

Let's Build a Simple Database

Writing a sqlite clone from scratch in C

[Overview](#)

[View on GitHub](#) (pull requests welcome)

This project is no longer under active development. You can read more [here](#). But if you'd like to keep learning how to make your own SQLite clone from scratch, or one of many other projects like Docker, Redis, Git or BitTorrent, try [CodeCrafters](#).



[Try interactive mode →](#)

How Does a Database Work?

- What format is data saved in? (in memory and on disk)
- When does it move from memory to disk?
- Why can there only be one primary key per table?
- How does rolling back a transaction work?
- How are indexes formatted?
- When and how does a full table scan happen?
- What format is a prepared statement saved in?

In short, how does a database **work**?

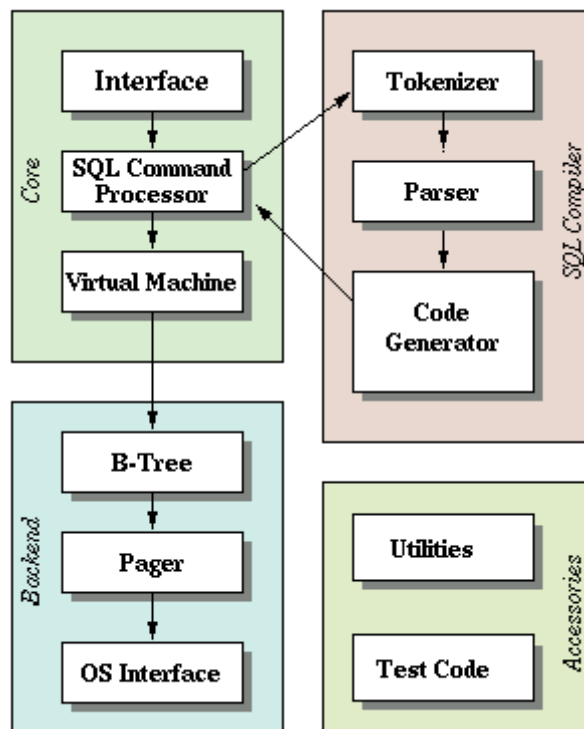
I'm building a clone of [sqlite](#) from scratch in C in order to understand, and I'm

going to document my process as I go.

Table of Contents

- [Part 1 - Introduction and Setting up the REPL](#)
- [Part 2 - World's Simplest SQL Compiler and Virtual Machine](#)
- [Part 3 - An In-Memory, Append-Only, Single-Table Database](#)
- [Part 4 - Our First Tests \(and Bugs\)](#)
- [Part 5 - Persistence to Disk](#)
- [Part 6 - The Cursor Abstraction](#)
- [Part 7 - Introduction to the B-Tree](#)
- [Part 8 - B-Tree Leaf Node Format](#)
- [Part 9 - Binary Search and Duplicate Keys](#)
- [Part 10 - Splitting a Leaf Node](#)
- [Part 11 - Recursively Searching the B-Tree](#)
- [Part 12 - Scanning a Multi-Level B-Tree](#)
- [Part 13 - Updating Parent Node After a Split](#)
- [Part 14 - Splitting Internal Nodes](#)
- [Part 15 - Where to go next](#)

"What I cannot create, I do not understand." – [Richard Feynman](#)



sqlite architecture (<https://www.sqlite.org/arch.html>)

[rss](#) | [subscribe by email](#)

This project is maintained by [cstack](#)

Hosted on GitHub Pages — Theme by [orderedlist](#)

Let's Build a Simple Database

Writing a sqlite clone from scratch in C

[Overview](#)

[View on GitHub](#) (pull requests welcome)

This project is no longer under active development. You can read more [here](#). But if you'd like to keep learning how to make your own SQLite clone from scratch, or one of many other projects like Docker, Redis, Git or BitTorrent, try [CodeCrafters](#).



Build Your Own SQLite

Try interactive mode →

Part 1 - Introduction and Setting up the REPL

[Part 2 - World's Simplest SQL Compiler and Virtual Machine >](#)

As a web developer, I use relational databases every day at my job, but they're a black box to me. Some questions I have:

- What format is data saved in? (in memory and on disk)
- When does it move from memory to disk?
- Why can there only be one primary key per table?
- How does rolling back a transaction work?
- How are indexes formatted?

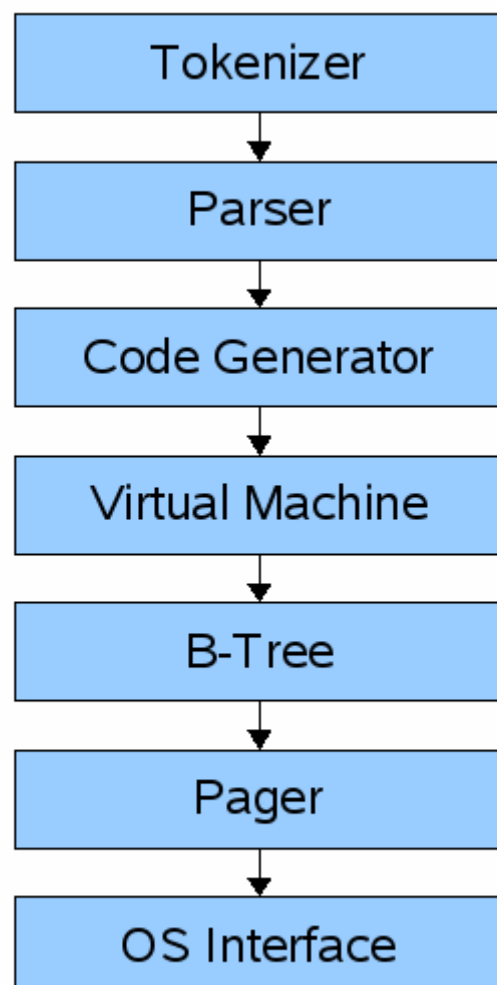
- When and how does a full table scan happen?
- What format is a prepared statement saved in?

In other words, how does a database **work**?

To figure things out, I'm writing a database from scratch. It's modeled off sqlite because it is designed to be small with fewer features than MySQL or PostgreSQL, so I have a better hope of understanding it. The entire database is stored in a single file!

Sqlite

There's lots of [documentation of sqlite internals](#) on their website, plus I've got a copy of [SQLite Database System: Design and Implementation](#).



A query goes through a chain of components in order to retrieve or modify data. The **front-end** consists of the:

- tokenizer
- parser
- code generator

The input to the front-end is a SQL query. the output is sqlite virtual machine bytecode (essentially a compiled program that can operate on the database).

The *back-end* consists of the:

- virtual machine
- B-tree
- pager
- os interface

The **virtual machine** takes bytecode generated by the front-end as instructions. It can then perform operations on one or more tables or indexes, each of which is stored in a data structure called a B-tree. The VM is essentially a big switch statement on the type of bytecode instruction.

Each **B-tree** consists of many nodes. Each node is one page in length. The B-tree can retrieve a page from disk or save it back to disk by issuing commands to the pager.

The **pager** receives commands to read or write pages of data. It is responsible for reading/writing at appropriate offsets in the database file. It also keeps a cache of recently-accessed pages in memory, and determines when those pages need to be written back to disk.

The **os interface** is the layer that differs depending on which operating system sqlite was compiled for. In this tutorial, I'm not going to support multiple platforms.

[A journey of a thousand miles begins with a single step](#), so let's start with something a little more straightforward: the REPL.

Making a Simple REPL

Sqlite starts a read-execute-print loop when you start it from the command line:

```
~ sqlite3
SQLite version 3.16.0 2016-11-04 19:09:39
```

```

Enter ".help" for usage hints.
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> create table users (id int, username varchar(255), email varchar(255));
sqlite> .tables
users
sqlite> .exit
~

```

To do that, our main function will have an infinite loop that prints the prompt, gets a line of input, then processes that line of input:

```

int main(int argc, char* argv[]) {
    InputBuffer* input_buffer = new_input_buffer();
    while (true) {
        print_prompt();
        read_input(input_buffer);

        if (strcmp(input_buffer->buffer, ".exit") == 0) {
            close_input_buffer(input_buffer);
            exit(EXIT_SUCCESS);
        } else {
            printf("Unrecognized command '%s'.\n", input_buffer->buffer);
        }
    }
}

```

We'll define InputBuffer as a small wrapper around the state we need to store to interact with `getline()`. (More on that in a minute)

```

typedef struct {
    char* buffer;
    size_t buffer_length;
    ssize_t input_length;
} InputBuffer;

InputBuffer* new_input_buffer() {
    InputBuffer* input_buffer = (InputBuffer*)malloc(sizeof(InputBuffer));
    input_buffer->buffer = NULL;
    input_buffer->buffer_length = 0;
    input_buffer->input_length = 0;
}

```

```
    return input_buffer;
}
```

Next, `print_prompt()` prints a prompt to the user. We do this before reading each line of input.

```
void print_prompt() { printf("db > "); }
```

To read a line of input, use `getline()`:

```
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

`lineptr` : a pointer to the variable we use to point to the buffer containing the read line. If it set to NULL it is mallocated by `getline` and should thus be freed by the user, even if the command fails.

`n` : a pointer to the variable we use to save the size of allocated buffer.

`stream` : the input stream to read from. We'll be reading from standard input.

`return value` : the number of bytes read, which may be less than the size of the buffer.

We tell `getline` to store the read line in `input_buffer->buffer` and the size of the allocated buffer in `input_buffer->buffer_length`. We store the return value in `input_buffer->input_length`.

`buffer` starts as null, so `getline` allocates enough memory to hold the line of input and makes `buffer` point to it.

```
void read_input(InputBuffer* input_buffer) {
    ssize_t bytes_read =
        getline(&(input_buffer->buffer), &(input_buffer->buffer_length),

    if (bytes_read <= 0) {
        printf("Error reading input\n");
        exit(EXIT_FAILURE);
    }

    // Ignore trailing newline
    input_buffer->input_length = bytes_read - 1;
```

```
input_buffer->buffer[bytes_read - 1] = 0;
}
```

Now it is proper to define a function that frees the memory allocated for an instance of `InputBuffer *` and the buffer element of the respective structure (getline allocates memory for `input_buffer->buffer` in `read_input`).

```
void close_input_buffer(InputBuffer* input_buffer) {
    free(input_buffer->buffer);
    free(input_buffer);
}
```

Finally, we parse and execute the command. There is only one recognized command right now : `.exit`, which terminates the program. Otherwise we print an error message and continue the loop.

```
if (strcmp(input_buffer->buffer, ".exit") == 0) {
    close_input_buffer(input_buffer);
    exit(EXIT_SUCCESS);
} else {
    printf("Unrecognized command '%s'.\n", input_buffer->buffer);
}
```

Let's try it out!

```
~ ./db
db > .tables
Unrecognized command '.tables'.
db > .exit
~
```

Alright, we've got a working REPL. In the next part, we'll start developing our command language. Meanwhile, here's the entire program from this part:

```
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    char* buffer;
```

```

    size_t buffer_length;
    ssize_t input_length;
} InputBuffer;

InputBuffer* new_input_buffer() {
    InputBuffer* input_buffer = malloc(sizeof(InputBuffer));
    input_buffer->buffer = NULL;
    input_buffer->buffer_length = 0;
    input_buffer->input_length = 0;

    return input_buffer;
}

void print_prompt() { printf("db > "); }

void read_input(InputBuffer* input_buffer) {
    ssize_t bytes_read =
        getline(&(input_buffer->buffer), &(input_buffer->buffer_

    if (bytes_read <= 0) {
        printf("Error reading input\n");
        exit(EXIT_FAILURE);
    }

    // Ignore trailing newline
    input_buffer->input_length = bytes_read - 1;
    input_buffer->buffer[bytes_read - 1] = 0;
}

void close_input_buffer(InputBuffer* input_buffer) {
    free(input_buffer->buffer);
    free(input_buffer);
}

int main(int argc, char* argv[]) {
    InputBuffer* input_buffer = new_input_buffer();
    while (true) {
        print_prompt();
        read_input(input_buffer);

        if (strcmp(input_buffer->buffer, ".exit") == 0) {
            close_input_buffer(input_buffer);

```

```
        exit(EXIT_SUCCESS);  
    } else {  
        printf("Unrecognized command '%s'.\n", input_buffer->buffer);  
    }  
}  
}
```

[Part 2 - World's Simplest SQL Compiler and Virtual Machine >](#)

[rss](#) | [subscribe by email](#)

This project is maintained by [cstack](#)

Hosted on GitHub Pages — Theme by [orderedlist](#)

Let's Build a Simple Database

Writing a sqlite clone from scratch in C

[Overview](#)

[View on GitHub \(pull requests welcome\)](#)

This project is no longer under active development. You can read more [here](#). But if you'd like to keep learning how to make your own SQLite clone from scratch, or one of many other projects like Docker, Redis, Git or BitTorrent, try [CodeCrafters](#).



Build Your Own SQLite

Try interactive mode →

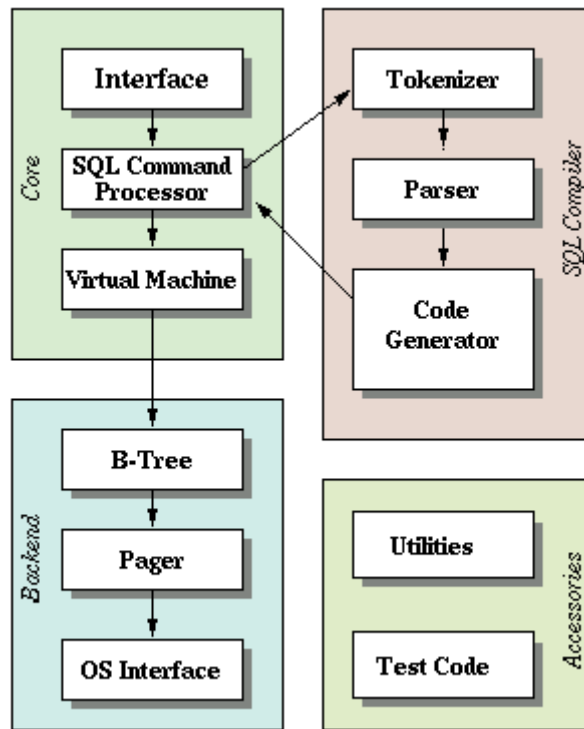
Part 2 - World's Simplest SQL Compiler and Virtual Machine

[< Part 1 - Introduction and Setting up the REPL](#)

[Part 3 - An In-Memory, Append-Only, Single-Table Database >](#)

We're making a clone of sqlite. The "front-end" of sqlite is a SQL compiler that parses a string and outputs an internal representation called bytecode.

This bytecode is passed to the virtual machine, which executes it.



SQLite Architecture (<https://www.sqlite.org/arch.html>)

Breaking things into two steps like this has a couple advantages:

- Reduces the complexity of each part (e.g. virtual machine does not worry about syntax errors)
- Allows compiling common queries once and caching the bytecode for improved performance

With this in mind, let's refactor our main function and support two new keywords in the process:

```

int main(int argc, char* argv[]) {
    InputBuffer* input_buffer = new_input_buffer();
    while (true) {
        print_prompt();
        read_input(input_buffer);

-       if (strcmp(input_buffer->buffer, ".exit") == 0) {
-           exit(EXIT_SUCCESS);
-       } else {
-           printf("Unrecognized command '%s'.\n", input_buffer->b
+       if (input_buffer->buffer[0] == '.') {
+           switch (do_meta_command(input_buffer)) {
+               case (META_COMMAND_SUCCESS):
+                   continue;
  
```

```

+         case (META_COMMAND_UNRECOGNIZED_COMMAND):
+             printf("Unrecognized command '%s'\n", input_buffer);
+             continue;
+     }
+ }
+
+ Statement statement;
+ switch (prepare_statement(input_buffer, &statement)) {
+     case (PREPARE_SUCCESS):
+         break;
+     case (PREPARE_UNRECOGNIZED_STATEMENT):
+         printf("Unrecognized keyword at start of '%s'.\n",
+             input_buffer->buffer);
+         continue;
+ }
+
+ execute_statement(&statement);
+ printf("Executed.\n");
+ }
+ }

```

Non-SQL statements like `.exit` are called “meta-commands”. They all start with a dot, so we check for them and handle them in a separate function.

Next, we add a step that converts the line of input into our internal representation of a statement. This is our hacky version of the sqlite front-end.

Lastly, we pass the prepared statement to `execute_statement`. This function will eventually become our virtual machine.

Notice that two of our new functions return enums indicating success or failure:

```

typedef enum {
    META_COMMAND_SUCCESS,
    META_COMMAND_UNRECOGNIZED_COMMAND
} MetaCommandResult;

typedef enum { PREPARE_SUCCESS, PREPARE_UNRECOGNIZED_STATEMENT

```

“Unrecognized statement”? That seems a bit like an exception. I prefer not to use exceptions (and C doesn’t even support them), so I’m using enum result codes wherever practical. The C compiler will complain if my switch statement

doesn't handle a member of the enum, so we can feel a little more confident we handle every result of a function. Expect more result codes to be added in the future.

do_meta_command is just a wrapper for existing functionality that leaves room for more commands:

```
MetaCommandResult do_meta_command(InputBuffer* input_buffer)
{
    if (strcmp(input_buffer->buffer, ".exit") == 0) {
        exit(EXIT_SUCCESS);
    } else {
        return META_COMMAND_UNRECOGNIZED_COMMAND;
    }
}
```

Our "prepared statement" right now just contains an enum with two possible values. It will contain more data as we allow parameters in statements:

```
typedef enum { STATEMENT_INSERT, STATEMENT_SELECT } StatementType;

typedef struct {
    StatementType type;
} Statement;
```

prepare_statement (our "SQL Compiler") does not understand SQL right now. In fact, it only understands two words:

```
PrepareResult prepare_statement(InputBuffer* input_buffer,
                                Statement* statement) {
    if (strncmp(input_buffer->buffer, "insert", 6) == 0) {
        statement->type = STATEMENT_INSERT;
        return PREPARE_SUCCESS;
    }
    if (strcmp(input_buffer->buffer, "select") == 0) {
        statement->type = STATEMENT_SELECT;
        return PREPARE_SUCCESS;
    }

    return PREPARE_UNRECOGNIZED_STATEMENT;
}
```

Note that we use `strncmp` for “insert” since the “insert” keyword will be followed by data. (e.g. `insert 1 cstack foo@bar.com`)

Lastly, `execute_statement` contains a few stubs:

```
void execute_statement(Statement* statement) {
    switch (statement->type) {
        case (STATEMENT_INSERT):
            printf("This is where we would do an insert.\n");
            break;
        case (STATEMENT_SELECT):
            printf("This is where we would do a select.\n");
            break;
    }
}
```

Note that it doesn’t return any error codes because there’s nothing that could go wrong yet.

With these refactors, we now recognize two new keywords!

```
~ ./db
db > insert foo bar
This is where we would do an insert.
Executed.
db > delete foo
Unrecognized keyword at start of 'delete foo'.
db > select
This is where we would do a select.
Executed.
db > .tables
Unrecognized command '.tables'
db > .exit
~
```

The skeleton of our database is taking shape... wouldn’t it be nice if it stored data? In the next part, we’ll implement `insert` and `select`, creating the world’s worst data store. In the mean time, here’s the entire diff from this part:

```
@@ -10,6 +10,23 @@ struct InputBuffer_t {
    } InputBuffer;
```

```

+typedef enum {
+ META_COMMAND_SUCCESS,
+ META_COMMAND_UNRECOGNIZED_COMMAND
+} MetaCommandResult;
+
+typedef enum { PREPARE_SUCCESS, PREPARE_UNRECOGNIZED_STATEMENT } PrepareResult;
+
+typedef enum { STATEMENT_INSERT, STATEMENT_SELECT } StatementType;
+
+typedef struct {
+ StatementType type;
+} Statement;
+
+
+InputBuffer* new_input_buffer() {
+ InputBuffer* input_buffer = malloc(sizeof(InputBuffer));
+ input_buffer->buffer = NULL;
@@ -40,17 +57,67 @@ void close_input_buffer(InputBuffer* input_buffer) {
+ free(input_buffer);
+ }
+
+
+MetaCommandResult do_meta_command(InputBuffer* input_buffer)
+ {
+ if (strcmp(input_buffer->buffer, ".exit") == 0) {
+ close_input_buffer(input_buffer);
+ exit(EXIT_SUCCESS);
+ } else {
+ return META_COMMAND_UNRECOGNIZED_COMMAND;
+ }
+ }
+
+
+PrepareResult prepare_statement(InputBuffer* input_buffer,
+ Statement* statement) {
+ if (strncmp(input_buffer->buffer, "insert", 6) == 0) {
+ statement->type = STATEMENT_INSERT;
+ return PREPARE_SUCCESS;
+ }
+ if (strcmp(input_buffer->buffer, "select") == 0) {
+ statement->type = STATEMENT_SELECT;
+ return PREPARE_SUCCESS;
+ }
+
+ return PREPARE_UNRECOGNIZED_STATEMENT;
+ }

```

```

+
+void execute_statement(Statement* statement) {
+  switch (statement->type) {
+    case (STATEMENT_INSERT):
+      printf("This is where we would do an insert.\n");
+      break;
+    case (STATEMENT_SELECT):
+      printf("This is where we would do a select.\n");
+      break;
+  }
+}
+
+int main(int argc, char* argv[]) {
+  InputBuffer* input_buffer = new_input_buffer();
+  while (true) {
+    print_prompt();
+    read_input(input_buffer);
+
-    if (strcmp(input_buffer->buffer, ".exit") == 0) {
-      close_input_buffer(input_buffer);
-      exit(EXIT_SUCCESS);
-    } else {
-      printf("Unrecognized command '%s'.\n", input_buffer->b
+    if (input_buffer->buffer[0] == '.') {
+      switch (do_meta_command(input_buffer)) {
+        case (META_COMMAND_SUCCESS):
+          continue;
+        case (META_COMMAND_UNRECOGNIZED_COMMAND):
+          printf("Unrecognized command '%s'\n", input_buffer
+          continue;
+      }
+    }
+
+    Statement statement;
+    switch (prepare_statement(input_buffer, &statement)) {
+      case (PREPARE_SUCCESS):
+        break;
+      case (PREPARE_UNRECOGNIZED_STATEMENT):
+        printf("Unrecognized keyword at start of '%s'.\n",
+              input_buffer->buffer);
+        continue;
+    }

```

```
+  
+     execute_statement(&statement);  
+     printf("Executed.\n");  
+ }  
}
```

[< Part 1 - Introduction and Setting up the REPL](#)

[Part 3 - An In-Memory, Append-Only, Single-Table Database >](#)

[rss](#) | [subscribe by email](#)

This project is maintained by [cstack](#)

Hosted on GitHub Pages — Theme by [orderedlist](#)

Let's Build a Simple Database

Writing a sqlite clone from scratch in C

[Overview](#)

[View on GitHub](#) (pull requests welcome)

This project is no longer under active development. You can read more [here](#). But if you'd like to keep learning how to make your own SQLite clone from scratch, or one of many other projects like Docker, Redis, Git or BitTorrent, try [CodeCrafters](#).



Try interactive mode →

Part 3 - An In-Memory, Append-Only, Single-Table Database

[< Part 2 - World's Simplest SQL Compiler and Virtual Machine](#)

[Part 4 - Our First Tests \(and Bugs\) >](#)

We're going to start small by putting a lot of limitations on our database. For now, it will:

- support two operations: inserting a row and printing all rows
- reside only in memory (no persistence to disk)
- support a single, hard-coded table

Our hard-coded table is going to store users and look like this:

column	type
id	integer
username	varchar(32)
email	varchar(255)

This is a simple schema, but it gets us to support multiple data types and multiple sizes of text data types.

insert statements are now going to look like this:

```
insert 1 cstack foo@bar.com
```

That means we need to upgrade our `prepare_statement` function to parse arguments

```
    if (strncmp(input_buffer->buffer, "insert", 6) == 0) {
        statement->type = STATEMENT_INSERT;
+       int args_assigned = sscanf(
+           input_buffer->buffer, "insert %d %s %s", &(statement
+           statement->row_to_insert.username, statement->row_to
+       if (args_assigned < 3) {
+           return PREPARE_SYNTAX_ERROR;
+       }
        return PREPARE_SUCCESS;
    }
    if (strcmp(input_buffer->buffer, "select") == 0) {
```

We store those parsed arguments into a new Row data structure inside the statement object:

```
+#define COLUMN_USERNAME_SIZE 32
+#define COLUMN_EMAIL_SIZE 255
+typedef struct {
+    uint32_t id;
+    char username[COLUMN_USERNAME_SIZE];
+    char email[COLUMN_EMAIL_SIZE];
+} Row;
+
```

```
typedef struct {
    StatementType type;
+   Row row_to_insert;  // only used by insert statement
} Statement;
```

Now we need to copy that data into some data structure representing the table. SQLite uses a B-tree for fast lookups, inserts and deletes. We'll start with something simpler. Like a B-tree, it will group rows into pages, but instead of arranging those pages as a tree it will arrange them as an array.

Here's my plan:

- Store rows in blocks of memory called pages
- Each page stores as many rows as it can fit
- Rows are serialized into a compact representation with each page
- Pages are only allocated as needed
- Keep a fixed-size array of pointers to pages

First we'll define the compact representation of a row:

```
#define size_of_attribute(Struct, Attribute) sizeof(((Struct*)0->Attribute))
+
+const uint32_t ID_SIZE = size_of_attribute(Row, id);
+const uint32_t USERNAME_SIZE = size_of_attribute(Row, username);
+const uint32_t EMAIL_SIZE = size_of_attribute(Row, email);
+const uint32_t ID_OFFSET = 0;
+const uint32_t USERNAME_OFFSET = ID_OFFSET + ID_SIZE;
+const uint32_t EMAIL_OFFSET = USERNAME_OFFSET + USERNAME_SIZE;
+const uint32_t ROW_SIZE = ID_SIZE + USERNAME_SIZE + EMAIL_SIZE;
```

This means the layout of a serialized row will look like this:

column	size (bytes)	offset
id	4	0
username	32	4
email	255	36
total	291	

We also need code to convert to and from the compact representation.

