PROJECT #0 - C++ PRIMER

Do not post your project on a public Github repository.

Last Updated: Aug 29, 2023

Overview

All the programming projects this semester will be written on the BusTub database management system. This system is written in C++. To make sure that you have the necessary C++ background, you must complete a simple programming assignment to assess your knowledge of basic C++ features. You will not be given a grade for this project, but you **must complete the project with a perfect score** before being allowed to proceed in the course. Any student unable to complete this assignment before the deadline will be asked to drop the course.

All of the code in this programming assignment must be written in C++. The projects will be specifically written for C++17, but we have found that it is generally sufficient to know C++11. If you have not used C++ before, here are some resources to help:

- 15-445 Bootcamp, which contains several small examples to get you familiar with C++11 features.
- A short tutorial on the language.
- cppreference has more detailed documentation of language internals.
- A Tour of C++ and Effective Modern C++ are also digitally available from the CMU library.

If you are using VSCode, we recommend you to install CMake Tools, C/C++ Extension Pack and clangd. After that, follow this tutorial to learn how to use the visual debugger in VSCode: Debug a C++ project in VS Code.

If you are using CLion, we recommend you to follow this tutorial: CLion Debugger Fundamentals.

If you prefer to use `gdb` for debugging, there are many tutorials available to teach you how to use `gdb`. Here are some that we have found useful:

- Debugging Under Unix: gdb Tutorial
- GDB Tutorial: Advanced Debugging Tips For C/C++ Programmers
- Give me 15 minutes & I'll change your view of GDB [VIDEO]

This is a single-person project that will be completed individually (i.e. no groups).

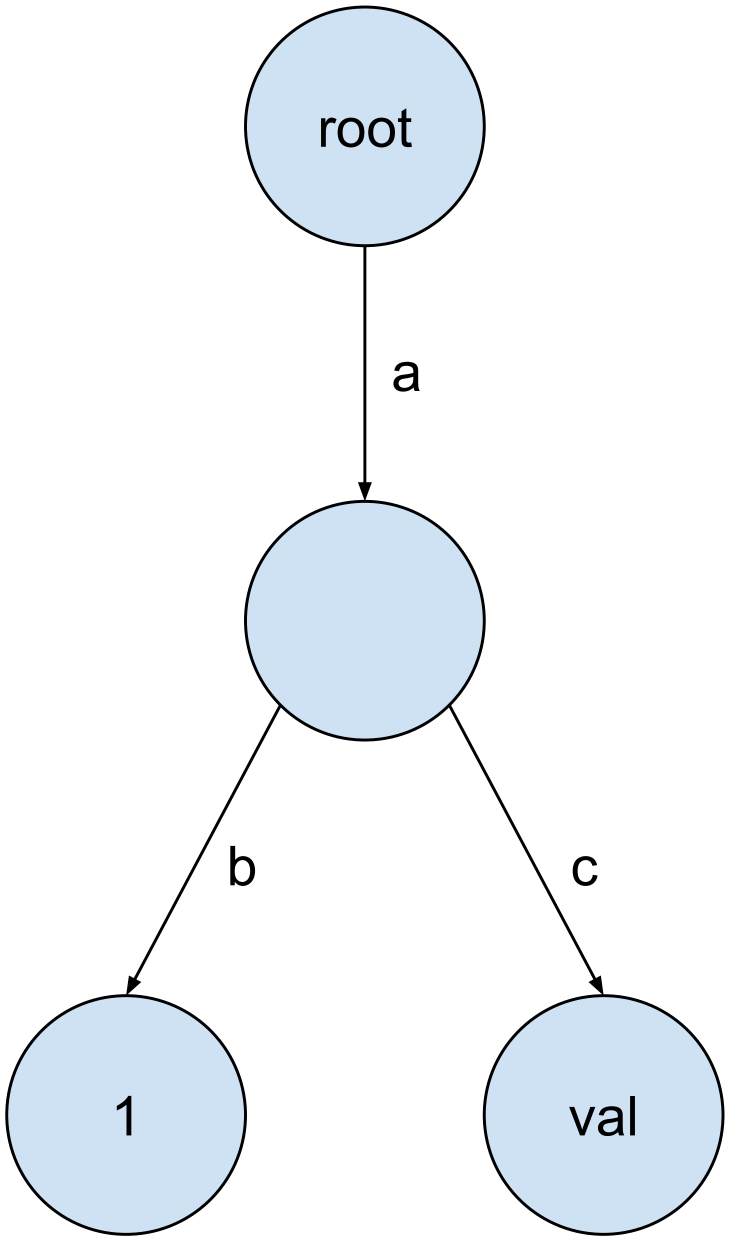
- **Release Date:** Aug 28, 2023
- **Due Date:** Sep 10, 2023 @ 11:59pm

