**EECS484 W17 Database Management Systems**

**Project 3 (100 points)**

**Due 11/17/2017**

**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://goo.gl/forms/0acXbn1LQRl9Wik92>

*You must work in pairs on this project.*

**Making partner submissions:**

Please join the same group with your partner **on the upcoming auto grader before making any submission** (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 version 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)*

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)**

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:**

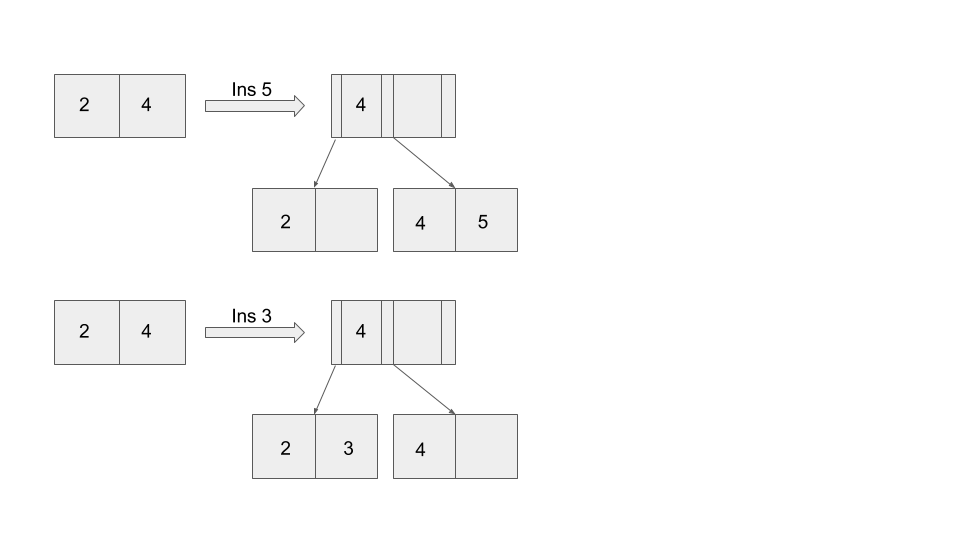
1. Implement Btree::insert()
   * Prefer splitting nodes over redistribution in your implementation.
2. Implement Btree::remove()
   * Prefer re-distribution over merging in your implementation.
3. Implement Btree::search\_range()
   * Include all Data\* that are between the given range (inclusive).

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

**Follow these general rules when implementing the above functions:**

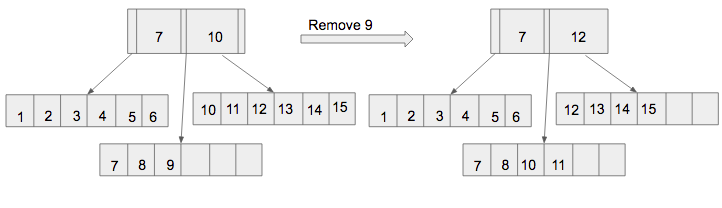
1. When splitting:
   * Push the smallest value of the new right split node up to the parent.
   * *For leaf nodes*: First split existing nodes evenly between the left and right node. Then insert the new value into the appropriate, valid node based on the existing values and the insert value (this is 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:**

****

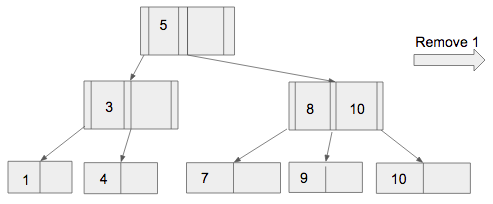
1. When redistributing:
   * *For leaf nodes:* Update the appropriate parent node with the smallest value in the right node. Redistribute with any valid left or right neighboring nodes, not necessarily sibling.
   * *For inner nodes:* Only redistribute with sibling nodes.
   * Prefer redistribution with 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.

