Data Structures 2020 Assignment 4

Publish date: 17/05/2020

Due date: 7/06/2020

Senior faculty referent: Dr. Sebastian Ben-Daniel

Junior faculty referents: Maor Zafry, Amihay Elboher

Assignment Structure

0. Integrity statement
1. Reading and asking
2. B-trees:
  i. Complete implementation
  ii. 2-pass insertion
3. Cuckoo Hashing - bonus

Important Implementation Notes

- It is strongly recommended to read the entire assignment at least twice 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 developers of Java, Git projects and so on) unless mentioned explicitly, and just for that section.
- 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.
- Your implementation should be as efficient as possible. Inefficient implementations will receive a partial score depending on the magnitude of the complexity. Basically, if you follow the instructions, i.e. implement according to our explanations and what you so in class, there will be no doubt whether your implementation is efficient or not.
- You may assume that your code will only be tested with proper inputs.
- Your code must not print any output that was not specifically requested in the exercises. The same stands for exceptions.
- Your code will be tested in computers using Java 8 and without any external packages. You must make sure that your code compiles and runs in such an environment. Code that will not compile or run **will receive a grade of 0**.
- Don't forget to sign and submit 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**.
- **Submission guidelines:**
  1. You must **submit** a single zipped folder.
  2. Use **zip** compression.
  3. The file name must be "assignment4" exactly: no spaces, lower case, four...
  4. When decompressing the file, the output must be a single folder with the same name - "assignment4" (no spaces, lower case, four...).
  5. Inside this folder will be:

- all the Java files you got from us, after you changed them according to the tasks, of course. If you added another Java file while solving the tasks, the additional Java files should be in this folder too.
- a PDF file with the credibility statement of task 0 filled and signed. The name of the PDF file must be your ID numbers: if you submit alone – the file name should be just your ID (for example “123456789”), and if you submit in pairs – your ID numbers separated by an underline (for example “123456789_012345678”). If you are male and female, ladies first (not really).

The guidelines are very clear and simple. DO NOT MAKE ANY MISTAKE HERE! No appeals will save your grade in such a case. We will be very sad to give you zero, but we will just do it, so please don’t make us the bad guys.

Section 0: Integrity Statement

Sign this statement and submit it as exercise 0.

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.

Name: _____

ID Number: _____

Signature: _____

Name: _____

ID Number: _____

Signature: _____

Section 1: Reading and Asking

Do not take it personally, but our experience shows that some students tend to read the instructions too fast or to skip here and there. It is reflected, for example, in the questions asked in the assignment forums. By the way, in other courses the situation was worse.

Let us tell you how you should read the assignment in detail:

1. You read this PDF file from the very first word to the very last one. If needed, take a break, a snap, a cup of coffee or clean the apartment or something. Just read it all, and patiently.
2. Read it again and mark or write down important points you see while reading.
3. After 1 and 2 you can start working. Section 3 is for the working time. So, reread the task you work on shortly, and anytime you are unsure about something, take a look in the task description again whether we related to that issue. In case it is more general issue, maybe there was a note about it in the "introduction" part. Didn't find the answer? Go to the FAQ. If the answer is not there too, you are invited to ask us in the forum, but please make a short scan to the former questions. Maybe you are not the first to ask that question.

"Yeah, you know, there are plenty of questions in the forum and it takes a long time to scan it! It takes $O(n)$ time, where n is the number of questions in the forum!!!"

```
public static boolean scanTheForum(Forum f, Question q){
        for(Question ques : f){
                if(ques.equals(q)){ // damn! It's so boring to check it…
                        return true;
                }
        }
        return false;
}
```

Well, that's right, but there are few reasons for that:

1. The titles are not informative. To deal with this problem, use informative titles. You can write in Hebrew if you like and/or feel more comfortable to do so. Mention the task (and sub-task) number(s). If the question is about some specific method – mention it. Add a couple of words about the topic. To cut it short, spend a moment to think about the title. Yes, it's that simple.
2. Forum is not WhatsApp, with all due respect. Write your question clearly. You are allowed to right more than a single sentence of 15 words. You should not be formal and all ("Hi" is OK, not necessarily "Hello", and even skip the greetings), just be clear, in favor of both you and us.

