Data Structures 2022 Assignment 4

Publish date: 24/05/2022

Due date: 14/06/2022

Senior faculty referent: Michal Shemesh

Junior faculty referents: Dor Amzaleg, Nadav Keren

Assignment Structure

0. Integrity statement
1. Backtracking in B-Trees – **Programming and Theoretical section**
2. Backtracking in AVL-Trees - **Programming and Theoretical section**
3. Theoretical questions

Important Implementation Notes

- It is strongly recommended to read the entire assignment and FAQ section (in the moodle) before you start writing your solutions.
- The assignment may be submitted either by pairs of students or by a sole student.
- You may **not** use generic data structures implemented by others. The only exception is using the List<T> and Deque<T> interfaces and their built-in implementations (such as LinkedList, ArrayList and ArrayDeque).
- Your code should be neat and well documented.
- When testing your code, you may use whatever tools you want, including classes and data structures created by others. The restriction above applies only to the code you submit to us.
- You **can assume** that all keys are unique, meaning that no duplicate values are used within our tests.
- Your implementation should be as efficient as possible. Inefficient implementations will receive a partial score depending on the magnitude of the complexity.
- As you have learned, in this course in general and specifically in this assignment the analysis of runtime complexity is always a worst-case analysis.
- Your code will be tested in the VPL environment, and therefore you must make sure that it compiles and runs in that environment. Code that will not compile **will receive a grade of 0**. We provide some basic sanity checks for you to make sure that your code compiles.
- Do not forget to sign the statement in section 0. Your code will be checked for plagiarism using automated tools and manually. The course faculty, CS department and the university regard plagiarism with all seriousness, and severe actions will be taken against anyone that was found to have plagiarized. A submitted assignment without a signed statement **will receive a grade of 0**.

Section 0: Integrity Statement

I assert that the work I submitted is entirely my own.

I have not received any part from any other student in the class, nor did I give parts of it for others to use.

I realize that if my work is found to contain code that is not originally my own, a formal case will be opened against me with the BGU disciplinary committee.

To sign and submit the integrity statement, fill your name/s in the method “signature” in the class IntegrityStatement. If you submit the assignment alone, write your full name, and if you submit the assignment in pair, write your full names separated by “and”. See the comment in the method “signature”.

Section 1: Backtracking in B-Trees

In this section, you are being given a partial implementation of B-Tree as seen in class; This code contains the insertion, split and search operations only. Also given is an implementation of Node class, along with a toString function for the tree.

In addition to the given operations on the data structure, we would like to support the operation:

Backtrack(S) – This method reverts the last *Insert(S,x)* performed on the data structure, and return the data-structure to **exactly the same** state prior to that action. This means that after backtracking, the data structure should look as if the backtracked action was never performed. If no *insert* actions were performed on the data structure, then the method should not change anything in the data structure.

To be able to backtrack an insertion, one can save only up to $\Theta(h)$ additional information for each insertion, that is additional information **proportional** to the **tree's height** in time of insertion.

- You may use **any implementation** of the List<T> and Deque<T> interfaces given by Java.
- The signature of Backtrack(S) is given in a *derived class* of BTree – ***BacktrackBTree*** - and should be implemented in that class.
- You may add additional private methods to the class ***BacktrackBTree***.
- You may add additional protected fields to the implementation of ***BTree***.
- You may add additional code to the implementation of *Insert(S,x)* in ***BTree***.
- You **may not** add any additional code to ***BTree*** otherwise.

Exercises:

1. Give an example of a series of insertions commands to a **degree-3 B-Tree** ($t = 3$), that after that series of insertions, using Backtrack(S) **doesn't yield** exactly the **same tree** as using Delete(S, x), where x is the last inserted key to the tree.

   Fill the example within the function:

   public static List<Integer> BTreeBacktrackingCounterExample()

   By appending the series of numbers to be inserted, by order of insertion. Notice that the **last insertion** will be the one being tested.
2. Implement the function **Backtrack(S)** in the class ***BacktrackBTree*** as described above in minimal time complexity.

Note: For your convenience, there is an enum that describes the possible balance operations possible in a B-Tree.

Section 2: Backtracking in AVL-Trees

In this section, you are being given an implementation of AVL-Tree as seen in class; This code contains the implementation of the insertion operation along the balancing operations of AVL Tree only. Also given are the implementations of a printing method and an in-order iterator.