Project Specification

In this project, you will implement a key-value store backed by a copy-on-write trie. Tries are efficient ordered-tree data structures for retrieving a value for a given key. To simplify the explanation, we will assume that the keys are variable-length strings, but in practice they can be any arbitrary type.

Each node in a trie can have multiple child nodes representing different possible next characters.

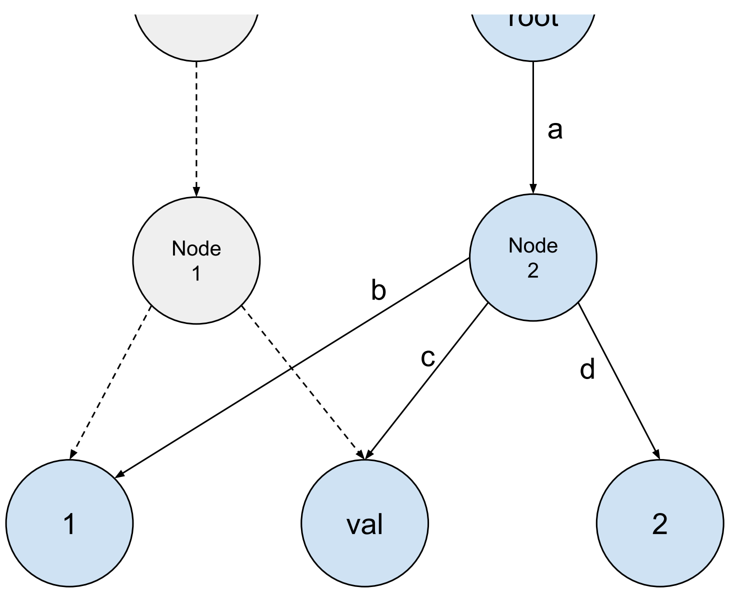
The key-value store you will implement can store string keys mapped to values of any type. The value of a key is stored in the node representing the last character of that



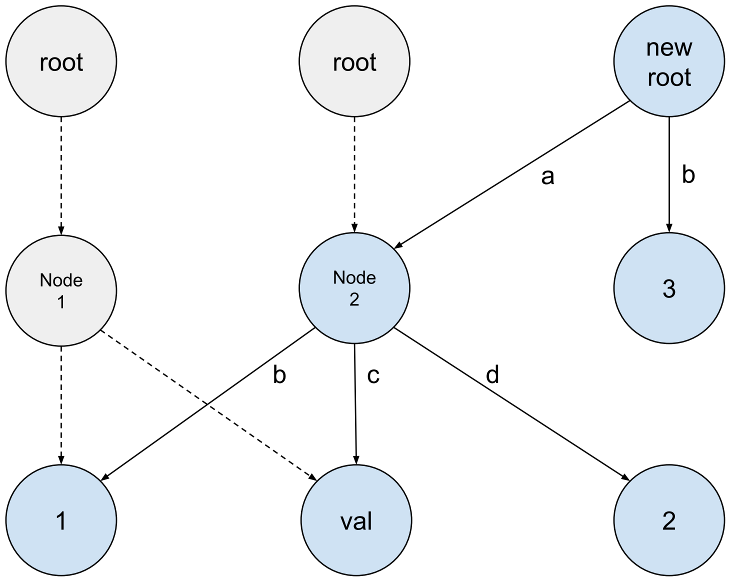
The two keys share the same parent node. The value 1 corresponding to key "ab" is stored in the left child, and the value "val" corresponding to key "ac" is stored in the right node.