In the end, you will also save time if you work according to these rules. We don't bite nails for you to ask us questions and answer immediately. As a matter of fact, the ones to save the most time are you, because it might take us a few hours to answer, even the day after if you submit the question late. "There are a short path which is long and a long path which is short". Choose yourself. And choose the latter.

Sorry again, but former bad experience led us to spend some moments for that issue, not snobbishness or alike.

Section 2: B-Tree

A note for all this section: the implementation should be as you saw in the lectures. We will mention it again in a few strategical points. One thing we declare here which should not be different from the lectures but better to be mentioned: you always use the left before the right (for example in shift – try the left brother first) and the predecessor before the successor.

Task 2.1: Complete the B-tree Implementation

The serious/relevant parts, at last! In this task you are given an implementation of the B-Tree data structure. As any (basic) data structure, B-Trees also have many different variants, and the implementation you got is also different from the implementation you have seen in the lectures.

In this sub-task you should implement the methods “insert” and “delete”.

For this goal you should make some minor changes in the current implementation, and we leave this problem for you. It’s not complicated…

We left the implementations of “add” and “remove”, and you may find them helpful.

DO NOT MAKE ANY CHANGES IN THE “toString” METHODS!!!

We will use those methods to check your code, and we assume they work exactly as it is in the code you got. No appeals on that issue will be accepted!

The former methods – “add” and “remove” should not work in the code you submit. We will not test them.

After task 2.2 there are some notes about the supplied implementation for your convenience.

<u>Some advice for how to work on this sub-task:</u>

1. You should know well how the algorithms of B-tree work, so repeat on the slides until you feel confident enough (or a bit more than that). You will use the slides a lot while working. In any case of doubt, decide according to the slides of the lectures (and not according to the slides of the practical sessions), and don’t forget – left before right and predecessor before successor.
2. Try to play with the given implementation: create a tree, insert some values, then remove some values and so on, and before printing the tree, think what you expect to get according to what you’ve learned and what you actually get. It will help you to figure out the changes you should make before implementing insertion and deletion.
3. Read the given implementation **carefully**. You can add comments where identifying suspected point/s while reading. It’s highly recommended not to change anything until reading all the code, or at least all the code sections that are related to what you want to change.

4. The order of changes should be done reasonably. We don't give you any clues here, because it's a very important ability you should develop. A simple and basic example is that it's not a good idea to implement the deletion before fixing the insertion.
5. The best way to check your changes is writing methods and call them from a main function. Any such method should test one specific case by building a new tree on operate on it somehow. Write these methods, i.e. the main and the test methods, in a separated class – not in the Java file that contains the B-tree class. When debugging any test, comment out the calls to any of the other test methods, and when you are done, check that/whether you still pass all the former tests.
6. It might be very useful to back up your code occasionally.

You are absolutely invited to share your tests with your classmates. Helping friends (giving and taking) is a crucial part of the university studies. **BUT**, beware not to share anything except for the tests themselves! Dishonesty will be treated harshly, and please don't try to attempt us.

Task 2.2: Two-pass Insertion

Advice: implement this sub-task not before you implement “insert” of task 2.1.

You’ve learned two insertion algorithms in the class – 1-pass and 2-pass. Here is a brief reminder for 2-pass insertion. In the 1-pass insertion, when inserting a new key to the tree, any node in the path from the root to the node the key should be inserted to is splat if found full. The idea is that when splitting a node, the middle key passes to the parent, so the parent should have room for it, or, of course, for the new key that should be inserted to this node, so it should have room for that key. In some cases, the splitting is redundant, since no key will pass from the child node to the node that is splat. This is the motivation behind the 2-pass algorithm.

