Assignment 1
Prefix Key Compression

For Assignment 1, you will be implementing prefix key compression for B+ Trees in the PostgreSQL engine. Prefix key compression will be described in class, and is described in section 10.8.1 of your text book. For this assignment, you are only asked to modify a single file, `nbinsert.c`. To grade your assignment, we will compile PostgreSQL with your modifications, and test the result for correctness.

1 Implementing Prefix Key Compression

For this assignment, you will be editing the `_bt_prefixKeyCompress` function in the file

`src/backend/access/nbtree/nbtinsert.c`

in the PostgreSQL source code. You should not need to modify any other functions for this project. We have modified the standard version of `nbtinsert.c` to include a skeleton of your solution.

We are making life simpler by looking at a special case: single-column indexes of SQL type `text` (PostgreSQL 's variable-length string data type). The compression logic in `_bt_prefixKeyCompress` is invoked only in this special case. For any other index key configuration, the code you write should simply not get called. Certainly the system should never crash on other index configurations.

In general, when a B+-tree leaf node split takes place, half of the data entries on the original node are moved onto a new "righthand" node - this happens in a routine called `_bt_split` in `nbtinsert.c`, which you should examine. The smallest entry in the resulting righthand node, which would ordinarily be copied up unchanged to the parent node during split, will have its key prefix-compressed before the copy occurs. PostgreSQL stashes a version of the resulting compressed key in a special slot on the original page (the so-called "high key" mentioned in the comments in `_bt_split`, which is maintained for concurrency control reasons that will are of no concern for this assignment. The compression is done by a routine we call `_bt_prefixKeyCompress`. The arguments to `_bt_prefixKeyCompress` are:

`Relation rel:` a data structure representing the actual index file, which is of type `Relation`.

`BTItem lowItem:` the highest index key remaining on the original (left) leaf page after the split, i.e. the key that immediately precedes, in alphabetical order, the one being compressed.

`BTItem highItem:` a fresh copy of the lowest index key on the new rightmost node, which we can prefix-compress before it gets copied up.

We have given you skeleton code in `_bt_PrefixKeyCompress` that extracts the actual text for the keys from the `BTItem` data structures for `lowItem` and `highItem` (which include both a key and a pointer) to eventually generate two corresponding C `char *` pointers, `lowp` and `highp`. Given these, you need to do three things:

1. Figure out how much you can truncate the string pointed to by `highp` by comparing it to the string pointed to by `lowp`. The length of the truncated string should be just long enough to distinguish the two.
2. Update the length field of the `highText` structure to set it to the length you computed in the previous step. See the comments in the code about including 4 bytes for the `vl_len` space.
3. Set the `toReturn` variable to the absolute difference in the length of the high key, pre- and post-compression.

As background, you should read through the code where `_bt_PrefixKeyCompress` is called, and generally poke around in `nbtinsert.c`. You can look for comments that say "CS448" to find things we added to `nbtinsert.c` to support prefix key compression and debugging. We have also provided code to output the contents of the B+-tree as text:

`_btdump(Relation r):` takes a B+-tree `Relation` structure, and outputs its pages in whatever order they physically appear in the file. This is available for you to call from the debugger, e.g. by typing `print _btdump(rel)` from a breakpoint in `_bt_prefixKeyCompress`. Be aware that it generates a lot of output.

`_btdumppage(Relation r, Buffer buf):` is the inner loop of `_btdump` that prints a particular index page from the buffer pool.

We have also put a call to `_btdumppage()` into the routine `_bt_insertonpg()` in `nbtinsert.c`, so that you can see the effects of your compression as internal index keys are generated and inserted. (If you want to call `_btdumppage` from the debugger, you first have to pin your page into the buffer pool via the `ReadBuffer()` function; see the code for `_btdump` for details.)

2 Deliverables

Please submit a single file: `nbtinsert.c`. Submit the file using the `submit` command, like this:

```
submit cs448 a1 .
```

Don't forget the dot (which refers to the current directory) at the end of this command. Make sure that the file `nbtinsert.c` is in the current directory before executing the `submit` command.

3 Grading

We will test your code on its ability to compress keys. We will also check to see that your index still works properly. You can verify these properties yourself by creating a table and an index, and loading some data into your database. The combination of debug messages from `_bt_prefixKeyCompress` and output from `_bt_dumppage` should help you validate your implementation.

We have provided some sample data for you to experiment with. You are, of course, free to use your own sample data as well. To load our data, first use `createdb` to create a database (refer to the PostgreSQL setup instructions for more details). To create and load the index, launch `psql` on your database and execute the following commands:

```
CREATE TABLE dict (id int4, word text);
CREATE INDEX dictix on dict(word);
COPY dict FROM '/u/cs448/public/a1/words.txt' WITH DELIMITER AS ' ';
```

Note that there's a space between the single quotes at the end of the `COPY` command. If you are working on your own machine, you can download a copy of `words.txt` using the link on the course web page.

You will see a lot of debugging output. You can examine the debugging output there to see the index page dumps, and any key compression that is happening.

Important note: There are 3 lines in the `_bt_prefixKeyCompress()` routine that must remain unchanged in order for the autograder to evaluate your code. They are clearly marked in the code. Be sure not to modify or delete them.