Task #1 - Copy-On-Write Trie

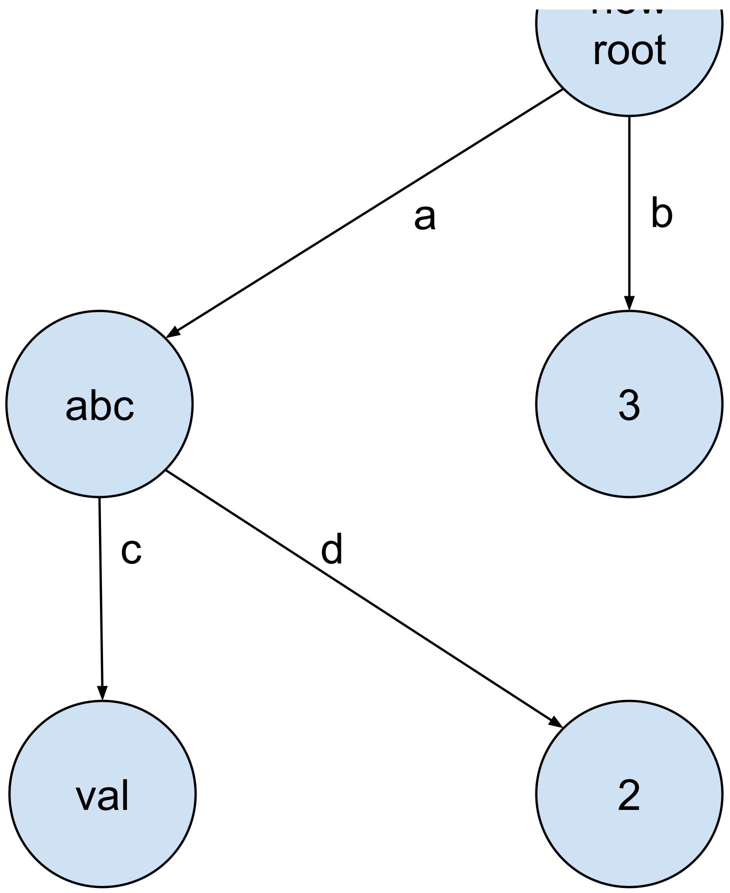
In this task, you will need to modify `trie.h` and `trie.cpp` to implement a copy-on-write trie. In a copy-on-write trie, operations do not directly modify the nodes of the original trie. Instead, new nodes are created for modified data, and a new root is returned for the newly-modified trie. Copy-on-write enables us to access the trie after each operation at any time with minimum overhead. Consider inserting `("ad", 2)` in the above example. We create a new `Node2` by reusing two of the child nodes from the original tree, and creating a new value node 2. (See figure below)



If we then insert `("b", 3)`, we will create a new root, a new node and reuse the previous nodes. In this way, we can get the content of the trie before and after each insertion operation. As long as we have the root object (`Trie` class), we can access the data inside the trie at that time. (See figure below)



One more example: if we then insert `("a", "abc")` and remove `("ab", 1)`, we can get the below trie. Note that parent nodes can have values, and **you will need to purge all unnecessary nodes after removal**. An empty trie is represented by `nullptr`.



Your trie must support three operations:

- `Get(key)`: Get the value corresponding to the key.
- `Put(key, value)`: Set the corresponding value to the key. Overwrite existing value if the key already exists. Note that the type of the value might be non-copyable (i.e., `std::unique_ptr<int>`). This method returns a new trie.
- `Delete(key)`: Delete the value for the key. This method returns a new trie.

None of these operations should be performed directly on the trie itself. You should create new trie nodes and reuse existing ones as much as possible.

To create a completely new node (i.e., a new leaf node without children), you can simply construct the object using the `TrieNodeWithValue` constructor and then make it into a smart pointer. To copy-on-write create a new node, you should use the `Clone` function on the `TrieNode` class. To reuse an existing node in the new trie, you can copy `std::shared_ptr<TrieNode>`: copying a shared pointer doesn't copy the underlying data. You should not manually allocate memory by using `new` and `delete` in this project. `std::shared_ptr` will deallocate the object when no one has a reference to the underlying object.