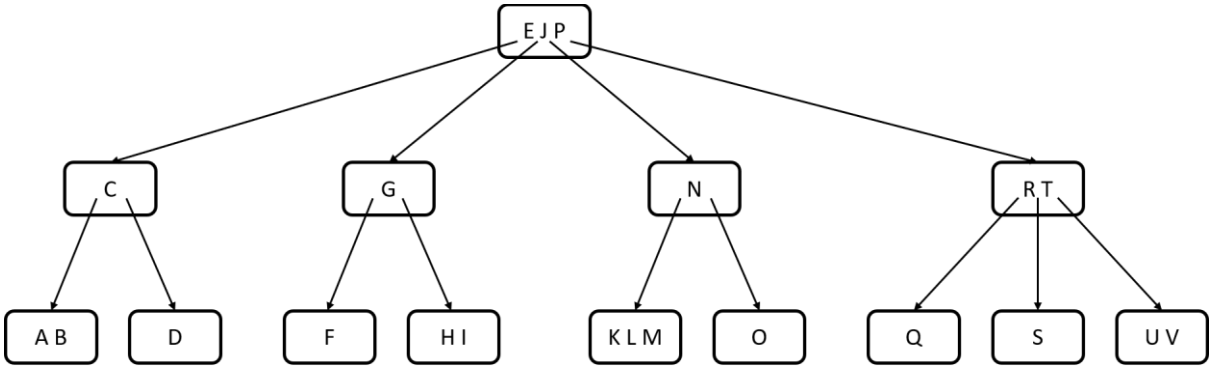
In the 2-pass algorithm only the nodes which should be splat are really splat. You saw in the lectures one way to do that, which declines two times: it goes down from the root to the right node – the first pass, and then it declines again from the uppermost node that should to be splat to the right node. The second decline starts from a node that was found during the first decline.

Here is another way you can implement the insertion (not a better one, just another alternative):

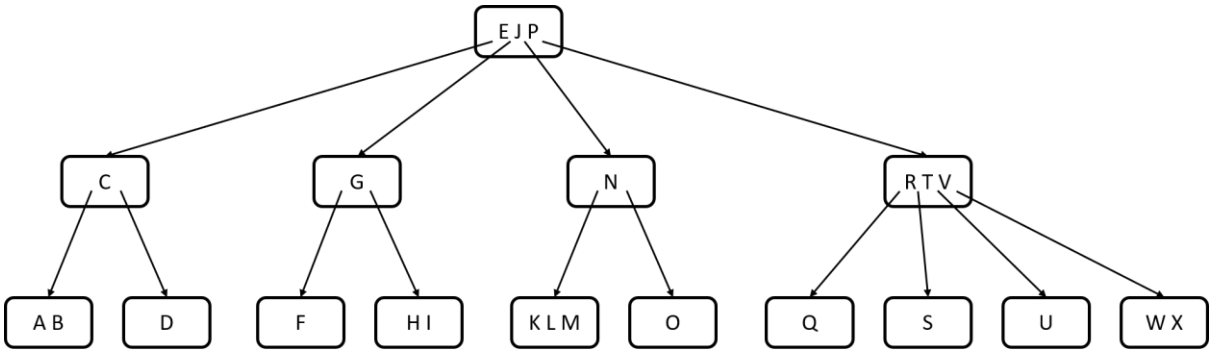
1. Finds the leaf that the key should be inserted to.
   This is the first pass – from the root to the correct leaf. This descent is just a search, meaning to say that no splat occurs while doing it.
2. Ascent while it is needed.
   When is it needed to ascent? If the current node has the maximal number of keys, then there’s no room for a new key to be inserted, so it should be splat. In such case, the parent node should have less than the maximal number of keys a node can hold. If this is not the case, the parent node should be splat, then the parent of the parent node will be checked also, i.e. there is a need to ascent.
3. From the node the algorithm has gotten to, it descents again – the “second” pass – and split any node on the way down to the leaf.

Let’s present an example:

In the tree below, the minimal rank is $t = 2$, so the number of keys in any node is less than or equals to 3. In this tree, the root is full, so, using 1-pass insertion, the root should be splat (what leads the tree to be higher by 1). Unlike that, the 2-pass insertion doesn’t split the root, since no matter which key is to be inserted, there’s room for it and for the needed changes in the lower levels.



For instance, if one inserts the key W to this tree using the 2-pass algorithm, the new key is inserted to the right place of the rightmost leaf without any split. If, in addition, he inserts X as well, then the only node that should be splat is the rightmost one, but the root can still stay as is without any splitting. After such insertions the tree looks like this:



A point of thought: the algorithm doesn't necessarily optimize the insertion in some aspect. Consider the case that the keys Y and Z are inserted too and try to think whether it was worth to use 2-pass instead 1-pass.

