COS 212

Assignment 3

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science
Deadline: 05/04/2017 at 12:00

General instructions:

- This assignment should be completed individually, no group effort is allowed.
- Be ready to upload your assignment well before the deadline as no extension will be granted.
- You may not import any of Java's built-in data structures. Doing so will result in a mark of 0. You may only make use native of arrays where applicable. If you require additional data structures, you will have to implement them yourself.
- If your code does not compile you will be awarded a mark of 0. Only the output of your program will be considered for marks but your code may be inspected for plagiarism.
- Read the entire assignment before you start coding.
- You will be afforded three upload opportunities.

After Completing This Assignment:

Upon successful completion of this assignment you will have implemented your own B^+-Tree with delayed splitting strategies.

Your Task

Refer to section 7.1 (specifically section 7.1.3) in the textbook. You will have to implement your own B^+-Tree as described in the relevant sections. Your B^+-Tree should contain `Integer` objects and should allow for `m` to be specified during object creation.

Download the archive `assignment3.tar.gz` from the course website. This archive contains partial code which you must complete in order to create a B^+-Tree. Each method is described in the comments within the partial code given to you. You have to write code to test your implementation in the `Main.java` file. Your `Main.java` file will be overwritten for marking purposes.

Your tree should allow for insertions and deletions as well as calculating various other properties. You may add any additional helper methods that you require but keep in mind that you must still produce a functioning B^+-Tree.

You must create your own makefile in which your code should be compiled with the following command:

```
javac *.java
```

Your makefile should also include a `run` rule which will execute your code if typed on the commandline as in `make run`.

Background

A B^+-Tree is a more advanced version of a B-Tree. All of the data in the tree is housed in the leaf nodes and the leaf nodes are linked in a linked-list structure. This is known as the `sequence set`. The part of the tree above the leaf nodes is a normal B-Tree and is called the `index set`. The parents of the leaf nodes may contain copies of the keys in their children. If a parent contains a copy of a key, then it is always the first key in the child to the right of the separator key. This arrangement only holds between the leaf nodes and their parents. As mentioned previously, the `index set` is a normal B-Tree and therefore parent nodes of non-leaf nodes do not contain copies of keys from their children.

Splitting Overflowing Nodes

One of the drawbacks of standard B-Trees and B^+-Trees is that overflowing nodes split immediately, even if there is space in other nodes of the tree to take on additional keys. Splits can become expensive when they probagate up in the tree and in the worst case all the way up to the root. Splitting in this extreme case causes much more additional space that will go unoccupied until it is eventually filled with keys, but this might not necessarily happen. Think about inserting the integers 1 to 10 000 into an initially empty B-Tree with `M = 5`. What would you observe?

Sharing Keys

B^*-Trees address the issue of unnecessary splits by delaying splits for as long as possible. This means that unoccupied space in the tree will have a chance to be utilized before splits are performed. For this assignment you will have to implement a version of the B^*-Tree key sharing strategy. An overflowing node has to send off only a single key and it's sibling then takes on one additional key. An overflowing node must always first check if it's left sibling can take on additional keys and then share a key if that is the case. If the left sibling is also full, then the node will have to check it's right sibling and share a key if the right sibling still has space. Be careful when sharing keys between nodes in the `index set` as they have children that also need to be taken into account.

Splitting and Redistribution of Keys

If a node does not have any siblings that can take on additional keys, a split needs to be performed. Refer to *figure 7.13* in the textbook for an example of a split operation. Remember that a copy of the right sibling's first key is sent to the parent for insertion only if the node that splits is a leaf node. In the case that a node in the `index set` splits, a normal B-Tree split operation is performed.

Examples of Output

The examples given in this section pertain to the standard B-Tree given in `figure 1` but they will apply exactly the same to the B^+-Tree which you have to implement. Consider the B-Tree in *figure 1*, assuming all insert and delete operations have been completed up until now. For this tree, $M = 5$. For the assignment, you will have to cater for any positive uneven M which is also large enough to allow for the operations to execute correctly.