In addition to the given operations on the data structure, we would like to support the operation Backtrack(S) as described in Section 1, and turn this structure into **Order Statistic Tree**, as seen in Recitation 5 (AVL Trees), that is adding the following operations:

Select(S, i) – Return the value of the i'th smallest value within the tree, for $1 \leq i \leq size(Tree)$, where *select(S, 1)* returns the minimal value, and *select(S, size(Tree))* returns the maximal value in S.

Rank(S, val) – Return the number of elements within the tree whose value is smaller than the value *val*. Notice that *val* does not need to be a member of the data structure.

To be able to backtrack an insertion, one can use only $\Theta(1)$ additional information for each insertion, that is a constant additional information nonrelated to the tree's size. As seen in class, **only** $\Theta(1)$ additional information needs to be stored for each insertion to enable the **Select** and **Rank** operations.

- You may use **any implementation** of Deque<T> or List<T> interface given by Java.
- The signatures of Backtrack(S), Select(S, i) and Rank(S, val) are given in a *derived class* of AVLTree – ***BacktrackAVLTree*** - and should be implemented in that class.
- You may add additional private methods to the class ***BacktrackAVLTree***.
- You may add additional private inner/nested-classes to ***BacktrackAVLTree***.
- You may add additional protected fields to the implementation of ***AVLTree***.
- You may add additional fields and methods to the protected inner-class ***AVLTree.Node***.
- You may add additional code to the implementation of *Insert(S,x)* and *InsertNode(S, node, x)* and *rotateLeft/Right()* in ***AVLTree***.
- You may assume that the index in ***Select(S, i)*** is valid (in the given range).
- You **may not** add any additional code to ***AVLTree*** otherwise.
- You **may not** upload any files that aren't part of the original skeleton files!

Exercises:

1. Give an example of a series of insertions commands to an AVL-Tree, that after that series of insertions, using Backtrack(S) **doesn't yield** exactly the **same tree** as using Delete(S, x), where x is the last inserted key to the tree.

   Fill the example within the function:

   public static List<Integer> AVLTreeBacktrackingCounterExample()

   By appending the series of numbers to be inserted, by order of insertion. Notice that the **last insertion** will be the one being tested.
2. Implement the function **Backtrack(S)** in the class ***BacktrackAVLTree*** as described above in minimal time complexity.
3. Implement the function **Select(i)** in the class ***BacktrackAVLTree*** **in** $\theta(\log n)$ **time complexity**.
4. Implement the function **Rank(val)** in the class ***BacktrackAVLTree*** **in** $\theta(\log n)$ **time complexity**.

Note: For your convenience, there is an enum that describes the possible imbalance cases within an AVL Tree.

Remark: Enums are useful when used alongside the **switch-case** statement structure, described in this Java tutorial by Oracle.

Section 3: Theoretical questions

In the following questions, write short but comprehensive answers. Each entry in the tables below should not exceed 2-3 sentences, and the total time and space complexity should be calculated for a single Insertion/Backtrack operation.

Notice: Your answer should contain **4** tables (described below), one for each of the questions 1-4, and two short answers for questions 5-6. Excessively lengthy answers will be **penalized**!

In the following questions, refer to the code you have written in Sections 1 and 2.

Answer questions **1** and **3** by filling in the following table:

Type	Usage	Space Complexity
The type of the variable	How the information is being used	How much space is needed for storing the information

Answer questions **2** and **4** by filling the following table:

Operation	Number of Repetitions	Total Time Complexity
Operation such as: Search, Balancing operation, Insertion…	The number of times used per backtrack operation	Total time of this operation used for each backtrack operation

1. What is the extra information you have added in the AVL to support the backtrack operation?
2. Give a **high-level** description of the backtracking operation in the AVL tree and its *time complexity*.
3. What is the extra information you have added in the B-Tree to support the backtrack operation?
4. Give a **high-level** description of the backtracking operation in the B-Tree and its *time complexity*.

Danny offered implementing the operation Backtrack(S) on B-Tree by saving a copy of the complete tree before the insertion of each of the keys to the tree. The backtrack operation is done by replacing the current's tree root by the root of the copy. Each answer should be only **a few (2-3) sentences long**.

5. What is the *time complexity* of Danny's implementation of the backtrack operation?
6. Is Danny's solution better than your implementation in Section 1?