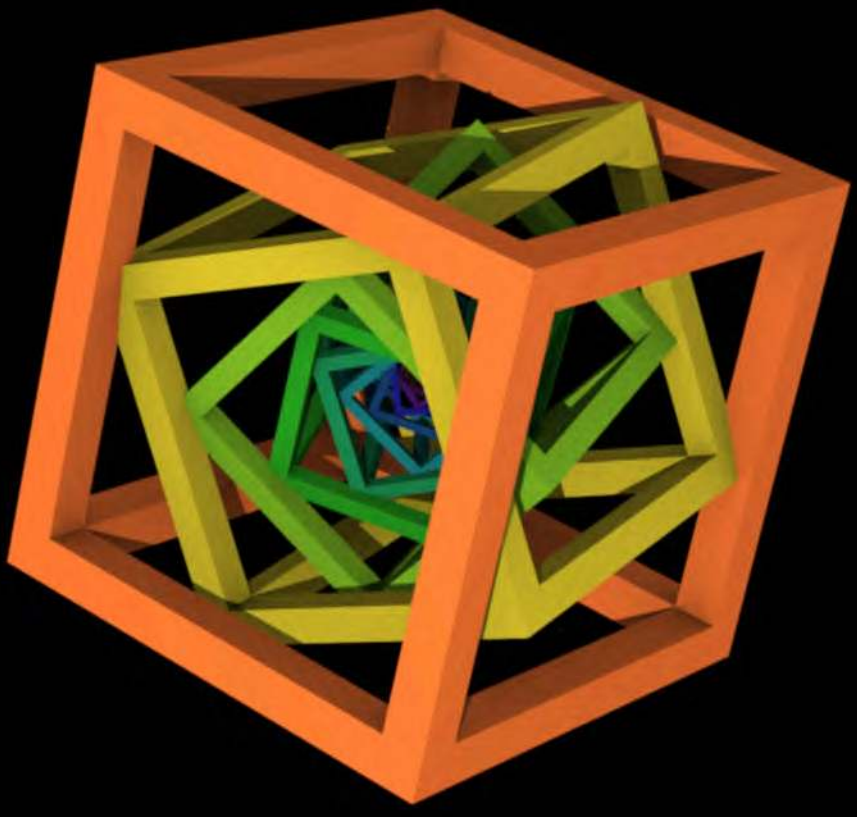
Build Your Own Redis with C/C++

Learn network programming and data structures
by coding from scratch



"What I cannot create, I do not understand"

https://build-your-own.org/

Build Your Own Redis with C/C++

Learn network programming and data structures

James Smith

build-your-own.org

2023-01-31

Contents

Part 1. Getting Started

Make a functioning server that responds to commands.

01. Introduction

What Is This Book About?

This book contains a step-by-step walkthrough of a simple implementation of a Redis-like server. It is intended as a practical guide or tutorial to network programming and the implementation and application of basic data structures in C.

What to Learn From This Book?

Redis could be considered one of the *building blocks* of modern computing that stood the test of time. The knowledge required for building such a project is broader and deeper than usual application-level development. Learning from such projects is a good way for software developers to level up their skills.

Redis is a good target for learning because it covers two important subjects of software engineering: network programming and data structures.

- While there are many guides on socket APIs or high-level libraries, network programming is more than calling APIs or libraries. It is important to understand core concepts such as the event loop, protocols, timers, etc, which this book will cover. The lack of understanding can result in fatal mistakes even if you are just employing high-level networking libraries or frameworks in your applications.
- Although many people learned some basic data structures from textbooks, there is still something more to learn. Data structures implemented in real projects often have some practical considerations which are not touched by textbooks. Learning how data structures are used in a non-toy environment (especially in C) is a unique experience from building Redis.

Like most real-world projects, Redis is a complex project built with lots of effort, which can be hard to grasp for beginners. Instead, this book takes an opposite approach: learning by building things *from scratch*.

Why From Scratch?

A couple of points:

- **To learn faster**. By building things from scratch, concepts can be introduced gradually. Starting from the small, adding things incrementally, and getting the big picture in the end.
- **To learn deeper**. While there are many materials explaining how an existing stuff works, the understanding obtained by reading those materials is often not the same as building the stuff yourself. It is easy to mistake memorization for understanding, and it's easier to pick up unimportant details than principles and basics.
- **To learn more**. The "from scratch" approach forces you to touch *every* aspect of the subject — there are no shortcuts to knowledge! And often not every aspect is known to you beforehand, you may discover "things I don't know I don't know" in the process.

Summarized in a quote from Feynman: "What I cannot create, I do not understand".

How to Use This Book?

This book follows a step-by-step approach. Each step builds on the previous one, adding a new concept. The full source code is provided on the web for reference purposes, readers are advised to tinker with it or DIY without it.

The code is written as direct and straightforwardly as the author could. It's mostly plain C with minimal C++ features. Don't worry if you don't know C, you just have the opportunity to do it in another language by yourself.

The end result is a mini Redis alike with only about 1200 lines of code. 1200 LoC seems low, but it illustrates many important aspects the book attempts to cover.

The techniques and approaches used in the book are not exactly the same as the real Redis. Some are intentionally simplified, and some are chosen to illustrate a general topic. Readers can learn even more by comparing different approaches.

The code used in this book is intended to run on Linux only, and can be downloaded at this URL:

https://build-your-own.org/redis/src.tgz

The contents and the source code of this book can be browsed online at:

https://build-your-own.org

02. Introduction to Sockets

This chapter is an introduction to socket programming. Readers are assumed to have basic knowledge of computer networking but no experience in network programming. This book does not contain every detail on how to use socket APIs, you are advised to read manpages and other network programming guides while learning from this book. (https://beej.us/ is a good source for socket APIs.)

Redis is an example of the server/client system. Multiple clients connect to a single server, and the server receives requests from TCP connections and sends responses back. There are several Linux system calls we need to learn before we can start socket programming.

The **`socket()`** syscall returns an fd. Here is a rough explanation of "fd" if you are unfamiliar with Unix systems: An fd is an integer that refers to something in the Linux kernel, like a TCP connection, a disk file, a listening port, or some other resources, etc.

The **`bind()`** and **`listen()`** syscall: the **`bind()`** associates an address to a socket fd, and the **`listen()`** enables us to accept connections to that address.

The **`accept()`** takes a listening fd, when a client makes a connection to the listening address, the **`accept()`** returns an fd that represents the connection socket. Here is the pseudo-code that explains the typical workflow of a server:

```
fd = socket()
bind(fd, address)
listen(fd)
while True:
    conn_fd = accept(fd)
    do_something_with(conn_fd)
    close(conn_fd)
```

The **`read()`** syscall receives data from a TCP connection. The **`write()`** syscall sends data. The **`close()`** syscall destroys the resource referred by the fd and recycles the fd number.

We have introduced the syscalls needed for server-side network programming. For the client side, the **`connect()`** syscall takes a socket fd and address and makes a TCP connection to that address. Here is the pseudo-code for the client:

```
fd = socket()
connect(fd, address)
do_something_with(fd)
close(fd)
```

The next chapter will help you get started using real code.