For the full specifications of these operations, please refer to the comments in the starter code. **Your implementation should store data as in the above examples.** Do not store the C-string terminator `\0` in your trie. Please also **avoid removing any `const` from the class definitions or use `mutable` / `const_cast` to bypass the const checks**.

Task #2 - Concurrent Key-Value Store

After you have a copy-on-write trie which can be used in a single-thread environment, implement a concurrent key-value store for a multithreaded environment. In this task, you will need to modify `trie_store.h` and `trie_store.cpp`. This key-value store also supports 3 operations:

- Delete(key): No return value.

For the original Trie class, everytime we modify the trie, we need to get the new root to access the new content. But for the concurrent key-value store, the `put` and `delete` methods do not have a return value. This requires you to use concurrency primitives to synchronize reads and writes so that no data is lost through the process.

Your concurrent key-value store should concurrently serve multiple readers *and* a single writer. That is to say, when someone is modifying the trie, reads can still be performed on the old root. When someone is reading, writes can still be performed without waiting for reads.

Also, if we get a reference to a value from the trie, we should be able to access it no matter how we modify the trie. The `Get` function from `Trie` only returns a pointer. If the trie node storing this value has been removed, the pointer will be dangling. Therefore, in `TrieStore`, we return a `ValueGuard` which stores both a reference to the value and the TrieNode corresponding to the root of the trie structure, so that the value can be accessed as we store the `ValueGuard`.

To achieve this, we have provided you with the pseudo code for `TrieStore::Get` in `trie_store.cpp`. Please read it carefully and think of how to implement `TrieStore::Put` and `TrieStore::Remove`.

Task #3 - Debugging

In this task, you will learn the basic techniques of debugging. You can choose any way you prefer for debugging, including but not limited to: using `cout` and `printf`, using CLion / VSCode debugger, using gdb, etc.

Please refer to `trie_debug_test.cpp` for instructions. You will need to set a breakpoint after the trie structure is generated and answer a few questions. You will need to fill in the answer in `trie_answer.h`.

Task #4 - SQL String Functions

Now it is time to dive into BusTub itself! You will need to implement `upper` and `lower` SQL functions. This can be done in 2 steps: (1) implement the function logic in `string_expression.h`. (2) register the function in BusTub, so that the SQL framework can call your function when the user executes a SQL, in `plan_func_call.cpp`.

To test your implementation, you can use `bustub-shell`:

```
cd build
make -j`nproc` shell
./bin/bustub-shell
bustub> select upper('AbCd'), lower('AbCd');
ABCD abcd
```

Your implementation should pass all 3 sqllogictest test cases.

```
cd build
make -j`nproc` sqllogictest
./bin/bustub-sqllogictest ../test/sql/p0.01-lower-upper.slt --verbose
./bin/bustub-sqllogictest ../test/sql/p0.02-function-error.slt --verbose
./bin/bustub-sqllogictest ../test/sql/p0.03-string-scan.slt --verbose
```

Note: If you see `BufferPoolManager is not implemented yet.` when running sqllogictest, this is normal and you can safely ignore this warning in project 0.

Instructions

If the below `git` concepts (e.g., repository, merge, pull, fork) do not make sense to you, please spend some time learning git first.

Follow the instructions to setup your own PRIVATE repository and your own development branch. If you have previuosly forked the repository through the GitHub UI (by clicking Fork), PLEASE DO NOT PUSH ANY CODE TO YOUR PUBLIC FORKED REPOSITORY! Make sure your repository is PRIVATE before you `git push` any of your code.

If the instructor makes any changes to the code, you can merge the changes to your code by keeping your private repository connected to the CMU-DB master repository. Execute the following commands to add a remote source:

```
$ git remote add public https://github.com/cmu-db/bustub.git
```

You can then pull down the latest changes as needed during the semester:

```
$ git fetch public
$ git merge public/master
```

Setting Up Your Development Environment

First install the packages that BusTub requires:

```
# Linux
$ sudo build_support/packages.sh
# macOS
$ build_support/packages.sh
```

See the README for additional information on how to setup different OS environments.

