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.

Sommario

[2 Struttura dati Tree 2](#_Toc482106960)

[3 Gestione dei duplicati in un ABR 2](#_Toc482106961)

[3.1 Strategia di risoluzione del problema 2](#_Toc482106962)

[3.2 Dettagli implementativi 2](#_Toc482106963)

[3.2.1 Funzione di inserimento 3](#_Toc482106964)

[3.2.2 Funzione di cancellazione 3](#_Toc482106965)

[4 Operazioni di base 4](#_Toc482106966)

[5 Funzioni offerte dalla libreria 4](#_Toc482106967)

[5.1 Funzioni semplici 4](#_Toc482106968)

[5.1.1 Descrizione funzioni semplici 4](#_Toc482106969)

[5.2 Funzioni complesse 4](#_Toc482106970)

[5.2.1 Descrizione funzioni complesse 5](#_Toc482106971)

[5.3 HeapSort 5](#_Toc482106972)

[5.3.1 Descrizione di heapSort 5](#_Toc482106973)

[6 Esempio d’utilizzo della libreria 5](#_Toc482106974)

# Struttura dati 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.

# 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.

## 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.

## Dettagli implementativi

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

### Funzione di inserimento



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 : Ο(h) con h altezza dell’albero

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

### 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.

# 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.

## 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.

### Funzione di creazione albero casuale con n nodi



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).

## Risultati dell’esperimento

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

### Limite asintotico inferiore dell’altezza di un albero generato casualmente

# Funzioni offerte dalla libreria

Per semplicità si distinguono le funzioni offerte dalla libreria in semplici e complesse come segue

## Funzioni semplici

Sono funzioni ordinarie per la gestione di un heap:



### Descrizione funzioni semplici

* isEmpty: ritorna 0 se l’heap è vuoto, 1 altrimenti
* size: ritorna un intero che rappresenta il numero di elementi presenti nell’heap
* min: ritorna l’elemento di minima priorità (nel caso di min heap il massimo dell’ordinamento) oppure 0 se l’heap è vuoto. Si consiglia di verificare prima se l’heap contiene almeno un elemento utilizzando il metodo isEmpty
* insert: inserisce l’intero k all’interno dell’heap
* delete: rimuove l’intero k dall’heap

## Funzioni complesse

Funzioni utili per la gestione di un heap:



### Descrizione funzioni complesse

* buildHeap: data una collezione di interi in ingresso e la sua dimensione, costruisce un heap e lo ritorna
* freeheap: libera la memoria dinamica allocata dall’heap cancellandolo in maniera corretta
* printHeap: stampa tutti gli elementi dell’heap in base alla loro posizione all’interno dello stesso

## HeapSort

È una funzione di ordinamento che non spreca memoria aggiuntiva e che ha complessità asintotica pari a *Θlog(N)* dove N è la dimensione della sequenza di ordinare data in ingresso.

Sfrutta le proprietà dell’heap ed è definita come segue



### Descrizione di heapSort

Ha come parametri d’ingresso la sequenza di interi da ordinare e la sua dimensione.

Dato che la sequenza è passata per riferimento la funzione in oggetto permette di modificare gli elementi della sequenza in modo da ordinarli in senso crescente.

Non c’è bisogno di creare un heap per poterla utilizzare in quanto questa operazione è svolta stesso all’interno del metodo e questo permette di poter utilizzare l’ordinamento tramite heap senza averne alcuna nozione.

# Esempio d’utilizzo della libreria

Nella libreria proposta vi è anche un esempio di utilizzo della stessa, composto da un menù principale che guida l’utente alla creazione di un heap e da un menù secondario che permette di svolgere operazioni su di esso.