In this sub-task, you are required to implement the 2-pass algorithm. You were given an unimplemented method named "insert2pass" that gets T value, which is the key to be inserted, and should work as depicted above, or more accurate – as was taught in the class. The method should return a boolean value: the returned value should be equivalent to the value returned from the "add" method in the given code. In different words – the method "insert2pass" should return true if and only if "add" would return true for the same tree and value parameter. Well, it means that it returns always true, because insertion never fails. Thank us for the answer afterwards…

Note about the motivations behind task 2:

Our major goal in this task is to make you familiar with B-trees, but more generally – with data structures programming. In the "real world" (including the university!), projects are not written from scratch more than once… Normally, you are integrated into a running project, then you study the parts of the code you are going to work on, and then you use them and maybe even change them. And this is an important sub-goal of this task: you should change and complete methods that you didn't write. Probably you would have implemented some things in different ways, at least here and there, but it will be too hard and long work to rewrite it. Therefore, you need to work with what you have gotten: "this is what you have, and you'll win with it"…

Few notes about the code you were supplied:

While reading the code we'd like you to notice some things:

1. The code is strongly based on what you have learned in the course introduction to computer science. We will not repeat on any of the things thoroughly, of course, but we do give you a

short reminder for the Comparable and Comparator interfaces. We also give you a short reminder about generics in Java. The reminders appear here below.

2. Unlike in the algorithm you have studied at class, the nodes in this implementation don't have a field to sign whether the node is a leaf. If you wondered, it's OK, because the only thing to take care of is that the implementation is correct. Using or not this or another field is just a question of implementation, not of correctness.
3. The code uses an imported package from java.util called Arrays. We don't expect you to read the code imported from this package but just to look for the documentation (if needed). It's not that complicated in that case…

Think well about how to read the code and in what order you read it. In this context, note how important it is for the names in the code to be informative. In addition, pay attention to the comments – where they appear and how concise they are. Our expectation is you to keep the same level of readability in the code you write and to comment in a similar way. You don't need to write JavaDoc comments, i.e. when writing comments about (above) a method, just explain it shortly as needed with regular comments (double-slash comments - //).

As we always say about the assignments (especially about the applied ones), that you need to read the whole assignment before starting to implement, we highly recommend you to read the code very well before writing even a single line of code. It's super critical, trust us... It was worth to say it again!

A short reminder about the Comparable and Comparator interfaces:

The Comparable interface requires any class that implements it to implement a method "compareTo". This method compares between the operating object and the object that is given as a parameter. Similarly – but not exactly the same, the Comparator interface requires any class that implements it to implement a method "compare". This method gets two (generic) objects and compares them. The comparison takes care of the order of the parameters and works in a similar way to the method "compareTo" of Comparable: the first parameter is equivalent to the operating object and the second parameter is equivalent to the only parameter.

In case that the operating object / first parameter is greater, the returned value is a positive int (not necessarily 1). If the operating object and the parameter are equal, or equivalent case in "compare" – if the two parameters are equal, the returned value is 0 (as int). In the last case the returned value is some negative int.

A short reminder about generics:

The idea behind generics is to implement only one class which will be compatible for any needed type. This makes the class as general as possible. In our case, for example, we don't want a B-tree for any type (e.g. int, double, String[], unknown types and so forth), so we use a general type (typically T or E are used for the generic type), and we don't assume anything about that type except for the things we declare in the implementation itself (see the next lines).
In our code the generic type T is declared in the signature of the class BTree:

```
public class BTree<T extends Comparable<T>> imlements ITree<T>
```

Therefore, the only thing we know about T is that it's of a Comparable type.

Task 3 (Bonus – 15 points): Cuckoo Hash Backtracking

Someone told us you missed the backtracking idea, so we are happy to give you another occasion! But there is no free lunch, so let's talk about a new data structure – cuckoo hash table.

OK, this task (task 3) is bonus but read it before you leave it aside. We are suspected that you even reached that line. Uff!..

In the cuckoo hash table data structure, in our version anyway (which is not the common version), there is one array and a family of hash functions. The following is a short explanation for what's going on in the code we give you. The explanations here are written in a bit intuitive manner for simplicity. After reading the notes here, it will be much easier for you to read the code. In our implementation there is a use of 2 hash functions only, but generally the family of the hash functions might be bigger, and most of the explanation is for d hash function $h_1 \dots h_d$.