03. Hello Server/Client

This chapter continues the introduction of socket programming. We'll write 2 simple (incomplete and broken) programs to demonstrate the syscalls from the last chapter. The first program is a server, it accepts connections from clients, reads a single message, and writes a single reply. The second program is a client, it connects to the server, writes a single message, and reads a single reply. Let's start with the server first.

First, we need to obtain a socket fd: **`int fd = socket(AF_INET, SOCK_STREAM, 0);`**

The **`AF_INET`** is for IPv4, use **`AF_INET6`** for IPv6 or dual-stack socket. For simplicity, we'll just use **`AF_INET`** throughout this book.

The **`SOCK_STREAM`** is for TCP. We won't use anything other than TCP in this book. All the 3 parameters of the **`socket()`** call are fixed in this book.

Next, we'll introduce a new syscall:

```
int val = 1;
setsockopt(fd, SOL_SOCKET, SO_REUSEADDR, &val, sizeof(val));
```

The **`setsockopt()`** call is used to configure various aspects of a socket. This particular call enables the **`SO_REUSEADDR`** option. Without this option, the server won't able to bind to the same address if restarted. Exercise to reader: find out what exactly is **`SO_REUSEADDR`** and why it is needed.

The next step is the **`bind()`** and **`listen()`**, we'll bind on the wildcard address **`0.0.0.0:1234`**:

```
    // bind, this is the syntax that deals with IPv4 addresses
    struct sockaddr_in addr = {};
    addr.sin_family = AF_INET;
    addr.sin_port = ntohs(1234);
    addr.sin_addr.s_addr = ntohl(0);    // wildcard address 0.0.0.0
    int rv = bind(fd, (const sockaddr *)&addr, sizeof(addr));
    if (rv) {
        die("bind()");
    }
```

```
    // listen
    rv = listen(fd, SOMAXCONN);
    if (rv) {
        die("listen()");
    }
```

Loop for each connection and do something with them.

```
    while (true) {
        // accept
        struct sockaddr_in client_addr = {};
        socklen_t socklen = sizeof(client_addr);
        int connfd = accept(fd, (struct sockaddr *)&client_addr, &socklen);
        if (connfd < 0) {
            continue;   // error
        }

        do_something(connfd);
        close(connfd);
    }
```

The **`do_something()`** function is simply read and write.

```
static void do_something(int connfd) {
    char rbuf[64] = {};
    ssize_t n = read(connfd, rbuf, sizeof(rbuf) - 1);
    if (n < 0) {
        msg("read() error");
        return;
    }
    printf("client says: %s\n", rbuf);

    char wbuf[] = "world";
    write(connfd, wbuf, strlen(wbuf));
}
```

Note that the **`read()`** and **`write()`** call returns the number of read or written bytes. A real programmer must deal with the return value of functions, but in this chapter, I have omitted lots of things for brevity. And the code in this chapter is not the correct way to do networking anyway.

The client program is much simpler:

```
    int fd = socket(AF_INET, SOCK_STREAM, 0);
    if (fd < 0) {
        die("socket()");
    }

    struct sockaddr_in addr = {};
    addr.sin_family = AF_INET;
    addr.sin_port = ntohs(1234);
    addr.sin_addr.s_addr = ntohl(INADDR_LOOPBACK);  // 127.0.0.1
    int rv = connect(fd, (const struct sockaddr *)&addr, sizeof(addr));
    if (rv) {
        die("connect");
    }

    char msg[] = "hello";
    write(fd, msg, strlen(msg));

    char rbuf[64] = {};
    ssize_t n = read(fd, rbuf, sizeof(rbuf) - 1);
    if (n < 0) {
        die("read");
    }
    printf("server says: %s\n", rbuf);
    close(fd);
```

Compile our programs with the following command line:

```
g++ -Wall -Wextra -O2 -g 03_server.cpp -o server
g++ -Wall -Wextra -O2 -g 03_client.cpp -o client
```

Run **`./server`** in a window and then run **`./client`** in another window. You should see the following results:

```
$ ./server
client says: hello
```

```
$ ./client
server says: world
```

Exercise for readers: read manpages of APIs used in this chapter, or find online tutorials for them. Make sure you know how to find helps on API usage since this book won't cover the details of API usage.

> - 03_client.cpp
> - 03_server.cpp

04. Protocol Parsing

Our server will be able to process multiple requests from a client, to do that we need to implement some sort of "protocol", at least to split requests apart from the TCP byte stream. The easiest way to split requests apart is by declaring how long the request is at the beginning of the request. Let's use the following scheme.

```
+-----+------+-----+------+--------
| len | msg1 | len | msg2 | more...
+-----+------+-----+------+--------
```

The protocol consists of 2 parts: a 4-byte little-endian integer indicating the length of the following request, and a variable length request.

Starts from the code from the last chapter, the loop of the server is modified to handle multiple requests:

```
    while (true) {
        // accept
        struct sockaddr_in client_addr = {};
        socklen_t socklen = sizeof(client_addr);
        int connfd = accept(fd, (struct sockaddr *)&client_addr, &socklen);
        if (connfd < 0) {
            continue;   // error
        }

        // only serves one client connection at once
        while (true) {
            int32_t err = one_request(connfd);
            if (err) {
                break;
            }
        }
        close(connfd);
    }
```

The **one_request** function only parses one request and replies, until something bad happens or the client connection is gone. Our server can only handle one connection at once until we introduce the event loop in later chapters.

Adding two helper functions before listing the **one_request** function:

```
static int32_t read_full(int fd, char *buf, size_t n) {
    while (n > 0) {
        ssize_t rv = read(fd, buf, n);
        if (rv <= 0) {
            return -1;  // error, or unexpected EOF
        }
        assert((size_t)rv <= n);
        n -= (size_t)rv;
        buf += rv;
    }
    return 0;
}

static int32_t write_all(int fd, const char *buf, size_t n) {
    while (n > 0) {
        ssize_t rv = write(fd, buf, n);
        if (rv <= 0) {
            return -1;  // error
        }
        assert((size_t)rv <= n);
        n -= (size_t)rv;
        buf += rv;
    }
    return 0;
}
```

Two things to note:

1. The **read()** syscall just returns whatever data is available in the kernel, or blocks if there is none. It's the application that is responsible for handling insufficient data. The **read_full()** function read from the kernel until it got exactly **n** bytes.
2. Likewise, the **write()** syscall can return successfully with partial data written if the kernel buffer is full, we need to keep trying when the **write()** returns fewer bytes than we need.

The **one_request** function did the actual work:

```
const size_t k_max_msg = 4096;

static int32_t one_request(int connfd) {
    // 4 bytes header
    char rbuf[4 + k_max_msg + 1];
    errno = 0;
    int32_t err = read_full(connfd, rbuf, 4);
    if (err) {
        if (errno == 0) {
            msg("EOF");
        } else {
            msg("read() error");
        }
        return err;
    }

    uint32_t len = 0;
    memcpy(&len, rbuf, 4);  // assume little endian
    if (len > k_max_msg) {
        msg("too long");
        return -1;
    }

    // request body
    err = read_full(connfd, &rbuf[4], len);
    if (err) {
        msg("read() error");
        return err;
    }

    // do something
    rbuf[4 + len] = '\0';
    printf("client says: %s\n", &rbuf[4]);

    // reply using the same protocol
    const char reply[] = "world";
    char wbuf[4 + sizeof(reply)];
    len = (uint32_t)strlen(reply);
```

```
    memcpy(wbuf, &len, 4);
    memcpy(&wbuf[4], reply, len);
    return write_all(connfd, wbuf, 4 + len);
}
```

For convenience, we added a limit to the maximum request size and use a large enough buffer to hold the request. Endianness used to be a consideration when parsing protocols, but it is less relevant today so we are just **memcpy**-ing integers.

