata locale.

Post condizione : Il valore è inserito all'interno dell'albero

Complessità in termini di tempo : O(h) con h altezza dell'albero

Complessità in termini di spazio : richiede lo spazio aggiuntivo per la creazione di un nuovo nodo.

3.2.2 Funzione di cancellazione

Sfruttando la strategia di risoluzione del problema su descritta non è stato necessario implementare una nuova funzione di cancellazione. Infatti questa funzione modifica l'albero che ha come radice il valore da eliminare, sostituendo questo con un nodo del sottoalbero destro che permette di rispettare ancora l'invariante di un albero binario di ricerca e, essendo i duplicati i candidati più adatti, ottenere così lo stesso risultato anche con gli alberi aventi chiavi duplicate.

4 ALTEZZA MEDIA DI ALBERI GENERATI CASUALMENTE

La traccia richiede di dimostrare in via sperimentale la relazione tra il numero di nodi e l'altezza media di alberi generati casualmente.

4.1 DETTAGLI IMPLEMENTATIVI

Per ottenere l'altezza media è stata implementata una funzione di creazione di un albero binario di ricerca avente un numero fissato di nodi generati in maniera casuale. Creando una quantità casuale di alberi in un array è stata poi fatta la media delle loro altezze.

4.1.1 Funzione di creazione albero casuale con n nodi

```
tree *newRandomBst(int nNodes){
    tree *head = NULL;
    while(nNodes>0){
        head = insertBstNode(head, rand());

        nNodes--;
    }
    return head;
}
```

Funzione iterativa che genera un numero casuale e lo inserisce in un nuovo albero tante volte quanto il numero indicato in input. La funzione di inserimento è quella classica di un ABR.

Elenco dei parametri:

- nNodes : il numero di nodi da creare

Ritorna un riferimento all'albero creato

Post condizione: l'albero ha al massimo nNodes elementi. (dipende da quante volte la funzione rand() genera un duplicato).

4.2 RISULTATI DELL'ESPERIMENTO

Dall'esperimento si evincono i seguenti risultati che tentano di esprimere l'altezza come funzione matematica del numero di nodi fissato.

4.2.1 Valori risultanti dalle prove ripetute dell'esperimento

La tabella di seguito mostra l'altezza media della serie di alberi in base al numero di nodi generati casualmente.

Numero di nodi generati	Altezza media della serie di alberi
1	0
2	2
3	1,61
6	3,048
7	3,34
14	4,96
30	6,552
31	6,638
61	7,61
123	8,074
155	8
246	8
495	8
775	8,073
3876	8,061
19380	8,07
96901	8,067
480456	8,08
969010	8,068

4.2.2 Ricerca della funzione matematica dell'altezza media

Dai risultati che si evincono dalla tabella si può notare che l'altezza rimane più o meno invariata con il crescere dei nodi assumendo valori simili alla funzione del logaritmo sul numero di nodi moltiplicata per una certa costante α. Data questa relazione è stato necessario calcolare il valore di α per ogni esperimento effettuato al fine di individuarne il valore massimo assunto per limitare superiormente l'altezza media delle serie di alberi generati casualmente.

