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 is described in section 10.8.1 of the 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 changing 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, and to generate some log output for debugging and marking purposes.

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 pointers to the actual text for the keys from the `BTItem` data structures `lowItem` and `highItem`. 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 `highItem` to truncate it by the amount 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.

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 Evaluation

To evaluate your code, we will check to see that your index still works properly by running queries that use the index. In addition, we will check the server log output produced by PostgreSQL to see whether compression is working properly.

You can test your implementation by creating a table with a `text` column, creating an index on that column, and then loading some data into the table. 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, as was described in Assignment 0. (Be sure that your database directory was initialized using the `--locale=C` option to `initdb`.) 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/words.txt' WITH DELIMITER AS ' ';
```

Note that there's a space between the single quotes at the end of the `COPY` command. You'll find these commands in a SQL command file located at `/u/cs448/public/words.sql`. You can type in these commands at the `psql` prompt or, better yet, just tell `psql` to load these commands from the command file using the `psql` command:

```
\i /u/cs448/public/words.sql;
```

If you are working on your own machine, you can copy both `words.txt` and `words.sql` from `/u/cs448/public` to your machine. Of course, you'll need to change the `COPY` command in `words.sql` to reflect the actual location of the data file (`words.txt`) on your machine.

The `_bt_PrefixKeyCompress` function includes some `elog` calls that produce server log data describing that each call to `_bt_PrefixKeyCompress`. To see these data, which are useful for debugging, you will need to start the PostgreSQL server with debugging turned on. To do this, give the `-d 1` option to the PostgreSQL server when you launch it, as in this example:

```
postmaster -d 1 -p <port-number> -D $HOME/pgdb
```

To generate even more debugging output, you can use `-d 2` instead of `-d 1`. Among other things, this will cause the contents of internal B+-tree nodes to be dumped to the log each time a new entry is inserted into them. The is done by a call to `_btdumppage()` in the routine `_bt_insertonpg()` in `nbtinsert.c`. Be aware that this will generate a lot of log output! There is also a function called `_btdump(Relation r)`, which takes a B+-tree `Relation` structure and outputs all of the pages of the tree in whatever order they physically appear in the file. This function is not currently called by the code in `nbtinsert.c`, but you may find it useful to invoke this function from the debugger. Since `_btdump` just calls `_btdumppage()` for each page in the B+-tree, it too will generate a lots of output.