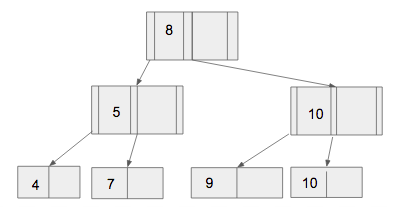
**Example with BTREE\_LEAF\_SIZE = 6, BTREE\_FANOUT = 3:**



1. 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. The variable size refers to the total number of data records stored in the tree, so it should be incremented every time a data record is successfully added, and decremented every time a data record is successfully removed.
* *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). You might want to create your own functions to check the validity of other aspects of your B+ tree.

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**](https://grader484.eecs.umich.edu/):

* 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* calledmain. **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/1h_0B1BXoaJLxeIIyFIan-_SaVrROBnbD8lbM64Euy84/edit?usp=sharing>

For this part, please fill out the Google form found at:

<https://goo.gl/forms/0acXbn1LQRl9Wik92>

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.

**Appendix: Dynamic Casting**

Parts of this project will likely require you to use dynamic casting of pointers. This appendix will offer a short explanation and example, which should be sufficient for what you need to do in this project.

Dynamic casting a pointer essentially means trying to change it from a pointer for one type of object to a pointer for another type. This is “dynamic”, as opposed to static casting, because the validity of the change is checked during the program’s run time, as opposed to compile time. Theuse of dynamic casting in this project will generally be to try and change pointers of a parent type (for example, Bnode) into pointers of an inherited type (for example, Bnode\_inner).

You can find an example of this in the given btree.cpp file, in the function Btree::search. Here’s some of the code:

Bnode\* current = root;

Bnode\_inner\* inner = dynamic\_cast<Bnode\_inner\*>(current);

while (inner) {

int find\_index = inner->find\_value\_gt(value);

current = inner->getChild(find\_index);

inner = dynamic\_cast<Bnode\_inner\*>(current);

}

So first, the Bnode pointer “current” is assigned. “current” will certainly point to a Bnode object, which could be of type Bnode, Bnode\_leaf, or Bnode\_inner. After this, the pointer is dynamic casted to a Bnode\_inner pointer. If the object that “current” pointed to is truly a Bnode\_inner object, this will succeed, and the “inner” pointer will point to the object.

However, if “current” points to a Bnode\_leaf instead of a Bnode\_inner object, then dynamic casting it to a Bnode\_inner pointer will fail, and the “inner” pointer will be assigned the value *nullptr*, which just means that it doesn’t point to anything.

A pointer can be treated as a boolean value: if it’s a valid pointer, it will be treated as TRUE, whereas if it’s *nullptr*, then it will be treated as false. As such, the while loop will be entered if “current” did point to a Bnode\_inner object, and it will be skipped otherwise.

Once the code is inside of the while loop, you can guarantee that “inner” points to a valid Bnode\_inner object. At that point, you can safely call functions like find\_value\_gt, which is only defined for Bnode\_inner objects.

As such, the effect of this code is to continuously iterate through the children of Bnode\_inner objects, reassigning “current” and “inner” until an it reaches an object that is not of type Bnode\_inner. At this point, “current” will be pointing to the Bnode child, and “inner” will be *nullptr*, as the loop only ends when the dynamic cast fails.

After this, the following code executes:

Bnode\_leaf\* leaf = dynamic\_cast<Bnode\_leaf\*>(current);

assert(leaf);

Since “current” does not point to a Bnode\_inner object once the loop is complete, it should be pointing to a Bnode\_leaf object instead (In this project, you should never be using any parent Bnode objects; only Bnode\_inner and Bnode\_leaf).

As such, this new dynamic cast should succeed, and “leaf” will point to a valid Bnode\_leaf object, so that you can use functions that are unique to Bnode\_leaf. The assert statement is just there to guarantee that the above logic is correct: namely, that any Bnode pointer is either a Bnode\_inner or a Bnode\_leaf pointer.