The client code for making requests and receiving replies:

```
static int32_t query(int fd, const char *text) {
    uint32_t len = (uint32_t)strlen(text);
    if (len > k_max_msg) {
        return -1;
    }

    char wbuf[4 + k_max_msg];
    memcpy(wbuf, &len, 4);  // assume little endian
    memcpy(&wbuf[4], text, len);
    if (int32_t err = write_all(fd, wbuf, 4 + len)) {
        return err;
    }

    // 4 bytes header
    char rbuf[4 + k_max_msg + 1];
    errno = 0;
    int32_t err = read_full(fd, rbuf, 4);
    if (err) {
        if (errno == 0) {
            msg("EOF");
        } else {
            msg("read() error");
        }
        return err;
    }
```

```
    memcpy(&len, rbuf, 4);  // assume little endian
    if (len > k_max_msg) {
        msg("too long");
        return -1;
    }

    // reply body
    err = read_full(fd, &rbuf[4], len);
    if (err) {
        msg("read() error");
        return err;
    }

    // do something
    rbuf[4 + len] = '\0';
    printf("server says: %s\n", &rbuf[4]);
    return 0;
}
```

Test our server by sending multiple commands:

```
int main() {
    int fd = socket(AF_INET, SOCK_STREAM, 0);
    if (fd < 0) {
        die("socket()");
    }

    // code omitted ...

    // multiple requests
    int32_t err = query(fd, "hello1");
    if (err) {
        goto L_DONE;
    }
    err = query(fd, "hello2");
    if (err) {
        goto L_DONE;
```

```
    }
    err = query(fd, "hello3");
    if (err) {
        goto L_DONE;
    }

L_DONE:
    close(fd);
    return 0;
}
```

Running the server and the client:

```
$ ./server
client says: hello1
client says: hello2
client says: hello3
EOF

$ ./client
server says: world
server says: world
server says: world
```

The protocol parsing code requires at least 2 **`read()`** syscalls per request. The number of syscalls can be reduced by using "buffered IO". That is: read as much as you can into a buffer at once, then try to parse multiple requests from that buffer. Readers are encouraged to try this as an exercise as it may be helpful to understand later chapters.

Notes on protocols: The protocol used in this chapter is the most simple practical protocol. Most real-world protocols are more complicated than this. Some use text instead of binary data. While text protocols have the advantage of being human-readable, text protocols do require more parsing than binary ones, which are more coding and error-prone. Another thing to complicate protocol parsing is that some protocols don't have a straight way to split messages apart, those protocols may use delimiters, or require further parsing to split messages. The use of delimiters in protocols can add another complication when the

protocol is carrying arbitrary data, as the delimiters in data need to be "escaped". We'll stick to the simple binary protocol for later chapters.

> - 04_client.cpp
> - 04_server.cpp

05. The Event Loop and Nonblocking IO

There are 3 ways to deal with concurrent connections in server-side network programming. They are: forking, multi-threading, and event loops. Forking creates new processes for each client connection to achieve concurrency. Multi-threading uses threads instead of processes. An event loop uses polling and nonblocking IO and usually runs on a single thread. Due to the overhead of processes and threads, most modern production-grade software uses event loops for networking.

The simplified pseudo-code for the event loop of our server is:

```
all_fds = [...]
while True:
    active_fds = poll(all_fds)
    for each fd in active_fds:
        do_something_with(fd)

def do_something_with(fd):
    if fd is a listening socket:
        add_new_client(fd)
    elif fd is a client connection:
        while work_not_done(fd):
            do_something_to_client(fd)

def do_something_to_client(fd):
    if should_read_from(fd):
        data = read_until_EAGAIN(fd)
        process_incoming_data(data)
    while should_write_to(fd):
        write_until_EAGAIN(fd)
    if should_close(fd):
        destroy_client(fd)
```

Instead of just doing things (reading, writing, or accepting) with fds, we use the **`poll`** operation to tell us which fd can be operated *immediately* without blocking. When we perform an IO operation on an fd, the operation should be performed in the *nonblocking* mode.

In blocking mode, **`read`** blocks the caller when there are no data in the kernel, **`write`** blocks when the write buffer is full, and **`accept`** blocks when there are no new connections in the kernel queue. In nonblocking mode, those operations either success without blocking, or fail with the errno **`EAGAIN`**, which means "not ready". Nonblocking operations that fail with **`EAGAIN`** must be retried after the readiness was notified by the **`poll`**.

The **`poll`** is the *sole* blocking operation in an event loop, everything else must be non-blocking; thus, a single thread can handle multiple concurrent connections. All blocking networking IO APIs, such as **`read`**, **`write`**, and **`accept`**, have a nonblocking mode. APIs that do not have a nonblocking mode, such as **`gethostbyname`**, and disk IOs, should be performed in thread pools, which will be covered in later chapters. Also, timers must be implemented within the event loop since we can't **`sleep`** waiting inside the event loop.

The syscall for setting an fd to nonblocking mode is **`fcntl`**:

```
static void fd_set_nb(int fd) {
    errno = 0;
    int flags = fcntl(fd, F_GETFL, 0);
    if (errno) {
        die("fcntl error");
        return;
    }

    flags |= O_NONBLOCK;

    errno = 0;
    (void)fcntl(fd, F_SETFL, flags);
    if (errno) {
        die("fcntl error");
    }
}
```

On Linux, besides the **`poll`** syscall, there are also **`select`** and **`epoll`**. The ancient **`select`** syscall is basically the same as the **`poll`**, except that the maximum fd number is limited to a small number, which makes it obsolete in modern applications. The **`epoll`** API consists of 3 syscalls: **`epoll_create`**, **`epoll_wait`**, and **`epoll_ctl`**. The **`epoll`** API is stateful, instead of supplying a set of fds as a syscall argument, **`epoll_ctl`** was used to manipulate an fd set created by **`epoll_create`**, which the **`epoll_wait`** is operating on.

We'll use the **`poll`** syscall in the next chapter since it's slightly less code than the stateful **`epoll`** API. However, the **`epoll`** API is preferable in real-world projects since the argument for the **`poll`** can become too large as the number of fds increases.

06. The Event Loop Implementation

This chapter walks through the real C++ code of an echo server.

The definition of **`struct Conn`**:

```
enum {
    STATE_REQ = 0,
    STATE_RES = 1,
    STATE_END = 2,  // mark the connection for deletion
};

struct Conn {
    int fd = -1;
    uint32_t state = 0;     // either STATE_REQ or STATE_RES
    // buffer for reading
    size_t rbuf_size = 0;
    uint8_t rbuf[4 + k_max_msg];
    // buffer for writing
    size_t wbuf_size = 0;
    size_t wbuf_sent = 0;
    uint8_t wbuf[4 + k_max_msg];
};
```

We need buffers for reading/writing, since in nonblocking mode, IO operations are often deferred.