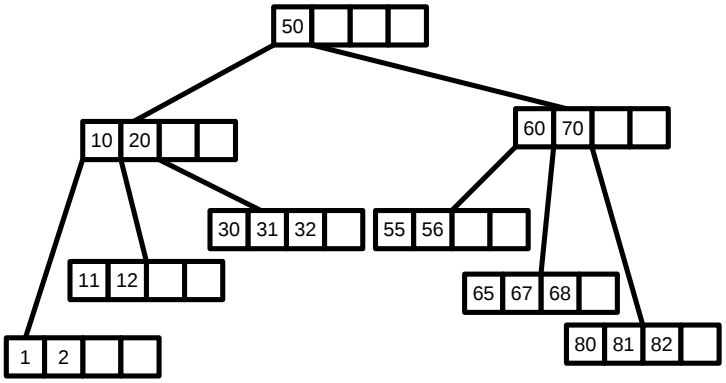

Figure 1: B-Tree

Nodes as Strings

A number of the methods in the `BPlusTree` class will require that nodes be represented as strings. The string representation of the node containing the keys 65, 67 and 68 is:

```
[65][67][68]
```

The string representation of the root node would be:

```
[50]
```

You must only show the keys currently contained in the nodes, each key enclosed with square brackets. The elements in the node should appear in ascending order from left to right. There should be no white space in the string representations of your nodes.

Searching

You will be required to return a string representation of all the nodes considered during a searching operation. The nodes should be comma separated with no white space. In the tree in `figure 1`, searching for the element 67 should yield the string:

```
[50],[60][70],[65][67][68]
```

A null node, where applicable, is indicated with the string:

```
*NULL*
```

Null nodes are used to indicate that the element searched for does not appear in the tree. Therefore, searching for the element 100 should yield the string:

```
[50],[60][70],[80][81][82],*NULL*
```

Once again, you should not have any white space in your strings.

Implementation Details

When nodes underflow or overflow they need siblings to participate in operations to restore the tree. The order in which you consider siblings may affect the resulting tree's representation. When siblings need to participate in actions, always consider the left sibling of a node first.

Overflowing Nodes

When a node overflows, first check if the left sibling has space to take additional keys. If the left sibling is full, check the right sibling. A sibling that still has space should take on only one additional key. If both siblings are full, then a split will occur. Keep in mind that the keys contained in the `index set` where copies from actual keys in the `sequence set`. They are not always deleted when a deletion does not cause an underflow in the `sequence set` because they remain valid separator keys. When redistributing nodes in the `sequence set`, you must ensure that the separator key is still valid. If it needs to be updated, it is updated to be a copy of the right sibling's first key between the sibling that share keys. Remember to treat the index set as a standard B-Tree with the key sharing strategy implemented under the pretense that all keys in the index set are "actual" keys.

Underflowing Nodes

If a node underflows, first check if the left sibling has enough keys to share. If it doesn't, check if the right sibling can share keys. A sibling that can share keys should only relinquish one key for the sharing operation. A sharing of keys in leaf nodes may invalidate the separator key in the parent. If the separator key in the parent becomes invalid, overwrite it with a copy of the right sibling's first key. If neither of the siblings have enough keys to share, then the underflowing node merges with its right sibling. The `index set` is treated as a standard B-Tree with the key sharing strategy implemented except if nodes in the index set merge. See the next section for further details.

Notes on Deletion in B^+-Trees

Deletion always starts in the leaf level as all of the keys are contained in the leafs. Separator keys are not deleted or updated in the `index set` if they remain valid. In general, separator keys are only deleted due to merging operations that start in the `sequence set`. Also be careful not to copy down keys from the parents of the nodes in the `sequence set`. Remember to treat the `index set` like a standard B-Tree with the key sharing strategy implemented except when nodes merge. In a standard B-Tree, if two siblings merge, then the separator key in their parent is moved down into the newly merged node. In B^+-Trees, instead of moving separator keys down from parents to merged nodes, the separator keys are simply deleted from the parents.

Submission Instructions:

Your `Main.java` file will be replaced for marking purposes. Once you are satisfied that your program is working, compress all of your code, including a working `makefile`, into a single archive (either .tar.gz, .tar.bz2 or .zip) and submit it for marking to the appropriate upload link (**Assignment 3**) before the deadline. Include any additional files you've written in your archive.