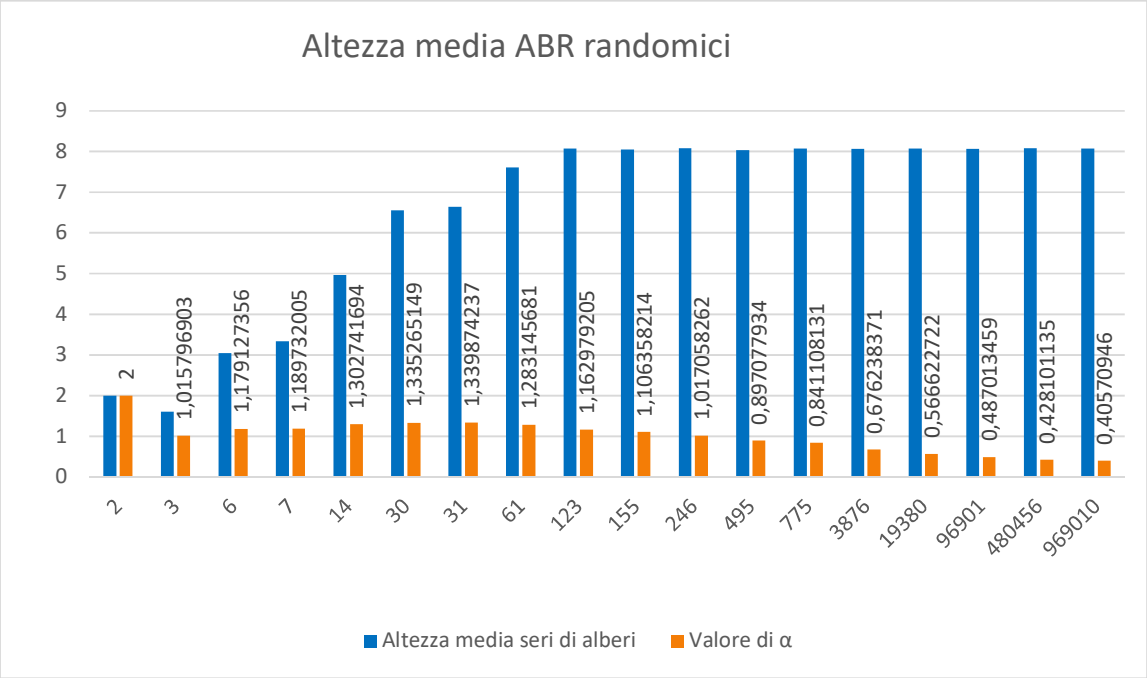
4.2.3 Ricerca del massimo valore di α

Di seguito la tabella che calcola α attraverso la formula $\alpha = \log_2(n)\ /h$ dove *n* è il numero di nodi e *h* è l'altezza media della serie di alberi

n	h	A
1	0	0
2	2	2
3	1,61	1,015797
6	3,048	1,179127
7	3,34	1,189732
14	4,96	1,302742
30	6,552	1,335265
31	6,638	1,339874
61	7,61	1,283146

123	8,074	1,162979
155	8	1,106358
246	8	1,017058
495	8	0,897078
775	8,073	0,841108
3876	8,061	0,676238
19380	8,07	0,566623
96901	8,067	0,487013
480456	8,08	0,428101
969010	8,068	0,405709

Il massimo valore assunto da α è 2 perché, al crescere di n, α assume valori sempre più piccoli. Attraverso il grafico che segue, è ancora più evidente l'andamento di h simile alla funzione $\log_2 n$.



4.3 STIMA ASINTOTICA DELL'ALTEZZA MEDIA

Dai presupposti dei paragrafi precedenti possiamo descrivere l'altezza attraverso la seguente espressione asintotica

$$\boldsymbol{h = O(log_2 n)}$$

5 FUNZIONE MERGE

È la funzione dedicata alla risoluzione del terzo punto della traccia. Ha lo scopo di effettuare l'unione insiemistica tra due alberi binari di ricerca dati in ingresso senza utilizzare memoria aggiuntiva.

5.1 DETTAGLI IMPLEMENTATIVI

La funzione è implementata dal seguente codice

```
tree *merge(tree *t1, tree *t2){
    tree* temp;
    if(t1!=NULL){
        if(t2==NULL){ //ATTACHES THE SUB-TREE
            t2=t1;
        } else if(t1->info>t2->info){
            temp=t1->left;
            t1->left=NULL; //BREAKS THE BST
            t2->right=merge(t1,t2->right);
            t2=merge(temp,t2);
        } else if(t1->info<t2->info){
            temp=t1->right;
            t1->right=NULL;
            t2->left=merge(t1,t2->left);
            t2=merge(temp,t2);
        } else {
            t2->left=merge(t1->left,t2->left);
            t2->right=merge(t1->right,t2->right);
            free(t1); //DELETES THE DUPLICATES IN T1
        }
    }
    if(t2!=NULL) updateH(t2); //UPDATES THE HEIGHT OF THE BST
    return t2;
}
```

È una funzione ricorsiva che, dati due alberi in ingresso, unisce il primo con il secondo.
Si basa su una ricerca della radice del primo albero nel secondo distinguendo così 5 casi innestati:

1. L'albero radicato in t1 è vuoto
2. L'albero radicato in t2 è vuoto
3. La chiave della radice del primo albero è maggiore di quella del secondo
4. La chiave della radice del primo albero è minore di quella del secondo
5. Le due radici hanno chiavi uguali

Elenco dei parametri:

- t1 : riferimento all'albero da unire
- t2 : riferimento all'albero in cui effettuare l'unione

Ritorna t2 aggiornato.