```
+void serialize_row(Row* source, void* destination) {
+  memcpy(destination + ID_OFFSET, &(source->id), ID_SIZE);
+  memcpy(destination + USERNAME_OFFSET, &(source->username),
+  memcpy(destination + EMAIL_OFFSET, &(source->email), EMAIL_
+}
+
+void deserialize_row(void* source, Row* destination) {
+  memcpy(&(destination->id), source + ID_OFFSET, ID_SIZE);
+  memcpy(&(destination->username), source + USERNAME_OFFSET,
+  memcpy(&(destination->email), source + EMAIL_OFFSET, EMAIL_
+}
```

Next, a Table structure that points to pages of rows and keeps track of how many rows there are:

```
+const uint32_t PAGE_SIZE = 4096;
+#define TABLE_MAX_PAGES 100
+const uint32_t ROWS_PER_PAGE = PAGE_SIZE / ROW_SIZE;
+const uint32_t TABLE_MAX_ROWS = ROWS_PER_PAGE * TABLE_MAX_PA
+
+typedef struct {
+  uint32_t num_rows;
+  void* pages[TABLE_MAX_PAGES];
+} Table;
```

I'm making our page size 4 kilobytes because it's the same size as a page used in the virtual memory systems of most computer architectures. This means one page in our database corresponds to one page used by the operating system. The operating system will move pages in and out of memory as whole units instead of breaking them up.

I'm setting an arbitrary limit of 100 pages that we will allocate. When we switch to a tree structure, our database's maximum size will only be limited by the maximum size of a file. (Although we'll still limit how many pages we keep in memory at once)

Rows should not cross page boundaries. Since pages probably won't exist next to each other in memory, this assumption makes it easier to read/write rows.

Speaking of which, here is how we figure out where to read/write in memory for a particular row:

```

+void* row_slot(Table* table, uint32_t row_num) {
+  uint32_t page_num = row_num / ROWS_PER_PAGE;
+  void* page = table->pages[page_num];
+  if (page == NULL) {
+    // Allocate memory only when we try to access page
+    page = table->pages[page_num] = malloc(PAGE_SIZE);
+  }
+  uint32_t row_offset = row_num % ROWS_PER_PAGE;
+  uint32_t byte_offset = row_offset * ROW_SIZE;
+  return page + byte_offset;
+}

```

Now we can make execute_statement read/write from our table structure:

```

-void execute_statement(Statement* statement) {
+ExecuteResult execute_insert(Statement* statement, Table* table) {
+  if (table->num_rows >= TABLE_MAX_ROWS) {
+    return EXECUTE_TABLE_FULL;
+  }
+
+  Row* row_to_insert = &(statement->row_to_insert);
+
+  serialize_row(row_to_insert, row_slot(table, table->num_rows));
+  table->num_rows += 1;
+
+  return EXECUTE_SUCCESS;
+}
+
+ExecuteResult execute_select(Statement* statement, Table* table) {
+  Row row;
+  for (uint32_t i = 0; i < table->num_rows; i++) {
+    deserialize_row(row_slot(table, i), &row);
+    print_row(&row);
+  }
+  return EXECUTE_SUCCESS;
+}
+
+ExecuteResult execute_statement(Statement* statement, Table* table) {
+  switch (statement->type) {
+    case (STATEMENT_INSERT):
+      printf("This is where we would do an insert.\n");
+      break;

```

```

+     return execute_insert(statement, table);
+     case (STATEMENT_SELECT):
-     printf("This is where we would do a select.\n");
-     break;
+     return execute_select(statement, table);
+ }
+ }

```

Lastly, we need to initialize the table, create the respective memory release function and handle a few more error cases:

```

+ Table* new_table() {
+     Table* table = (Table*)malloc(sizeof(Table));
+     table->num_rows = 0;
+     for (uint32_t i = 0; i < TABLE_MAX_PAGES; i++) {
+         table->pages[i] = NULL;
+     }
+     return table;
+ }
+
+ void free_table(Table* table) {
+     for (int i = 0; table->pages[i]; i++) {
+         free(table->pages[i]);
+     }
+     free(table);
+ }

```

```

int main(int argc, char* argv[]) {
+ Table* table = new_table();
+ InputBuffer* input_buffer = new_input_buffer();
+ while (true) {
+     print_prompt();
@@ -105,13 +203,22 @@ int main(int argc, char* argv[]) {
+     switch (prepare_statement(input_buffer, &statement)) {
+         case (PREPARE_SUCCESS):
+             break;
+         case (PREPARE_SYNTAX_ERROR):
+             printf("Syntax error. Could not parse statement.\n");
+             continue;
+         case (PREPARE_UNRECOGNIZED_STATEMENT):
+             printf("Unrecognized keyword at start of '%s'.\n",

```

```

        input_buffer->buffer);
    continue;
}

-   execute_statement(&statement);
-   printf("Executed.\n");
+   switch (execute_statement(&statement, table)) {
+       case (EXECUTE_SUCCESS):
+           printf("Executed.\n");
+           break;
+       case (EXECUTE_TABLE_FULL):
+           printf("Error: Table full.\n");
+           break;
+   }
}
}

```

With those changes we can actually save data in our database!

```

~ ./db
db > insert 1 cstack foo@bar.com
Executed.
db > insert 2 bob bob@example.com
Executed.
db > select
(1, cstack, foo@bar.com)
(2, bob, bob@example.com)
Executed.
db > insert foo bar 1
Syntax error. Could not parse statement.
db > .exit
~

```

Now would be a great time to write some tests, for a couple reasons:

- We're planning to dramatically change the data structure storing our table, and tests would catch regressions.
- There are a couple edge cases we haven't tested manually (e.g. filling up the table)

We'll address those issues in the next part. For now, here's the complete diff from this part:

```

@@ -2,6 +2,7 @@
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
+#include <stdint.h>

typedef struct {
    char* buffer;
@@ -10,6 +11,105 @@ typedef struct {
    } InputBuffer;

+typedef enum { EXECUTE_SUCCESS, EXECUTE_TABLE_FULL } ExecuteResult;
+
+typedef enum {
+    META_COMMAND_SUCCESS,
+    META_COMMAND_UNRECOGNIZED_COMMAND
+} MetaCommandResult;
+
+typedef enum {
+    PREPARE_SUCCESS,
+    PREPARE_SYNTAX_ERROR,
+    PREPARE_UNRECOGNIZED_STATEMENT
+} PrepareResult;
+
+typedef enum { STATEMENT_INSERT, STATEMENT_SELECT } StatementType;
+
+#define COLUMN_USERNAME_SIZE 32
+#define COLUMN_EMAIL_SIZE 255
+typedef struct {
+    uint32_t id;
+    char username[COLUMN_USERNAME_SIZE];
+    char email[COLUMN_EMAIL_SIZE];
+} Row;
+
+typedef struct {
+    StatementType type;
+    Row row_to_insert; //only used by insert statement
+} Statement;
+
+#define size_of_attribute(Struct, Attribute) sizeof(((Struct*)0)->Attribute)
+
+const uint32_t ID_SIZE = size_of_attribute(Row, id);

```

```

+const uint32_t USERNAME_SIZE = size_of_attribute(Row, username);
+const uint32_t EMAIL_SIZE = size_of_attribute(Row, email);
+const uint32_t ID_OFFSET = 0;
+const uint32_t USERNAME_OFFSET = ID_OFFSET + ID_SIZE;
+const uint32_t EMAIL_OFFSET = USERNAME_OFFSET + USERNAME_SIZE;
+const uint32_t ROW_SIZE = ID_SIZE + USERNAME_SIZE + EMAIL_SIZE;
+
+const uint32_t PAGE_SIZE = 4096;
+#define TABLE_MAX_PAGES 100
+const uint32_t ROWS_PER_PAGE = PAGE_SIZE / ROW_SIZE;
+const uint32_t TABLE_MAX_ROWS = ROWS_PER_PAGE * TABLE_MAX_PAGES;
+
+typedef struct {
+    uint32_t num_rows;
+    void* pages[TABLE_MAX_PAGES];
+} Table;
+
+void print_row(Row* row) {
+    printf("(%d, %s, %s)\n", row->id, row->username, row->email);
+}
+
+void serialize_row(Row* source, void* destination) {
+    memcpy(destination + ID_OFFSET, &(source->id), ID_SIZE);
+    memcpy(destination + USERNAME_OFFSET, &(source->username), USERNAME_SIZE);
+    memcpy(destination + EMAIL_OFFSET, &(source->email), EMAIL_SIZE);
+}
+
+void deserialize_row(void *source, Row* destination) {
+    memcpy(&(destination->id), source + ID_OFFSET, ID_SIZE);
+    memcpy(&(destination->username), source + USERNAME_OFFSET, USERNAME_SIZE);
+    memcpy(&(destination->email), source + EMAIL_OFFSET, EMAIL_SIZE);
+}
+
+void* row_slot(Table* table, uint32_t row_num) {
+    uint32_t page_num = row_num / ROWS_PER_PAGE;
+    void *page = table->pages[page_num];
+    if (page == NULL) {
+        // Allocate memory only when we try to access page
+        page = table->pages[page_num] = malloc(PAGE_SIZE);
+    }
+    uint32_t row_offset = row_num % ROWS_PER_PAGE;
+    uint32_t byte_offset = row_offset * ROW_SIZE;

```

```

+   return page + byte_offset;
+}
+
+Table* new_table() {
+   Table* table = (Table*)malloc(sizeof(Table));
+   table->num_rows = 0;
+   for (uint32_t i = 0; i < TABLE_MAX_PAGES; i++) {
+       table->pages[i] = NULL;
+   }
+   return table;
+}
+
+void free_table(Table* table) {
+   for (int i = 0; table->pages[i]; i++) {
+       free(table->pages[i]);
+   }
+   free(table);
+}
+
+   InputBuffer* new_input_buffer() {
+       InputBuffer* input_buffer = (InputBuffer*)malloc(sizeof(InputBuffer));
+       input_buffer->buffer = NULL;
@@ -40,17 +140,105 @@ void close_input_buffer(InputBuffer* input_buffer) {
+       free(input_buffer);
+   }
+
+MetaCommandResult do_meta_command(InputBuffer* input_buffer,
+   if (strcmp(input_buffer->buffer, ".exit") == 0) {
+       close_input_buffer(input_buffer);
+       free_table(table);
+       exit(EXIT_SUCCESS);
+   } else {
+       return META_COMMAND_UNRECOGNIZED_COMMAND;
+   }
+}
+
+PrepareResult prepare_statement(InputBuffer* input_buffer,
+                               Statement* statement) {
+   if (strncmp(input_buffer->buffer, "insert", 6) == 0) {
+       statement->type = STATEMENT_INSERT;
+       int args_assigned = sscanf(
+           input_buffer->buffer, "insert %d %s %s", &(statement->

```

```

+     statement->row_to_insert.username, statement->row_to_
+ );
+ if (args_assigned < 3) {
+     return PREPARE_SYNTAX_ERROR;
+ }
+ return PREPARE_SUCCESS;
+ }
+ if (strcmp(input_buffer->buffer, "select") == 0) {
+     statement->type = STATEMENT_SELECT;
+     return PREPARE_SUCCESS;
+ }
+
+ return PREPARE_UNRECOGNIZED_STATEMENT;
+}
+
+ExecuteResult execute_insert(Statement* statement, Table* ta
+ if (table->num_rows >= TABLE_MAX_ROWS) {
+     return EXECUTE_TABLE_FULL;
+ }
+
+ Row* row_to_insert = &(amp;statement->row_to_insert);
+
+ serialize_row(row_to_insert, row_slot(table, table->num_row
+ table->num_rows += 1;
+
+ return EXECUTE_SUCCESS;
+}
+
+ExecuteResult execute_select(Statement* statement, Table* ta
+ Row row;
+ for (uint32_t i = 0; i < table->num_rows; i++) {
+     deserialize_row(row_slot(table, i), &row);
+     print_row(&row);
+ }
+ return EXECUTE_SUCCESS;
+}
+
+ExecuteResult execute_statement(Statement* statement, Table
+ switch (statement->type) {
+     case (STATEMENT_INSERT):
+         return execute_insert(statement, table);
+     case (STATEMENT_SELECT):

```

```

+         return execute_select(statement, table);
+     }
+ }
+
+ int main(int argc, char* argv[]) {
+     Table* table = new_table();
+     InputBuffer* input_buffer = new_input_buffer();
+     while (true) {
+         print_prompt();
+         read_input(input_buffer);
+
+         if (strcmp(input_buffer->buffer, ".exit") == 0) {
+             close_input_buffer(input_buffer);
+             exit(EXIT_SUCCESS);
+         } else {
+             printf("Unrecognized command '%s'.\n", input_buffer->b
+ if (input_buffer->buffer[0] == '.') {
+             switch (do_meta_command(input_buffer, table)) {
+                 case (META_COMMAND_SUCCESS):
+                     continue;
+                 case (META_COMMAND_UNRECOGNIZED_COMMAND):
+                     printf("Unrecognized command '%s'\n", input_buffer
+                     continue;
+             }
+         }
+
+         Statement statement;
+         switch (prepare_statement(input_buffer, &statement)) {
+             case (PREPARE_SUCCESS):
+                 break;
+             case (PREPARE_SYNTAX_ERROR):
+                 printf("Syntax error. Could not parse statement.\n");
+                 continue;
+             case (PREPARE_UNRECOGNIZED_STATEMENT):
+                 printf("Unrecognized keyword at start of '%s'.\n",
+                     input_buffer->buffer);
+                 continue;
+         }
+
+         switch (execute_statement(&statement, table)) {
+             case (EXECUTE_SUCCESS):
+                 printf("Executed.\n");

```

```
+         break;
+     case (EXECUTE_TABLE_FULL):
+         printf("Error: Table full.\n");
+         break;
+     }
+ }
+ }
```

[< Part 2 - World's Simplest SQL Compiler and Virtual Machine](#)

[Part 4 - Our First Tests \(and Bugs\) >](#)

[rss](#) | [subscribe by email](#)

This project is maintained by [cstack](#)

Hosted on GitHub Pages — Theme by [orderedlist](#)

Let's Build a Simple Database

Writing a sqlite clone from scratch in C

[Overview](#)

[View on GitHub](#) (pull requests welcome)

This project is no longer under active development. You can read more [here](#). But if you'd like to keep learning how to make your own SQLite clone from scratch, or one of many other projects like Docker, Redis, Git or BitTorrent, try [CodeCrafters](#).



Build Your Own SQLite

Try interactive mode →

Part 4 - Our First Tests (and Bugs)

[< Part 3 - An In-Memory, Append-Only, Single-Table Database](#)

[Part 5 - Persistence to Disk >](#)

We've got the ability to insert rows into our database and to print out all rows. Let's take a moment to test what we've got so far.

I'm going to use [rspec](#) to write my tests because I'm familiar with it, and the syntax is fairly readable.

I'll define a short helper to send a list of commands to our database program then make assertions about the output:

```

describe 'database' do
  def run_script(commands)
    raw_output = nil
    IO.popen("./db", "r+") do |pipe|
      commands.each do |command|
        pipe.puts command
      end

      pipe.close_write

      # Read entire output
      raw_output = pipe.gets(nil)
    end
    raw_output.split("\n")
  end

  it 'inserts and retrieves a row' do
    result = run_script([
      "insert 1 user1 person1@example.com",
      "select",
      ".exit",
    ])
    expect(result).to match_array([
      "db > Executed.",
      "db > (1, user1, person1@example.com)",
      "Executed.",
      "db > ",
    ])
  end
end
end

```

This simple test makes sure we get back what we put in. And indeed it passes:

```
bundle exec rspec
```

```
.
```

```

Finished in 0.00871 seconds (files took 0.09506 seconds to load)
1 example, 0 failures

```

Now it's feasible to test inserting a large number of rows into the database:

```

it 'prints error message when table is full' do
  script = (1..1401).map do |i|
    "insert #{i} user#{i} person#{i}@example.com"
  end
  script << ".exit"
  result = run_script(script)
  expect(result[-2]).to eq('db > Error: Table full.')
end

```

Running tests again...

```
bundle exec rspec
```

```
..
```

```

Finished in 0.01553 seconds (files took 0.08156 seconds to load)
2 examples, 0 failures

```

Sweet, it works! Our db can hold 1400 rows right now because we set the maximum number of pages to 100, and 14 rows can fit in a page.

Reading through the code we have so far, I realized we might not handle storing text fields correctly. Easy to test with this example:

```

it 'allows inserting strings that are the maximum length' do
  long_username = "a"*32
  long_email = "a"*255
  script = [
    "insert 1 #{long_username} #{long_email}",
    "select",
    ".exit",
  ]
  result = run_script(script)
  expect(result).to match_array([
    "db > Executed.",
    "db > (1, #{long_username}, #{long_email})",
    "Executed.",
    "db > ",
  ])
end

```

And the test fails!

Failures:

- 1) database allows inserting strings that are the maximum length
Failure/Error: raw_output.**split**("\\n")

ArgumentError:

```
invalid byte sequence in UTF-8
# ./spec/main_spec.rb:14:in `split'
# ./spec/main_spec.rb:14:in `run_script'
# ./spec/main_spec.rb:48:in `block (2 levels) in <top (required)>'
```

If we try it ourselves, we'll see that there's some weird characters when we try to print out the row. (I'm abbreviating the long strings):

```
db > insert 1 aaaaa... aaaaa...
Executed.
db > select
(1, aaaaa...aaa\0, aaaaa...aaa\0)
Executed.
db >
```

What's going on? If you take a look at our definition of a Row, we allocate exactly 32 bytes for username and exactly 255 bytes for email. But **C strings** are supposed to end with a null character, which we didn't allocate space for. The solution is to allocate one additional byte:

```
const uint32_t COLUMN_EMAIL_SIZE = 255;
typedef struct {
    uint32_t id;
-   char username[COLUMN_USERNAME_SIZE];
-   char email[COLUMN_EMAIL_SIZE];
+   char username[COLUMN_USERNAME_SIZE + 1];
+   char email[COLUMN_EMAIL_SIZE + 1];
} Row;
```

And indeed that fixes it:

```
bundle exec rspec
...

Finished in 0.0188 seconds (files took 0.08516 seconds to load)
```

3 examples, 0 failures

We should not allow inserting usernames or emails that are longer than column size. The spec for that looks like this:

```
it 'prints error message if strings are too long' do
  long_username = "a"*33
  long_email = "a"*256
  script = [
    "insert 1 #{long_username} #{long_email}",
    "select",
    ".exit",
  ]
  result = run_script(script)
  expect(result).to match_array([
    "db > String is too long.",
    "db > Executed.",
    "db > ",
  ])
end
```

In order to do this we need to upgrade our parser. As a reminder, we're currently using `scanf()`:

```
if (strncmp(input_buffer->buffer, "insert", 6) == 0) {
  statement->type = STATEMENT_INSERT;
  int args_assigned = sscanf(
    input_buffer->buffer, "insert %d %s %s", &(statement->row_to_insert.id,
    statement->row_to_insert.username, statement->row_to_insert.email);
  if (args_assigned < 3) {
    return PREPARE_SYNTAX_ERROR;
  }
  return PREPARE_SUCCESS;
}
```

But `scanf` has some disadvantages. If the string it's reading is larger than the buffer it's reading into, it will cause a buffer overflow and start writing into unexpected places. We want to check the length of each string before we copy it into a Row structure. And to do that, we need to divide the input by spaces.

I'm going to use `strtok()` to do that. I think it's easiest to understand if you see it

in action:

```
+PrepareResult prepare_insert(InputBuffer* input_buffer, Statement* statement) {
+   statement->type = STATEMENT_INSERT;
+
+   char* keyword = strtok(input_buffer->buffer, " ");
+   char* id_string = strtok(NULL, " ");
+   char* username = strtok(NULL, " ");
+   char* email = strtok(NULL, " ");
+
+   if (id_string == NULL || username == NULL || email == NULL)
+       return PREPARE_SYNTAX_ERROR;
+ }
+
+   int id = atoi(id_string);
+   if (strlen(username) > COLUMN_USERNAME_SIZE) {
+       return PREPARE_STRING_TOO_LONG;
+   }
+   if (strlen(email) > COLUMN_EMAIL_SIZE) {
+       return PREPARE_STRING_TOO_LONG;
+   }
+
+   statement->row_to_insert.id = id;
+   strcpy(statement->row_to_insert.username, username);
+   strcpy(statement->row_to_insert.email, email);
+
+   return PREPARE_SUCCESS;
+}
+
+PrepareResult prepare_statement(InputBuffer* input_buffer,
+                                Statement* statement) {
+   if (strncmp(input_buffer->buffer, "insert", 6) == 0) {
+       return prepare_insert(input_buffer, statement);
+       statement->type = STATEMENT_INSERT;
+       int args_assigned = sscanf(
+           input_buffer->buffer, "insert %d %s %s", &(statement->row_to_insert.id),
+           statement->row_to_insert.username, statement->row_to_insert.email);
+       if (args_assigned < 3) {
+           return PREPARE_SYNTAX_ERROR;
+       }
+       return PREPARE_SUCCESS;
+   }
+}
```

Calling `strtok` successively on the input buffer breaks it into substrings by inserting a null character whenever it reaches a delimiter (space, in our case). It returns a pointer to the start of the substring.

We can call `strlen()` on each text value to see if it's too long.

We can handle the error like we do any other error code:

```
enum PrepareResult_t {
    PREPARE_SUCCESS,
+   PREPARE_STRING_TOO_LONG,
    PREPARE_SYNTAX_ERROR,
    PREPARE_UNRECOGNIZED_STATEMENT
};
```

```
switch (prepare_statement(input_buffer, &statement)) {
    case (PREPARE_SUCCESS):
        break;
+   case (PREPARE_STRING_TOO_LONG):
+       printf("String is too long.\n");
+       continue;
    case (PREPARE_SYNTAX_ERROR):
        printf("Syntax error. Could not parse statement.\n");
        continue;
```

Which makes our test pass

```
bundle exec rspec
```

```
....
```

```
Finished in 0.02284 seconds (files took 0.116 seconds to load
4 examples, 0 failures
```

While we're here, we might as well handle one more error case:

```
it 'prints an error message if id is negative' do
  script = [
    "insert -1 cstack foo@bar.com",
    "select",
    ".exit",
  ]
```

```

result = run_script(script)
expect(result).to match_array([
  "db > ID must be positive.",
  "db > Executed.",
  "db > ",
])
end

```

```

enum PrepareResult_t {
    PREPARE_SUCCESS,
+   PREPARE_NEGATIVE_ID,
    PREPARE_STRING_TOO_LONG,
    PREPARE_SYNTAX_ERROR,
    PREPARE_UNRECOGNIZED_STATEMENT
@@ -148,9 +147,6 @@ PrepareResult prepare_insert(InputBuffer*
    }

    int id = atoi(id_string);
+   if (id < 0) {
+       return PREPARE_NEGATIVE_ID;
+   }
    if (strlen(username) > COLUMN_USERNAME_SIZE) {
        return PREPARE_STRING_TOO_LONG;
    }
@@ -230,9 +226,6 @@ int main(int argc, char* argv[]) {
    switch (prepare_statement(input_buffer, &statement)) {
        case (PREPARE_SUCCESS):
            break;
+       case (PREPARE_NEGATIVE_ID):
+           printf("ID must be positive.\n");
+           continue;
        case (PREPARE_STRING_TOO_LONG):
            printf("String is too long.\n");
            continue;

```

Alright, that's enough testing for now. Next is a very important feature: persistence! We're going to save our database to a file and read it back out again.

It's gonna be great.

Here's the complete diff for this part:

@@ -22,6 +22,8 @@

```
enum PrepareResult_t {
    PREPARE_SUCCESS,
+   PREPARE_NEGATIVE_ID,
+   PREPARE_STRING_TOO_LONG,
    PREPARE_SYNTAX_ERROR,
    PREPARE_UNRECOGNIZED_STATEMENT
};
```

@@ -34,8 +36,8 @@

```
#define COLUMN_EMAIL_SIZE 255
typedef struct {
    uint32_t id;
-   char username[COLUMN_USERNAME_SIZE];
-   char email[COLUMN_EMAIL_SIZE];
+   char username[COLUMN_USERNAME_SIZE + 1];
+   char email[COLUMN_EMAIL_SIZE + 1];
} Row;
```

@@ -150,18 +152,40 @@ MetaCommandResult do_meta_command(InputBuffer* input_buffer, Statement* statement) {

```
-PrepareResult prepare_statement(InputBuffer* input_buffer,
-                                Statement* statement) {
-   if (strncmp(input_buffer->buffer, "insert", 6) == 0) {
+PrepareResult prepare_insert(InputBuffer* input_buffer, Statement* statement) {
    statement->type = STATEMENT_INSERT;
-   int args_assigned = sscanf(
-       input_buffer->buffer, "insert %d %s %s", &(statement->row_to_insert.id),
-       statement->row_to_insert.username, statement->row_to_insert.email);
-   };
-   if (args_assigned < 3) {
+
+   char* keyword = strtok(input_buffer->buffer, " ");
+   char* id_string = strtok(NULL, " ");
+   char* username = strtok(NULL, " ");
+   char* email = strtok(NULL, " ");
+
+   if (id_string == NULL || username == NULL || email == NULL)
        return PREPARE_SYNTAX_ERROR;
    }
```

```

+
+ int id = atoi(id_string);
+ if (id < 0) {
+     return PREPARE_NEGATIVE_ID;
+ }
+ if (strlen(username) > COLUMN_USERNAME_SIZE) {
+     return PREPARE_STRING_TOO_LONG;
+ }
+ if (strlen(email) > COLUMN_EMAIL_SIZE) {
+     return PREPARE_STRING_TOO_LONG;
+ }
+
+ statement->row_to_insert.id = id;
+ strcpy(statement->row_to_insert.username, username);
+ strcpy(statement->row_to_insert.email, email);
+
+     return PREPARE_SUCCESS;
+
+}
+PrepareResult prepare_statement(InputBuffer* input_buffer,
+                               Statement* statement) {
+    if (strncmp(input_buffer->buffer, "insert", 6) == 0) {
+        return prepare_insert(input_buffer, statement);
+    }
+    if (strcmp(input_buffer->buffer, "select") == 0) {
+        statement->type = STATEMENT_SELECT;
@@ -223,6 +247,12 @@ int main(int argc, char* argv[]) {
+    switch (prepare_statement(input_buffer, &statement)) {
+        case (PREPARE_SUCCESS):
+            break;
+        case (PREPARE_NEGATIVE_ID):
+            printf("ID must be positive.\n");
+            continue;
+        case (PREPARE_STRING_TOO_LONG):
+            printf("String is too long.\n");
+            continue;
+        case (PREPARE_SYNTAX_ERROR):
+            printf("Syntax error. Could not parse statement.\n");
+            continue;

```

And we added tests:

```

+describe 'database' do
+  def run_script(commands)
+    raw_output = nil
+    IO.popen("./db", "r+") do |pipe|
+      commands.each do |command|
+        pipe.puts command
+      end
+
+      pipe.close_write
+
+      # Read entire output
+      raw_output = pipe.gets(nil)
+    end
+    raw_output.split("\n")
+  end
+
+  it 'inserts and retrieves a row' do
+    result = run_script([
+      "insert 1 user1 person1@example.com",
+      "select",
+      ".exit",
+    ])
+    expect(result).to match_array([
+      "db > Executed.",
+      "db > (1, user1, person1@example.com)",
+      "Executed.",
+      "db > ",
+    ])
+  end
+
+  it 'prints error message when table is full' do
+    script = (1..1401).map do |i|
+      "insert #{i} user#{i} person#{i}@example.com"
+    end
+    script << ".exit"
+    result = run_script(script)
+    expect(result[-2]).to eq('db > Error: Table full.')
+  end
+
+  it 'allows inserting strings that are the maximum length' do
+    long_username = "a"*32
+    long_email = "a"*255
  
```

```

+   script = [
+     "insert 1 #{long_username} #{long_email}",
+     "select",
+     ".exit",
+   ]
+   result = run_script(script)
+   expect(result).to match_array([
+     "db > Executed.",
+     "db > (1, #{long_username}, #{long_email})",
+     "Executed.",
+     "db > ",
+   ])
+ end
+
+ it 'prints error message if strings are too long' do
+   long_username = "a"*33
+   long_email = "a"*256
+   script = [
+     "insert 1 #{long_username} #{long_email}",
+     "select",
+     ".exit",
+   ]
+   result = run_script(script)
+   expect(result).to match_array([
+     "db > String is too long.",
+     "db > Executed.",
+     "db > ",
+   ])
+ end
+
+ it 'prints an error message if id is negative' do
+   script = [
+     "insert -1 cstack foo@bar.com",
+     "select",
+     ".exit",
+   ]
+   result = run_script(script)
+   expect(result).to match_array([
+     "db > ID must be positive.",
+     "db > Executed.",
+     "db > ",
+   ])

```

```
+ end
+end
```

[< Part 3 - An In-Memory, Append-Only, Single-Table Database](#)

[Part 5 - Persistence to Disk >](#)

[rss](#) | [subscribe by email](#)

This project is maintained by [cstack](#)

Hosted on GitHub Pages — Theme by [orderedlist](#)

Let's Build a Simple Database

Writing a sqlite clone from scratch in C

[Overview](#)

[View on GitHub](#) (pull requests welcome)

This project is no longer under active development. You can read more [here](#). But if you'd like to keep learning how to make your own SQLite clone from scratch, or one of many other projects like Docker, Redis, Git or BitTorrent, try [CodeCrafters](#).



Build Your Own SQLite

Try interactive mode →

Part 5 - Persistence to Disk

[< Part 4 - Our First Tests \(and Bugs\)](#)

[Part 6 - The Cursor Abstraction >](#)

"Nothing in the world can take the place of persistence." – [Calvin Coolidge](#)

Our database lets you insert records and read them back out, but only as long as you keep the program running. If you kill the program and start it back up, all your records are gone. Here's a spec for the behavior we want:

```
it 'keeps data after closing connection' do
```

```

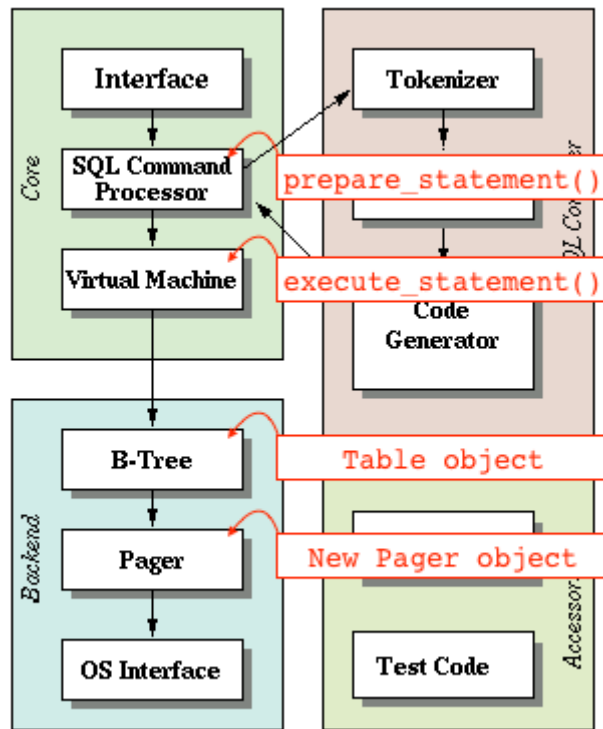
result1 = run_script([
    "insert 1 user1 person1@example.com",
    ".exit",
])
expect(result1).to match_array([
    "db > Executed.",
    "db > ",
])
result2 = run_script([
    "select",
    ".exit",
])
expect(result2).to match_array([
    "db > (1, user1, person1@example.com)",
    "Executed.",
    "db > ",
])
end

```

Like sqlite, we're going to persist records by saving the entire database to a file.

We already set ourselves up to do that by serializing rows into page-sized memory blocks. To add persistence, we can simply write those blocks of memory to a file, and read them back into memory the next time the program starts up.

To make this easier, we're going to make an abstraction called the pager. We ask the pager for page number x , and the pager gives us back a block of memory. It first looks in its cache. On a cache miss, it copies data from disk into memory (by reading the database file).



How our program matches up with SQLite architecture

The Pager accesses the page cache and the file. The Table object makes requests for pages through the pager:

```
+typedef struct {
+  int file_descriptor;
+  uint32_t file_length;
+  void* pages[TABLE_MAX_PAGES];
+} Pager;
+
+  typedef struct {
-   void* pages[TABLE_MAX_PAGES];
+   Pager* pager;
+   uint32_t num_rows;
+ } Table;
```

I'm renaming `new_table()` to `db_open()` because it now has the effect of opening a connection to the database. By opening a connection, I mean:

- opening the database file
- initializing a pager data structure
- initializing a table data structure

```
-Table* new_table() {
```

```

+Table* db_open(const char* filename) {
+  Pager* pager = pager_open(filename);
+  uint32_t num_rows = pager->file_length / ROW_SIZE;
+
+  Table* table = malloc(sizeof(Table));
+  table->num_rows = 0;
+  table->pager = pager;
+  table->num_rows = num_rows;

+  return table;
+}

```

`db_open()` in turn calls `pager_open()`, which opens the database file and keeps track of its size. It also initializes the page cache to all NULLs.