The **`state`** is used to decide what to do with the connection. There are 2 states for an ongoing connection. The **`STATE_REQ`** is for reading requests and the **`STATE_RES`** is for sending responses.

The code for the event loop:

```
int main() {
    int fd = socket(AF_INET, SOCK_STREAM, 0);
    if (fd < 0) {
        die("socket()");
```

```
    }

    // bind, listen and etc
    // code omitted...

    // a map of all client connections, keyed by fd
    std::vector<Conn *> fd2conn;

    // set the listen fd to nonblocking mode
    fd_set_nb(fd);

    // the event loop
    std::vector<struct pollfd> poll_args;
    while (true) {
        // prepare the arguments of the poll()
        poll_args.clear();
        // for convenience, the listening fd is put in the first position
        struct pollfd pfd = {fd, POLLIN, 0};
        poll_args.push_back(pfd);
        // connection fds
        for (Conn *conn : fd2conn) {
            if (!conn) {
                continue;
            }
            struct pollfd pfd = {};
            pfd.fd = conn->fd;
            pfd.events = (conn->state == STATE_REQ) ? POLLIN : POLLOUT;
            pfd.events = pfd.events | POLLERR;
            poll_args.push_back(pfd);
        }

        // poll for active fds
        // the timeout argument doesn't matter here
        int rv = poll(poll_args.data(), (nfds_t)poll_args.size(), 1000);
        if (rv < 0) {
            die("poll");
        }
```

```
        // process active connections
        for (size_t i = 1; i < poll_args.size(); ++i) {
            if (poll_args[i].revents) {
                Conn *conn = fd2conn[poll_args[i].fd];
                connection_io(conn);
                if (conn->state == STATE_END) {
                    // client closed normally, or something bad happened.
                    // destroy this connection
                    fd2conn[conn->fd] = NULL;
                    (void)close(conn->fd);
                    free(conn);
                }
            }
        }

        // try to accept a new connection if the listening fd is active
        if (poll_args[0].revents) {
            (void)accept_new_conn(fd2conn, fd);
        }
    }

    return 0;
}
```

The first thing in our event loop is setting up arguments of **`poll`**. The listening fd is polled with the **`POLLIN`** flag. For the connection fd, the state of the struct **`Conn`** determines the poll flag. In this particular case, the poll flag is either reading (**`POLLIN`**) or writing (**`POLLOUT`**), never both. If using **`epoll`**, the first thing in an event loop is usually updating the fd set with **`epoll_ctl`**.

The **`poll`** also takes a timeout argument which can be used to implement timers, in our case, this argument doesn't matter, just set it to a big number. After the return of **`poll`**, we are notified by which fd are ready for reading/writing and act accordingly.

The **`accept_new_conn()`** function accepts a new connection and creates the **`struct Conn`** object:

```
static void conn_put(std::vector<Conn *> &fd2conn, struct Conn *conn) {
    if (fd2conn.size() <= (size_t)conn->fd) {
        fd2conn.resize(conn->fd + 1);
    }
    fd2conn[conn->fd] = conn;
}

static int32_t accept_new_conn(std::vector<Conn *> &fd2conn, int fd) {
    // accept
    struct sockaddr_in client_addr = {};
    socklen_t socklen = sizeof(client_addr);
    int connfd = accept(fd, (struct sockaddr *)&client_addr, &socklen);
    if (connfd < 0) {
        msg("accept() error");
        return -1;  // error
    }

    // set the new connection fd to nonblocking mode
    fd_set_nb(connfd);
    // creating the struct Conn
    struct Conn *conn = (struct Conn *)malloc(sizeof(struct Conn));
    if (!conn) {
        close(connfd);
        return -1;
    }
    conn->fd = connfd;
    conn->state = STATE_REQ;
    conn->rbuf_size = 0;
    conn->wbuf_size = 0;
    conn->wbuf_sent = 0;
    conn_put(fd2conn, conn);
    return 0;
}
```

The **`connection_io()`** is the state machine for client connections:

```
static void connection_io(Conn *conn) {
    if (conn->state == STATE_REQ) {
        state_req(conn);
    } else if (conn->state == STATE_RES) {
        state_res(conn);
    } else {
        assert(0);  // not expected
    }
}
```

The **STATE_REQ** state is for reading:

```
static void state_req(Conn *conn) {
    while (try_fill_buffer(conn)) {}
}

static bool try_fill_buffer(Conn *conn) {
    // try to fill the buffer
    assert(conn->rbuf_size < sizeof(conn->rbuf));
    ssize_t rv = 0;
    do {
        size_t cap = sizeof(conn->rbuf) - conn->rbuf_size;
        rv = read(conn->fd, &conn->rbuf[conn->rbuf_size], cap);
    } while (rv < 0 && errno == EINTR);
    if (rv < 0 && errno == EAGAIN) {
        // got EAGAIN, stop.
        return false;
    }
    if (rv < 0) {
        msg("read() error");
        conn->state = STATE_END;
        return false;
    }
    if (rv == 0) {
        if (conn->rbuf_size > 0) {
            msg("unexpected EOF");
        } else {
```

```
            msg("EOF");
        }
        conn->state = STATE_END;
        return false;
    }

    conn->rbuf_size += (size_t)rv;
    assert(conn->rbuf_size <= sizeof(conn->rbuf) - conn->rbuf_size);

    // Try to process requests one by one.
    // Why is there a loop? Please read the explanation of "pipelining".
    while (try_one_request(conn)) {}
    return (conn->state == STATE_REQ);
}
```

There are lots of things to unpack here. To understand this function, let's review the pseudo-code from the last chapter:

```
def do_something_to_client(fd):
    if should_read_from(fd):
        data = read_until_EAGAIN(fd)
        process_incoming_data(data)
    # code omitted...
```

The **`try_fill_buffer()`** function fills the read buffer with data. Since the size of the read buffer is limited, the read buffer could be full before we hit **`EAGAIN`**, so we need to process data immediately after reading to clear some read buffer space, then the **`try_fill_buffer()`** is looped until we hit **`EAGAIN`**.

The **`read`** syscall (and any other syscalls) need to be retried after getting the errno **`EINTR`**. The **`EINTR`** means the syscall was interrupted by a signal, the retrying is needed even if our application does not make use of signals.

The **`try_one_request`** function handles the incoming data, but why is this in a loop? Is there more than one request in the read buffer? The answer is yes. For a request/response protocol, clients are not limited to sending one request and waiting for the response at a time, clients can save some latency by sending multiple requests without waiting for

responses in between, this mode of operation is called "pipelining". Thus we can't assume that the read buffer contains at most one request.