Insertion:

Calculate $h_1(k)$, where k is the key to insert. If the cell of that index is occupied – what called a collision, try to hash k by h_2. It means that we check if the cell of index $h_2(k)$ is occupied. If not, k will be inserted there, and if yes, h_3 will be used. The process proceeds in that way until h_d, and if there is a collision again, k is inserted to the cell of index $h_i(k)$, where i is chosen randomly. The only demand is that $h_i(k)$ will be different from the current index (this is the idea behind the check whether $pos == kick\_pos$ in the helper method of insertion). But what about the former element that was there? This one is taken out (or kicked out…) and go on a journey for a new house. It does exactly the same process, and if it doesn't find an empty cell… well, the expelled element is not a sucker, and it behaves nicely in the same way as before: kicking out the former resident of $h_j(array[h_i(k)])$ (j is chosen randomly again with the mentioned demand) and elegantly sits in the empty cell. Don't worry, the third element is not a child, and it does exactly the same as was done towards him if needed, and so on.

And what if there is a cycle of collisions? A Groicé Broch! There are some ways to deal with that, and we use just one simple way. The data structure holds a list called "stash", and k is inserted to this list.

Deletion:

The key parameter, denoted as k, might reside in any of the cells of indices $h_i(k), \forall i \in \{1 \dots d\}$, so any of these cells is checked if not yet found. When k is found, it is deleted. If it is not in the array, it still may be in the stash. If it is there – delete it, and if not – do nothing. No exceptions should be thrown in any situation.

Search:

Similar to deletion but without the deletion itself: check in the array. Found? Great, otherwise check the stash. Found? Good, else the element is not in the table.

Note about time complexity

Though the complexity issue is out of our objectives in this assignment, you can see that the search and deletion are executed in $O(d)$ time in the **worst case**! d is in $O(1)$, of course, so $O(d) = O(1)$. The insertion complexity may be executed in $O(n)$, but on average it takes $O(1)$ too.

Implement backtracking for cuckoo hash table

You should implement the method "undo", which doesn't get any parameters, and cancels the last insertion. And now in more detail:

We take care just about "effective operations", i.e. successful insertion or successful deletion but **not** search and unsuccessful operations. Now, let assume that the last k effective operations were insertions, then calling "undo" succeeds k times. In the next call (the $(k+1)^{th}$ call) nothing happens. The counting of k calls starts from the last deletion or from the beginning – the latter of them. Canceled insertions are not considered in the counting. For example, if 10 insertions were executed, then 2 deletions, 4 insertions, 2 searches and 1 undo, then $k = 3$. Why? The deletions exclude the first 10 insertions, then we count just the first 3 insertions, because the 4th was canceled by the undo. The searches are ignored, since search is not an effective operation.

This method never throws an exception.

You can add fields, methods and classes as you wish.

You can import any built-in library, unless you find something that approximately solves the task, if such exist… Basically, it means that you can import libraries from java.util, java.lang or any another standard library.

Critical note!!!

Who, for God sake, is the weird guy who decided to call this data structure "cuckoo"?.. OK, so in computer science there are many names of that kind, and even worse. The only reason to use such names is very simple and very good – it reflects some idea that occurs somewhat in the data structure / algorithm / whatever. In our case, the reason is derived from the way the cuckoo bird behaves when it is born (see this staggering video!). When the cuckoo fledgling hatches from the egg, it's in a cruel competition against the other chicks, that are not necessarily hatched yet. Mommy and daddy birds will not be able to feed more than one offspring, so they do not interrupt – and even expect – the competition to be over. Oh yeah, just one winner survives… and takes it all! The only rule is very simple: throw any other guy out of the nest. The next chronicles of the nice family are not relevant very much for now. Well, this data structure is not that cruel (and this task as well), but the idea of throwing someone who resides in someone else's place appears in that data structure. So maybe this is not very a appropriate name (educationally speaking), but you must agree that this is an outstanding name!

It was a critical note, seriously. You probably wonder why. When you study something or work in any area, you must love it. For that, you should be interested in things around, to ask question, to get into the world. Well, nothing would have happened if you didn't know the idea behind the name "cuckoo", of course, but as computer scientists it's good to be interested in the area as a world and not just as an intellectual area, for most of the people anyway. This will make everything much more nice, and you will suddenly figure out that you really like computer science. The love for the area is something that can come naturally, no doubt, but it helps to develop that love, and it is possible. Think about it…

Good luck, and try to enjoy (: (: (: (: (: (:):):