```

+Pager* pager_open(const char* filename) {
+  int fd = open(filename,
+
+                O_RDWR |      // Read/Write mode
+                O_CREAT,      // Create file if it does not exist
+                S_IWUSR |      // User write permission
+                S_IRUSR       // User read permission
+
+                );
+
+  if (fd == -1) {
+    printf("Unable to open file\n");
+    exit(EXIT_FAILURE);
+  }
+
+  off_t file_length = lseek(fd, 0, SEEK_END);
+
+  Pager* pager = malloc(sizeof(Pager));
+  pager->file_descriptor = fd;
+  pager->file_length = file_length;
+
+  for (uint32_t i = 0; i < TABLE_MAX_PAGES; i++) {
+    pager->pages[i] = NULL;
+  }
+
+  return pager;
+}

```

Following our new abstraction, we move the logic for fetching a page into its

own method:

```
void* row_slot(Table* table, uint32_t row_num) {
    uint32_t page_num = row_num / ROWS_PER_PAGE;
-   void* page = table->pages[page_num];
-   if (page == NULL) {
-       // Allocate memory only when we try to access page
-       page = table->pages[page_num] = malloc(PAGE_SIZE);
-   }
+   void* page = get_page(table->pager, page_num);
    uint32_t row_offset = row_num % ROWS_PER_PAGE;
    uint32_t byte_offset = row_offset * ROW_SIZE;
    return page + byte_offset;
}
```

The `get_page()` method has the logic for handling a cache miss. We assume pages are saved one after the other in the database file: Page 0 at offset 0, page 1 at offset 4096, page 2 at offset 8192, etc. If the requested page lies outside the bounds of the file, we know it should be blank, so we just allocate some memory and return it. The page will be added to the file when we flush the cache to disk later.

```
+void* get_page(Pager* pager, uint32_t page_num) {
+   if (page_num > TABLE_MAX_PAGES) {
+       printf("Tried to fetch page number out of bounds. %d > %d\n",
+              page_num, TABLE_MAX_PAGES);
+       exit(EXIT_FAILURE);
+   }
+
+   if (pager->pages[page_num] == NULL) {
+       // Cache miss. Allocate memory and load from file.
+       void* page = malloc(PAGE_SIZE);
+       uint32_t num_pages = pager->file_length / PAGE_SIZE;
+
+       // We might save a partial page at the end of the file
+       if (pager->file_length % PAGE_SIZE) {
+           num_pages += 1;
+       }
+
+       if (page_num <= num_pages) {
+           lseek(pager->file_descriptor, page_num * PAGE_SIZE, SEEK_SET);
+           ssize_t bytes_read = read(pager->file_descriptor, page,
```

```

+     if (bytes_read == -1) {
+         printf("Error reading file: %d\n", errno);
+         exit(EXIT_FAILURE);
+     }
+ }
+
+ pager->pages[page_num] = page;
+ }
+
+ return pager->pages[page_num];
+}

```

For now, we'll wait to flush the cache to disk until the user closes the connection to the database. When the user exits, we'll call a new method called `db_close()`, which

- flushes the page cache to disk
- closes the database file
- frees the memory for the Pager and Table data structures

```

+void db_close(Table* table) {
+    Pager* pager = table->pager;
+    uint32_t num_full_pages = table->num_rows / ROWS_PER_PAGE;
+
+    for (uint32_t i = 0; i < num_full_pages; i++) {
+        if (pager->pages[i] == NULL) {
+            continue;
+        }
+        pager_flush(pager, i, PAGE_SIZE);
+        free(pager->pages[i]);
+        pager->pages[i] = NULL;
+    }
+
+    // There may be a partial page to write to the end of the
+    // This should not be needed after we switch to a B-tree
+    uint32_t num_additional_rows = table->num_rows % ROWS_PER_PAGE;
+    if (num_additional_rows > 0) {
+        uint32_t page_num = num_full_pages;
+        if (pager->pages[page_num] != NULL) {
+            pager_flush(pager, page_num, num_additional_rows * ROWS_PER_PAGE);
+            free(pager->pages[page_num]);
+            pager->pages[page_num] = NULL;
+        }
+    }
+}

```

```

+     }
+ }
+
+ int result = close(pager->file_descriptor);
+ if (result == -1) {
+     printf("Error closing db file.\n");
+     exit(EXIT_FAILURE);
+ }
+ for (uint32_t i = 0; i < TABLE_MAX_PAGES; i++) {
+     void* page = pager->pages[i];
+     if (page) {
+         free(page);
+         pager->pages[i] = NULL;
+     }
+ }
+ free(pager);
+ free(table);
+}
+
-MetaCommandResult do_meta_command(InputBuffer* input_buffer)
+MetaCommandResult do_meta_command(InputBuffer* input_buffer,
+    if (strcmp(input_buffer->buffer, ".exit") == 0) {
+    db_close(table);
+    exit(EXIT_SUCCESS);
+    } else {
+        return META_COMMAND_UNRECOGNIZED_COMMAND;

```

In our current design, the length of the file encodes how many rows are in the database, so we need to write a partial page at the end of the file. That's why `pager_flush()` takes both a page number and a size. It's not the greatest design, but it will go away pretty quickly when we start implementing the B-tree.

```

+void pager_flush(Pager* pager, uint32_t page_num, uint32_t s
+ if (pager->pages[page_num] == NULL) {
+     printf("Tried to flush null page\n");
+     exit(EXIT_FAILURE);
+ }
+
+ off_t offset = lseek(pager->file_descriptor, page_num * PA
+
+ if (offset == -1) {
+     printf("Error seeking: %d\n", errno);

```

```

+     exit(EXIT_FAILURE);
+ }
+
+ ssize_t bytes_written =
+     write(pager->file_descriptor, pager->pages[page_num],
+
+
+ if (bytes_written == -1) {
+     printf("Error writing: %d\n", errno);
+     exit(EXIT_FAILURE);
+ }
+ }
+}

```

Lastly, we need to accept the filename as a command-line argument. Don't forget to also add the extra argument to `do_meta_command`:

```

int main(int argc, char* argv[]) {
- Table* table = new_table();
+ if (argc < 2) {
+     printf("Must supply a database filename.\n");
+     exit(EXIT_FAILURE);
+ }
+
+ char* filename = argv[1];
+ Table* table = db_open(filename);
+
+ InputBuffer* input_buffer = new_input_buffer();
+ while (true) {
+     print_prompt();
+     read_input(input_buffer);
+
+     if (input_buffer->buffer[0] == '.') {
-         switch (do_meta_command(input_buffer)) {
+         switch (do_meta_command(input_buffer, table)) {

```

With these changes, we're able to close then reopen the database, and our records are still there!

```

~ ./db mydb.db
db > insert 1 cstack foo@bar.com
Executed.
db > insert 2 voltorb volty@example.com

```

```
Executed.  
db > .exit  
~  
~ ./db mydb.db  
db > select  
(1, cstack, foo@bar.com)  
(2, voltorb, volty@example.com)  
Executed.  
db > .exit  
~
```

For extra fun, let's take a look at `mydb.db` to see how our data is being stored. I'll use `vim` as a hex editor to look at the memory layout of the file:

```
vim mydb.db  
:%!xxd
```

```
00000000: 0100 0000 6373 7461 636b 0000 0000 0000 ....cstack.....
00000010: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000020: 0000 0000 0066 6f6f 4062 6172 2e63 6f6d .....foo@bar.com
00000030: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000040: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000050: id `1` in username `cstack` terminated
00000060: little-endian by null character
00000070: byte order
00000080: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000090: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000000a0: 0000 0000 ce00 8562 1a63 1cad 6061 d258 .....b.c..`a.X
000000b0: ff7f 0000 485e d258 ff7f 0000 305e d258 ....H^X....0^X
000000c0: ff7f 0000 605e d258 ff7f 0000 0200 0000 ....`^X.....
000000d0: 0000 0000 105e d258 ff7f 0000 49b2 d109 .....^X....I...
000000e0: 0100 0000 305e d258 ff7f 0000 305e d258 ....0^X....0^X
000000f0: ff7f 0000 485e d258 ff7f 0000 0200 0000 ....H^X.....
00000100: 0100 0000 email terminated
00000110: 0100 0000 by null character
00000120: 0000 0000 0002 0000 007b 6f6c 746f 7262 .....voltorb
00000130: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000140: 0000 0000 0000 0000 0000 766f 6c74 7940 .....volty@
00000150: 6578 616d 706c 652e 636f 6d00 0000 0000 example.com.....
00000160: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000170: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000180: 0000 0000 0000 0000 0000 0000 0000 0000 .....
00000190: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000001a0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000001b0: 0000 0000 0000 0000 0000 0000 0000 0000 .....
000001c0: 0000 0000 0000 0000 00ce 0085 621a 631c .....b.c.
000001d0: ad60 61d2 58ff 7f00 0048 5ed2 58ff 7f00 ..`a.X....H^X...
000001e0: 0030 5ed2 58ff 7f00 0060 5ed2 58ff 7f00 .0^X....`^X...
000001f0: 0002 0000 0000 0000 0010 5ed2 58ff 7f00 .....^X...
00000200: 0049 b2d1 0901 0000 0030 5ed2 58ff 7f00 .I.....0^X...
00000210: 0030 5ed2 58ff 7f00 0048 5ed2 58ff 7f00 .0^X....H^X...
00000220: 0000 90ed 0601 0000 0000 0000 0002 0000 .....
00000230: 0020 a5d1 0901 0000 00c0 a4d1 0901 0000 .
00000240: 0000 0000 0000 0000 0000 0a
```

Current File Format

The first four bytes are the id of the first row (4 bytes because we store a `uint32_t`). It's stored in little-endian byte order, so the least significant byte comes first (01), followed by the higher-order bytes (00 00 00). We used `memcpy()` to copy bytes from our `Row` struct into the page cache, so that means the struct was laid out in memory in little-endian byte order. That's an attribute of the machine I compiled the program for. If we wanted to write a database file on my machine, then read it on a big-endian machine, we'd have to change our `serialize_row()` and `deserialize_row()` methods to always store and read bytes in the same order.

The next 33 bytes store the username as a null-terminated string. Apparently "cstack" in ASCII hexadecimal is 63 73 74 61 63 6b, followed by a null character (00). The rest of the 33 bytes are unused.

The next 256 bytes store the email in the same way. Here we can see some random junk after the terminating null character. This is most likely due to uninitialized memory in our Row struct. We copy the entire 256-byte email buffer into the file, including any bytes after the end of the string. Whatever was in memory when we allocated that struct is still there. But since we use a terminating null character, it has no effect on behavior.

NOTE: If we wanted to ensure that all bytes are initialized, it would suffice to use `strncpy` instead of `memcpy` while copying the username and email fields of rows in `serialize_row`, like so:

```
void serialize_row(Row* source, void* destination) {
    memcpy(destination + ID_OFFSET, &(source->id), ID_SIZE);
-   memcpy(destination + USERNAME_OFFSET, &(source->username), USERNAME_SIZE);
-   memcpy(destination + EMAIL_OFFSET, &(source->email), EMAIL_SIZE);
+   strncpy(destination + USERNAME_OFFSET, source->username, USERNAME_SIZE);
+   strncpy(destination + EMAIL_OFFSET, source->email, EMAIL_SIZE);
}
```

Conclusion

Alright! We've got persistence. It's not the greatest. For example if you kill the program without typing `.exit`, you lose your changes. Additionally, we're writing all pages back to disk, even pages that haven't changed since we read them from disk. These are issues we can address later.

Next time we'll introduce cursors, which should make it easier to implement the B-tree.

Until then!

Complete Diff

```
+#include <errno.h>
+#include <fcntl.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdint.h>
+#include <unistd.h>
```

```

    struct InputBuffer_t {
        char* buffer;
@@ -62,9 +65,16 @@ const uint32_t PAGE_SIZE = 4096;
        const uint32_t ROWS_PER_PAGE = PAGE_SIZE / ROW_SIZE;
        const uint32_t TABLE_MAX_ROWS = ROWS_PER_PAGE * TABLE_MAX_PA

+typedef struct {
+    int file_descriptor;
+    uint32_t file_length;
+    void* pages[TABLE_MAX_PAGES];
+} Pager;
+
+    typedef struct {
+        uint32_t num_rows;
+    - void* pages[TABLE_MAX_PAGES];
+    + Pager* pager;
+    } Table;

@@ -84,32 +94,81 @@ void deserialize_row(void *source, Row* d
    memcpy(&(destination->email), source + EMAIL_OFFSET, EMAIL_
    }

+void* get_page(Pager* pager, uint32_t page_num) {
+    if (page_num > TABLE_MAX_PAGES) {
+        printf("Tried to fetch page number out of bounds. %d > %d", page_num,
+            TABLE_MAX_PAGES);
+        exit(EXIT_FAILURE);
+    }
+
+    if (pager->pages[page_num] == NULL) {
+        // Cache miss. Allocate memory and load from file.
+        void* page = malloc(PAGE_SIZE);
+        uint32_t num_pages = pager->file_length / PAGE_SIZE;
+
+        // We might save a partial page at the end of the file
+        if (pager->file_length % PAGE_SIZE) {
+            num_pages += 1;
+        }
+
+        if (page_num <= num_pages) {
+            lseek(pager->file_descriptor, page_num * PAGE_SIZE,
+                ssize_t bytes_read = read(pager->file_descriptor, p

```

```

+         if (bytes_read == -1) {
+             printf("Error reading file: %d\n", errno);
+             exit(EXIT_FAILURE);
+         }
+     }
+
+     pager->pages[page_num] = page;
+ }
+
+ return pager->pages[page_num];
+}
+
+ void* row_slot(Table* table, uint32_t row_num) {
+     uint32_t page_num = row_num / ROWS_PER_PAGE;
+ - void *page = table->pages[page_num];
+ - if (page == NULL) {
+ -     // Allocate memory only when we try to access page
+ -     page = table->pages[page_num] = malloc(PAGE_SIZE);
+ - }
+ void *page = get_page(table->pager, page_num);
+     uint32_t row_offset = row_num % ROWS_PER_PAGE;
+     uint32_t byte_offset = row_offset * ROW_SIZE;
+     return page + byte_offset;
+ }
+
+ -Table* new_table() {
+ - Table* table = malloc(sizeof(Table));
+ - table->num_rows = 0;
+ +Pager* pager_open(const char* filename) {
+ + int fd = open(filename,
+ +     O_RDWR |           // Read/Write mode
+ +     O_CREAT,           // Create file if it does not exist
+ +     S_IWUSR |          // User write permission
+ +     S_IRUSR            // User read permission
+ + );
+ +
+ + if (fd == -1) {
+ +     printf("Unable to open file\n");
+ +     exit(EXIT_FAILURE);
+ + }
+ +
+ + off_t file_length = lseek(fd, 0, SEEK_END);

```

```

+
+ Pager* pager = malloc(sizeof(Pager));
+ pager->file_descriptor = fd;
+ pager->file_length = file_length;
+
+     for (uint32_t i = 0; i < TABLE_MAX_PAGES; i++) {
-         table->pages[i] = NULL;
+         pager->pages[i] = NULL;
+     }
-     return table;
+
+     return pager;
+ }

-void free_table(Table* table) {
-     for (int i = 0; table->pages[i]; i++) {
-         free(table->pages[i]);
-     }
-     free(table);
+Table* db_open(const char* filename) {
+     Pager* pager = pager_open(filename);
+     uint32_t num_rows = pager->file_length / ROW_SIZE;
+
+     Table* table = malloc(sizeof(Table));
+     table->pager = pager;
+     table->num_rows = num_rows;
+
+     return table;
+ }

    InputBuffer* new_input_buffer() {
@@ -142,10 +201,76 @@ void close_input_buffer(InputBuffer* in
    free(input_buffer);
}

+void pager_flush(Pager* pager, uint32_t page_num, uint32_t s
+ if (pager->pages[page_num] == NULL) {
+     printf("Tried to flush null page\n");
+     exit(EXIT_FAILURE);
+ }
+
+
+ off_t offset = lseek(pager->file_descriptor, page_num * PA

```

```

+             SEEK_SET);
+
+ if (offset == -1) {
+     printf("Error seeking: %d\n", errno);
+     exit(EXIT_FAILURE);
+ }
+
+ ssize_t bytes_written = write(
+     pager->file_descriptor, pager->pages[page_num], size
+ );
+
+ if (bytes_written == -1) {
+     printf("Error writing: %d\n", errno);
+     exit(EXIT_FAILURE);
+ }
+}
+
+void db_close(Table* table) {
+    Pager* pager = table->pager;
+    uint32_t num_full_pages = table->num_rows / ROWS_PER_PAGE;
+
+    for (uint32_t i = 0; i < num_full_pages; i++) {
+        if (pager->pages[i] == NULL) {
+            continue;
+        }
+        pager_flush(pager, i, PAGE_SIZE);
+        free(pager->pages[i]);
+        pager->pages[i] = NULL;
+    }
+
+    // There may be a partial page to write to the end of the
+    // This should not be needed after we switch to a B-tree
+    uint32_t num_additional_rows = table->num_rows % ROWS_PER_PAGE;
+    if (num_additional_rows > 0) {
+        uint32_t page_num = num_full_pages;
+        if (pager->pages[page_num] != NULL) {
+            pager_flush(pager, page_num, num_additional_rows * PAGE_SIZE);
+            free(pager->pages[page_num]);
+            pager->pages[page_num] = NULL;
+        }
+    }
+}
+

```

```

+   int result = close(pager->file_descriptor);
+   if (result == -1) {
+       printf("Error closing db file.\n");
+       exit(EXIT_FAILURE);
+   }
+   for (uint32_t i = 0; i < TABLE_MAX_PAGES; i++) {
+       void* page = pager->pages[i];
+       if (page) {
+           free(page);
+           pager->pages[i] = NULL;
+       }
+   }
+
+   free(pager);
+   free(table);
+}
+
+MetaCommandResult do_meta_command(InputBuffer* input_buffer,
+    if (strcmp(input_buffer->buffer, ".exit") == 0) {
+        close_input_buffer(input_buffer);
+        free_table(table);
+        db_close(table);
+        exit(EXIT_SUCCESS);
+    } else {
+        return META_COMMAND_UNRECOGNIZED_COMMAND;
+    }
@@ -182,6 +308,7 @@ PrepareResult prepare_insert(InputBuffer*
+    return PREPARE_SUCCESS;
+
+}
+
+PrepareResult prepare_statement(InputBuffer* input_buffer,
+                                Statement* statement) {
+    if (strncmp(input_buffer->buffer, "insert", 6) == 0) {
@@ -227,7 +354,14 @@ ExecuteResult execute_statement(Statemen
+    }

+    int main(int argc, char* argv[]) {
+        Table* table = new_table();
+        if (argc < 2) {
+            printf("Must supply a database filename.\n");
+            exit(EXIT_FAILURE);
+        }

```

```

+
+ char* filename = argv[1];
+ Table* table = db_open(filename);
+
+   InputBuffer* input_buffer = new_input_buffer();
+   while (true) {
+       print_prompt();

```

And the diff to our tests:

```

    describe 'database' do
+   before do
+       `rm -rf test.db`
+   end
+
+   def run_script(commands)
+       raw_output = nil
-       IO.popen("./db", "r+") do |pipe|
+       IO.popen("./db test.db", "r+") do |pipe|
+           commands.each do |command|
+               pipe.puts command
+           end
@@ -28,6 +32,27 @@ describe 'database' do
    ])
  end

+ it 'keeps data after closing connection' do
+   result1 = run_script([
+     "insert 1 user1 person1@example.com",
+     ".exit",
+   ])
+   expect(result1).to match_array([
+     "db > Executed.",
+     "db > ",
+   ])
+
+   result2 = run_script([
+     "select",
+     ".exit",
+   ])
+   expect(result2).to match_array([
+     "db > (1, user1, person1@example.com)",

```

```
+      "Executed.",  
+      "db > ",  
+    ])  
+  end  
+  
+  it 'prints error message when table is full' do  
+    script = (1..1401).map do |i|  
+      "insert #{i} user#{i} person#{i}@example.com"+    end  
+  end  
+end
```

[< Part 4 - Our First Tests \(and Bugs\)](#)

[Part 6 - The Cursor Abstraction >](#)

[rss](#) | [subscribe by email](#)

This project is maintained by [cstack](#)

Hosted on GitHub Pages — Theme by [orderedlist](#)

Let's Build a Simple Database

Writing a sqlite clone from scratch in C

[Overview](#)

[View on GitHub](#) (pull requests welcome)

This project is no longer under active development. You can read more [here](#). But if you'd like to keep learning how to make your own SQLite clone from scratch, or one of many other projects like Docker, Redis, Git or BitTorrent, try [CodeCrafters](#).



Build Your Own SQLite

Try interactive mode →

Part 6 - The Cursor Abstraction

[< Part 5 - Persistence to Disk](#)

[Part 7 - Introduction to the B-Tree >](#)

This should be a shorter part than the last one. We're just going to refactor a bit to make it easier to start the B-Tree implementation.

We're going to add a Cursor object which represents a location in the table. Things you might want to do with cursors:

- Create a cursor at the beginning of the table
- Create a cursor at the end of the table

- Access the row the cursor is pointing to
- Advance the cursor to the next row

Those are the behaviors we're going to implement now. Later, we will also want to:

- Delete the row pointed to by a cursor
- Modify the row pointed to by a cursor
- Search a table for a given ID, and create a cursor pointing to the row with that ID

Without further ado, here's the Cursor type:

```
+typedef struct {
+  Table* table;
+  uint32_t row_num;
+  bool end_of_table; // Indicates a position one past the last row
+} Cursor;
```

Given our current table data structure, all you need to identify a location in a table is the row number.

A cursor also has a reference to the table it's part of (so our cursor functions can take just the cursor as a parameter).

Finally, it has a boolean called `end_of_table`. This is so we can represent a position past the end of the table (which is somewhere we may want to insert a row).

`table_start()` and `table_end()` create new cursors:

```
+Cursor* table_start(Table* table) {
+  Cursor* cursor = malloc(sizeof(Cursor));
+  cursor->table = table;
+  cursor->row_num = 0;
+  cursor->end_of_table = (table->num_rows == 0);
+
+  return cursor;
+}
+
+Cursor* table_end(Table* table) {
+  Cursor* cursor = malloc(sizeof(Cursor));
+  cursor->table = table;
```

```

+ cursor->row_num = table->num_rows;
+ cursor->end_of_table = true;
+
+ return cursor;
+}

```

Our `row_slot()` function will become `cursor_value()`, which returns a pointer to the position described by the cursor:

```

-void* row_slot(Table* table, uint32_t row_num) {
+void* cursor_value(Cursor* cursor) {
+  uint32_t row_num = cursor->row_num;
+  uint32_t page_num = row_num / ROWS_PER_PAGE;
-  void* page = get_page(table->pager, page_num);
+  void* page = get_page(cursor->table->pager, page_num);
+  uint32_t row_offset = row_num % ROWS_PER_PAGE;
+  uint32_t byte_offset = row_offset * ROW_SIZE;
+  return page + byte_offset;
}

```

Advancing the cursor in our current table structure is as simple as incrementing the row number. This will be a bit more complicated in a B-tree.

```

+void cursor_advance(Cursor* cursor) {
+  cursor->row_num += 1;
+  if (cursor->row_num >= cursor->table->num_rows) {
+    cursor->end_of_table = true;
+  }
+}

```

Finally we can change our “virtual machine” methods to use the cursor abstraction. When inserting a row, we open a cursor at the end of table, write to that cursor location, then close the cursor.

```

    Row* row_to_insert = &(statement->row_to_insert);
+  Cursor* cursor = table_end(table);

-  serialize_row(row_to_insert, row_slot(table, table->num_rows));
+  serialize_row(row_to_insert, cursor_value(cursor));
    table->num_rows += 1;

```

```

+   free(cursor);
+
+   return EXECUTE_SUCCESS;
+ }

```

When selecting all rows in the table, we open a cursor at the start of the table, print the row, then advance the cursor to the next row. Repeat until we've reached the end of the table.