Listing the **`try_one_request`** function:

```
static bool try_one_request(Conn *conn) {
    // try to parse a request from the buffer
    if (conn->rbuf_size < 4) {
        // not enough data in the buffer. Will retry in the next iteration
        return false;
    }
    uint32_t len = 0;
    memcpy(&len, &conn->rbuf[0], 4);
    if (len > k_max_msg) {
        msg("too long");
        conn->state = STATE_END;
        return false;
    }
    if (4 + len > conn->rbuf_size) {
        // not enough data in the buffer. Will retry in the next iteration
        return false;
    }

    // got one request, do something with it
    printf("client says: %.*s\n", len, &conn->rbuf[4]);

    // generating echoing response
    memcpy(&conn->wbuf[0], &len, 4);
    memcpy(&conn->wbuf[4], &conn->rbuf[4], len);
    conn->wbuf_size = 4 + len;

    // remove the request from the buffer.
    // note: frequent memmove is inefficient.
    // note: need better handling for production code.
    size_t remain = conn->rbuf_size - 4 - len;
    if (remain) {
        memmove(conn->rbuf, &conn->rbuf[4 + len], remain);
    }
    conn->rbuf_size = remain;
```

```
    // change state
    conn->state = STATE_RES;
    state_res(conn);

    // continue the outer loop if the request was fully processed
    return (conn->state == STATE_REQ);
}
```

The **`try_one_request`** function takes one request from the read buffer, generates a response, then transits to the **`STATE_RES`** state.

The code for the state **`STATE_RES`**:

```
static void state_res(Conn *conn) {
    while (try_flush_buffer(conn)) {}
}

static bool try_flush_buffer(Conn *conn) {
    ssize_t rv = 0;
    do {
        size_t remain = conn->wbuf_size - conn->wbuf_sent;
        rv = write(conn->fd, &conn->wbuf[conn->wbuf_sent], remain);
    if (rv < 0 && errno == EAGAIN) {
        // got EAGAIN, stop.
        return false;
    }
    if (rv < 0) {
        msg("write() error");
        conn->state = STATE_END;
        return false;
    }
    conn->wbuf_sent += (size_t)rv;
    assert(conn->wbuf_sent <= conn->wbuf_size);
    if (conn->wbuf_sent == conn->wbuf_size) {
        // response was fully sent, change state back
        conn->state = STATE_REQ;
        conn->wbuf_sent = 0;
```

```
        conn->wbuf_size = 0;
        return false;
    }
    // still got some data in wbuf, could try to write again
    return true;
}
```

The above code flushes the write buffer until it got **`EAGAIN`**, or transits back to the **`STATE_REQ`** if the flushing is done.

To test our server, we can run the client from chapter 04 since the protocol is identical. We can also modify the client to demonstrate pipelining client:

```
// the `query` function was simply splited into `send_req` and `read_res`.
static int32_t send_req(int fd, const char *text);
static int32_t read_res(int fd);

int main() {
    int fd = socket(AF_INET, SOCK_STREAM, 0);
    if (fd < 0) {
        die("socket()");
    }

    // code omitted...

    // multiple pipelined requests
    const char *query_list[3] = {"hello1", "hello2", "hello3"};
    for (size_t i = 0; i < 3; ++i) {
        int32_t err = send_req(fd, query_list[i]);
        if (err) {
            goto L_DONE;
        }
    }
    for (size_t i = 0; i < 3; ++i) {
        int32_t err = read_res(fd);
        if (err) {
            goto L_DONE;
```

```
        }
    }

L_DONE:
    close(fd);
    return 0;
}
```

Exercises:

1. Try to use **`epoll`** instead of **`poll`** in the event loop. This should be easy.
2. We are using **`memmove`** to reclaim read buffer space. However, **`memmove`** on every request is unnecessary, change the code the perform **`memmove`** only before **`read`**.
3. In the **`state_res`** function, **`write`** was performed for a single response. In pipelined sceneries, we could buffer multiple responses and flush them in the end with a single **`write`** call. Note that the write buffer could be full in the middle.

> - 06_client.cpp
> - 06_server.cpp

07. Basic Server: get, set, del

With the event loop code from the last chapter, we can finally start adding commands to our server.

The "command" in our design is a list of strings, like **`set key val`**. We'll encode the "command" with the following scheme.

```
+------+-----+------+-----+------+-----+-----+------+
| nstr | len | str1 | len | str2 | ... | len | strn |
+------+-----+------+-----+------+-----+-----+------+
```

The **`nstr`** is the number of strings and the **`len`** is the length of the following string. Both are 32-bit integers.

The response is a 32-bit status code followed by the response string.

```
+-----+---------+
| res | data... |
+-----+---------+
```

Starts with the **`try_one_request`** function.

```
static bool try_one_request(Conn *conn) {
    // try to parse a request from the buffer
    if (conn->rbuf_size < 4) {
        // not enough data in the buffer. Will retry in the next iteration
        return false;
    }
    uint32_t len = 0;
    memcpy(&len, &conn->rbuf[0], 4);
    if (len > k_max_msg) {
        msg("too long");
        conn->state = STATE_END;
        return false;
    }
    if (4 + len > conn->rbuf_size) {
        // not enough data in the buffer. Will retry in the next iteration
        return false;
    }
```

```
    // got one request, generate the response.
    uint32_t rescode = 0;
    uint32_t wlen = 0;
    int32_t err = do_request(
        &conn->rbuf[4], len,
        &rescode, &conn->wbuf[4 + 4], &wlen
    );
    if (err) {
        conn->state = STATE_END;
        return false;
    }
    wlen += 4;
    memcpy(&conn->wbuf[0], &wlen, 4);
    memcpy(&conn->wbuf[4], &rescode, 4);
    conn->wbuf_size = 4 + wlen;

    // remove the request from the buffer.
    // note: frequent memmove is inefficient.
    // note: need better handling for production code.
    size_t remain = conn->rbuf_size - 4 - len;
    if (remain) {
        memmove(conn->rbuf, &conn->rbuf[4 + len], remain);
    }
    conn->rbuf_size = remain;

    // change state
    conn->state = STATE_RES;
    state_res(conn);

    // continue the outer loop if the request was fully processed
    return (conn->state == STATE_REQ);
}
```

The **`do_request`** function handles the request. Only 3 commands (get, set, del) are recognized now.

```
static int32_t do_request(
    const uint8_t *req, uint32_t reqlen,
    uint32_t *rescode, uint8_t *res, uint32_t *reslen)
{
    std::vector<std::string> cmd;
    if (0 != parse_req(req, reqlen, cmd)) {
        msg("bad req");
        return -1;
    }
    if (cmd.size() == 2 && cmd_is(cmd[0], "get")) {
        *rescode = do_get(cmd, res, reslen);
    } else if (cmd.size() == 3 && cmd_is(cmd[0], "set")) {
        *rescode = do_set(cmd, res, reslen);
    } else if (cmd.size() == 2 && cmd_is(cmd[0], "del")) {
        *rescode = do_del(cmd, res, reslen);
    } else {
        // cmd is not recognized
        *rescode = RES_ERR;
        const char *msg = "Unknown cmd";
        strcpy((char *)res, msg);
        *reslen = strlen(msg);
        return 0;
    }
    return 0;
}
```

The parsing of the command is straightforward:

```
static int32_t parse_req(
    const uint8_t *data, size_t len, std::vector<std::string> &out)
{
    if (len < 4) {
        return -1;
    }
    uint32_t n = 0;
    memcpy(&n, &data[0], 4);
    if (n > k_max_args) {
```

```
        return -1;
    }

    size_t pos = 4;
    while (n--) {
        if (pos + 4 > len) {
            return -1;
        }
        uint32_t sz = 0;
        memcpy(&sz, &data[pos], 4);
        if (pos + 4 + sz > len) {
            return -1;
        }
        out.push_back(std::string((char *)&data[pos + 4], sz));
        pos += 4 + sz;
    }

    if (pos != len) {
        return -1;  // trailing garbage
    }
    return 0;
}
```

