EECS484 W17 Database Management Systems
Project 3 (100 points)
Due 11:55PM, March 24th 2017

Please read the following instructions before starting the project:

There are two parts to this project:

The **first part** will require you to implement certain B+ tree functions. We have provided you with pieces of skeleton code so that you don't have to start from scratch. It is advised that you familiarize yourself with B+ tree operations (thoroughly) before starting.
For this part, please submit a .zip of your .cpp, .h and Makefile that are appropriate for your build of the B+tree.

The **second part** provides you an opportunity to gain hands-on experience in creating indices and understanding query optimizers in Postgres (note: this is not sqlplus/oracle). For this part, please fill out the Google form found at:
https://docs.google.com/a/umich.edu/forms/d/e/1FAIpQLSdmq51YIhqmM0U03JIMSc4Yq-KAzQmlzJjrSY0fKqoxq9-Y7A/viewform

You are allowed (and encouraged) to work in pairs on this project.

If you are working in a pair:
Please join the same group with your partner **on the upcoming auto grader <u>before</u> <u>making any submission</u>** (refer to the announcement for more information on joining groups) and indicate this also on the Google form. We will be expecting only one submission per group.

It is advisable to use some revision control system.
HOWEVER DO NOT *post code related to this project to a public resource or repository. Doing so will be considered a violation of the honor code. (To make a private repo, you can upgrade to a student github account that comes with 5 private repo's, or use bitbucket, or any other similar alternatives)*

Late days: *A flat 15% penalty on the whole project will be deducted if you submit within 4 days after the deadline.*

By submitting this project, you are agreeing to abide by the Honor Code:
I have neither given nor received unauthorized aid on this assignment, nor have I concealed any violations of the Honor Code.

Part 1: B+ Tree Implementation (80 points + 5 bonus)

UPDATE & CLARIFICATION

- **"Size" variable of B+ tree refers to the number of data records in the tree. You should increment for each successful insertion and decrement for each successful removal.**

For this part of the project, you are expected to implement the basic functionality of a B+ tree like the algorithms we have talked about in class. Specifically, you must implement the `Btree::insert()`, `Btree::remove()`, and `Btree::search_range()` functions. **Duplicated data entries should be rejected from being stored into this B+ tree.**

We have provided you with some skeleton code as well as the implementations of some basic functions - so that you don't have to start from scratch. It is advised that you follow the current design of our code, as we believe this design will make it easier to implement your code. However, If you are really motivated, feel free to change the design (see details about this at the end).

We have provided you with an implementation of `Btree::search()` as a simple example of how you can interface with the current skeleton code we have given you.

We have also provided you with some basic operations for Bnode's (B+ tree node classes). You can find these in `bnode_inner.h`, `bnode_leaf.h` and their implementations in `impl.cpp`. Feel free to add more functionality, if you believe you need them. We have purposely not given you the implementation for:

```
Bnode_inner::merge()
Bnode_inner::redistribute()
Bnode_leaf::redistribute()
Bnode_leaf::split()
```

What you need to do:

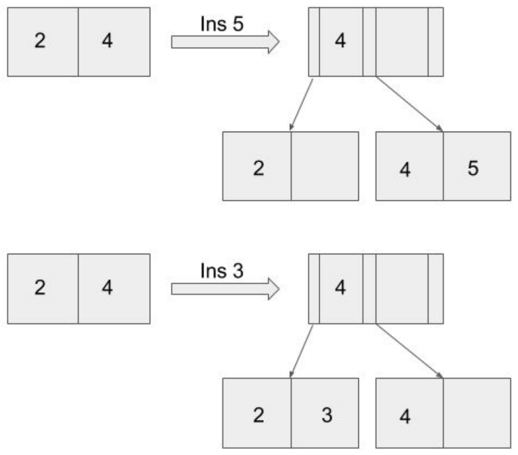
1. Implement `Btree::insert()`
   - Prefer splitting nodes over redistribution in your implementation.
2. Implement `Btree::remove()`
   - Prefer re-distribution over merging in your implementation.
3. Implement `Btree::search_range()`
   - Include all Data* that are between the given range (inclusive).

Specifics on how to merge, redistribute or split are listed on the next few pages.

Follow these general rules when implementing the above functions:

1. When splitting:
    - Push the smallest value of the new right split node up to the parent.
    - *For leaf nodes*: First split existing nodes evenly between the left and right node. Then insert the new value into the appropriate, valid node based on the existing values and the insert value (this is a different choice than the choice we have typically been doing in class). When both nodes are valid, add the value to the left node.