```

    ExecuteResult execute_select(Statement* statement, Table* table) {
+   Cursor* cursor = table_start(table);
+
+   Row row;
-   for (uint32_t i = 0; i < table->num_rows; i++) {
-       deserialize_row(row_slot(table, i), &row);
+   while (!(cursor->end_of_table)) {
+       deserialize_row(cursor_value(cursor), &row);
+       print_row(&row);
+       cursor_advance(cursor);
+   }
+
+   free(cursor);
+
+   return EXECUTE_SUCCESS;
+ }

```

Alright, that's it! Like I said, this was a shorter refactor that should help us as we rewrite our table data structure into a B-Tree. `execute_select()` and `execute_insert()` can interact with the table entirely through the cursor without assuming anything about how the table is stored.

Here's the complete diff to this part:

```

@@ -78,6 +78,13 @@ struct {
+   Table;

+typedef struct {
+   Table* table;
+   uint32_t row_num;
+   bool end_of_table; // Indicates a position one past the last row
+} Cursor;
+

```

```

void print_row(Row* row) {
    printf("(%d, %s, %s)\n", row->id, row->username, row->email);
}
@@ -126,12 +133,38 @@ void* get_page(Pager* pager, uint32_t page_num) {
    return pager->pages[page_num];
}

-void* row_slot(Table* table, uint32_t row_num) {
-    uint32_t page_num = row_num / ROWS_PER_PAGE;
-    void *page = get_page(table->pager, page_num);
-    uint32_t row_offset = row_num % ROWS_PER_PAGE;
-    uint32_t byte_offset = row_offset * ROW_SIZE;
-    return page + byte_offset;
+Cursor* table_start(Table* table) {
+    Cursor* cursor = malloc(sizeof(Cursor));
+    cursor->table = table;
+    cursor->row_num = 0;
+    cursor->end_of_table = (table->num_rows == 0);
+
+    return cursor;
+}
+
+Cursor* table_end(Table* table) {
+    Cursor* cursor = malloc(sizeof(Cursor));
+    cursor->table = table;
+    cursor->row_num = table->num_rows;
+    cursor->end_of_table = true;
+
+    return cursor;
+}
+
+void* cursor_value(Cursor* cursor) {
+    uint32_t row_num = cursor->row_num;
+    uint32_t page_num = row_num / ROWS_PER_PAGE;
+    void *page = get_page(cursor->table->pager, page_num);
+    uint32_t row_offset = row_num % ROWS_PER_PAGE;
+    uint32_t byte_offset = row_offset * ROW_SIZE;
+    return page + byte_offset;
+}
+
+void cursor_advance(Cursor* cursor) {
+    cursor->row_num += 1;

```

```

+   if (cursor->row_num >= cursor->table->num_rows) {
+       cursor->end_of_table = true;
+   }
+   }

    Pager* pager_open(const char* filename) {
@@ -327,19 +360,28 @@ ExecuteResult execute_insert(Statement*
    }

    Row* row_to_insert = &(amp;statement->row_to_insert);
+   Cursor* cursor = table_end(table);

-   serialize_row(row_to_insert, row_slot(table, table->num_rows));
+   serialize_row(row_to_insert, cursor_value(cursor));
    table->num_rows += 1;

+   free(cursor);
+
    return EXECUTE_SUCCESS;
}

ExecuteResult execute_select(Statement* statement, Table* table) {
+   Cursor* cursor = table_start(table);
+
    Row row;
-   for (uint32_t i = 0; i < table->num_rows; i++) {
-       deserialize_row(row_slot(table, i), &row);
+   while (!(cursor->end_of_table)) {
+       deserialize_row(cursor_value(cursor), &row);
+       print_row(&row);
+       cursor_advance(cursor);
+   }
+
+   free(cursor);
+
    return EXECUTE_SUCCESS;
}

```

[rss](#) | [subscribe by email](#)

This project is maintained by [cstack](#)

Hosted on GitHub Pages — Theme by [orderedlist](#)

Let's Build a Simple Database

Writing a sqlite clone from scratch in C

[Overview](#)

[View on GitHub](#) (pull requests welcome)

This project is no longer under active development. You can read more [here](#). But if you'd like to keep learning how to make your own SQLite clone from scratch, or one of many other projects like Docker, Redis, Git or BitTorrent, try [CodeCrafters](#).



Build Your Own SQLite

[Try interactive mode →](#)

Part 7 - Introduction to the B-Tree

[< Part 6 - The Cursor Abstraction](#)

[Part 8 - B-Tree Leaf Node Format >](#)

The B-Tree is the data structure SQLite uses to represent both tables and indexes, so it's a pretty central idea. This article will just introduce the data structure, so it won't have any code.

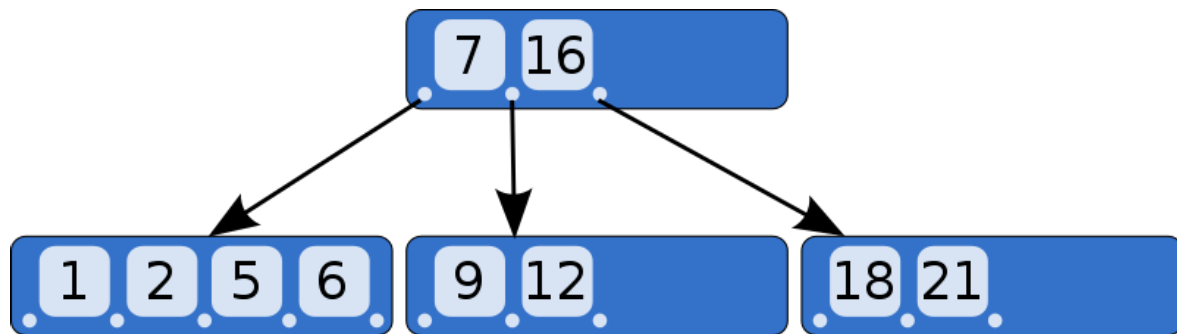
Why is a tree a good data structure for a database?

- Searching for a particular value is fast (logarithmic time)
- Inserting / deleting a value you've already found is fast (constant-ish time)

to rebalance)

- Traversing a range of values is fast (unlike a hash map)

A B-Tree is different from a binary tree (the “B” probably stands for the inventor’s name, but could also stand for “balanced”). Here’s an example B-Tree:



example B-Tree (<https://en.wikipedia.org/wiki/File:B-tree.svg>)

Unlike a binary tree, each node in a B-Tree can have more than 2 children. Each node can have up to m children, where m is called the tree’s “order”. To keep the tree mostly balanced, we also say nodes have to have at least $m/2$ children (rounded up).

Exceptions:

- Leaf nodes have 0 children
- The root node can have fewer than m children but must have at least 2
- If the root node is a leaf node (the only node), it still has 0 children

The picture from above is a B-Tree, which SQLite uses to store indexes. To store tables, SQLite uses a variation called a B+ tree.

	B-tree	B+ tree
Pronounced	“Bee Tree”	“Bee Plus Tree”
Used to store	Indexes	Tables
Internal nodes store keys	Yes	Yes
Internal nodes store values	Yes	No
Number of children per node	Less	More
Internal nodes vs. leaf nodes	Same structure	Different structure

Until we get to implementing indexes, I’m going to talk solely about B+ trees, but I’ll just refer to it as a B-tree or a btree.

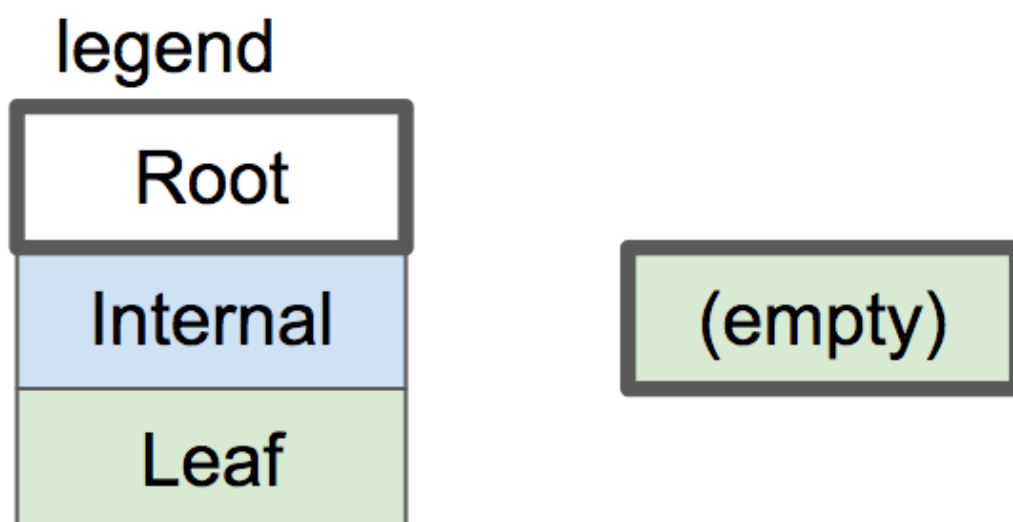
Nodes with children are called “internal” nodes. Internal nodes and leaf nodes are structured differently:

For an order-m tree...	Internal Node	Leaf Node
Stores	keys and pointers to children	keys and values
Number of keys	up to m-1	as many as will fit
Number of pointers	number of keys + 1	none
Number of values	none	number of keys
Key purpose	used for routing	paired with value
Stores values?	No	Yes

Let’s work through an example to see how a B-tree grows as you insert elements into it. To keep things simple, the tree will be order 3. That means:

- up to 3 children per internal node
- up to 2 keys per internal node
- at least 2 children per internal node
- at least 1 key per internal node

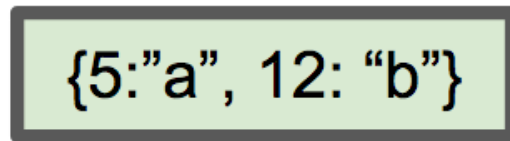
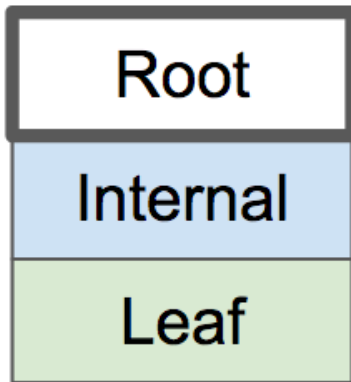
An empty B-tree has a single node: the root node. The root node starts as a leaf node with zero key/value pairs:



empty btree

If we insert a couple key/value pairs, they are stored in the leaf node in sorted order.

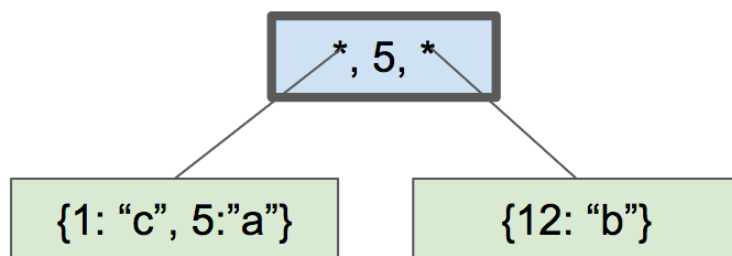
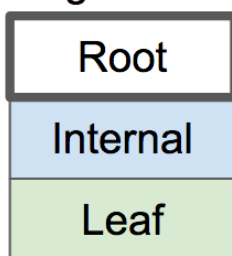
legend



one-node btree

Let's say that the capacity of a leaf node is two key/value pairs. When we insert another, we have to split the leaf node and put half the pairs in each node. Both nodes become children of a new internal node which will now be the root node.

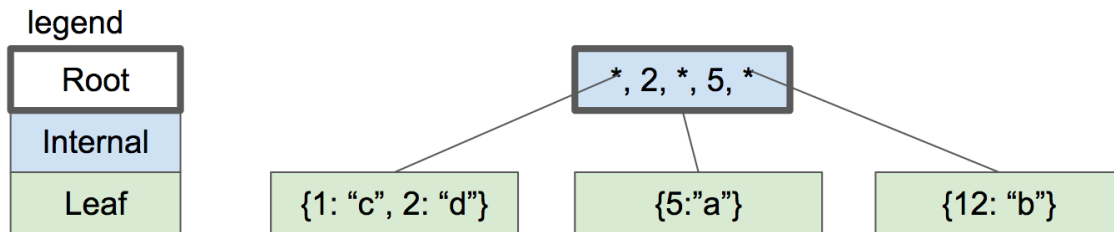
legend



two-level btree

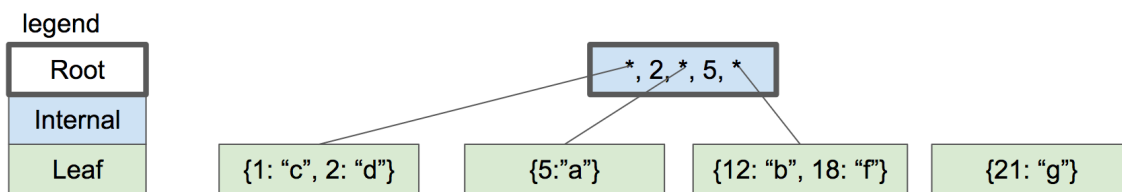
The internal node has 1 key and 2 pointers to child nodes. If we want to look up a key that is less than or equal to 5, we look in the left child. If we want to look up a key greater than 5, we look in the right child.

Now let's insert the key "2". First we look up which leaf node it would be in if it was present, and we arrive at the left leaf node. The node is full, so we split the leaf node and create a new entry in the parent node.



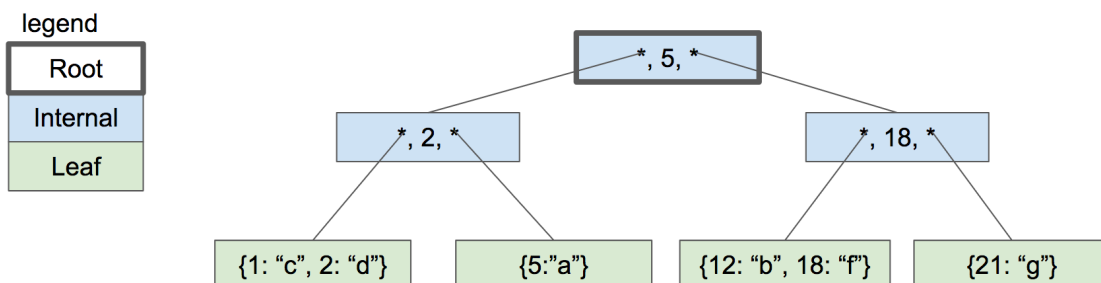
four-node btree

Let's keep adding keys. 18 and 21. We get to the point where we have to split again, but there's no room in the parent node for another key/pointer pair.



no room in internal node

The solution is to split the root node into two internal nodes, then create new root node to be their parent.



three-level btree

The depth of the tree only increases when we split the root node. Every leaf node has the same depth and close to the same number of key/value pairs, so the tree remains balanced and quick to search.

I'm going to hold off on discussion of deleting keys from the tree until after we've implemented insertion.

When we implement this data structure, each node will correspond to one page. The root node will exist in page 0. Child pointers will simply be the page number that contains the child node.

Next time, we start implementing the btree!

[< Part 6 - The Cursor Abstraction](#)

[Part 8 - B-Tree Leaf Node Format >](#)

[rss](#) | [subscribe by email](#)

This project is maintained by [cstack](#)

Hosted on GitHub Pages — Theme by [orderedlist](#)

Let's Build a Simple Database

Writing a sqlite clone from scratch in C

[Overview](#)

[View on GitHub](#) (pull requests welcome)

This project is no longer under active development. You can read more [here](#). But if you'd like to keep learning how to make your own SQLite clone from scratch, or one of many other projects like Docker, Redis, Git or BitTorrent, try [CodeCrafters](#).



Build Your Own SQLite

Try interactive mode →

Part 8 - B-Tree Leaf Node Format

[< Part 7 - Introduction to the B-Tree](#)

[Part 9 - Binary Search and Duplicate Keys >](#)

We're changing the format of our table from an unsorted array of rows to a B-Tree. This is a pretty big change that is going to take multiple articles to implement. By the end of this article, we'll define the layout of a leaf node and support inserting key/value pairs into a single-node tree. But first, let's recap the reasons for switching to a tree structure.

Alternative Table Formats

With the current format, each page stores only rows (no metadata) so it is pretty space efficient. Insertion is also fast because we just append to the end. However, finding a particular row can only be done by scanning the entire table. And if we want to delete a row, we have to fill in the hole by moving every row that comes after it.

If we stored the table as an array, but kept rows sorted by id, we could use binary search to find a particular id. However, insertion would be slow because we would have to move a lot of rows to make space.

Instead, we're going with a tree structure. Each node in the tree can contain a variable number of rows, so we have to store some information in each node to keep track of how many rows it contains. Plus there is the storage overhead of all the internal nodes which don't store any rows. In exchange for a larger database file, we get fast insertion, deletion and lookup.

	Unsorted Array of rows	Sorted Array of rows	Tree of nodes
Pages contain	only data	only data	metadata, primary keys, and data
Rows per page	more	more	fewer
Insertion	$O(1)$	$O(n)$	$O(\log(n))$
Deletion	$O(n)$	$O(n)$	$O(\log(n))$
Lookup by id	$O(n)$	$O(\log(n))$	$O(\log(n))$

Node Header Format

Leaf nodes and internal nodes have different layouts. Let's make an enum to keep track of node type:

```
+typedef enum { NODE_INTERNAL, NODE_LEAF } NodeType;
```

Each node will correspond to one page. Internal nodes will point to their children by storing the page number that stores the child. The btree asks the pager for a particular page number and gets back a pointer into the page cache. Pages are stored in the database file one after the other in order of page number.

Nodes need to store some metadata in a header at the beginning of the page. Every node will store what type of node it is, whether or not it is the root node, and a pointer to its parent (to allow finding a node's siblings). I define constants for the size and offset of every header field:

```

+/*
+ * Common Node Header Layout
+ */
+const uint32_t NODE_TYPE_SIZE = sizeof(uint8_t);
+const uint32_t NODE_TYPE_OFFSET = 0;
+const uint32_t IS_ROOT_SIZE = sizeof(uint8_t);
+const uint32_t IS_ROOT_OFFSET = NODE_TYPE_SIZE;
+const uint32_t PARENT_POINTER_SIZE = sizeof(uint32_t);
+const uint32_t PARENT_POINTER_OFFSET = IS_ROOT_OFFSET + IS_ROOT_SIZE;
+const uint8_t COMMON_NODE_HEADER_SIZE =
+    NODE_TYPE_SIZE + IS_ROOT_SIZE + PARENT_POINTER_SIZE;

```

Leaf Node Format

In addition to these common header fields, leaf nodes need to store how many "cells" they contain. A cell is a key/value pair.

```

+/*
+ * Leaf Node Header Layout
+ */
+const uint32_t LEAF_NODE_NUM_CELLS_SIZE = sizeof(uint32_t);
+const uint32_t LEAF_NODE_NUM_CELLS_OFFSET = COMMON_NODE_HEADER_SIZE;
+const uint32_t LEAF_NODE_HEADER_SIZE =
+    COMMON_NODE_HEADER_SIZE + LEAF_NODE_NUM_CELLS_SIZE;

```

The body of a leaf node is an array of cells. Each cell is a key followed by a value (a serialized row).

```

+/*
+ * Leaf Node Body Layout
+ */
+const uint32_t LEAF_NODE_KEY_SIZE = sizeof(uint32_t);
+const uint32_t LEAF_NODE_KEY_OFFSET = 0;
+const uint32_t LEAF_NODE_VALUE_SIZE = ROW_SIZE;
+const uint32_t LEAF_NODE_VALUE_OFFSET =
+    LEAF_NODE_KEY_OFFSET + LEAF_NODE_KEY_SIZE;

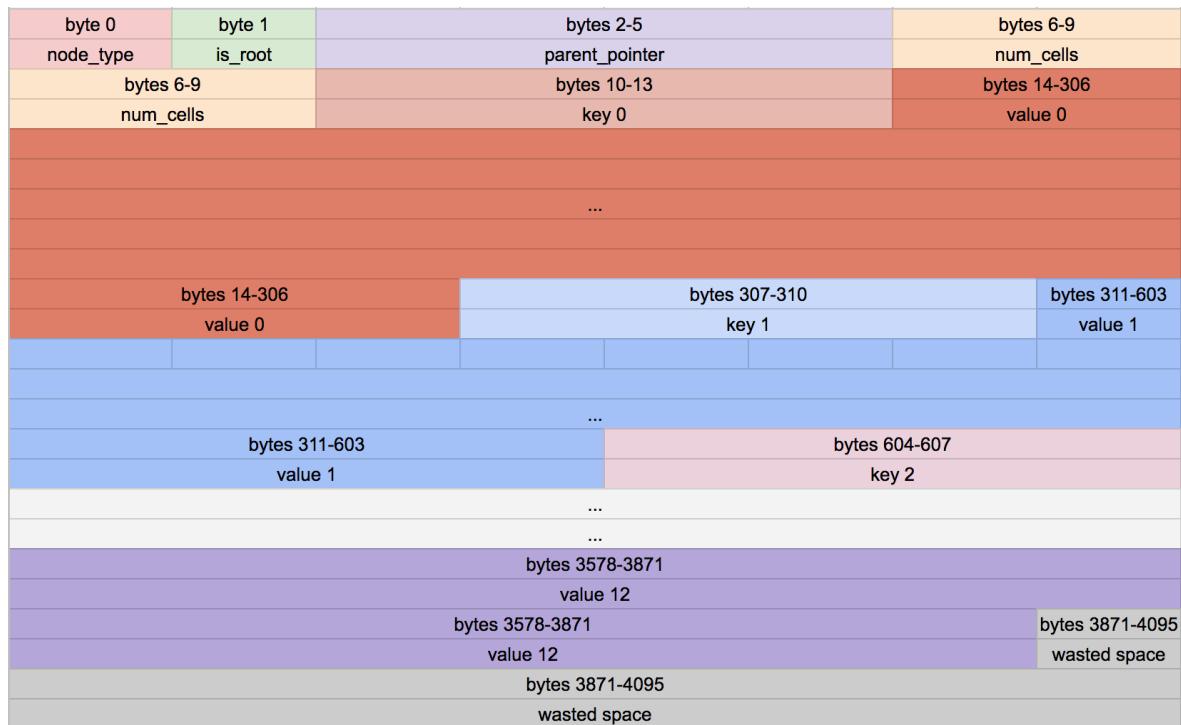
```

```

+const uint32_t LEAF_NODE_CELL_SIZE = LEAF_NODE_KEY_SIZE + LEAF_NODE_VALUE_SIZE;
+const uint32_t LEAF_NODE_SPACE_FOR_CELLS = PAGE_SIZE - LEAF_NODE_HEADER_SIZE;
+const uint32_t LEAF_NODE_MAX_CELLS =
+    LEAF_NODE_SPACE_FOR_CELLS / LEAF_NODE_CELL_SIZE;

```

Based on these constants, here's what the layout of a leaf node looks like currently:



Our leaf node format

It's a little space inefficient to use an entire byte per boolean value in the header, but this makes it easier to write code to access those values.

Also notice that there's some wasted space at the end. We store as many cells as we can after the header, but the leftover space can't hold an entire cell. We leave it empty to avoid splitting cells between nodes.

Accessing Leaf Node Fields

The code to access keys, values and metadata all involve pointer arithmetic using the constants we just defined.

```

+uint32_t* leaf_node_num_cells(void* node) {
+    return node + LEAF_NODE_NUM_CELLS_OFFSET;
+}
+
+void* leaf_node_cell(void* node, uint32_t cell_num) {

```

```

+ return node + LEAF_NODE_HEADER_SIZE + cell_num * LEAF_NODE_
+}
+
+uint32_t* leaf_node_key(void* node, uint32_t cell_num) {
+ return leaf_node_cell(node, cell_num);
+}
+
+void* leaf_node_value(void* node, uint32_t cell_num) {
+ return leaf_node_cell(node, cell_num) + LEAF_NODE_KEY_SIZE
+}
+
+void initialize_leaf_node(void* node) { *leaf_node_num_cells
+

```

These methods return a pointer to the value in question, so they can be used both as a getter and a setter.

Changes to Pager and Table Objects

Every node is going to take up exactly one page, even if it's not full. That means our pager no longer needs to support reading/writing partial pages.

```

-void pager_flush(Pager* pager, uint32_t page_num, uint32_t s
+void pager_flush(Pager* pager, uint32_t page_num) {
    if (pager->pages[page_num] == NULL) {
        printf("Tried to flush null page\n");
        exit(EXIT_FAILURE);
@@ -242,7 +337,7 @@ void pager_flush(Pager* pager, uint32_t p
    }

    ssize_t bytes_written =
-    write(pager->file_descriptor, pager->pages[page_num],
+    write(pager->file_descriptor, pager->pages[page_num],

    if (bytes_written == -1) {
        printf("Error writing: %d\n", errno);

```

```

void db_close(Table* table) {
    Pager* pager = table->pager;
-    uint32_t num_full_pages = table->num_rows / ROWS_PER_PAGE;

-    for (uint32_t i = 0; i < num_full_pages; i++) {

```

```

+   for (uint32_t i = 0; i < pager->num_pages; i++) {
        if (pager->pages[i] == NULL) {
            continue;
        }
-   pager_flush(pager, i, PAGE_SIZE);
+   pager_flush(pager, i);
        free(pager->pages[i]);
        pager->pages[i] = NULL;
    }

-   // There may be a partial page to write to the end of the
-   // This should not be needed after we switch to a B-tree
-   uint32_t num_additional_rows = table->num_rows % ROWS_PER_PAGE;
-   if (num_additional_rows > 0) {
-       uint32_t page_num = num_full_pages;
-       if (pager->pages[page_num] != NULL) {
-           pager_flush(pager, page_num, num_additional_rows * ROW_SIZE);
-           free(pager->pages[page_num]);
-           pager->pages[page_num] = NULL;
-       }
-   }
-
    int result = close(pager->file_descriptor);
    if (result == -1) {
        printf("Error closing db file.\n");
    }
}

```

Now it makes more sense to store the number of pages in our database rather than the number of rows. The number of pages should be associated with the pager object, not the table, since it's the number of pages used by the database, not a particular table. A btree is identified by its root node page number, so the table object needs to keep track of that.

```

const uint32_t PAGE_SIZE = 4096;
const uint32_t TABLE_MAX_PAGES = 100;
-const uint32_t ROWS_PER_PAGE = PAGE_SIZE / ROW_SIZE;
-const uint32_t TABLE_MAX_ROWS = ROWS_PER_PAGE * TABLE_MAX_PAGES;

typedef struct {
    int file_descriptor;
    uint32_t file_length;
+   uint32_t num_pages;
    void* pages[TABLE_MAX_PAGES];
}

```

```

} Pager;

typedef struct {
    Pager* pager;
-   uint32_t num_rows;
+   uint32_t root_page_num;
} Table;

```

```

@@ -127,6 +200,10 @@ void* get_page(Pager* pager, uint32_t page_num)
{
    if (page_num >= pager->num_pages) {
        pager->pages[page_num] = page;
+
+   if (page_num >= pager->num_pages) {
+       pager->num_pages = page_num + 1;
+   }
    }

    return pager->pages[page_num];

```

```

@@ -184,6 +269,12 @@ Pager* pager_open(const char* filename)
{
    Pager* pager = malloc(sizeof(Pager));
    pager->file_descriptor = fd;
    pager->file_length = file_length;
+   pager->num_pages = (file_length / PAGE_SIZE);
+
+   if (file_length % PAGE_SIZE != 0) {
+       printf("Db file is not a whole number of pages. Corrupt\n");
+       exit(EXIT_FAILURE);
+   }

    for (uint32_t i = 0; i < TABLE_MAX_PAGES; i++) {
        pager->pages[i] = NULL;
    }
}

```

Changes to the Cursor Object

A cursor represents a position in the table. When our table was a simple array of rows, we could access a row given just the row number. Now that it's a tree, we identify a position by the page number of the node, and the cell number within that node.