The "implementation" of 3 commands:

```
enum {
    RES_OK = 0,
    RES_ERR = 1,
    RES_NX = 2,
};

// The data structure for the key space. This is just a placeholder
// until we implement a hashtable in the next chapter.
static std::map<std::string, std::string> g_map;

static uint32_t do_get(
    const std::vector<std::string> &cmd, uint8_t *res, uint32_t *reslen)
```

```
{
    if (!g_map.count(cmd[1])) {
        return RES_NX;
    }
    std::string &val = g_map[cmd[1]];
    assert(val.size() <= k_max_msg);
    memcpy(res, val.data(), val.size());
    *reslen = (uint32_t)val.size();
    return RES_OK;
}

static uint32_t do_set(
    const std::vector<std::string> &cmd, uint8_t *res, uint32_t *reslen)
{
    (void)res;
    (void)reslen;
    g_map[cmd[1]] = cmd[2];
    return RES_OK;
}

static uint32_t do_del(
    const std::vector<std::string> &cmd, uint8_t *res, uint32_t *reslen)
{
    (void)res;
    (void)reslen;
    g_map.erase(cmd[1]);
    return RES_OK;
}
```

Now it's time to test with our client:

```
static int32_t send_req(int fd, const std::vector<std::string> &cmd) {
    uint32_t len = 4;
    for (const std::string &s : cmd) {
        len += 4 + s.size();
    }
    if (len > k_max_msg) {
```

```
        return -1;
    }

    char wbuf[4 + k_max_msg];
    memcpy(&wbuf[0], &len, 4);  // assume little endian
    uint32_t n = cmd.size();
    memcpy(&wbuf[4], &n, 4);
    size_t cur = 8;
    for (const std::string &s : cmd) {
        uint32_t p = (uint32_t)s.size();
        memcpy(&wbuf[cur], &p, 4);
        memcpy(&wbuf[cur + 4], s.data(), s.size());
        cur += 4 + s.size();
    }
    return write_all(fd, wbuf, 4 + len);
}

static int32_t read_res(int fd) {
    // code omitted...

    // print the result
    uint32_t rescode = 0;
    if (len < 4) {
        msg("bad response");
        return -1;
    }
    memcpy(&rescode, &rbuf[4], 4);
    printf("server says: [%u] %.*s\n", rescode, len - 4, &rbuf[8]);
    return 0;
}

int main(int argc, char **argv) {
    int fd = socket(AF_INET, SOCK_STREAM, 0);
    if (fd < 0) {
        die("socket()");
    }

    // code omitted...
```

```
    std::vector<std::string> cmd;
    for (int i = 1; i < argc; ++i) {
        cmd.push_back(argv[i]);
    }
    int32_t err = send_req(fd, cmd);
    if (err) {
        goto L_DONE;
    }
    err = read_res(fd);
    if (err) {
        goto L_DONE;
    }

L_DONE:
    close(fd);
    return 0;
}
```

Testing commands:

```
$ ./client get k
server says: [2]
$ ./client set k v
server says: [0]
$ ./client get k
server says: [0] v
$ ./client del k
server says: [0]
$ ./client get k
server says: [2]
$ ./client aaa bbb
server says: [1] Unknown cmd
```

> - 07_client.cpp
> - 07_server.cpp

Part 2. Essential Topics

Learn more network programming and data structures.

08. Data Structure: Hashtables

This chapter fills the placeholder code in the last chapter's server. We'll start by implementing a hashtable. Hashtables are often the obvious data structure for holding an unknown amount of key-value data that does not require ordering.

There are two kinds of hashtables: chaining and open addressing. Their primary difference is collision resolution. Open addressing seeks another free slot in the event of a collision while chaining simply groups conflicting keys with a linked list. There are many variants of open addressing due to the need to find free slots, while the chaining hashtable is pretty much a fixed design. The hashtable used in our server is a chaining one. A chaining hashtable is easy to code; it doesn't require much choice-making.

The definition of our data types:

```
// hashtable node, should be embedded into the payload
struct HNode {
    HNode *next = NULL;
    uint64_t hcode = 0;
};

// a simple fixed-sized hashtable
struct HTab {
    HNode **tab = NULL;
    size_t mask = 0;
    size_t size = 0;
};
```

When the size of the hashtable is the power of two, the indexing operation is a simple bit mask with the hash code.

```
// n must be a power of 2
static void h_init(HTab *htab, size_t n) {
    assert(n > 0 && ((n - 1) & n) == 0);
    htab->tab = (HNode **)calloc(sizeof(HNode *), n);
    htab->mask = n - 1;
```

```
    htab->size = 0;
}

// hashtable insertion
static void h_insert(HTab *htab, HNode *node) {
    size_t pos = node->hcode & htab->mask;
    HNode *next = htab->tab[pos];
    node->next = next;
    htab->tab[pos] = node;
    htab->size++;
}
```

The lookup subroutine is simply a list traversal:

```
// hashtable look up subroutine.
// Pay attention to the return value. It returns the address of
// the parent pointer that owns the target node,
// which can be used to delete the target node.
static HNode **h_lookup(
    HTab *htab, HNode *key, bool (*cmp)(HNode *, HNode *))
{
    if (!htab->tab) {
        return NULL;
    }

    size_t pos = key->hcode & htab->mask;
    HNode **from = &htab->tab[pos];
    while (*from) {
        if (cmp(*from, key)) {
            return from;
        }
        from = &(*from)->next;
    }
    return NULL;
}
```

Deleting is easy. Notice how the use of pointers enables succinct code. The **`from`** pointer

can be either an item of the array or from a node, yet the code doesn't differentiate.

```
// remove a node from the chain
static HNode *h_detach(HTab *htab, HNode **from) {
    HNode *node = *from;
    *from = (*from)->next;
    htab->size--;
    return node;
}
```

Our hashtable is fixed in size, we need to migrate to a bigger one when the load factor is too high. There is an extra consideration when using hashtables in Redis. Resizing a large hashtable requires moving a lot of nodes to a new table, which can stall the server for some time. This shall be avoided by not moving everything at once, instead, we keep two hashtables and gradually move nodes between them. Here is the final hashtable interface:

```
// the real hashtable interface.
// it uses 2 hashtables for progressive resizing.
struct HMap {
    HTab ht1;
    HTab ht2;
    size_t resizing_pos = 0;
};
```

The lookup subroutine now help with resizing:

```
HNode *hm_lookup(
    HMap *hmap, HNode *key, bool (*cmp)(HNode *, HNode *))
{
    hm_help_resizing(hmap);
    HNode **from = h_lookup(&hmap->ht1, key, cmp);
    if (!from) {
        from = h_lookup(&hmap->ht2, key, cmp);
    }
    return from ? *from : NULL;
}
```

The **`hm_help_resizing`** function is the subroutine for gradually moving nodes:

```
const size_t k_resizing_work = 128;

static void hm_help_resizing(HMap *hmap) {
    if (hmap->ht2.tab == NULL) {
        return;
    }

    size_t nwork = 0;
    while (nwork < k_resizing_work && hmap->ht2.size > 0) {
        // scan for nodes from ht2 and move them to ht1
        HNode **from = &hmap->ht2.tab[hmap->resizing_pos];
        if (!*from) {
            hmap->resizing_pos++;
            continue;
        }

        h_insert(&hmap->ht1, h_detach(&hmap->ht2, from));
        nwork++;
    }

    if (hmap->ht2.size == 0) {
        // done
        free(hmap->ht2.tab);
        hmap->ht2 = HTab{};
    }
}
```