Pre condizione : i due alberi non devono avere chiavi duplicate.

Post condizione : l'albero ritornato contiene l'unione insiemistica dei due alberi e il riferimento al primo albero è inutilizzabile poiché potrebbe perdere dei collegamenti.

5.1.1 Descrizione delle situazioni

Presi in esame i casi in cui si possono trovare i due alberi, di seguito sono descritte le azioni che svolge l'algoritmo su di essi:

- Se l'albero radicato in t1 è vuoto, banalmente si ritorna t2
- Se l'albero radicato in t2 è vuoto, si ritorna t1 che verrà attaccato ad un nodo che permette all'albero binario di ricerca di rispettarne le proprietà
- Se la chiave presente in t1 è maggiore di quella presente in t2, allora viene rotto il collegamento con il figlio sinistro di t1 e verranno effettuate due chiamate ricorsive:
    a. La prima sul sottoalbero destro di t2 e su t1
    b. La seconda su t2 e sul vecchio figlio sinistro di t1 (perché potrebbe avere chiave minore di quella di t2)
- Se la chiave presente in t1 è minore di quella presente in t2 , la situazione è speculare a quella descritta nel punto precedente
- Se le chiavi sono uguali, vengono effettuate le chiamate ricorsive sui rispettivi sottoalberi destri e sinistri ed infine viene deallocato il nodo t1.

5.2 Descrizione della complessità asintotica

Dato che l'algoritmo in esame effettua una ricerca della chiave della radice t1 nell'albero t2 e, se non la trova, attacca tutto l'albero radicato in t1 è facile notare che si esegue una discesa lungo un percorso di t2 per ogni chiamata ricorsiva.

L'algoritmo di merge quindi costa un $\boldsymbol{O(h)}$ dove *h* è l'altezza dell'albero t2 in ingresso. Non occupa spazio aggiuntivo rendendo però l'albero radicato in t1 inutilizzabile.

6 Funzione Rotation

È la funzione che risolve il quarto punto della traccia. Permette di effettuare un numero di volte dato in input una rotazione destra o sinistra.

6.1 Dettagli implementativi

Le funzioni di rotazione destra e sinistra sono le stesse utilizzate per la gestione di un albero AVL e quindi rispettano tutte le proprietà di ordinamento di un albero binario di ricerca. La funzione "master" rotation è implementata come segue

```
tree *rotation(tree* t,int n,int direction){
    if(direction<0 || direction>1){
        ABRERROR=-1;
        return t;
    }
    if(t==NULL){
        ABRERROR=-2;
        return t;
    }
    while(n>0){
        if(direction==0){ //LEFT ROTATION
            t=lRotation(t);
        } else { // RIGHT ROTATION
            t=rRotation(t);
        }
        n--;
    }
    return t;
}
```

È una funzione iterativa che decrementa il valore n passato in input fino a che non assume il valore 0 effettuando la rotazione dettata dal parametro direction.

Elenco dei parametri:

- t : riferimento all'albero da ruotare
- n : è il numero di volte che si deve ruotare l'albero
- direction : verso della rotazione codificato come intero (0 equivale a rotazione sinistra e 1 a quella destra)

Ritorna l'albero aggiornato in quanto le rotazioni modificano la radice.

Pre condizione : il parametro direction deve essere 0 o 1 in base alla direzione scelta. L'albero deve avere figlio destro/sinistro se si vuole effettuare una rotazione destra/sinistra.

Post condizione : la rotazione desiderata è stata effettuata n volte sull'albero. L'altezza dell'albero è aggiornata.

Errori generabili (codice presente nella variabile ABRERROR):

- -1 : il valore del parametro direction è diverso da 0 o 1
- -2 : impossibile applicare rotazione su di un albero vuoto
- -3 : impossibile effettuare rotazione destra perché il sottoalbero destro è vuoto
- -4 : impossibile effettuare rotazione sinistra perché il sottoalbero sinistro è vuoto

Costo in termini di tempo : $\Theta(n)$ dove n è il numero di rotazioni da effettuare dato che le rotazioni singole hanno un costo asintotico costante.