```

typedef struct {
    Table* table;
-   uint32_t row_num;
+   uint32_t page_num;
+   uint32_t cell_num;
    bool end_of_table; // Indicates a position one past the last cell
} Cursor;

```

```

Cursor* table_start(Table* table) {
    Cursor* cursor = malloc(sizeof(Cursor));
    cursor->table = table;
-   cursor->row_num = 0;
-   cursor->end_of_table = (table->num_rows == 0);
+   cursor->page_num = table->root_page_num;
+   cursor->cell_num = 0;
+
+   void* root_node = get_page(table->pager, table->root_page_num);
+   uint32_t num_cells = *leaf_node_num_cells(root_node);
+   cursor->end_of_table = (num_cells == 0);

    return cursor;
}

```

```

Cursor* table_end(Table* table) {
    Cursor* cursor = malloc(sizeof(Cursor));
    cursor->table = table;
-   cursor->row_num = table->num_rows;
+   cursor->page_num = table->root_page_num;
+
+   void* root_node = get_page(table->pager, table->root_page_num);
+   uint32_t num_cells = *leaf_node_num_cells(root_node);
+   cursor->cell_num = num_cells;
    cursor->end_of_table = true;

    return cursor;
}

```

```

void* cursor_value(Cursor* cursor) {
-   uint32_t row_num = cursor->row_num;
-   uint32_t page_num = row_num / ROWS_PER_PAGE;

```

```

+ uint32_t page_num = cursor->page_num;
  void* page = get_page(cursor->table->pager, page_num);
- uint32_t row_offset = row_num % ROWS_PER_PAGE;
- uint32_t byte_offset = row_offset * ROW_SIZE;
- return page + byte_offset;
+ return leaf_node_value(page, cursor->cell_num);
}

```

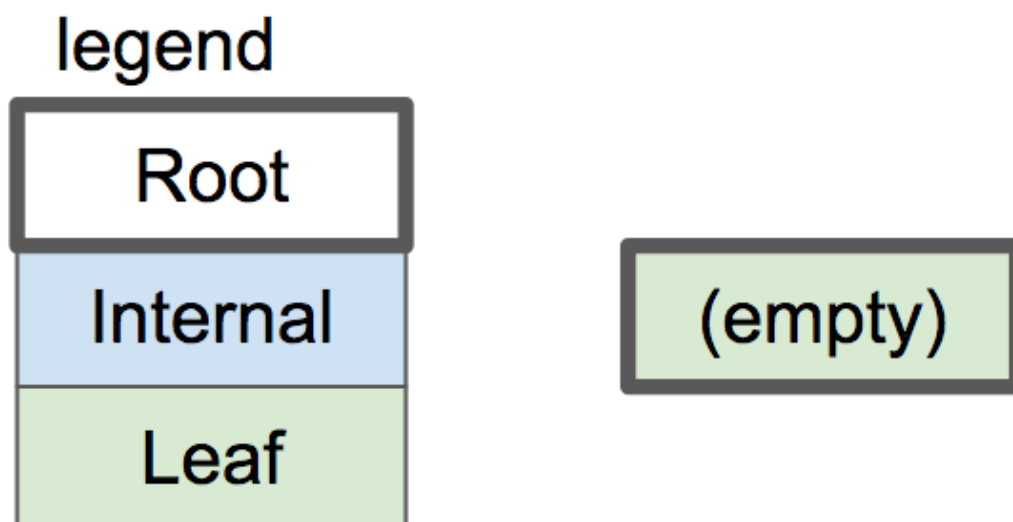
```

void cursor_advance(Cursor* cursor) {
- cursor->row_num += 1;
- if (cursor->row_num >= cursor->table->num_rows) {
+ uint32_t page_num = cursor->page_num;
+ void* node = get_page(cursor->table->pager, page_num);
+
+ cursor->cell_num += 1;
+ if (cursor->cell_num >= (*leaf_node_num_cells(node))) {
    cursor->end_of_table = true;
  }
}

```

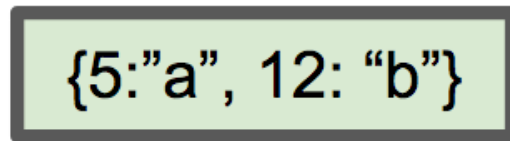
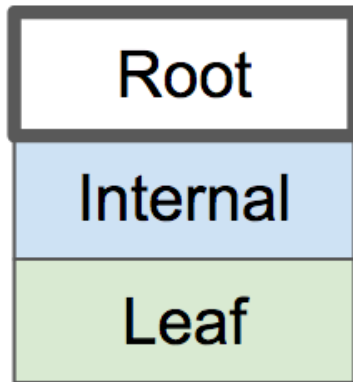
Insertion Into a Leaf Node

In this article we're only going to implement enough to get a single-node tree. Recall from last article that a tree starts out as an empty leaf node:



Key/value pairs can be added until the leaf node is full:

legend



one-node btree

When we open the database for the first time, the database file will be empty, so we initialize page 0 to be an empty leaf node (the root node):

```
Table* db_open(const char* filename) {
    Pager* pager = pager_open(filename);
-   uint32_t num_rows = pager->file_length / ROW_SIZE;

    Table* table = malloc(sizeof(Table));
    table->pager = pager;
-   table->num_rows = num_rows;
+   table->root_page_num = 0;
+
+   if (pager->num_pages == 0) {
+       // New database file. Initialize page 0 as leaf node.
+       void* root_node = get_page(pager, 0);
+       initialize_leaf_node(root_node);
+   }

    return table;
}
```

Next we'll make a function for inserting a key/value pair into a leaf node. It will take a cursor as input to represent the position where the pair should be inserted.

```
+void leaf_node_insert(Cursor* cursor, uint32_t key, Row* val)
+ void* node = get_page(cursor->table->pager, cursor->page_n
```

```

+
+  uint32_t num_cells = *leaf_node_num_cells(node);
+  if (num_cells >= LEAF_NODE_MAX_CELLS) {
+    // Node full
+    printf("Need to implement splitting a leaf node.\n");
+    exit(EXIT_FAILURE);
+  }
+
+  if (cursor->cell_num < num_cells) {
+    // Make room for new cell
+    for (uint32_t i = num_cells; i > cursor->cell_num; i--)
+      memcpy(leaf_node_cell(node, i), leaf_node_cell(node, i + 1),
+             LEAF_NODE_CELL_SIZE);
+  }
+
+  *(leaf_node_num_cells(node)) += 1;
+  *(leaf_node_key(node, cursor->cell_num)) = key;
+  serialize_row(value, leaf_node_value(node, cursor->cell_num), cursor->cell_num);
+}
+

```

We haven't implemented splitting yet, so we error if the node is full. Next we shift cells one space to the right to make room for the new cell. Then we write the new key/value into the empty space.

Since we assume the tree only has one node, our `execute_insert()` function simply needs to call this helper method:

```

ExecuteResult execute_insert(Statement* statement, Table* table) {
-  if (table->num_rows >= TABLE_MAX_ROWS) {
+  void* node = get_page(table->pager, table->root_page_num);
+  if ((*leaf_node_num_cells(node) >= LEAF_NODE_MAX_CELLS)) {
+    return EXECUTE_TABLE_FULL;
+  }

  Row* row_to_insert = &(statement->row_to_insert);
  Cursor* cursor = table_end(table);

-  serialize_row(row_to_insert, cursor_value(cursor));
-  table->num_rows += 1;
+  leaf_node_insert(cursor, row_to_insert->id, row_to_insert)
}

```

```
free(cursor);
```

With those changes, our database should work as before! Except now it returns a “Table Full” error much sooner, since we can’t split the root node yet.

How many rows can the leaf node hold?

Command to Print Constants

I’m adding a new meta command to print out a few constants of interest.

```
+void print_constants() {
+  printf("ROW_SIZE: %d\n", ROW_SIZE);
+  printf("COMMON_NODE_HEADER_SIZE: %d\n", COMMON_NODE_HEADER_SIZE);
+  printf("LEAF_NODE_HEADER_SIZE: %d\n", LEAF_NODE_HEADER_SIZE);
+  printf("LEAF_NODE_CELL_SIZE: %d\n", LEAF_NODE_CELL_SIZE);
+  printf("LEAF_NODE_SPACE_FOR_CELLS: %d\n", LEAF_NODE_SPACE_FOR_CELLS);
+  printf("LEAF_NODE_MAX_CELLS: %d\n", LEAF_NODE_MAX_CELLS);
+}
+
@@ -294,6 +376,14 @@ MetaCommandResult do_meta_command(InputBuffer *input_buffer) {
    if (strcmp(input_buffer->buffer, ".exit") == 0) {
        db_close(table);
        exit(EXIT_SUCCESS);
+   } else if (strcmp(input_buffer->buffer, ".constants") == 0) {
+       printf("Constants:\n");
+       print_constants();
+       return META_COMMAND_SUCCESS;
    } else {
        return META_COMMAND_UNRECOGNIZED_COMMAND;
    }
}
```

I’m also adding a test so we get alerted when those constants change:

```
+  it 'prints constants' do
+    script = [
+      ".constants",
+      ".exit",
+    ]
+    result = run_script(script)
+  end
```

```

+     expect(result).to match_array([
+       "db > Constants:",
+       "ROW_SIZE: 293",
+       "COMMON_NODE_HEADER_SIZE: 6",
+       "LEAF_NODE_HEADER_SIZE: 10",
+       "LEAF_NODE_CELL_SIZE: 297",
+       "LEAF_NODE_SPACE_FOR_CELLS: 4086",
+       "LEAF_NODE_MAX_CELLS: 13",
+       "db > ",
+     ])
+   end

```

So our table can hold 13 rows right now!

Tree Visualization

To help with debugging and visualization, I'm also adding a meta command to print out a representation of the btree.

```

+void print_leaf_node(void* node) {
+  uint32_t num_cells = *leaf_node_num_cells(node);
+  printf("leaf (size %d)\n", num_cells);
+  for (uint32_t i = 0; i < num_cells; i++) {
+    uint32_t key = *leaf_node_key(node, i);
+    printf("  - %d : %d\n", i, key);
+  }
+}
+
+

```

```

@@ -294,6 +376,14 @@ MetaCommandResult do_meta_command(InputBuffer*
    if (strcmp(input_buffer->buffer, ".exit") == 0) {
        db_close(table);
        exit(EXIT_SUCCESS);
+   } else if (strcmp(input_buffer->buffer, ".btree") == 0) {
+       printf("Tree:\n");
+       print_leaf_node(get_page(table->pager, 0));
+       return META_COMMAND_SUCCESS;
    } else if (strcmp(input_buffer->buffer, ".constants") == 0) {
        printf("Constants:\n");
        print_constants();
        return META_COMMAND_SUCCESS;
    } else {

```

```
    return META_COMMAND_UNRECOGNIZED_COMMAND;
}
```

And a test

```
+ it 'allows printing out the structure of a one-node btree'
+   script = [3, 1, 2].map do |i|
+     "insert #{i} user#{i} person#{i}@example.com"
+   end
+   script << ".btree"
+   script << ".exit"
+   result = run_script(script)
+
+   expect(result).to match_array([
+     "db > Executed.",
+     "db > Executed.",
+     "db > Executed.",
+     "db > Tree:",
+     "leaf (size 3)",
+     "  - 0 : 3",
+     "  - 1 : 1",
+     "  - 2 : 2",
+     "db > "
+   ])
+ end
```

Uh oh, we're still not storing rows in sorted order. You'll notice that `execute_insert()` inserts into the leaf node at the position returned by `table_end()`. So rows are stored in the order they were inserted, just like before.

Next Time

This all might seem like a step backwards. Our database now stores fewer rows than it did before, and we're still storing rows in unsorted order. But like I said at the beginning, this is a big change and it's important to break it up into manageable steps.

Next time, we'll implement finding a record by primary key, and start storing rows in sorted order.

Complete Diff

```

@@ -62,29 +62,101 @@ const uint32_t ROW_SIZE = ID_SIZE + USERI

    const uint32_t PAGE_SIZE = 4096;
    #define TABLE_MAX_PAGES 100
-const uint32_t ROWS_PER_PAGE = PAGE_SIZE / ROW_SIZE;
-const uint32_t TABLE_MAX_ROWS = ROWS_PER_PAGE * TABLE_MAX_PA

typedef struct {
    int file_descriptor;
    uint32_t file_length;
+ uint32_t num_pages;
    void* pages[TABLE_MAX_PAGES];
} Pager;

typedef struct {
    Pager* pager;
- uint32_t num_rows;
+ uint32_t root_page_num;
} Table;

typedef struct {
    Table* table;
- uint32_t row_num;
+ uint32_t page_num;
+ uint32_t cell_num;
    bool end_of_table; // Indicates a position one past the l
} Cursor;

+typedef enum { NODE_INTERNAL, NODE_LEAF } NodeType;
+
+/*
+ * Common Node Header Layout
+ */
+const uint32_t NODE_TYPE_SIZE = sizeof(uint8_t);
+const uint32_t NODE_TYPE_OFFSET = 0;
+const uint32_t IS_ROOT_SIZE = sizeof(uint8_t);
+const uint32_t IS_ROOT_OFFSET = NODE_TYPE_SIZE;
+const uint32_t PARENT_POINTER_SIZE = sizeof(uint32_t);
+const uint32_t PARENT_POINTER_OFFSET = IS_ROOT_OFFSET + IS_R
+const uint8_t COMMON_NODE_HEADER_SIZE =
+    NODE_TYPE_SIZE + IS_ROOT_SIZE + PARENT_POINTER_SIZE;
+

```

```

+/*
+ * Leaf Node Header Layout
+ */
+const uint32_t LEAF_NODE_NUM_CELLS_SIZE = sizeof(uint32_t);
+const uint32_t LEAF_NODE_NUM_CELLS_OFFSET = COMMON_NODE_HEADER_SIZE;
+const uint32_t LEAF_NODE_HEADER_SIZE =
+    COMMON_NODE_HEADER_SIZE + LEAF_NODE_NUM_CELLS_SIZE;
+
+/*
+ * Leaf Node Body Layout
+ */
+const uint32_t LEAF_NODE_KEY_SIZE = sizeof(uint32_t);
+const uint32_t LEAF_NODE_KEY_OFFSET = 0;
+const uint32_t LEAF_NODE_VALUE_SIZE = ROW_SIZE;
+const uint32_t LEAF_NODE_VALUE_OFFSET =
+    LEAF_NODE_KEY_OFFSET + LEAF_NODE_KEY_SIZE;
+const uint32_t LEAF_NODE_CELL_SIZE = LEAF_NODE_KEY_SIZE + LEAF_NODE_VALUE_SIZE;
+const uint32_t LEAF_NODE_SPACE_FOR_CELLS = PAGE_SIZE - LEAF_NODE_HEADER_SIZE;
+const uint32_t LEAF_NODE_MAX_CELLS =
+    LEAF_NODE_SPACE_FOR_CELLS / LEAF_NODE_CELL_SIZE;
+
+uint32_t* leaf_node_num_cells(void* node) {
+    return node + LEAF_NODE_NUM_CELLS_OFFSET;
+}
+
+void* leaf_node_cell(void* node, uint32_t cell_num) {
+    return node + LEAF_NODE_HEADER_SIZE + cell_num * LEAF_NODE_CELL_SIZE;
+}
+
+uint32_t* leaf_node_key(void* node, uint32_t cell_num) {
+    return leaf_node_cell(node, cell_num);
+}
+
+void* leaf_node_value(void* node, uint32_t cell_num) {
+    return leaf_node_cell(node, cell_num) + LEAF_NODE_KEY_SIZE;
+}
+
+void print_constants() {
+    printf("ROW_SIZE: %d\n", ROW_SIZE);
+    printf("COMMON_NODE_HEADER_SIZE: %d\n", COMMON_NODE_HEADER_SIZE);
+    printf("LEAF_NODE_HEADER_SIZE: %d\n", LEAF_NODE_HEADER_SIZE);
+    printf("LEAF_NODE_CELL_SIZE: %d\n", LEAF_NODE_CELL_SIZE);
+}

```

```
+ printf("LEAF_NODE_SPACE_FOR_CELLS: %d\n", LEAF_NODE_SPACE_FOR_CELLS);
+ printf("LEAF_NODE_MAX_CELLS: %d\n", LEAF_NODE_MAX_CELLS);
+
+
+void print_leaf_node(void* node) {
+  uint32_t num_cells = *leaf_node_num_cells(node);
+  printf("leaf (size %d)\n", num_cells);
+  for (uint32_t i = 0; i < num_cells; i++) {
+    uint32_t key = *leaf_node_key(node, i);
+    printf("  - %d : %d\n", i, key);
+  }
+}
+
+
+void print_row(Row* row) {
+  printf("(%d, %s, %s)\n", row->id, row->username, row->email);
+}
@@ -101,6 +173,8 @@ void deserialize_row(void *source, Row* destination) {
+  memcpy(&(destination->email), source + EMAIL_OFFSET, EMAIL_SIZE);
+}
+
+
+void initialize_leaf_node(void* node) { *leaf_node_num_cells(node) = 0; }
+
+
+void* get_page(Pager* pager, uint32_t page_num) {
+  if (page_num > TABLE_MAX_PAGES) {
+    printf("Tried to fetch page number out of bounds. %d > %d\n", page_num, TABLE_MAX_PAGES);
+    return NULL;
+  }
+
+  pager->pages[page_num] = page;
+
+  if (page_num >= pager->num_pages) {
+    pager->num_pages = page_num + 1;
+  }
+
+  return pager->pages[page_num];
+}
@@ -136,8 +214,12 @@ void* get_page(Pager* pager, uint32_t page_num) {
+Cursor* table_start(Table* table) {
+  Cursor* cursor = malloc(sizeof(Cursor));
+  cursor->table = table;
+  cursor->row_num = 0;
+  cursor->end_of_table = (table->num_rows == 0);
+}
```

```

+ cursor->page_num = table->root_page_num;
+ cursor->cell_num = 0;
+
+ void* root_node = get_page(table->pager, table->root_page_
+ uint32_t num_cells = *leaf_node_num_cells(root_node);
+ cursor->end_of_table = (num_cells == 0);

    return cursor;
}
@@ -145,24 +227,28 @@ Cursor* table_start(Table* table) {
    Cursor* table_end(Table* table) {
        Cursor* cursor = malloc(sizeof(Cursor));
        cursor->table = table;
-        cursor->row_num = table->num_rows;
+        cursor->page_num = table->root_page_num;
+
+        void* root_node = get_page(table->pager, table->root_page_
+        uint32_t num_cells = *leaf_node_num_cells(root_node);
+        cursor->cell_num = num_cells;
        cursor->end_of_table = true;

        return cursor;
    }

    void* cursor_value(Cursor* cursor) {
-        uint32_t row_num = cursor->row_num;
-        uint32_t page_num = row_num / ROWS_PER_PAGE;
+        uint32_t page_num = cursor->page_num;
        void* page = get_page(cursor->table->pager, page_num);
-        uint32_t row_offset = row_num % ROWS_PER_PAGE;
-        uint32_t byte_offset = row_offset * ROW_SIZE;
-        return page + byte_offset;
+        return leaf_node_value(page, cursor->cell_num);
    }

    void cursor_advance(Cursor* cursor) {
-        cursor->row_num += 1;
-        if (cursor->row_num >= cursor->table->num_rows) {
+        uint32_t page_num = cursor->page_num;
+        void* node = get_page(cursor->table->pager, page_num);
+
+        cursor->cell_num += 1;

```

```

+   if (cursor->cell_num >= (*leaf_node_num_cells(node))) {
+       cursor->end_of_table = true;
+   }
+ }
@@ -185,6 +271,12 @@ Pager* pager_open(const char* filename)
    Pager* pager = malloc(sizeof(Pager));
    pager->file_descriptor = fd;
    pager->file_length = file_length;
+   pager->num_pages = (file_length / PAGE_SIZE);
+
+   if (file_length % PAGE_SIZE != 0) {
+       printf("Db file is not a whole number of pages. Corrupt
+       exit(EXIT_FAILURE);
+   }

    for (uint32_t i = 0; i < TABLE_MAX_PAGES; i++) {
        pager->pages[i] = NULL;
@@ -194,11 +285,15 @@ Pager* pager_open(const char* filename)
@@ -195,11 +287,16 @@ Pager* pager_open(const char* filename)

Table* db_open(const char* filename) {
    Pager* pager = pager_open(filename);
-   uint32_t num_rows = pager->file_length / ROW_SIZE;

    Table* table = malloc(sizeof(Table));
    table->pager = pager;
-   table->num_rows = num_rows;
+   table->root_page_num = 0;
+
+   if (pager->num_pages == 0) {
+       // New database file. Initialize page 0 as leaf node.
+       void* root_node = get_page(pager, 0);
+       initialize_leaf_node(root_node);
+   }

    return table;
}
@@ -234,7 +331,7 @@ void close_input_buffer(InputBuffer* input
    free(input_buffer);
}

-void pager_flush(Pager* pager, uint32_t page_num, uint32_t s

```

```

+void pager_flush(Pager* pager, uint32_t page_num) {
    if (pager->pages[page_num] == NULL) {
        printf("Tried to flush null page\n");
        exit(EXIT_FAILURE);
@@ -242,7 +337,7 @@ void pager_flush(Pager* pager, uint32_t p
@@ -249,7 +346,7 @@ void pager_flush(Pager* pager, uint32_t p
    }

    ssize_t bytes_written =
-        write(pager->file_descriptor, pager->pages[page_num],
+        write(pager->file_descriptor, pager->pages[page_num],

    if (bytes_written == -1) {
        printf("Error writing: %d\n", errno);
@@ -252,29 +347,16 @@ void pager_flush(Pager* pager, uint32_t
@@ -260,29 +357,16 @@ void pager_flush(Pager* pager, uint32_t

void db_close(Table* table) {
    Pager* pager = table->pager;
-    uint32_t num_full_pages = table->num_rows / ROWS_PER_PAGE;

-    for (uint32_t i = 0; i < num_full_pages; i++) {
+    for (uint32_t i = 0; i < pager->num_pages; i++) {
        if (pager->pages[i] == NULL) {
            continue;
        }
-        pager_flush(pager, i, PAGE_SIZE);
+        pager_flush(pager, i);
        free(pager->pages[i]);
        pager->pages[i] = NULL;
    }

-    // There may be a partial page to write to the end of the
-    // This should not be needed after we switch to a B-tree
-    uint32_t num_additional_rows = table->num_rows % ROWS_PER_P
-    if (num_additional_rows > 0) {
-        uint32_t page_num = num_full_pages;
-        if (pager->pages[page_num] != NULL) {
-            pager_flush(pager, page_num, num_additional_rows * ROW
-            free(pager->pages[page_num]);
-            pager->pages[page_num] = NULL;
-        }

```

```

- }
-
    int result = close(pager->file_descriptor);
    if (result == -1) {
        printf("Error closing db file.\n");
@@ -305,6 +389,14 @@ MetaCommandResult do_meta_command(InputB
    if (strcmp(input_buffer->buffer, ".exit") == 0) {
        db_close(table);
        exit(EXIT_SUCCESS);
+ } else if (strcmp(input_buffer->buffer, ".btree") == 0) {
+     printf("Tree:\n");
+     print_leaf_node(get_page(table->pager, 0));
+     return META_COMMAND_SUCCESS;
+ } else if (strcmp(input_buffer->buffer, ".constants") == 0
+     printf("Constants:\n");
+     print_constants();
+     return META_COMMAND_SUCCESS;
    } else {
        return META_COMMAND_UNRECOGNIZED_COMMAND;
    }
@@ -354,16 +446,39 @@ PrepareResult prepare_statement(InputBu
    return PREPARE_UNRECOGNIZED_STATEMENT;
}

+void leaf_node_insert(Cursor* cursor, uint32_t key, Row* val
+ void* node = get_page(cursor->table->pager, cursor->page_n
+
+ uint32_t num_cells = *leaf_node_num_cells(node);
+ if (num_cells >= LEAF_NODE_MAX_CELLS) {
+     // Node full
+     printf("Need to implement splitting a leaf node.\n");
+     exit(EXIT_FAILURE);
+ }
+
+ if (cursor->cell_num < num_cells) {
+     // Make room for new cell
+     for (uint32_t i = num_cells; i > cursor->cell_num; i--)
+         memcpy(leaf_node_cell(node, i), leaf_node_cell(node, i
+             LEAF_NODE_CELL_SIZE);
+ }
+ }
+

```

```

+  *(leaf_node_num_cells(node)) += 1;
+  *(leaf_node_key(node, cursor->cell_num)) = key;
+  serialize_row(value, leaf_node_value(node, cursor->cell_num));
+}
+
+ ExecuteResult execute_insert(Statement* statement, Table* table) {
-   if (table->num_rows >= TABLE_MAX_ROWS) {
+   void* node = get_page(table->pager, table->root_page_num);
+   if ((*leaf_node_num_cells(node) >= LEAF_NODE_MAX_CELLS)) {
+       return EXECUTE_TABLE_FULL;
+   }

+   Row* row_to_insert = &(statement->row_to_insert);
+   Cursor* cursor = table_end(table);

-   serialize_row(row_to_insert, cursor_value(cursor));
-   table->num_rows += 1;
+   leaf_node_insert(cursor, row_to_insert->id, row_to_insert)

+   free(cursor);

```

And the specs:

```

+   it 'allows printing out the structure of a one-node btree'
+     script = [3, 1, 2].map do |i|
+       "insert #{i} user#{i} person#{i}@example.com"
+     end
+     script << ".btree"
+     script << ".exit"
+     result = run_script(script)
+
+     expect(result).to match_array([
+       "db > Executed.",
+       "db > Executed.",
+       "db > Executed.",
+       "db > Tree:",
+       "leaf (size 3)",
+       "  - 0 : 3",
+       "  - 1 : 1",
+       "  - 2 : 2",
+       "db > "
+     ])

```

```
+ end
+
+ it 'prints constants' do
+   script = [
+     ".constants",
+     ".exit",
+   ]
+   result = run_script(script)
+
+   expect(result).to match_array([
+     "db > Constants:",
+     "ROW_SIZE: 293",
+     "COMMON_NODE_HEADER_SIZE: 6",
+     "LEAF_NODE_HEADER_SIZE: 10",
+     "LEAF_NODE_CELL_SIZE: 297",
+     "LEAF_NODE_SPACE_FOR_CELLS: 4086",
+     "LEAF_NODE_MAX_CELLS: 13",
+     "db > ",
+   ])
+ end
end
```

[< Part 7 - Introduction to the B-Tree](#)

[Part 9 - Binary Search and Duplicate Keys >](#)

[rss](#) | [subscribe by email](#)

This project is maintained by [cstack](#)

Hosted on GitHub Pages — Theme by [orderedlist](#)

Let's Build a Simple Database

Writing a sqlite clone from scratch in C

[Overview](#)

[View on GitHub](#) (pull requests welcome)

This project is no longer under active development. You can read more [here](#). But if you'd like to keep learning how to make your own SQLite clone from scratch, or one of many other projects like Docker, Redis, Git or BitTorrent, try [CodeCrafters](#).



Build Your Own SQLite

Try interactive mode →

Part 9 - Binary Search and Duplicate Keys

[< Part 8 - B-Tree Leaf Node Format](#)

[Part 10 - Splitting a Leaf Node >](#)

Last time we noted that we're still storing keys in unsorted order. We're going to fix that problem, plus detect and reject duplicate keys.

Right now, our `execute_insert()` function always chooses to insert at the end of the table. Instead, we should search the table for the correct place to insert, then insert there. If the key already exists there, return an error.

```

ExecuteResult execute_insert(Statement* statement, Table* table) {
    void* node = get_page(table->pager, table->root_page_num);
    - if ((*leaf_node_num_cells(node) >= LEAF_NODE_MAX_CELLS)) {
+   uint32_t num_cells = (*leaf_node_num_cells(node));
+   if (num_cells >= LEAF_NODE_MAX_CELLS) {
        return EXECUTE_TABLE_FULL;
    }

    Row* row_to_insert = &(amp;statement->row_to_insert);
    - Cursor* cursor = table_end(table);
+   uint32_t key_to_insert = row_to_insert->id;
+   Cursor* cursor = table_find(table, key_to_insert);
+
+   if (cursor->cell_num < num_cells) {
+       uint32_t key_at_index = *leaf_node_key(node, cursor->cell_num);
+       if (key_at_index == key_to_insert) {
+           return EXECUTE_DUPLICATE_KEY;
+       }
+   }

    leaf_node_insert(cursor, row_to_insert->id, row_to_insert)
}

```

We don't need the `table_end()` function anymore.

```

-Cursor* table_end(Table* table) {
-   Cursor* cursor = malloc(sizeof(Cursor));
-   cursor->table = table;
-   cursor->page_num = table->root_page_num;
-
-   void* root_node = get_page(table->pager, table->root_page_num);
-   uint32_t num_cells = *leaf_node_num_cells(root_node);
-   cursor->cell_num = num_cells;
-   cursor->end_of_table = true;
-
-   return cursor;
-}

```

We'll replace it with a method that searches the tree for a given key.

```

+/*
+Return the position of the given key.

```

```

+If the key is not present, return the position
+where it should be inserted
+*/
+Cursor* table_find(Table* table, uint32_t key) {
+  uint32_t root_page_num = table->root_page_num;
+  void* root_node = get_page(table->pager, root_page_num);
+
+  if (get_node_type(root_node) == NODE_LEAF) {
+    return leaf_node_find(table, root_page_num, key);
+  } else {
+    printf("Need to implement searching an internal node\n");
+    exit(EXIT_FAILURE);
+  }
+}
+}

```

I'm stubbing out the branch for internal nodes because we haven't implemented internal nodes yet. We can search the leaf node with binary search.