The insertion subroutine will trigger resizing should the table become too full:

```
const size_t k_max_load_factor = 8;

void hm_insert(HMap *hmap, HNode *node) {
    if (!hmap->ht1.tab) {
        h_init(&hmap->ht1, 4);
    }
```

```
    h_insert(&hmap->ht1, node);

    if (!hmap->ht2.tab) {
        // check whether we need to resize
        size_t load_factor = hmap->ht1.size / (hmap->ht1.mask + 1);
        if (load_factor >= k_max_load_factor) {
            hm_start_resizing(hmap);
        }
    }
    hm_help_resizing(hmap);
}

static void hm_start_resizing(HMap *hmap) {
    assert(hmap->ht2.tab == NULL);
    // create a bigger hashtable and swap them
    hmap->ht2 = hmap->ht1;
    h_init(&hmap->ht1, (hmap->ht1.mask + 1) * 2);
    hmap->resizing_pos = 0;
}
```

The subroutine for removing a key. Nothing interesting.

```
HNode *hm_pop(
    HMap *hmap, HNode *key, bool (*cmp)(HNode *, HNode *))
{
    hm_help_resizing(hmap);
    HNode **from = h_lookup(&hmap->ht1, key, cmp);
    if (from) {
        return h_detach(&hmap->ht1, from);
    }
    from = h_lookup(&hmap->ht2, key, cmp);
    if (from) {
        return h_detach(&hmap->ht2, from);
    }
    return NULL;
}
```

The hashtable implementation is done. Let's add them to the server. Looking at the **`struct HNode`** again, this structure contains no data, how do we actually use that? The answer is called "intrusive data structure":

```
// the structure for the key
struct Entry {
    struct HNode node;
    std::string key;
    std::string val;
};
```

Instead of making our data structure contain data, the hashtable node structure is embedded into the payload data. This is the standard way of creating generic data structures in C. Besides making the data structure fully generic, this technique also has the advantage of reducing unnecessary memory management. The structure node is not separately allocated but is part of the payload data, and the data structure code does not own the payload but merely organizes the data. This may be quite a new idea to you if you learned data structures from textbooks, which is probably using **`void *`** or C++ templates or even macros.

Listing the **`do_get`** function to see how the intrusive data structure is used:

```
// The data structure for the key space.
static struct {
    HMap db;
} g_data;

static uint32_t do_get(
    std::vector<std::string> &cmd, uint8_t *res, uint32_t *reslen)
{
    Entry key;
    key.key.swap(cmd[1]);
    key.node.hcode = str_hash((uint8_t *)key.key.data(), key.key.size());

    HNode *node = hm_lookup(&g_data.db, &key.node, &entry_eq);
    if (!node) {
        return RES_NX;
```

```
    }

    const std::string &val = container_of(node, Entry, node)->val;
    assert(val.size() <= k_max_msg);
    memcpy(res, val.data(), val.size());
    *reslen = (uint32_t)val.size();
    return RES_OK;
}

static bool entry_eq(HNode *lhs, HNode *rhs) {
    struct Entry *le = container_of(lhs, struct Entry, node);
    struct Entry *re = container_of(rhs, struct Entry, node);
    return lhs->hcode == rhs->hcode && le->key == re->key;
}
```

The **hm_lookup** function returns a pointer to **HNode**, which is a member of the **Entry**, we need some pointer arithmetics to convert that pointer to an **Entry** pointer. The **container_of** macro is commonly used in C projects for this purpose:

```
#define container_of(ptr, type, member) ({                  \
    const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
    (type *)( (char *)__mptr - offsetof(type, member) );})
```

The **do_set** and **do_del** are both trivial.

```
static uint32_t do_set(
    std::vector<std::string> &cmd, uint8_t *res, uint32_t *reslen)
{
    (void)res;
    (void)reslen;

    Entry key;
    key.key.swap(cmd[1]);
    key.node.hcode = str_hash((uint8_t *)key.key.data(), key.key.size());

    HNode *node = hm_lookup(&g_data.db, &key.node, &entry_eq);
```

```
    if (node) {
        container_of(node, Entry, node)->val.swap(cmd[2]);
    } else {
        Entry *ent = new Entry();
        ent->key.swap(key.key);
        ent->node.hcode = key.node.hcode;
        ent->val.swap(cmd[2]);
        hm_insert(&g_data.db, &ent->node);
    }
    return RES_OK;
}

static uint32_t do_del(
    std::vector<std::string> &cmd, uint8_t *res, uint32_t *reslen)
{
    (void)res;
    (void)reslen;

    Entry key;
    key.key.swap(cmd[1]);
    key.node.hcode = str_hash((uint8_t *)key.key.data(), key.key.size());

    HNode *node = hm_pop(&g_data.db, &key.node, &entry_eq);
    if (node) {
        delete container_of(node, Entry, node);
    }
    return RES_OK;
}
```

Exercises:

1. Our hashtable triggers resizing when the load factor is too high, should we also shrink the hashtable when the load factor is too low? Can the shrinking be performed automatically?

> - 08_server.cpp
> - hashtable.cpp
> - hashtable.h

09. Data Serialization

For now, our server protocol response is an error code plus a string. What if we need to return more complicated data? For example, we might add the **`keys`** command that returns a list of strings. We have already encoded the list-of-strings data in the request protocol. In this chapter, we will generalize the encoding to handle different types of data. This is often called "serialization".

Our serialization protocol consists of five types of data:

```
enum {
    SER_NIL = 0,
    SER_ERR = 1,
    SER_STR = 2,
    SER_INT = 3,
    SER_ARR = 4,
};
```

The **`SER_NIL`** is like **`NULL`**, the **`SER_ERR`** is for returning error code and message, the **`SER_STR`** and **`SER_INT`** are for string and **`int64`**, and the **`SER_ARR`** is for arrays.

Code listing starts with the **`try_one_request`** function:

```
static bool try_one_request(Conn *conn) {
    // code omitted...

    // parse the request
    std::vector<std::string> cmd;
    if (0 != parse_req(&conn->rbuf[4], len, cmd)) {
        msg("bad req");
        conn->state = STATE_END;
        return false;
    }

    // got one request, generate the response.
    std::string out;
    do_request(cmd, out);
```

```
    // pack the response into the buffer
    if (4 + out.size() > k_max_msg) {
        out.clear();
        out_err(out, ERR_2BIG, "response is too big");
    }
    uint32_t wlen = (uint32_t)out.size();
    memcpy(&conn->wbuf[0], &wlen, 4);
    memcpy(&conn->wbuf[4], out.data(), out.size());
    conn->wbuf_size = 4 + wlen;

    // code omitted...
}
```

For convenience, **`std::string`** was used to hold the response data. Production-grade projects often have more sophisticated ways to manage buffers.