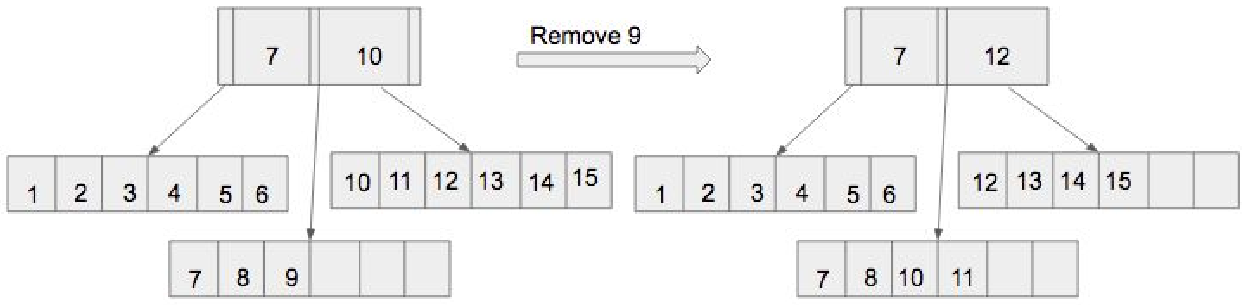
      Example with `BTREE_LEAF_SIZE = 2`, `BTREE_FANOUT` = 3:



[1]

2. When redistributing:
    - *For leaf nodes:* Update the appropriate parent node with the smallest value in the right node. Redistribute with any valid left or right neighboring nodes, not necessarily sibling.
    - *For inner nodes:* Only redistribute with sibling nodes.
    - Prefer redistribution with right node over left node.
    - Rhs (right hand side) and lhs should have the same number of values if possible. When this is impossible (due to odd number of values) - rhs should have 1 more values than lhs.

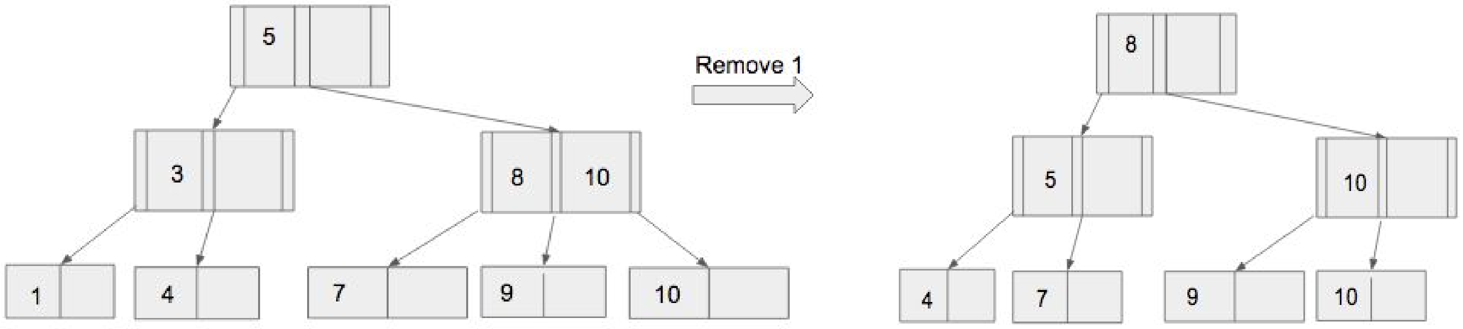
      Example with `BTREE_LEAF_SIZE = 6`, `BTREE_FANOUT` = 3:



3. When merging:
    - *For leaf nodes:* Merge with any valid left or right neighboring nodes, not necessarily sibling.
    - *For inner nodes:* Only merge with sibling node.
    - Prefer merging with right node over left node.
    - Always merge the right node values into the left node, then delete the rhs node and do stuff to rebalance the tree
    - Merge can cause, redistribution, make sure to follow the rules for redistribution.

      Example with `BTREE_LEAF_SIZE = 2`, `BTREE_FANOUT` = 3:

For further clarifications that you think this spec might have missed, feel free to post on piazza and one of the instructors will answer you.



Other notes:

- Remember to maintain the private variable `size`(**number of data records**) while implementing the above functions.
- *Always* be sure to use the constants defined in constants.h (such as `BTREE_LEAF_SIZE`, `BTREE_FANOUT`). `BTREE_FANOUT` represents the maximum fanout of each inner node.
- The capacity of each inner node must always be in between `(BTREE_FANOUT-1)/2` and `BTREE_FANOUT-1` (except when this is the root node).
- The capacity of the leaf nodes must be between `BTREE_LEAF_SIZE/2` and `BTREE_LEAF_SIZE` (except when this is the root node).
- `BTREE_FANOUT` is guaranteed to be odd and it is greater than 2.
- `BTREE_LEAF_SIZE` is guaranteed to be even and it is greater than 1.
- The B+ tree should initially start empty, with its root node being a Bnode_leaf*. When this leaf node has to split, then change the root node to be a Bnode_inner. The root should turn back into a Bnode_leaf when there size<BTREE_LEAF_SIZE. Otherwise, the root node should stay as an inner node.