```

+Cursor* leaf_node_find(Table* table, uint32_t page_num, uint32_t key) {
+  void* node = get_page(table->pager, page_num);
+  uint32_t num_cells = *leaf_node_num_cells(node);
+
+  Cursor* cursor = malloc(sizeof(Cursor));
+  cursor->table = table;
+  cursor->page_num = page_num;
+
+  // Binary search
+  uint32_t min_index = 0;
+  uint32_t one_past_max_index = num_cells;
+  while (one_past_max_index != min_index) {
+    uint32_t index = (min_index + one_past_max_index) / 2;
+    uint32_t key_at_index = *leaf_node_key(node, index);
+    if (key == key_at_index) {
+      cursor->cell_num = index;
+      return cursor;
+    }
+    if (key < key_at_index) {
+      one_past_max_index = index;
+    } else {
+      min_index = index + 1;
+    }
+  }
+}

```

```

+ }
+
+ cursor->cell_num = min_index;
+ return cursor;
+}

```

This will either return

- the position of the key,
- the position of another key that we'll need to move if we want to insert the new key, or
- the position one past the last key

Since we're now checking node type, we need functions to get and set that value in a node.

```

+NodeType get_node_type(void* node) {
+  uint8_t value = *((uint8_t*)(node + NODE_TYPE_OFFSET));
+  return (NodeType)value;
+}
+
+void set_node_type(void* node, NodeType type) {
+  uint8_t value = type;
+  *((uint8_t*)(node + NODE_TYPE_OFFSET)) = value;
+}

```

We have to cast to `uint8_t` first to ensure it's serialized as a single byte.

We also need to initialize node type.

```

-void initialize_leaf_node(void* node) { *leaf_node_num_cells
+void initialize_leaf_node(void* node) {
+  set_node_type(node, NODE_LEAF);
+  *leaf_node_num_cells(node) = 0;
+}

```

Lastly, we need to make and handle a new error code.

```

-enum ExecuteResult_t { EXECUTE_SUCCESS, EXECUTE_TABLE_FULL }
+enum ExecuteResult_t {
+  EXECUTE_SUCCESS,
+  EXECUTE_DUPLICATE_KEY,

```

```
+ EXECUTE_TABLE_FULL
+};
```

```
        case (EXECUTE_SUCCESS):
            printf("Executed.\n");
            break;
+       case (EXECUTE_DUPLICATE_KEY):
+           printf("Error: Duplicate key.\n");
+           break;
        case (EXECUTE_TABLE_FULL):
            printf("Error: Table full.\n");
            break;
```

With these changes, our test can change to check for sorted order:

```
        "db > Executed.",
        "db > Tree:",
        "leaf (size 3)",
-       "  - 0 : 3",
-       "  - 1 : 1",
-       "  - 2 : 2",
+       "  - 0 : 1",
+       "  - 1 : 2",
+       "  - 2 : 3",
        "db > "
    ])
end
```

And we can add a new test for duplicate keys:

```
+ it 'prints an error message if there is a duplicate id' do
+   script = [
+     "insert 1 user1 person1@example.com",
+     "insert 1 user1 person1@example.com",
+     "select",
+     ".exit",
+   ]
+   result = run_script(script)
+   expect(result).to match_array([
+     "db > Executed.",
+     "db > Error: Duplicate key.",
+   ])
```

```
+      "db > (1, user1, person1@example.com)",  
+      "Executed.",  
+      "db > ",  
+    ])  
+  end
```

That's it! Next up: implement splitting leaf nodes and creating internal nodes.

[< Part 8 - B-Tree Leaf Node Format](#)

[Part 10 - Splitting a Leaf Node >](#)

[rss](#) | [subscribe by email](#)

This project is maintained by [cstack](#)

Hosted on GitHub Pages — Theme by [orderedlist](#)

Let's Build a Simple Database

Writing a sqlite clone from scratch in C

[Overview](#)

[View on GitHub](#) (pull requests welcome)

This project is no longer under active development. You can read more [here](#). But if you'd like to keep learning how to make your own SQLite clone from scratch, or one of many other projects like Docker, Redis, Git or BitTorrent, try [CodeCrafters](#).



Build Your Own SQLite

Try interactive mode →

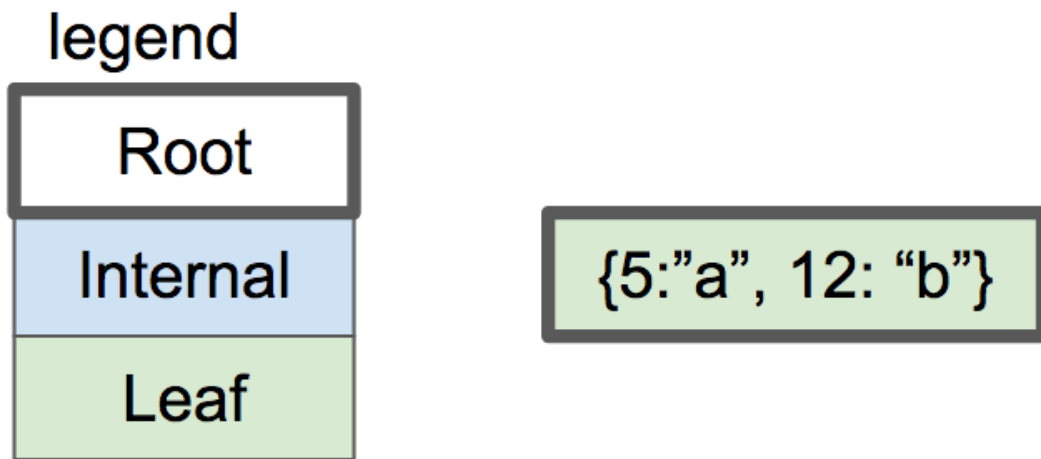
Part 10 - Splitting a Leaf Node

[< Part 9 - Binary Search and Duplicate Keys](#)

[Part 11 - Recursively Searching the B-Tree >](#)

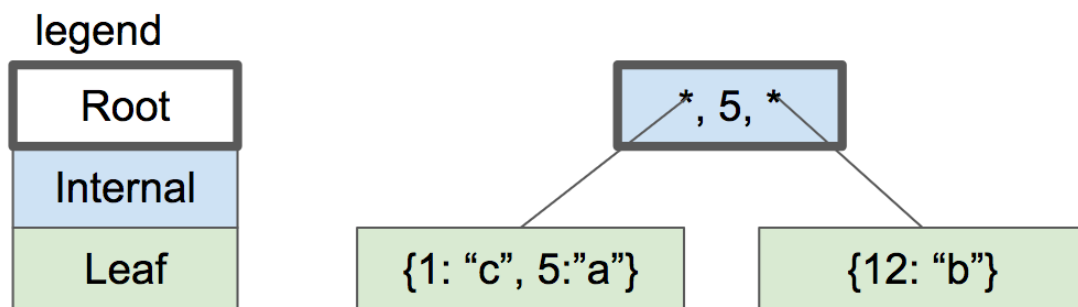
Our B-Tree doesn't feel like much of a tree with only one node. To fix that, we need some code to split a leaf node in twain. And after that, we need to create an internal node to serve as a parent for the two leaf nodes.

Basically our goal for this article is to go from this:



one-node btree

to this:



two-level btree

First things first, let's remove the error handling for a full leaf node:

```
void leaf_node_insert(Cursor* cursor, uint32_t key, Row* value) {
    void* node = get_page(cursor->table->pager, cursor->page_no);

    uint32_t num_cells = *leaf_node_num_cells(node);
    if (num_cells >= LEAF_NODE_MAX_CELLS) {
        // Node full
        - printf("Need to implement splitting a leaf node.\n");
        - exit(EXIT_FAILURE);
        + leaf_node_split_and_insert(cursor, key, value);
        + return;
    }
}
```

```

ExecuteResult execute_insert(Statement* statement, Table* table,
    void* node = get_page(table->pager, table->root_page_num);
    uint32_t num_cells = (*leaf_node_num_cells)(node));
-   if (num_cells >= LEAF_NODE_MAX_CELLS) {
-       return EXECUTE_TABLE_FULL;
-   }

    Row* row_to_insert = &(amp;statement->row_to_insert);
    uint32_t key_to_insert = row_to_insert->id;

```

Splitting Algorithm

Easy part's over. Here's a description of what we need to do from [SQLite Database System: Design and Implementation](#)

If there is no space on the leaf node, we would split the existing entries residing there and the new one (being inserted) into two equal halves: lower and upper halves. (Keys on the upper half are strictly greater than those on the lower half.) We allocate a new leaf node, and move the upper half into the new node.

Let's get a handle to the old node and create the new node:

```

+void leaf_node_split_and_insert(Cursor* cursor, uint32_t key
+   /*
+   Create a new node and move half the cells over.
+   Insert the new value in one of the two nodes.
+   Update parent or create a new parent.
+   */
+
+   void* old_node = get_page(cursor->table->pager, cursor->page_num);
+   uint32_t new_page_num = get_unused_page_num(cursor->table->pager);
+   void* new_node = get_page(cursor->table->pager, new_page_num);
+   initialize_leaf_node(new_node);

```

Next, copy every cell into its new location:

```

+   /*
+   All existing keys plus new key should be divided
+   evenly between old (left) and new (right) nodes.
+   Starting from the right, move each key to correct position
+   */
+   for (int32_t i = LEAF_NODE_MAX_CELLS; i >= 0; i--) {

```

```

+ void* destination_node;
+ if (i >= LEAF_NODE_LEFT_SPLIT_COUNT) {
+     destination_node = new_node;
+ } else {
+     destination_node = old_node;
+ }
+ uint32_t index_within_node = i % LEAF_NODE_LEFT_SPLIT_COUNT;
+ void* destination = leaf_node_cell(destination_node, index_within_node);
+
+ if (i == cursor->cell_num) {
+     serialize_row(value, destination);
+ } else if (i > cursor->cell_num) {
+     memcpy(destination, leaf_node_cell(old_node, i - 1), LEAF_CELL_SIZE);
+ } else {
+     memcpy(destination, leaf_node_cell(old_node, i), LEAF_CELL_SIZE);
+ }
+ }

```

Update cell counts in each node's header:

```

+ /* Update cell count on both leaf nodes */
+ *(leaf_node_num_cells(old_node)) = LEAF_NODE_LEFT_SPLIT_COUNT;
+ *(leaf_node_num_cells(new_node)) = LEAF_NODE_RIGHT_SPLIT_COUNT;

```

Then we need to update the nodes' parent. If the original node was the root, it had no parent. In that case, create a new root node to act as the parent. I'll stub out the other branch for now:

```

+ if (is_node_root(old_node)) {
+     return create_new_root(cursor->table, new_page_num);
+ } else {
+     printf("Need to implement updating parent after split\n");
+     exit(EXIT_FAILURE);
+ }
+ }
+ }

```

Allocating New Pages

Let's go back and define a few new functions and constants. When we created a new leaf node, we put it in a page decided by `get_unused_page_num()`:

```

+/*
+Until we start recycling free pages, new pages will always
+go onto the end of the database file
+*/
+uint32_t get_unused_page_num(Pager* pager) { return pager->n

```

For now, we're assuming that in a database with N pages, page numbers 0 through $N-1$ are allocated. Therefore we can always allocate page number N for new pages. Eventually after we implement deletion, some pages may become empty and their page numbers unused. To be more efficient, we could re-allocate those free pages.

Leaf Node Sizes

To keep the tree balanced, we evenly distribute cells between the two new nodes. If a leaf node can hold N cells, then during a split we need to distribute $N+1$ cells between two nodes (N original cells plus one new one). I'm arbitrarily choosing the left node to get one more cell if $N+1$ is odd.

```

+const uint32_t LEAF_NODE_RIGHT_SPLIT_COUNT = (LEAF_NODE_MAX_
+const uint32_t LEAF_NODE_LEFT_SPLIT_COUNT =
+      (LEAF_NODE_MAX_CELLS + 1) - LEAF_NODE_RIGHT_SPLIT_COUNT;

```

Creating a New Root

Here's how [SQLite Database System](#) explains the process of creating a new root node:

Let N be the root node. First allocate two nodes, say L and R . Move lower half of N into L and the upper half into R . Now N is empty. Add $\langle L, K, R \rangle$ in N , where K is the max key in L . Page N remains the root. Note that the depth of the tree has increased by one, but the new tree remains height balanced without violating any B+-tree property.

At this point, we've already allocated the right child and moved the upper half into it. Our function takes the right child as input and allocates a new page to store the left child.

```

+void create_new_root(Table* table, uint32_t right_child_page
+  /*
+   Handle splitting the root.
+   Old root copied to new page, becomes left child.

```

```

+ Address of right child passed in.
+ Re-initialize root page to contain the new root node.
+ New root node points to two children.
+ */
+
+ void* root = get_page(table->pager, table->root_page_num);
+ void* right_child = get_page(table->pager, right_child_page_num);
+ uint32_t left_child_page_num = get_unused_page_num(table->pager);
+ void* left_child = get_page(table->pager, left_child_page_num);

```

The old root is copied to the left child so we can reuse the root page:

```

+ /* Left child has data copied from old root */
+ memcpy(left_child, root, PAGE_SIZE);
+ set_node_root(left_child, false);

```

Finally we initialize the root page as a new internal node with two children.

```

+ /* Root node is a new internal node with one key and two children */
+ initialize_internal_node(root);
+ set_node_root(root, true);
+ *internal_node_num_keys(root) = 1;
+ *internal_node_child(root, 0) = left_child_page_num;
+ uint32_t left_child_max_key = get_node_max_key(left_child);
+ *internal_node_key(root, 0) = left_child_max_key;
+ *internal_node_right_child(root) = right_child_page_num;
+}

```

Internal Node Format

Now that we're finally creating an internal node, we have to define its layout. It starts with the common header, then the number of keys it contains, then the page number of its rightmost child. Internal nodes always have one more child pointer than they have keys. That extra child pointer is stored in the header.

```

+/*
+ * Internal Node Header Layout
+ */
+const uint32_t INTERNAL_NODE_NUM_KEYS_SIZE = sizeof(uint32_t);
+const uint32_t INTERNAL_NODE_NUM_KEYS_OFFSET = COMMON_NODE_HEADER_SIZE;
+const uint32_t INTERNAL_NODE_RIGHT_CHILD_SIZE = sizeof(uint32_t);

```

```

+const uint32_t INTERNAL_NODE_RIGHT_CHILD_OFFSET =
+    INTERNAL_NODE_NUM_KEYS_OFFSET + INTERNAL_NODE_NUM_KEYS_SIZE;
+const uint32_t INTERNAL_NODE_HEADER_SIZE = COMMON_NODE_HEADER_SIZE
+    + INTERNAL_NODE_NUM_KEYS_SIZE
+    + INTERNAL_NODE_RIGHT_CHILD_OFFSET;

```

The body is an array of cells where each cell contains a child pointer and a key. Every key should be the maximum key contained in the child to its left.

```

+/*
+ * Internal Node Body Layout
+ */
+const uint32_t INTERNAL_NODE_KEY_SIZE = sizeof(uint32_t);
+const uint32_t INTERNAL_NODE_CHILD_SIZE = sizeof(uint32_t);
+const uint32_t INTERNAL_NODE_CELL_SIZE =
+    INTERNAL_NODE_CHILD_SIZE + INTERNAL_NODE_KEY_SIZE;

```

Based on these constants, here's how the layout of an internal node will look:

byte 0	byte 1	bytes 2-5	bytes 6-9
node_type	is_root	parent_pointer	num keys
bytes 6-9		bytes 10-13	bytes 14-17
num keys		right child pointer	child pointer 0
bytes 14-17		bytes 18-21	bytes 22-25
child pointer 0		key 0	child pointer 1
bytes 22-25		bytes 26-29	...
child pointer 1		key 1	...
...		...	bytes 4086-4089
...		...	child pointer 509
bytes 4086-4089		bytes 4090-4093	bytes 4094-4095
child pointer 509		key 509	wasted space

Our internal node format

Notice our huge branching factor. Because each child pointer / key pair is so small, we can fit 510 keys and 511 child pointers in each internal node. That means we'll never have to traverse many layers of the tree to find a given key!

# internal node layers	max # leaf nodes	Size of all leaf nodes
0	$511^0 = 1$	4 KB
1	$511^1 = 512$	~2 MB
2	$511^2 = 261,121$	~1 GB
3	$511^3 = 133,432,831$	~550 GB

In actuality, we can't store a full 4 KB of data per leaf node due to the overhead

of the header, keys, and wasted space. But we can search through something like 500 GB of data by loading only 4 pages from disk. This is why the B-Tree is a useful data structure for databases.

Here are the methods for reading and writing to an internal node:

```
+uint32_t* internal_node_num_keys(void* node) {
+ return node + INTERNAL_NODE_NUM_KEYS_OFFSET;
+}
+
+uint32_t* internal_node_right_child(void* node) {
+ return node + INTERNAL_NODE_RIGHT_CHILD_OFFSET;
+}
+
+uint32_t* internal_node_cell(void* node, uint32_t cell_num) {
+ return node + INTERNAL_NODE_HEADER_SIZE + cell_num * INTERNAL_NODE_CELL_SIZE;
+}
+
+uint32_t* internal_node_child(void* node, uint32_t child_num,
+ uint32_t num_keys = *internal_node_num_keys(node);
+ if (child_num > num_keys) {
+     printf("Tried to access child_num %d > num_keys %d\n", child_num, num_keys);
+     exit(EXIT_FAILURE);
+ } else if (child_num == num_keys) {
+     return internal_node_right_child(node);
+ } else {
+     return internal_node_cell(node, child_num);
+ }
+}
+
+uint32_t* internal_node_key(void* node, uint32_t key_num) {
+ return internal_node_cell(node, key_num) + INTERNAL_NODE_KEY_OFFSET;
+}
```

For an internal node, the maximum key is always its right key. For a leaf node, it's the key at the maximum index:

```
+uint32_t get_node_max_key(void* node) {
+ switch (get_node_type(node)) {
+     case NODE_INTERNAL:
+         return *internal_node_key(node, *internal_node_num_keys(node));
+     case NODE_LEAF:
+         return *internal_node_key(node, *internal_node_num_keys(node) - 1);
+ }
```

```
+         return *leaf_node_key(node, *leaf_node_num_cells(node)
+     }
+ }
```

Keeping Track of the Root

We're finally using the `is_root` field in the common node header. Recall that we use it to decide how to split a leaf node:

```
if (is_node_root(old_node)) {
    return create_new_root(cursor->table, new_page_num);
} else {
    printf("Need to implement updating parent after split\n");
    exit(EXIT_FAILURE);
}
}
```

Here are the getter and setter:

```
+bool is_node_root(void* node) {
+    uint8_t value = *((uint8_t*)(node + IS_ROOT_OFFSET));
+    return (bool)value;
+}
+
+void set_node_root(void* node, bool is_root) {
+    uint8_t value = is_root;
+    *((uint8_t*)(node + IS_ROOT_OFFSET)) = value;
+}
```

Initializing both types of nodes should default to setting `is_root` to false:

```
void initialize_leaf_node(void* node) {
    set_node_type(node, NODE_LEAF);
+    set_node_root(node, false);
    *leaf_node_num_cells(node) = 0;
}

+void initialize_internal_node(void* node) {
+    set_node_type(node, NODE_INTERNAL);
+    set_node_root(node, false);
+    *internal_node_num_keys(node) = 0;
```

```
+}
```

We should set `is_root` to true when creating the first node of the table:

```
    // New database file. Initialize page 0 as leaf node.
    void* root_node = get_page(pager, 0);
    initialize_leaf_node(root_node);
+   set_node_root(root_node, true);
    }

    return table;
```

Printing the Tree

To help us visualize the state of the database, we should update our `.bt` ree metacommand to print a multi-level tree.

I'm going to replace the current `print_leaf_node()` function

```
-void print_leaf_node(void* node) {
-   uint32_t num_cells = *leaf_node_num_cells(node);
-   printf("leaf (size %d)\n", num_cells);
-   for (uint32_t i = 0; i < num_cells; i++) {
-       uint32_t key = *leaf_node_key(node, i);
-       printf("  - %d : %d\n", i, key);
-   }
-}
```

with a new recursive function that takes any node, then prints it and its children. It takes an indentation level as a parameter, which increases with each recursive call. I'm also adding a tiny helper function to indent.

```
+void indent(uint32_t level) {
+   for (uint32_t i = 0; i < level; i++) {
+       printf("  ");
+   }
+}
+
+void print_tree(Pager* pager, uint32_t page_num, uint32_t indent) {
+   void* node = get_page(pager, page_num);
+   uint32_t num_keys, child;
```

```

+
+ switch (get_node_type(node)) {
+   case (NODE_LEAF):
+     num_keys = *leaf_node_num_cells(node);
+     indent(indentation_level);
+     printf("- leaf (size %d)\n", num_keys);
+     for (uint32_t i = 0; i < num_keys; i++) {
+       indent(indentation_level + 1);
+       printf("- %d\n", *leaf_node_key(node, i));
+     }
+     break;
+   case (NODE_INTERNAL):
+     num_keys = *internal_node_num_keys(node);
+     indent(indentation_level);
+     printf("- internal (size %d)\n", num_keys);
+     for (uint32_t i = 0; i < num_keys; i++) {
+       child = *internal_node_child(node, i);
+       print_tree(pager, child, indentation_level + 1);
+
+       indent(indentation_level + 1);
+       printf("- key %d\n", *internal_node_key(node, i));
+     }
+     child = *internal_node_right_child(node);
+     print_tree(pager, child, indentation_level + 1);
+     break;
+ }
+}

```

And update the call to the print function, passing an indentation level of zero.

```

    } else if (strcmp(input_buffer->buffer, ".btree") == 0) {
      printf("Tree:\n");
      - print_leaf_node(get_page(table->pager, 0));
      + print_tree(table->pager, 0, 0);
      return META_COMMAND_SUCCESS;
    }

```

Here's a test case for the new printing functionality!

```

+ it 'allows printing out the structure of a 3-leaf-node btree' do
+   script = (1..14).map do |i|
+     "insert #{i} user#{i} person#{i}@example.com"
+   end

```

```

+   script << ".btree"
+   script << "insert 15 user15 person15@example.com"
+   script << ".exit"
+   result = run_script(script)
+
+   expect(result[14...(result.length)]).to match_array([
+     "db > Tree:",
+     "- internal (size 1)",
+     "  - leaf (size 7)",
+     "    - 1",
+     "    - 2",
+     "    - 3",
+     "    - 4",
+     "    - 5",
+     "    - 6",
+     "    - 7",
+     "  - key 7",
+     "  - leaf (size 7)",
+     "    - 8",
+     "    - 9",
+     "    - 10",
+     "    - 11",
+     "    - 12",
+     "    - 13",
+     "    - 14",
+     "db > Need to implement searching an internal node",
+   ])
+ end

```

The new format is a little simplified, so we need to update the existing .btree test:

```

      "db > Executed.",
      "db > Executed.",
      "db > Tree:",
-     "leaf (size 3)",
-     "  - 0 : 1",
-     "  - 1 : 2",
-     "  - 2 : 3",
+     "- leaf (size 3)",
+     "  - 1",
+     "  - 2",

```

```
+      "    - 3",  
      "db > "  
    ])  
end
```

Here's the .btree output of the new test on its own:

```
Tree:  
- internal (size 1)  
  - leaf (size 7)  
    - 1  
    - 2  
    - 3  
    - 4  
    - 5  
    - 6  
    - 7  
  - key 7  
  - leaf (size 7)  
    - 8  
    - 9  
    - 10  
    - 11  
    - 12  
    - 13  
    - 14
```

On the least indented level, we see the root node (an internal node). It says size 1 because it has one key. Indented one level, we see a leaf node, a key, and another leaf node. The key in the root node (7) is the maximum key in the first leaf node. Every key greater than 7 is in the second leaf node.

A Major Problem

If you've been following along closely you may notice we've missed something big. Look what happens if we try to insert one additional row:

```
db > insert 15 user15 person15@example.com  
Need to implement searching an internal node
```

Whoops! Who wrote that TODO message? :P

Next time we'll continue the epic B-tree saga by implementing search on a multi-level tree.

[< Part 9 - Binary Search and Duplicate Keys](#)

[Part 11 - Recursively Searching the B-Tree >](#)

[rss](#) | [subscribe by email](#)

This project is maintained by [cstack](#)

Hosted on GitHub Pages — Theme by [orderedlist](#)

Let's Build a Simple Database

Writing a sqlite clone from scratch in C

[Overview](#)

[View on GitHub](#) (pull requests welcome)

This project is no longer under active development. You can read more [here](#). But if you'd like to keep learning how to make your own SQLite clone from scratch, or one of many other projects like Docker, Redis, Git or BitTorrent, try [CodeCrafters](#).



Build Your Own SQLite

Try interactive mode →

Part 11 - Recursively Searching the B-Tree

[< Part 10 - Splitting a Leaf Node](#)

[Part 12 - Scanning a Multi-Level B-Tree >](#)

Last time we ended with an error inserting our 15th row:

```
db > insert 15 user15 person15@example.com
Need to implement searching an internal node
```

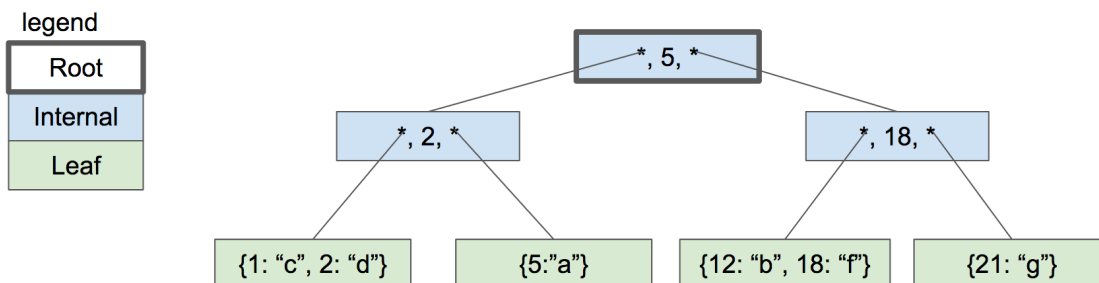
First, replace the code stub with a new function call.