7 BILANCIAMENTO TRAMITE ROTAZIONI

Funzione che permette il bilanciamento tramite rotazioni di un albero binario di ricerca avente nodi di tipo Tree.

7.1 DETTAGLI IMPLEMENTATIVI

L'implementazione della funzione è la seguente

```
tree *balanceBst(tree *t){
    if(t!=NULL){
        while(abs(height(t->left)-height(t->right))>1){ //WHILE BST IS NOT BALANCED
            if(height(t->left)>height(t->right))
                    //VIOLATION IS ON THE LEFT SUB-TREE
                    t=lBalance(t);
            else t=rBalance(t); //VIOLATION IS ON THE RIGHT SUB-TREE
        }
    }
    return t;
}
```

È una funzione iterativa che, dato un albero in ingresso, esegue gli algoritmi di ribilanciamento a destra o a sinistra in base a quale sottoalbero viola la condizione di bilanciamento. A differenza degli AVL per il quale è necessario effettuare al più una rotazione, è stato necessario effettuare un ciclo che termina solo quando l'albero d'ingresso è bilanciato.

Gli algoritmi di ribilanciamento destro e sinistro utilizzano la funzione rotation, descritta nel capitolo 6, con numero di rotazioni da effettuare pari ad uno per ogni rotazione. Di seguito il codice sia del bilanciamento destro che quello sinistro

```
tree* rBalance(tree* t){
    // WHEN THE TREE IS NOT BALANCED BEACAUSE OF THE RIGHT SUB-TREE
    tree *dx=t->right;
    if(dx!=NULL){
        if(height(dx->right)<height(dx->left)){
            t->right=rotation(t->right,1,0);
            //LEFT ROTATION ON THE RIGHT SUB-TREE
        }
        t=rotation(t,1,1); //RIGHT ROTATION ON THE ROOT
    }
    return t;
}

tree* lBalance(tree* t){
    // WHEN THE TREE IS NOT BALANCED BEACAUSE OF THE LEFT SUB-TREE
    tree *sx=t->left;
    if(sx!=NULL){
        if(height(sx->left)<height(sx->right)){
            t->left=rotation(t->left,1,1);
            //RIGHT ROTATION ON THE LEFT SUB-TREE
        }
        t=rotation(t,1,0); //LEFT ROTATION ON THE ROOT
    }
    return t;
}
```

Il costo asintotico in termini di tempo è un *O(h)* dove h è l'altezza dell'albero (caso dell'albero degenere).

8 STAMPA GRAFICA DI UN ALBERO

Le funzioni seguenti sono state implementate in modo tale da poter stampare un albero graficamente come richiesto dal punto opzionale della traccia.

8.1 DETTAGLI IMPLEMENTATIVI

L'idea implementativa è quella di calcolare inizialmente il numero di spazi da dover stampare per ottenere una stampa della radice e, successivamente stampare per livelli i nodi dell'albero

```c
void print(tree *head, int spaces, int height){
    int i=0;
    if (head != NULL){
        if (height == 1){
            //IF THE NODE IS ON THE LEVEL THAT WE ARE CURRENTLY PRINTING ON
            for(i=spaces; i>0; i--)
                printf(" "); //PRINT SOME SPACES
            printf("%d", head->info);
        }
        else if (height > 1){ //GO TO THE LEFT AND THE RIGHT SUB-TREES
            print(head->left, spaces/2, height-1);
            print(head->right, (spaces)+(sizeof(int)), height-1);
        }
    }
}

void printBst(tree *head){
    int h = height(head);
    int area = pow(2,h); // SPACES IS EQUALS TO THE NUMBER OF THE LEAVES IN A FULL
BST
    int i;
    int base = (area*2)/h;
    for (i=1; i<=h; i++){ //TO DRAW IN LEVELS
        print(head,base,i);
        printf("\n\n");
    }
}
```

La funzione printBst utilizza la funzione print che serve appunto a stampare i vari livelli.

La funzione ricorsiva di appoggio print verifica prima se il nodo preso in analisi è all'altezza desiderata e, se rispetta questo canone, stampa spazi e il valore della chiave.

La stampa non è ottimizzata per essere vista in console in quanto essa non è sufficientemente grande per poter contenere tutti i caratteri utili alla rappresentazione dell'albero.