Test cases

We have provided you with a printing function that would print out your B+tree to an output stream. We will be comparing this print output to the solution's print output to determine correctness. *Note: these print lines all start with "@@@ …", such lines will be the only lines that will be compared during grading (so feel free to debug info, but these lines SHOULD NOT start with @@@).*

We have provided you with a mainfile.cpp that includes some test cases including the examples above. The equivalent output for two settings (BTREE_FANOUT=3, BTREE_LEAF_SIZE=2 and BTREE_FANOUT=3, BTREE_LEAF_SIZE=6) has been provided in the file `expected_3_2.out` `and expected_3_6.out, respectively`. Feel free to use this as a start to figure out if your B+tree class is correct or not. At least two test cases will be from these public test cases but we will have additional test cases. Therefore, it is highly recommended for you to create your own test cases and parse the output yourself to see if you have implemented the algorithm correctly.

Also, a `Btree::isValid()` function has been provided for you to test if your B+tree has maintained ordering correctly. This **does not** check if your B+ tree is implemented correctly. It serves merely as a sanity check and catch obvious errors (obvious, as in, obvious to a computer, but not necessarily to your eyes).

OPTIONAL: Test Case Competition [5 bonus points]

To get you thinking about corner cases we will also be offering up to 5 bonus points for groups that come up with good test cases. You can submit up to 5 test cases (named `test1.cpp`, `test2.cpp`…, `test5.cpp`) with a maximum of 15 insertions or deletions in total per test case. Assume the following `BTREE_FANOUT` and `BTREE_LEAF_SIZE` values for the 5 test cases:

- test1.cpp: `BTREE_FANOUT=3 BTREE_LEAF_SIZE=2`
- test2.cpp: `BTREE_FANOUT=3 BTREE_LEAF_SIZE=4`
- test3.cpp: `BTREE_FANOUT=3 BTREE_LEAF_SIZE=6`
- test4.cpp: `BTREE_FANOUT=5 BTREE_LEAF_SIZE=4`
- test5.cpp: `BTREE_FANOUT=5 BTREE_LEAF_SIZE=6`

You will receive points based on how many test cases your peers fail to pass correctly. The group with the most test cases that most other groups fail will receive 5 points, runner-up will get 4 points… etc. Groups with ties shall receive the same number of points.
Each of these test files should contain a `main()` and should compile correctly when replaced with `mainfile.cpp`

Part 1 Submission & Grading Details

Please make a group submission if you have a partner. Submit to the auto grader:

- ☐ p3.zip (case sensitive, zip file) including:
    - *.cpp
    - *.h
    - Makefile
    - testX.cpp (where X is a number 1-5, see below)

You will receive points (out of 80) based on how many of the auto-grader test cases you pass. The bonus points will depend on how many test cases your peers fail to pass correctly, as described above.

For efficient grading, ensure that, once unzipped, we can build your project by typing

```
$    make
```

which will *create an executable* called `main`. **What is currently in your `mainfile.cpp` does not matter, we will be supplying our own `mainfile.cpp` when grading your projects.**

Please **DO NOT EDIT the following files**:

- impl.cpp   (make sure that you do not change the prototypes)
- constants.h

When grading, these files (in addition to `mainfile.cpp`) will be replaced with our own versions, so make sure that you do not include functional code in them.

We will be assuming that we can create an object called `Btree`, *which has at least the functions that we have given you in the skeleton code*. You can add more member functions to this class, if you want, but make sure it will compile, as described above.

You are free to change whatever else you like. There is a certain amount of flexibility in this project. *If you are unsure of what might break the grading of this project, you are welcomed to post on piazza.*

WARNING: Any submissions should be entirely your own work. Any submissions of code that is not your own is a violation of the Honor Code and will have severe consequences. Any attempt to using network libraries and printing result of the test cases without B+ implementation also violates the Honor Code.

Part 2: Postgres (20 points)

Instructions for logging into the Postgres server as well as some troubleshooting information can be found at:
https://docs.google.com/a/umich.edu/document/d/1dSXxniWAuOpN7wFv3HGI69Ab1tZ3NxmFNrs_2z9c2xg/edit?usp=sharing

For this part, please fill out the Google form found at:
https://docs.google.com/a/umich.edu/forms/d/e/1FAIpQLSdmq51YIhqmM0U03JIMSc4Yq-KAzQmlzJjrSY0fKqoxq9-Y7A/viewform

No other submission is necessary for this part. If you are working in a group, make sure to indicate this in the form. Only one submission is necessary. And you are free to re-submit before the deadline.