To build the system from the commandline, execute the following commands:

```
$ mkdir build
$ cd build
$ cmake -DCMAKE_BUILD_TYPE=Debug ..
$ make -j`nproc`
```

We recommend always configuring CMake in debug mode. This will enable you to output debug messages and check for memory leaks (more on this in below sections).

Testing

You can test the individual components of this assignment using our testing framework. We use GTest for unit test cases. You can compile and run each test individually from the command-line:

```
$ cd build
$ make trie_test trie_store_test -j$(nproc)
$ make trie_noncopy_test trie_store_noncopy_test -j$(nproc)
$ ./test/trie_test
$ ./test/trie_noncopy_test
$ ./test/trie_store_test
$ ./test/trie_store_noncopy_test
```

In this project, there are no hidden tests. In the future, the provided tests in the starter code are only a subset of the all the tests that we will use to evaluate and grade your project. You should write additional test cases on your own to check the complete functionality of your implementation.

Formatting

Your code must follow the Google C++ Style Guide. We use Clang to automatically check the quality of your source code. Your project grade will be **zero** if your submission fails any of these checks.

Execute the following commands to check your syntax. The `format` target will automatically correct your code. The `check-lint` and `check-clang-tidy` targets will print errors that you must manually fix to conform to our style guide.

```
$ make format
$ make check-clang-tidy-p0
```

Memory Leaks

For this project, we use LLVM Address Sanitizer (ASAN) and Leak Sanitizer (LSAN) to check for memory errors. To enable ASAN and LSAN, configure CMake in debug mode and run tests as you normally would. If there is memory error, you will see a memory error report. Note that macOS **only supports address sanitizer without leak sanitizer**.

In some cases, address sanitizer might affect the usability of the debugger. In this case, you might need to disable all sanitizers by configuring the CMake project with:

```
$ cmake -DCMAKE_BUILD_TYPE=Debug -DBUSTUB_SANITIZER= ..
```

Development Hints

You can use `BUSTUB_ASSERT` for assertions in debug mode. Note that the statements within `BUSTUB_ASSERT` will NOT be executed in release mode. If you have something to assert in all cases, use `BUSTUB_ENSURE` instead.

We will test your implementation in release mode. To compile your solution in release mode,

```
$ mkdir build_rel && cd build_rel
$ cmake -DCMAKE_BUILD_TYPE=Release ..
```

Post all of your questions about this project on Piazza. Do **not** email the TAs directly with questions.

TAs will **not** look into your code or help you debug in this project.

Grading Rubric

In order to pass this project, you must ensure your code follows the following guidelines:

1. Does the submission successfully execute all of the test cases and produce the correct answer?
2. Does the submission execute without any memory leaks?
3. Does the submission follow the code formatting and style policies?

Note that we will use additional test cases to grade your submission that are more complex than the sample test cases that we provide you in future projects.

Late Policy

There are no late days for this project.

You will submit your implementation to Gradescope:

- https://www.gradescope.com/courses/579715

Run this command in `build` directory and it will create a `zip` archive called `project0-submission.zip` that you can submit to Gradescope.

```
$ make submit-p0
```

Although you are allowed submit your answers as many times as you like, you should **not** treat Gradescope as your only debugging tool. Most students submit their projects near the deadline, and thus Gradescope will take longer to process the requests. You may not get feedback in a timely manner to help you debug problems. Furthermore, the output from Gradescope is unlikely to be as informative as the output from a debugger (like `gdb`), provided you invest some time in learning to use it.

CMU students should use the Gradescope course code announced on Piazza.

Collaboration Policy

- Every student must work individually on this assignment.
- Students are allowed to discuss high-level details about the project with others.
- Students are **not** allowed to copy the contents of a white-board after a group meeting with other students.
- Students are **not** allowed to copy solutions from another person.
- In this project, you are allowed to search on Google or ask ChatGPT high-level questions like "what is trie", "how to use `std::move`".

WARNING: All of the code for this project must be your own. You may not copy source code from other students or other sources that you find on the web. Plagiarism **will not** be tolerated. See CMU's Policy on Academic Integrity for additional information.