**SECONDA PARTE PROGETTO DI ALGORITMI E STRUTTURE DATI**

Simone Molinaro n°147082 (147082@spes.uniud.it)

Mateusz Kwiatosz n°148330 (148330@spes.uniud.it)

Leo Barić n°137424 (137424@spes.uniud.it)

**Introduzione**

Per l’analisi dei tempi di esecuzione abbiamo utilizzato Java. In questa seconda parte del progetto l’obiettivo era quella di valutare i tempi di esecuzione di tre diverse tipologie di strutture dati ad albero: Binary Search Tree, Red Black Tree e AVL.

Abbiamo affrontato il problema utilizzando due metodi nei confronti dei BST: il caso migliore e il caso peggiore.

**Caso Migliore**

In questo primo caso affrontato abbiamo generato una serie di numeri casuali. Questo significa che tali numeri non presentano alcun ordinamento.

Di seguito si può notare il grafico ottenuto:

Come si può notare, abbiamo dei grafici molto simili. Infatti, per tutte e tre le diverse tipologie di albero la funzione indica un andamento logaritmico. Il risultato può sembrare scontato per i RBT e AVL, in quanto entrambe le tipologie svolgono delle operazioni di ribilanciamento dell’albero per ogni inserimento, mentre non lo è per i BST. I Binary Search Tree non hanno funzioni di ribilanciamento e il loro costo logaritmico è legato al fatto che la generazione dei numeri è stata casuale. Questo ha portato alla costruzione di un albero bilanciato.

Come anticipato, per verificare la nostra affermazione abbiamo deciso di valutare le strutture dati implementate anche per un caso peggiore.

**Caso Peggiore**

Il caso peggiore si verifica nel momento in cui vengono inseriti in sequenza una serie di nodi in ordine crescente (o decrescente). In queste strutture dati come i BST che non fanno operazioni di ribilanciamento dovrebbero avere un costo lineare rispetto al numero di nodi inseriti.

Di seguito il grafico relativo ai tempi di esecuzione del caso peggiore.

Come si può notare in questo caso la funzione per gli alberi RBT e AVL mantiene un andamento logaritmico (o comunque un costo che non varia molto all’aumentare del numero di nodi) mentre, come prevedibile, il grafico per l’albero BST è lineare rispetto al numero di nodi. Questo è dovuto al fatto che inserendo dei nodi con chiave in ordine crescente nell’albero e, non avendo operazioni di ribilanciamento, si sbilancia verso destra. Questo comportamento causa aumento di costo per le operazioni di inserimento e ricerca. Otteniamo così un costo lineare e non più logaritmico.

**Considerazioni sul codice**

Per implementare entrambe le soluzioni ed ottenere dei costi in linea con quanto ci aspettavamo, abbiamo deciso di creare, prima di eseguire le operazioni di ricerca e inserimento, una lista di nodi che avremmo poi utilizzato per generare gli alberi. In questo modo abbiamo azzerato i costi di inizializzazione delle variabili nodo. Questo ci ha permesso di ottenere una misurazione molto più precisa sulle operazioni implementate dagli alberi.

Durante i test e le verifiche che abbiamo fatto, ci siamo resi conto che nel caso peggiore l’albero BST con una n maggiore a 23.000 generava un errore di StackOverflow. Tale errore è dovuto all’elevato numero di chiamate ricorsive che ha causato il rapido riempimento dello stack di memoria associato al processo. Abbiamo sviluppato anche una versione iterativa della funzione di ricerca in modo da poter completare tutte le esecuzioni, ma abbiamo deciso di non caricarla in quanto a livello grafico pensiamo sia già abbastanza chiaro il comportamento di un BST nel caso peggiore.