```

    if (get_node_type(root_node) == NODE_LEAF) {
        return leaf_node_find(table, root_page_num, key);
    } else {
-       printf("Need to implement searching an internal node\n");
-       exit(EXIT_FAILURE);
+       return internal_node_find(table, root_page_num, key);
    }
}

```

This function will perform binary search to find the child that should contain the given key. Remember that the key to the right of each child pointer is the maximum key contained by that child.



three-level btree

So our binary search compares the key to find and the key to the right of the child pointer:

```

+Cursor* internal_node_find(Table* table, uint32_t page_num,
+ void* node = get_page(table->pager, page_num);
+ uint32_t num_keys = *internal_node_num_keys(node);
+
+ /* Binary search to find index of child to search */
+ uint32_t min_index = 0;
+ uint32_t max_index = num_keys; /* there is one more child
+
+ while (min_index != max_index) {
+     uint32_t index = (min_index + max_index) / 2;
+     uint32_t key_to_right = *internal_node_key(node, index);
+     if (key_to_right >= key) {
+         max_index = index;
+     } else {
+         min_index = index + 1;
+     }
+ }

```

```
+ }
```

Also remember that the children of an internal node can be either leaf nodes or more internal nodes. After we find the correct child, call the appropriate search function on it:

```
+ uint32_t child_num = *internal_node_child(node, min_index)
+ void* child = get_page(table->pager, child_num);
+ switch (get_node_type(child)) {
+   case NODE_LEAF:
+     return leaf_node_find(table, child_num, key);
+   case NODE_INTERNAL:
+     return internal_node_find(table, child_num, key);
+ }
+}
```

Tests

Now inserting a key into a multi-node btree no longer results in an error. And we can update our test:

```
      "    - 12",
      "    - 13",
      "    - 14",
-     "db > Need to implement searching an internal node",
+     "db > Executed.",
+     "db > ",
    ])
  end
```

I also think it's time we revisit another test. The one that tries inserting 1400 rows. It still errors, but the error message is new. Right now, our tests don't handle it very well when the program crashes. If that happens, let's just use the output we've gotten so far:

```
    raw_output = nil
    IO.popen("./db test.db", "r+") do |pipe|
      commands.each do |command|
-       pipe.puts command
+       begin
+         pipe.puts command
```

```
+         rescue Errno::EPIPE
+           break
+         end
      end

      pipe.close_write
```

And that reveals that our 1400-row test outputs this error:

```
      end
      script << ".exit"
      result = run_script(script)
-     expect(result[-2]).to eq('db > Error: Table full.')
+     expect(result.last(2)).to match_array([
+       "db > Executed.",
+       "db > Need to implement updating parent after split",
+     ])
    end
```

Looks like that's next on our to-do list!

[< Part 10 - Splitting a Leaf Node](#)

[Part 12 - Scanning a Multi-Level B-Tree >](#)

[rss](#) | [subscribe by email](#)

This project is maintained by [cstack](#)

Hosted on GitHub Pages — Theme by [orderedlist](#)

Let's Build a Simple Database

Writing a sqlite clone from scratch in C

[Overview](#)

[View on GitHub](#) (pull requests welcome)

This project is no longer under active development. You can read more [here](#). But if you'd like to keep learning how to make your own SQLite clone from scratch, or one of many other projects like Docker, Redis, Git or BitTorrent, try [CodeCrafters](#).



Build Your Own SQLite

Try interactive mode →

Part 12 - Scanning a Multi-Level B-Tree

[< Part 11 - Recursively Searching the B-Tree](#)

[Part 13 - Updating Parent Node After a Split >](#)

We now support constructing a multi-level btree, but we've broken select statements in the process. Here's a test case that inserts 15 rows and then tries to print them.

```
+ it 'prints all rows in a multi-level tree' do
+   script = []
+   (1..15).each do |i|
```

```

+     script << "insert #{i} user#{i} person#{i}@example.com
+     end
+     script << "select"
+     script << ".exit"
+     result = run_script(script)
+
+     expect(result[15...result.length]).to match_array([
+       "db > (1, user1, person1@example.com)",
+       "(2, user2, person2@example.com)",
+       "(3, user3, person3@example.com)",
+       "(4, user4, person4@example.com)",
+       "(5, user5, person5@example.com)",
+       "(6, user6, person6@example.com)",
+       "(7, user7, person7@example.com)",
+       "(8, user8, person8@example.com)",
+       "(9, user9, person9@example.com)",
+       "(10, user10, person10@example.com)",
+       "(11, user11, person11@example.com)",
+       "(12, user12, person12@example.com)",
+       "(13, user13, person13@example.com)",
+       "(14, user14, person14@example.com)",
+       "(15, user15, person15@example.com)",
+       "Executed.", "db > ",
+     ])
+   end

```

But when we run that test case right now, what actually happens is:

```

db > select
(2, user1, person1@example.com)
Executed.

```

That's weird. It's only printing one row, and that row looks corrupted (notice the id doesn't match the username).

The weirdness is because `execute_select()` begins at the start of the table, and our current implementation of `table_start()` returns cell 0 of the root node. But the root of our tree is now an internal node which doesn't contain any rows. The data that was printed must have been left over from when the root node was a leaf. `execute_select()` should really return cell 0 of the leftmost leaf node.

So get rid of the old implementation:

```
-Cursor* table_start(Table* table) {
-  Cursor* cursor = malloc(sizeof(Cursor));
-  cursor->table = table;
-  cursor->page_num = table->root_page_num;
-  cursor->cell_num = 0;
-
-  void* root_node = get_page(table->pager, table->root_page_num);
-  uint32_t num_cells = *leaf_node_num_cells(root_node);
-  cursor->end_of_table = (num_cells == 0);
-
-  return cursor;
-}
```

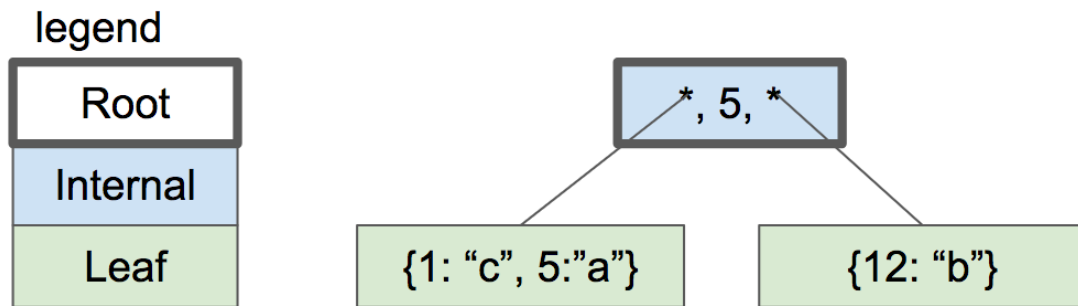
And add a new implementation that searches for key 0 (the minimum possible key). Even if key 0 does not exist in the table, this method will return the position of the lowest id (the start of the left-most leaf node).

```
+Cursor* table_start(Table* table) {
+  Cursor* cursor = table_find(table, 0);
+
+  void* node = get_page(table->pager, cursor->page_num);
+  uint32_t num_cells = *leaf_node_num_cells(node);
+  cursor->end_of_table = (num_cells == 0);
+
+  return cursor;
+}
```

With those changes, it still only prints out one node's worth of rows:

```
db > select
(1, user1, person1@example.com)
(2, user2, person2@example.com)
(3, user3, person3@example.com)
(4, user4, person4@example.com)
(5, user5, person5@example.com)
(6, user6, person6@example.com)
(7, user7, person7@example.com)
Executed.
db >
```

With 15 entries, our btree consists of one internal node and two leaf nodes, which looks something like this:



structure of our btree

To scan the entire table, we need to jump to the second leaf node after we reach the end of the first. To do that, we're going to save a new field in the leaf node header called "next_leaf", which will hold the page number of the leaf's sibling node on the right. The rightmost leaf node will have a next_leaf value of 0 to denote no sibling (page 0 is reserved for the root node of the table anyway).

Update the leaf node header format to include the new field:

```
const uint32_t LEAF_NODE_NUM_CELLS_SIZE = sizeof(uint32_t);
const uint32_t LEAF_NODE_NUM_CELLS_OFFSET = COMMON_NODE_HEADER_SIZE;
const uint32_t LEAF_NODE_HEADER_SIZE =
- COMMON_NODE_HEADER_SIZE + LEAF_NODE_NUM_CELLS_SIZE;
+const uint32_t LEAF_NODE_NEXT_LEAF_SIZE = sizeof(uint32_t);
+const uint32_t LEAF_NODE_NEXT_LEAF_OFFSET =
+ LEAF_NODE_NUM_CELLS_OFFSET + LEAF_NODE_NUM_CELLS_SIZE;
+const uint32_t LEAF_NODE_HEADER_SIZE = COMMON_NODE_HEADER_SIZE
+ LEAF_NODE_NUM_CELLS_SIZE
+ LEAF_NODE_NEXT_LEAF_SIZE;
```

Add a method to access the new field:

```
+uint32_t* leaf_node_next_leaf(void* node) {
+ return node + LEAF_NODE_NEXT_LEAF_OFFSET;
+}
```

Set next_leaf to 0 by default when initializing a new leaf node:

```
@@ -322,6 +330,7 @@ void initialize_leaf_node(void* node) {
    set_node_type(node, NODE_LEAF);
    set_node_root(node, false);
    *leaf_node_num_cells(node) = 0;
+   *leaf_node_next_leaf(node) = 0; // 0 represents no sibling
}
```

Whenever we split a leaf node, update the sibling pointers. The old leaf's sibling becomes the new leaf, and the new leaf's sibling becomes whatever used to be the old leaf's sibling.

```
@@ -659,6 +671,8 @@ void leaf_node_split_and_insert(Cursor* c
    uint32_t new_page_num = get_unused_page_num(cursor->table->
    void* new_node = get_page(cursor->table->pager, new_page_num);
    initialize_leaf_node(new_node);
+   *leaf_node_next_leaf(new_node) = *leaf_node_next_leaf(old_node);
+   *leaf_node_next_leaf(old_node) = new_page_num;
```

Adding a new field changes a few constants:

```
it 'prints constants' do
  script = [
    ".constants",
@@ -199,9 +228,9 @@ describe 'database' do
    "db > Constants:",
    "ROW_SIZE: 293",
    "COMMON_NODE_HEADER_SIZE: 6",
-    "LEAF_NODE_HEADER_SIZE: 10",
+    "LEAF_NODE_HEADER_SIZE: 14",
    "LEAF_NODE_CELL_SIZE: 297",
-    "LEAF_NODE_SPACE_FOR_CELLS: 4086",
+    "LEAF_NODE_SPACE_FOR_CELLS: 4082",
    "LEAF_NODE_MAX_CELLS: 13",
    "db > ",
  ]
end
```

Now whenever we want to advance the cursor past the end of a leaf node, we can check if the leaf node has a sibling. If it does, jump to it. Otherwise, we're at the end of the table.

```
@@ -428,7 +432,15 @@ void cursor_advance(Cursor* cursor) {
```

```

        cursor->cell_num += 1;
        if (cursor->cell_num >= (*leaf_node_num_cells(node))) {
-         cursor->end_of_table = true;
+         /* Advance to next leaf node */
+         uint32_t next_page_num = *leaf_node_next_leaf(node);
+         if (next_page_num == 0) {
+             /* This was rightmost leaf */
+             cursor->end_of_table = true;
+         } else {
+             cursor->page_num = next_page_num;
+             cursor->cell_num = 0;
+         }
        }
    }
}

```

After those changes, we actually print 15 rows...

```

db > select
(1, user1, person1@example.com)
(2, user2, person2@example.com)
(3, user3, person3@example.com)
(4, user4, person4@example.com)
(5, user5, person5@example.com)
(6, user6, person6@example.com)
(7, user7, person7@example.com)
(8, user8, person8@example.com)
(9, user9, person9@example.com)
(10, user10, person10@example.com)
(11, user11, person11@example.com)
(12, user12, person12@example.com)
(13, user13, person13@example.com)
(1919251317, 14, on14@example.com)
(15, user15, person15@example.com)
Executed.
db >

```

...but one of them looks corrupted

```

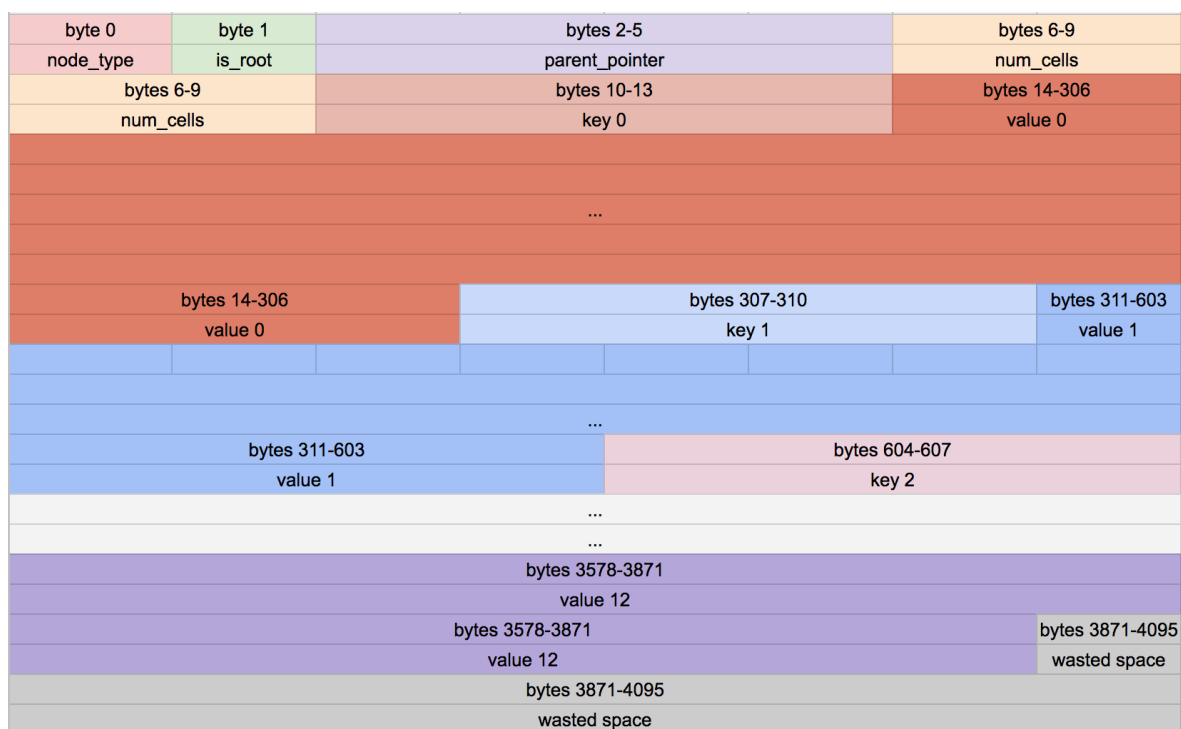
(1919251317, 14, on14@example.com)

```

After some debugging, I found out it's because of a bug in how we split leaf nodes:

```
@@ -676,7 +690,9 @@ void leaf_node_split_and_insert(Cursor* c
    void* destination = leaf_node_cell(destination_node, index_wi
    if (i == cursor->cell_num) {
-       serialize_row(value, destination);
+       serialize_row(value,
+                   leaf_node_value(destination_node, index_wi
+       *leaf_node_key(destination_node, index_within_node) = l
    } else if (i > cursor->cell_num) {
        memcpy(destination, leaf_node_cell(old_node, i - 1), L
    } else {
```

Remember that each cell in a leaf node consists of first a key then a value:



Original leaf node format

We were writing the new row (value) into the start of the cell, where the key should go. That means part of the username was going into the section for id (hence the crazy large id).

After fixing that bug, we finally print out the entire table as expected:

```
db > select
```

```
(1, user1, person1@example.com)
(2, user2, person2@example.com)
(3, user3, person3@example.com)
(4, user4, person4@example.com)
(5, user5, person5@example.com)
(6, user6, person6@example.com)
(7, user7, person7@example.com)
(8, user8, person8@example.com)
(9, user9, person9@example.com)
(10, user10, person10@example.com)
(11, user11, person11@example.com)
(12, user12, person12@example.com)
(13, user13, person13@example.com)
(14, user14, person14@example.com)
(15, user15, person15@example.com)
Executed.
db >
```

Whew! One bug after another, but we're making progress.

Until next time.

[< Part 11 - Recursively Searching the B-Tree](#)

[Part 13 - Updating Parent Node After a Split >](#)

[rss](#) | [subscribe by email](#)

This project is maintained by [cstack](#)

Hosted on GitHub Pages — Theme by [orderedlist](#)

Let's Build a Simple Database

Writing a sqlite clone from scratch in C

[Overview](#)

[View on GitHub \(pull requests welcome\)](#)

This project is no longer under active development. You can read more [here](#). But if you'd like to keep learning how to make your own SQLite clone from scratch, or one of many other projects like Docker, Redis, Git or BitTorrent, try [CodeCrafters](#).



Build Your Own SQLite

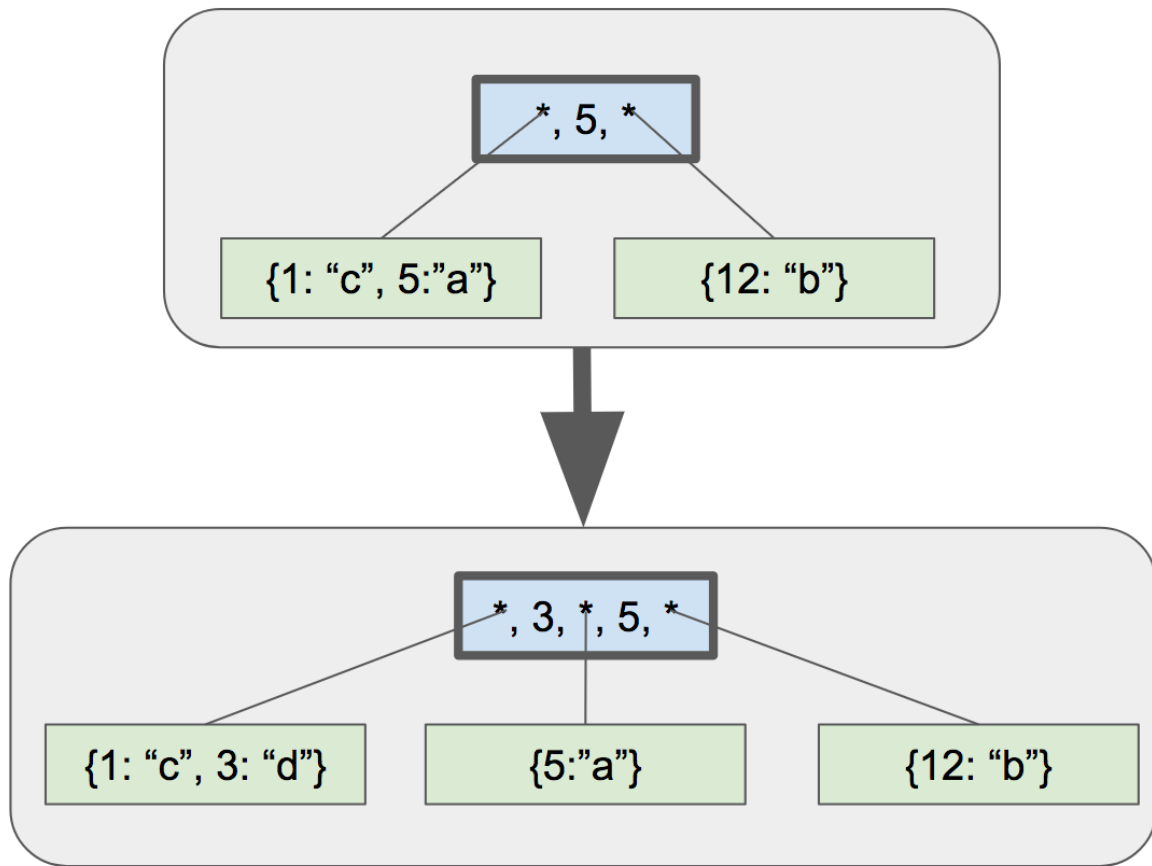
Try interactive mode →

Part 13 - Updating Parent Node After a Split

[< Part 12 - Scanning a Multi-Level B-Tree](#)

[Part 14 - Splitting Internal Nodes >](#)

For the next step on our epic b-tree implementation journey, we're going to handle fixing up the parent node after splitting a leaf. I'm going to use the following example as a reference:



Example of updating internal node

In this example, we add the key "3" to the tree. That causes the left leaf node to split. After the split we fix up the tree by doing the following:

1. Update the first key in the parent to be the maximum key in the left child ("3")
2. Add a new child pointer / key pair after the updated key
 - The new pointer points to the new child node
 - The new key is the maximum key in the new child node ("5")

So first things first, replace our stub code with two new function calls:

`update_internal_node_key()` for step 1 and `internal_node_insert()` for step 2

```
@@ -670,9 +725,11 @@ void leaf_node_split_and_insert(Cursor*
    */

    void* old_node = get_page(cursor->table->pager, cursor->pa
+   uint32_t old_max = get_node_max_key(old_node);
    uint32_t new_page_num = get_unused_page_num(cursor->table->
    void* new_node = get_page(cursor->table->pager, new_page_n
    initialize_leaf_node(new_node);
+   *node_parent(new_node) = *node_parent(old_node);
```

```

*leaf_node_next_leaf(new_node) = *leaf_node_next_leaf(old_node);
*leaf_node_next_leaf(old_node) = new_page_num;

@@ -709,8 +766,12 @@ void leaf_node_split_and_insert(Cursor* cursor, Table* table,
    if (is_node_root(old_node)) {
        return create_new_root(cursor->table, new_page_num);
    } else {
-       printf("Need to implement updating parent after split\n");
-       exit(EXIT_FAILURE);
+       uint32_t parent_page_num = *node_parent(old_node);
+       uint32_t new_max = get_node_max_key(old_node);
+       void* parent = get_page(cursor->table->pager, parent_page_num);
+
+       update_internal_node_key(parent, old_max, new_max);
+       internal_node_insert(cursor->table, parent_page_num, new_max);
+       return;
    }
}

```

In order to get a reference to the parent, we need to start recording in each node a pointer to its parent node.

```

+uint32_t* node_parent(void* node) { return node + PARENT_POINTER; }

```

```

@@ -660,6 +675,48 @@ void create_new_root(Table* table, uint32_t new_page_num) {
    uint32_t left_child_max_key = get_node_max_key(left_child);
    *internal_node_key(root, 0) = left_child_max_key;
    *internal_node_right_child(root) = right_child_page_num;
+   *node_parent(left_child) = table->root_page_num;
+   *node_parent(right_child) = table->root_page_num;
}

```

Now we need to find the affected cell in the parent node. The child doesn't know its own page number, so we can't look for that. But it does know its own maximum key, so we can search the parent for that key.

```

+void update_internal_node_key(void* node, uint32_t old_key, uint32_t new_key) {
+   uint32_t old_child_index = internal_node_find_child(node, old_key);
+   *internal_node_key(node, old_child_index) = new_key;
}

```

Inside `internal_node_find_child()` we'll reuse some code we already have for finding a key in an internal node. Refactor `internal_node_find()` to use the new helper method.

```

-Cursor* internal_node_find(Table* table, uint32_t page_num,
- void* node = get_page(table->pager, page_num);
+uint32_t internal_node_find_child(void* node, uint32_t key)
+ /*
+  Return the index of the child which should contain
+  the given key.
+  */
+
+  uint32_t num_keys = *internal_node_num_keys(node);
+
+  /* Binary search to find index of child to search */
+  /* Binary search */
+  uint32_t min_index = 0;
+  uint32_t max_index = num_keys; /* there is one more child
@@ -386,7 +394,14 @@ Cursor* internal_node_find(Table* table,
    }
}

-  uint32_t child_num = *internal_node_child(node, min_index)
+  return min_index;
+}
+
+Cursor* internal_node_find(Table* table, uint32_t page_num,
+ void* node = get_page(table->pager, page_num);
+
+  uint32_t child_index = internal_node_find_child(node, key)
+  uint32_t child_num = *internal_node_child(node, child_index);
+  void* child = get_page(table->pager, child_num);
+  switch (get_node_type(child)) {
+    case NODE_LEAF:

```

Now we get to the heart of this article, implementing `internal_node.insert()`. I'll explain it in pieces.

[illegible]

```

+ Add a new child/key pair to parent that corresponds to child
+ */
+
+ void* parent = get_page(table->pager, parent_page_num);
+ void* child = get_page(table->pager, child_page_num);
+ uint32_t child_max_key = get_node_max_key(child);
+ uint32_t index = internal_node_find_child(parent, child_max_key);
+
+ uint32_t original_num_keys = *internal_node_num_keys(parent);
+ *internal_node_num_keys(parent) = original_num_keys + 1;
+
+ if (original_num_keys >= INTERNAL_NODE_MAX_CELLS) {
+     printf("Need to implement splitting internal node\n");
+     exit(EXIT_FAILURE);
+ }

```

The index where the new cell (child/key pair) should be inserted depends on the maximum key in the new child. In the example we looked at, `child_max_key` would be 5 and index would be 1.

If there's no room in the internal node for another cell, throw an error. We'll implement that later.

Now let's look at the rest of the function:

```

+
+ uint32_t right_child_page_num = *internal_node_right_child_page_num;
+ void* right_child = get_page(table->pager, right_child_page_num);
+
+ if (child_max_key > get_node_max_key(right_child)) {
+     /* Replace right child */
+     *internal_node_child(parent, original_num_keys) = right_child;
+     *internal_node_key(parent, original_num_keys) =
+         get_node_max_key(right_child);
+     *internal_node_right_child(parent) = child_page_num;
+ } else {
+     /* Make room for the new cell */
+     for (uint32_t i = original_num_keys; i > index; i--) {
+         void* destination = internal_node_cell(parent, i);
+         void* source = internal_node_cell(parent, i - 1);
+         memcpy(destination, source, INTERNAL_NODE_CELL_SIZE);
+     }
+ }

```

```
+    *internal_node_child(parent, index) = child_page_num;
+    *internal_node_key(parent, index) = child_max_key;
+ }
+}
```

Because we store the rightmost child pointer separately from the rest of the child/key pairs, we have to handle things differently if the new child is going to become the rightmost child.

In our example, we would get into the `else` block. First we make room for the new cell by shifting other cells one space to the right. (Although in our example there are 0 cells to shift)

Next, we write the new child pointer and key into the cell determined by `index`.

To reduce the size of testcases needed, I'm hardcoding `INTERNAL_NODE_MAX_CELLS` for now

```
@@ -126,6 +126,8 @@ const uint32_t INTERNAL_NODE_KEY_SIZE = s
    const uint32_t INTERNAL_NODE_CHILD_SIZE = sizeof(uint32_t);
    const uint32_t INTERNAL_NODE_CELL_SIZE =
        INTERNAL_NODE_CHILD_SIZE + INTERNAL_NODE_KEY_SIZE;
+/* Keep this small for testing */
+const uint32_t INTERNAL_NODE_MAX_CELLS = 3;
```

Speaking of tests, our large-dataset test gets past our old stub and gets to our new one:

```
@@ -65,7 +65,7 @@ describe 'database' do
    result = run_script(script)
    expect(result.last(2)).to match_array([
        "db > Executed.",
-        "db > Need to implement updating parent after split",
+        "db > Need to implement splitting internal node",
    ])
}
```

Very satisfying, I know.

I'll add another test that prints a four-node tree. Just so we test more cases than sequential ids, this test will add records in a pseudorandom order.

```
+ it 'allows printing out the structure of a 4-leaf-node btree'
```

```

+   script = [
+       "insert 18 user18 person18@example.com",
+       "insert 7 user7 person7@example.com",
+       "insert 10 user10 person10@example.com",
+       "insert 29 user29 person29@example.com",
+       "insert 23 user23 person23@example.com",
+       "insert 4 user4 person4@example.com",
+       "insert 14 user14 person14@example.com",
+       "insert 30 user30 person30@example.com",
+       "insert 15 user15 person15@example.com",
+       "insert 26 user26 person26@example.com",
+       "insert 22 user22 person22@example.com",
+       "insert 19 user19 person19@example.com",
+       "insert 2 user2 person2@example.com",
+       "insert 1 user1 person1@example.com",
+       "insert 21 user21 person21@example.com",
+       "insert 11 user11 person11@example.com",
+       "insert 6 user6 person6@example.com",
+       "insert 20 user20 person20@example.com",
+       "insert 5 user5 person5@example.com",
+       "insert 8 user8 person8@example.com",
+       "insert 9 user9 person9@example.com",
+       "insert 3 user3 person3@example.com",
+       "insert 12 user12 person12@example.com",
+       "insert 27 user27 person27@example.com",
+       "insert 17 user17 person17@example.com",
+       "insert 16 user16 person16@example.com",
+       "insert 13 user13 person13@example.com",
+       "insert 24 user24 person24@example.com",
+       "insert 25 user25 person25@example.com",
+       "insert 28 user28 person28@example.com",
+       ".btree",
+       ".exit",
+   ]
+   result = run_script(script)

```

As-is, it will output this:

```

- internal (size 3)
- leaf (size 7)
- 1
- 2

```

- 3
- 4
- 5
- 6
- 7
- key 1
- leaf (size 8)
 - 8
 - 9
 - 10
 - 11
 - 12
 - 13
 - 14
 - 15
- key 15
- leaf (size 7)
 - 16
 - 17
 - 18
 - 19
 - 20
 - 21
 - 22
- key 22
- leaf (size 8)
 - 23
 - 24
 - 25
 - 26
 - 27
 - 28
 - 29
 - 30

db >

Look carefully and you'll spot a bug:

- 5
- 6
- 7
- key 1

The key there should be 7, not 1!

After a bunch of debugging, I discovered this was due to some bad pointer arithmetic.

```
uint32_t* internal_node_key(void* node, uint32_t key_num) {  
-   return internal_node_cell(node, key_num) + INTERNAL_NODE_C  
+   return (void*)internal_node_cell(node, key_num) + INTERNAL_  
}
```

INTERNAL_NODE_CHILD_SIZE is 4. My intention here was to add 4 bytes to the result of `internal_node_cell()`, but since `internal_node_cell()` returns a `uint32_t*`, this it was actually adding `4 * sizeof(uint32_t)` bytes. I fixed it by casting to a `void*` before doing the arithmetic.

NOTE! [Pointer arithmetic on void pointers is not part of the C standard and may not work with your compiler](#). I may do an article in the future on portability, but I'm leaving my void pointer arithmetic for now.

Alright. One more step toward a fully-operational btree implementation. The next step should be splitting internal nodes. Until then!

[< Part 12 - Scanning a Multi-Level B-Tree](#)

[Part 14 - Splitting Internal Nodes >](#)

[rss](#) | [subscribe by email](#)

This project is maintained by [cstack](#)

Hosted on GitHub Pages — Theme by [orderedlist](#)

Let's Build a Simple Database

Writing a sqlite clone from scratch in C

[Overview](#)

[View on GitHub](#) (pull requests welcome)

This project is no longer under active development. You can read more [here](#). But if you'd like to keep learning how to make your own SQLite clone from scratch, or one of many other projects like Docker, Redis, Git or BitTorrent, try [CodeCrafters](#).



Build Your Own SQLite

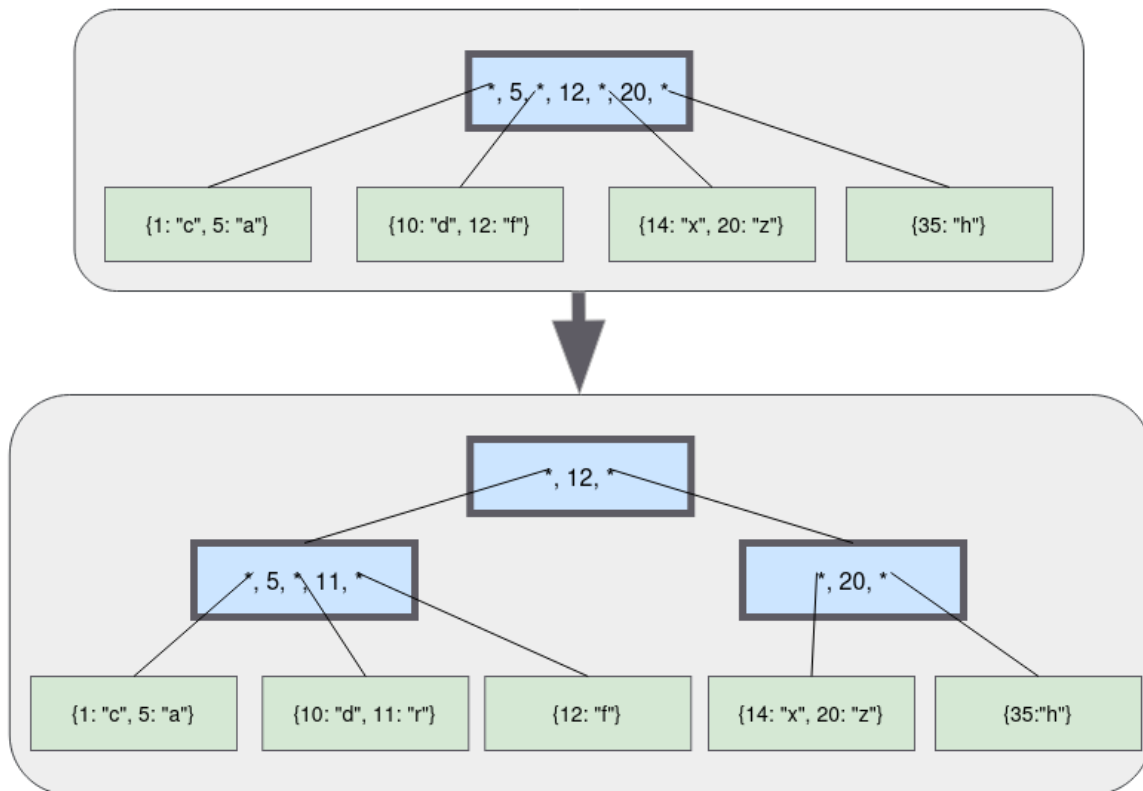
Try interactive mode →

Part 14 - Splitting Internal Nodes

[< Part 13 - Updating Parent Node After a Split](#)

[Part 15 - Where to go next >](#)

The next leg of our journey will be splitting internal nodes which are unable to accommodate new keys. Consider the example below:



Example of splitting an internal

In this example, we add the key "11" to the tree. This will cause our root to split. When splitting an internal node, we will have to do a few things in order to keep everything straight:

1. Create a sibling node to store $(n-1)/2$ of the original node's keys
2. Move these keys from the original node to the sibling node
3. Update the original node's key in the parent to reflect its new max key after splitting
4. Insert the sibling node into the parent (could result in the parent also being split)

We will begin by replacing our stub code with the call to `internal_node_split_and_insert`

```
+void internal_node_split_and_insert(Table* table, uint32_t p
+
+                               uint32_t child_page_num);
+
+void internal_node_insert(Table* table, uint32_t parent_page
+                               uint32_t child_page_num) {
+
+    /*
@@ -685,25 +714,39 @@ void internal_node_insert(Table* table,
+
+    void* parent = get_page(table->pager, parent_page_num);
```

```

void* child = get_page(table->pager, child_page_num);
- uint32_t child_max_key = get_node_max_key(child);
+ uint32_t child_max_key = get_node_max_key(table->pager, child_page_num);
uint32_t index = internal_node_find_child(parent, child_max_key);

uint32_t original_num_keys = *internal_node_num_keys(parent);
- *internal_node_num_keys(parent) = original_num_keys + 1;

if (original_num_keys >= INTERNAL_NODE_MAX_CELLS) {
-   printf("Need to implement splitting internal node\n");
-   exit(EXIT_FAILURE);
+   internal_node_split_and_insert(table, parent_page_num, child_page_num, child_max_key);
+   return;
}

uint32_t right_child_page_num = *internal_node_right_child(parent);
+ /*
+  * An internal node with a right child of INVALID_PAGE_NUM is
+  * invalid.
+  */
+ if (right_child_page_num == INVALID_PAGE_NUM) {
+     *internal_node_right_child(parent) = child_page_num;
+     return;
+ }
+
void* right_child = get_page(table->pager, right_child_page_num);
+ /*
+  * If we are already at the max number of cells for a node, we
+  * cannot insert a new key before splitting. Incrementing without inserting a new key
+  * and immediately calling internal_node_split_and_insert has
+  * the effect of creating a new key at (max_cells + 1) with an uninitialized value.
+  */
+ *internal_node_num_keys(parent) = original_num_keys + 1;

- if (child_max_key > get_node_max_key(right_child)) {
+ if (child_max_key > get_node_max_key(table->pager, right_child_page_num)) {
+     /* Replace right child */
+     *internal_node_child(parent, original_num_keys) = right_child;
+     *internal_node_key(parent, original_num_keys) =
-         get_node_max_key(right_child);
+         get_node_max_key(table->pager, right_child_page_num);
+     *internal_node_right_child(parent) = child_page_num;

```

There are three important changes we are making here aside from replacing the stub:

- First, `internal_node_split_and_insert` is forward-declared because we will be calling `internal_node_insert` in its definition to avoid code duplication.
- In addition, we are moving the logic which increments the parent's number of keys further down in the function definition to ensure that this does not happen before the split.
- Finally, we are ensuring that a child node inserted into an empty internal node will become that internal node's right child without any other operations being performed, since an empty internal node has no keys to manipulate.

The changes above require that we be able to identify an empty node - to this end, we will first define a constant which represents an invalid page number that is the child of every empty node.

```
#define INVALID_PAGE_NUM UINT32_MAX
```

Now, when an internal node is initialized, we initialize its right child with this invalid page number.

```
@@ -330,6 +335,12 @@ void initialize_internal_node(void* node)
    set_node_type(node, NODE_INTERNAL);
    set_node_root(node, false);
    *internal_node_num_keys(node) = 0;
+   /*
+    * Necessary because the root page number is 0; by not initializing
+    * node's right child to an invalid page number when initializing,
+    * we end up with 0 as the node's right child, which makes the node
+    * the root.
+    */
+   *internal_node_right_child(node) = INVALID_PAGE_NUM;
}
```

This step was made necessary by a problem that the comment above attempts to summarize - when initializing an internal node without explicitly initializing the right child field, the value of that field at runtime could be 0 depending on the compiler or the architecture of the machine on which the program is being executed. Since we are using 0 as our root page number, this means that a newly allocated internal node will be a parent of the root.

We have introduced some guards in our `internal_node_child` function to throw an error in the case of an attempt to access an invalid page.

```
@@ -186,9 +188,19 @@ uint32_t* internal_node_child(void* node
    printf("Tried to access child_num %d > num_keys %d\n", child_num, num_keys);
    exit(EXIT_FAILURE);
} else if (child_num == num_keys) {
-    return internal_node_right_child(node);
+    uint32_t* right_child = internal_node_right_child(node);
+    if (*right_child == INVALID_PAGE_NUM) {
+        printf("Tried to access right child of node, but was invalid page\n");
+        exit(EXIT_FAILURE);
+    }
+    return right_child;
} else {
-    return internal_node_cell(node, child_num);
+    uint32_t* child = internal_node_cell(node, child_num);
+    if (*child == INVALID_PAGE_NUM) {
+        printf("Tried to access child %d of node, but was invalid page\n");
+        exit(EXIT_FAILURE);
+    }
+    return child;
}
}
```

One additional guard is needed in our `print_tree` function to ensure that we do not attempt to print an empty node, as that would involve trying to access an invalid page.

```
@@ -294,15 +305,17 @@ void print_tree(Pager* pager, uint32_t num_keys,
    num_keys = *internal_node_num_keys(node);
    indent(indentation_level);
    printf("- internal (size %d)\n", num_keys);
-    for (uint32_t i = 0; i < num_keys; i++) {
-        child = *internal_node_child(node, i);
+    if (num_keys > 0) {
+        for (uint32_t i = 0; i < num_keys; i++) {
+            child = *internal_node_child(node, i);
+            print_tree(pager, child, indentation_level + 1);
+
+            indent(indentation_level + 1);
+            printf("- key %d\n", *internal_node_key(node, i));
+        }
+    }
}
```

```

+         }
+         child = *internal_node_right_child(node);
+         print_tree(pager, child, indentation_level + 1);
-
-         indent(indentation_level + 1);
-         printf("- key %d\n", *internal_node_key(node, i));
+     }
-     child = *internal_node_right_child(node);
-     print_tree(pager, child, indentation_level + 1);
+     break;
+ }
+ }

```

Now for the headliner, `internal_node_split_and_insert`. We will first provide it in its entirety, and then break it down by steps.

```

+void internal_node_split_and_insert(Table* table, uint32_t p
+         uint32_t child_page_num) {
+     uint32_t old_page_num = parent_page_num;
+     void* old_node = get_page(table->pager, parent_page_num);
+     uint32_t old_max = get_node_max_key(table->pager, old_node
+
+     void* child = get_page(table->pager, child_page_num);
+     uint32_t child_max = get_node_max_key(table->pager, child)
+
+     uint32_t new_page_num = get_unused_page_num(table->pager);
+
+     /*
+     Declaring a flag before updating pointers which
+     records whether this operation involves splitting the root
+     if it does, we will insert our newly created node during
+     the step where the table's new root is created. If it does
+     not, we have to insert the newly created node into its par
+     after the old node's keys have been transferred over. We a
+     able to do this if the newly created node's parent is not i
+     initialized root node, because in that case its parent may
+     keys aside from our old node which we are splitting. If th
+     need to find a place for our newly created node in its par
+     cannot insert it at the correct index if it does not yet h
+     */
+     uint32_t splitting_root = is_node_root(old_node);
+
+

```

```

+ void* parent;
+ void* new_node;
+ if (splitting_root) {
+     create_new_root(table, new_page_num);
+     parent = get_page(table->pager, table->root_page_num);
+     /*
+      * If we are splitting the root, we need to update old_node
+      * to the new root's left child, new_page_num will already be
+      * the new root's right child
+      */
+     old_page_num = *internal_node_child(parent, 0);
+     old_node = get_page(table->pager, old_page_num);
+ } else {
+     parent = get_page(table->pager, *node_parent(old_node));
+     new_node = get_page(table->pager, new_page_num);
+     initialize_internal_node(new_node);
+ }
+
+ uint32_t* old_num_keys = internal_node_num_keys(old_node);
+
+ uint32_t cur_page_num = *internal_node_right_child(old_node);
+ void* cur = get_page(table->pager, cur_page_num);
+
+ /*
+  * First put right child into new node and set right child of
+  * new node to cur_page_num
+  */
+ internal_node_insert(table, new_page_num, cur_page_num);
+ *node_parent(cur) = new_page_num;
+ *internal_node_right_child(old_node) = INVALID_PAGE_NUM;
+ /*
+  * For each key until you get to the middle key, move the key
+  * to the new node
+  */
+ for (int i = INTERNAL_NODE_MAX_CELLS - 1; i > INTERNAL_NODE_MAX_CELLS / 2; i--) {
+     cur_page_num = *internal_node_child(old_node, i);
+     cur = get_page(table->pager, cur_page_num);
+
+     internal_node_insert(table, new_page_num, cur_page_num);
+     *node_parent(cur) = new_page_num;
+
+     (*old_num_keys)--;
+ }
+
+ }

```

```

+  /*
+   Set child before middle key, which is now the highest key,
+   and decrement number of keys
+  */
+  *internal_node_right_child(old_node) = *internal_node_child
+  (*old_num_keys)--;
+
+  /*
+   Determine which of the two nodes after the split should co
+   and insert the child
+  */
+  uint32_t max_after_split = get_node_max_key(table->pager,
+
+
+  uint32_t destination_page_num = child_max < max_after_spli
+
+  internal_node_insert(table, destination_page_num, child_pa
+  *node_parent(child) = destination_page_num;
+
+  update_internal_node_key(parent, old_max, get_node_max_key
+
+  if (!splitting_root) {
+      internal_node_insert(table, *node_parent(old_node), new_pa
+      *node_parent(new_node) = *node_parent(old_node);
+  }
+}
+
+

```

The first thing we need to do is create a variable to store the page number of the node we are splitting (the old node from here out). This is necessary because the page number of the old node will change if it happens to be the table's root node. We also need to remember what the node's current max is, because that value represents its key in the parent, and that key will need to be updated with the old node's new maximum after the split occurs.

```

+  uint32_t old_page_num = parent_page_num;
+  void* old_node = get_page(table->pager, parent_page_num);
+  uint32_t old_max = get_node_max_key(table->pager, old_node

```

The next important step is the branching logic which depends on whether the old node is the table's root node. We will need to keep track of this value for later use; as the comment attempts to convey, we run into a problem if we do not store this information at the beginning of our function definition - if we are

not splitting the root, we cannot insert our newly created sibling node into the old node's parent right away, because it does not yet contain any keys and therefore will not be placed at the right index among the other key/child pairs which may or may not already be present in the parent node.

```
+ uint32_t splitting_root = is_node_root(old_node);
+
+ void* parent;
+ void* new_node;
+ if (splitting_root) {
+     create_new_root(table, new_page_num);
+     parent = get_page(table->pager, table->root_page_num);
+     /*
+     If we are splitting the root, we need to update old_node
+     to the new root's left child, new_page_num will already be
+     the new root's right child
+     */
+     old_page_num = *internal_node_child(parent, 0);
+     old_node = get_page(table->pager, old_page_num);
+ } else {
+     parent = get_page(table->pager, *node_parent(old_node));
+     new_node = get_page(table->pager, new_page_num);
+     initialize_internal_node(new_node);
+ }
```

Once we have settled the question of splitting or not splitting the root, we begin moving keys from the old node to its sibling. We must first move the old node's right child and set its right child field to an invalid page to indicate that it is empty. Now, we loop over the old node's remaining keys, performing the following steps on each iteration:

1. Obtain a reference to the old node's key and child at the current index
2. Insert the child into the sibling node
3. Update the child's parent value to point to the sibling node
4. Decrement the old node's number of keys

```
+ uint32_t* old_num_keys = internal_node_num_keys(old_node);
+
+ uint32_t cur_page_num = *internal_node_right_child(old_node);
+ void* cur = get_page(table->pager, cur_page_num);
+
+ /*
```

```

+ First put right child into new node and set right child of
+ */
+ internal_node_insert(table, new_page_num, cur_page_num);
+ *node_parent(cur) = new_page_num;
+ *internal_node_right_child(old_node) = INVALID_PAGE_NUM;
+ /*
+ For each key until you get to the middle key, move the key
+ */
+ for (int i = INTERNAL_NODE_MAX_CELLS - 1; i > INTERNAL_NODE_MAX_CELLS / 2; i--)
+     cur_page_num = *internal_node_child(old_node, i);
+     cur = get_page(table->pager, cur_page_num);
+
+     internal_node_insert(table, new_page_num, cur_page_num);
+     *node_parent(cur) = new_page_num;
+
+     (*old_num_keys)--;
+ }

```

Step 4 is important, because it serves the purpose of “erasing” the key/child pair from the old node. Although we are not actually freeing the memory at that byte offset in the old node’s page, by decrementing the old node’s number of keys we are making that memory location inaccessible, and the bytes will be overwritten the next time a child is inserted into the old node.

Also note the behavior of our loop invariant - if our maximum number of internal node keys changes in the future, our logic ensures that both our old node and our sibling node will end up with $(n-1)/2$ keys after the split, with the 1 remaining node going to the parent. If an even number is chosen as the maximum number of nodes, $n/2$ nodes will remain with the old node while $(n-1)/2$ will be moved to the sibling node. This logic would be straightforward to revise as needed.

Once the keys to be moved have been, we set the old node’s i ’th child as its right child and decrement its number of keys.

```

+ /*
+ Set child before middle key, which is now the highest key,
+ and decrement number of keys
+ */
+ *internal_node_right_child(old_node) = *internal_node_child(old_node, i);
+ (*old_num_keys)--;

```

We then insert the child node into either the old node or the sibling node depending on the value of its max key.

```
+ uint32_t max_after_split = get_node_max_key(table->pager, old_node);
+
+ uint32_t destination_page_num = child_max < max_after_split ? old_page_num : sibling_page_num;
+
+ internal_node_insert(table, destination_page_num, child_page_num);
+ *node_parent(child) = destination_page_num;
```

Finally, we update the old node's key in its parent, and insert the sibling node and update the sibling node's parent pointer if necessary.

```
+ update_internal_node_key(parent, old_max, get_node_max_key(old_node));
+
+ if (!splitting_root) {
+     internal_node_insert(table, *node_parent(old_node), new_page_num);
+     *node_parent(new_node) = *node_parent(old_node);
+ }
```

One important change required to support this new logic is in our `create_new_root` function. Before, we were only taking into account situations where the new root's children would be leaf nodes. If the new root's children are instead internal nodes, we need to do two things:

1. Correctly initialize the root's new children to be internal nodes
2. In addition to the call to `memcpy`, we need to insert each of the root's keys into its new left child and update the parent pointer of each of those children

```
@@ -661,22 +680,40 @@ void create_new_root(Table* table, uint32_t root_page_num,
    uint32_t left_child_page_num = get_unused_page_num(table->pager);
    void* left_child = get_page(table->pager, left_child_page_num);

+ if (get_node_type(root) == NODE_INTERNAL) {
+     initialize_internal_node(right_child);
+     initialize_internal_node(left_child);
+ }
+
+ /* Left child has data copied from old root */
+ memcpy(left_child, root, PAGE_SIZE);
```

```

    set_node_root(left_child, false);

+   if (get_node_type(left_child) == NODE_INTERNAL) {
+       void* child;
+       for (int i = 0; i < *internal_node_num_keys(left_child);
+           child = get_page(table->pager, *internal_node_child(le
+               *node_parent(child) = left_child_page_num;
+       }
+       child = get_page(table->pager, *internal_node_right_chil
+       *node_parent(child) = left_child_page_num;
+   }
+
    /* Root node is a new internal node with one key and two ch
    initialize_internal_node(root);
    set_node_root(root, true);
    *internal_node_num_keys(root) = 1;
    *internal_node_child(root, 0) = left_child_page_num;
-   uint32_t left_child_max_key = get_node_max_key(left_child)
+   uint32_t left_child_max_key = get_node_max_key(table->page
    *internal_node_key(root, 0) = left_child_max_key;
    *internal_node_right_child(root) = right_child_page_num;
    *node_parent(left_child) = table->root_page_num;
    *node_parent(right_child) = table->root_page_num;
}

```

Another important change has been made to `get_node_max_key`, as mentioned at the beginning of this article. Since an internal node's key represents the maximum of the tree pointed to by the child to its left, and that child can be a tree of arbitrary depth, we need to walk down the right children of that tree until we get to a leaf node, and then take the maximum key of that leaf node.

```

+uint32_t get_node_max_key(Pager* pager, void* node) {
+   if (get_node_type(node) == NODE_LEAF) {
+       return *leaf_node_key(node, *leaf_node_num_cells(node) -
+   }
+   void* right_child = get_page(pager, *internal_node_right_ch
+   return get_node_max_key(pager, right_child);
+}

```

We have written a single test to demonstrate that our `print_tree` function still works after the introduction of internal node splitting.

```
+ it 'allows printing out the structure of a 7-leaf-node btree'
+   script = [
+     "insert 58 user58 person58@example.com",
+     "insert 56 user56 person56@example.com",
+     "insert 8 user8 person8@example.com",
+     "insert 54 user54 person54@example.com",
+     "insert 77 user77 person77@example.com",
+     "insert 7 user7 person7@example.com",
+     "insert 25 user25 person25@example.com",
+     "insert 71 user71 person71@example.com",
+     "insert 13 user13 person13@example.com",
+     "insert 22 user22 person22@example.com",
+     "insert 53 user53 person53@example.com",
+     "insert 51 user51 person51@example.com",
+     "insert 59 user59 person59@example.com",
+     "insert 32 user32 person32@example.com",
+     "insert 36 user36 person36@example.com",
+     "insert 79 user79 person79@example.com",
+     "insert 10 user10 person10@example.com",
+     "insert 33 user33 person33@example.com",
+     "insert 20 user20 person20@example.com",
+     "insert 4 user4 person4@example.com",
+     "insert 35 user35 person35@example.com",
+     "insert 76 user76 person76@example.com",
+     "insert 49 user49 person49@example.com",
+     "insert 24 user24 person24@example.com",
+     "insert 70 user70 person70@example.com",
+     "insert 48 user48 person48@example.com",
+     "insert 39 user39 person39@example.com",
+     "insert 15 user15 person15@example.com",
+     "insert 47 user47 person47@example.com",
+     "insert 30 user30 person30@example.com",
+     "insert 86 user86 person86@example.com",
+     "insert 31 user31 person31@example.com",
+     "insert 68 user68 person68@example.com",
+     "insert 37 user37 person37@example.com",
+     "insert 66 user66 person66@example.com",
+     "insert 63 user63 person63@example.com",
+     "insert 40 user40 person40@example.com",
+     "insert 78 user78 person78@example.com",
+     "insert 19 user19 person19@example.com",
+     "insert 46 user46 person46@example.com",
```

```

+     "insert 14 user14 person14@example.com",
+     "insert 81 user81 person81@example.com",
+     "insert 72 user72 person72@example.com",
+     "insert 6 user6 person6@example.com",
+     "insert 50 user50 person50@example.com",
+     "insert 85 user85 person85@example.com",
+     "insert 67 user67 person67@example.com",
+     "insert 2 user2 person2@example.com",
+     "insert 55 user55 person55@example.com",
+     "insert 69 user69 person69@example.com",
+     "insert 5 user5 person5@example.com",
+     "insert 65 user65 person65@example.com",
+     "insert 52 user52 person52@example.com",
+     "insert 1 user1 person1@example.com",
+     "insert 29 user29 person29@example.com",
+     "insert 9 user9 person9@example.com",
+     "insert 43 user43 person43@example.com",
+     "insert 75 user75 person75@example.com",
+     "insert 21 user21 person21@example.com",
+     "insert 82 user82 person82@example.com",
+     "insert 12 user12 person12@example.com",
+     "insert 18 user18 person18@example.com",
+     "insert 60 user60 person60@example.com",
+     "insert 44 user44 person44@example.com",
+     ".btree",
+     ".exit",
+ ]
+ result = run_script(script)
+
+ expect(result[64...(result.length)]).to match_array([
+     "db > Tree:",
+     "- internal (size 1)",
+     "  - internal (size 2)",
+     "    - leaf (size 7)",
+     "      - 1",
+     "      - 2",
+     "      - 4",
+     "      - 5",
+     "      - 6",
+     "      - 7",
+     "      - 8",
+     "    - key 8",

```

```
+      "      - leaf (size 11)",
+      "      - 9",
+      "      - 10",
+      "      - 12",
+      "      - 13",
+      "      - 14",
+      "      - 15",
+      "      - 18",
+      "      - 19",
+      "      - 20",
+      "      - 21",
+      "      - 22",
+      "      - key 22",
+      "      - leaf (size 8)",
+      "      - 24",
+      "      - 25",
+      "      - 29",
+      "      - 30",
+      "      - 31",
+      "      - 32",
+      "      - 33",
+      "      - 35",
+      "      - key 35",
+      "      - internal (size 3)",
+      "      - leaf (size 12)",
+      "      - 36",
+      "      - 37",
+      "      - 39",
+      "      - 40",
+      "      - 43",
+      "      - 44",
+      "      - 46",
+      "      - 47",
+      "      - 48",
+      "      - 49",
+      "      - 50",
+      "      - 51",
+      "      - key 51",
+      "      - leaf (size 11)",
+      "      - 52",
+      "      - 53",
+      "      - 54",
```

```
+      "      - 55",
+      "      - 56",
+      "      - 58",
+      "      - 59",
+      "      - 60",
+      "      - 63",
+      "      - 65",
+      "      - 66",
+      "      - key 66",
+      "      - leaf (size 7)",
+      "      - 67",
+      "      - 68",
+      "      - 69",
+      "      - 70",
+      "      - 71",
+      "      - 72",
+      "      - 75",
+      "      - key 75",
+      "      - leaf (size 8)",
+      "      - 76",
+      "      - 77",
+      "      - 78",
+      "      - 79",
+      "      - 81",
+      "      - 82",
+      "      - 85",
+      "      - 86",
+      "db > ",
+    ])
+  end
```

[< Part 13 - Updating Parent Node After a Split](#)

[Part 15 - Where to go next >](#)

[rss](#) | [subscribe by email](#)

This project is maintained by [cstack](#)

Hosted on GitHub Pages — Theme by [orderedlist](#)

Let's Build a Simple Database

Writing a sqlite clone from scratch in C

[Overview](#)

[View on GitHub](#) (pull requests welcome)

This project is no longer under active development. You can read more [here](#). But if you'd like to keep learning how to make your own SQLite clone from scratch, or one of many other projects like Docker, Redis, Git or BitTorrent, try [CodeCrafters](#).



Build Your Own SQLite

Try interactive mode →

Part 15 - Where to go next

[< Part 14 - Splitting Internal Nodes](#)

This project is no longer under active development.

But if you'd like to keep learning how to make your own SQLite clone from scratch, or one of many other projects like Docker, Redis, Git or BitTorrent, try [CodeCrafters](#).

CodeCrafters maintains a [pretty comprehensive list of "Build your own X" tutorials](#) including "Build your own Database".

Plus, if your company has a learning and development budget, you can use it to

pay for CodeCrafter's paid service:



Build Your Own SQLite

Try interactive mode →

If you use my referral link, I get a commision.

[< Part 14 - Splitting Internal Nodes](#)

[rss](#) | [subscribe by email](#)

This project is maintained by [cstack](#)

Hosted on GitHub Pages — Theme by [orderedlist](#)