A new command **`keys`** was added to the **`do_request`** handler:

```
static void do_request(std::vector<std::string> &cmd, std::string &out) {
    if (cmd.size() == 1 && cmd_is(cmd[0], "keys")) {
        do_keys(cmd, out);
    } else if (cmd.size() == 2 && cmd_is(cmd[0], "get")) {
        do_get(cmd, out);
    } else if (cmd.size() == 3 && cmd_is(cmd[0], "set")) {
        do_set(cmd, out);
    } else if (cmd.size() == 2 && cmd_is(cmd[0], "del")) {
        do_del(cmd, out);
    } else {
        // cmd is not recognized
        out_err(out, ERR_UNKNOWN, "Unknown cmd");
    }
}
```

The code for our serialization protocol:

```
static void out_nil(std::string &out) {
    out.push_back(SER_NIL);
}

static void out_str(std::string &out, const std::string &val) {
    out.push_back(SER_STR);
    uint32_t len = (uint32_t)val.size();
    out.append((char *)&len, 4);
    out.append(val);
}

static void out_int(std::string &out, int64_t val) {
    out.push_back(SER_INT);
    out.append((char *)&val, 8);
}

static void out_err(std::string &out, int32_t code, const std::string &msg) {
    out.push_back(SER_ERR);
    out.append((char *)&code, 4);
    uint32_t len = (uint32_t)msg.size();
    out.append((char *)&len, 4);
    out.append(msg);
}

static void out_arr(std::string &out, uint32_t n) {
    out.push_back(SER_ARR);
    out.append((char *)&n, 4);
}
```

As we can see, our serialization protocol starts with one byte of data type, followed by various types of payload data. Arrays come with their size first, then their possibly nested elements.

The **do_keys** function generates a response consisting of a list of strings:

```
static void h_scan(HTab *tab, void (*f)(HNode *, void *), void *arg) {
    if (tab->size == 0) {
```

```
        return;
    }
    for (size_t i = 0; i < tab->mask + 1; ++i) {
        HNode *node = tab->tab[i];
        while (node) {
            f(node, arg);
            node = node->next;
        }
    }
}

static void cb_scan(HNode *node, void *arg) {
    std::string &out = *(std::string *)arg;
    out_str(out, container_of(node, Entry, node)->key);
}

static void do_keys(std::vector<std::string> &cmd, std::string &out) {
    (void)cmd;
    out_arr(out, (uint32_t)hm_size(&g_data.db));
    h_scan(&g_data.db.ht1, &cb_scan, &out);
    h_scan(&g_data.db.ht2, &cb_scan, &out);
}
```

The **`del`** command responds with an integer indicating whether the deletion took place.

```
static void do_del(std::vector<std::string> &cmd, std::string &out) {
    Entry key;
    key.key.swap(cmd[1]);
    key.node.hcode = str_hash((uint8_t *)key.key.data(), key.key.size());

    HNode *node = hm_pop(&g_data.db, &key.node, &entry_eq);
    if (node) {
        delete container_of(node, Entry, node);
    }
    return out_int(out, node ? 1 : 0);
}
```

The code for other commands is of nothing interesting, there is no need to list them.

Listing the client "deserialization" code:

```
static int32_t on_response(const uint8_t *data, size_t size) {
    if (size < 1) {
        msg("bad response");
        return -1;
    }
    switch (data[0]) {
    case SER_NIL:
        printf("(nil)\n");
        return 1;
    case SER_ERR:
        if (size < 1 + 8) {
            msg("bad response");
            return -1;
        }
        {
            int32_t code = 0;
            uint32_t len = 0;
            memcpy(&code, &data[1], 4);
            memcpy(&len, &data[1 + 4], 4);
            if (size < 1 + 8 + len) {
                msg("bad response");
                return -1;
            }
            printf("(err) %d %.*s\n", code, len, &data[1 + 8]);
            return 1 + 8 + len;
        }
    case SER_STR:
        // code omited...
    case SER_INT:
        // code omited...
    case SER_ARR:
        if (size < 1 + 4) {
            msg("bad response");
            return -1;
        }
```

```
        {
            uint32_t len = 0;
            memcpy(&len, &data[1], 4);
            printf("(arr) len=%u\n", len);
            size_t arr_bytes = 1 + 4;
            for (uint32_t i = 0; i < len; ++i) {
                int32_t rv = on_response(&data[arr_bytes], size - arr_bytes);
                if (rv < 0) {
                    return rv;
                }
                arr_bytes += (size_t)rv;
            }
            printf("(arr) end\n");
            return (int32_t)arr_bytes;
        }
    default:
        msg("bad response");
        return -1;
    }
}
```

Testing our new server/client:

```
$ ./client asdf
(err) 1 Unknown cmd
$ ./client get asdf
(nil)
$ ./client set k v
(nil)
$ ./client get k
(str) v
$ ./client keys
(arr) len=1
(str) k
(arr) end
$ ./client del k
(int) 1
$ ./client del k
(int) 0
$ ./client keys
(arr) len=0
(arr) end
```

> - 09_client.cpp
> - 09_server.cpp
> - hashtable.cpp
> - hashtable.h

10. The AVL Tree: Implementation & Testing

While Redis is often referred to as a key-value store, the "value" part of Redis is not restricted to plain strings, lists, hashmaps, and sorted sets are quite nice things to have. Redis is also referred to as the "data structure server" due to its rich set of data structures. Redis is often used as an in-memory cache, and when storing data in memory, there is an advantage of freely using data structures. The sorted set data structure in Redis is quite a unique and useful thing. Not only it offers the ability to sort your data in order, but also has the unique feature of querying ordered data by rank. If you put 20M records into a sorted set, you can get the record that ranked at 10M, without going through the first 10M records, this is a feat that can not be emulated by current SQL databases.

As the name "sorted set" implies, it's a data structure for sorting. Trees, balanced binary trees, are popular data structures for storing sorted data. Among various data structures, the author found the AVL tree particularly simple and easy to code, which will be used in this book to implement sorted set. The real Redis project uses skiplist which is also considered easy to code.

The idea of the AVL tree is to restrict the height difference between the left subtree and the right subtree. The height difference between subtrees is restricted to be at most one, never reaching two. When inserting/removing nodes from an AVL tree, the height difference can temporarily reach two, which is then fixed by the node rotations. The rotation operation is the basis of balanced binary trees, which is also used by other balanced trees like the RB tree. After the rotation, a node with a subtree height difference of two is reduced back to be at most one.

Let's start with the tree node:

```
struct AVLNode {
    uint32_t depth = 0;
    uint32_t cnt = 0;
    AVLNode *left = NULL;
    AVLNode *right = NULL;
    AVLNode *parent = NULL;
};

static void avl_init(AVLNode *node) {
```

```
    node->depth = 1;
    node->cnt = 1;
    node->left = node->right = node->parent = NULL;
}
```

This is a regular binary tree node with extra fields. The **depth** field is the height of the tree. The **cnt** field is the size of the tree, this field is not specific to the AVL tree, it is used to implement the rank-based query, which will be explained in the next chapter.

Listing some helper functions:

```
static uint32_t avl_depth(AVLNode *node) {
    return node ? node->depth : 0;
}

static uint32_t avl_cnt(AVLNode *node) {
    return node ? node->cnt : 0;
}

static uint32_t max(uint32_t lhs, uint32_t rhs) {
    return lhs < rhs ? rhs : lhs;
}

// maintaining the depth and cnt field
static void avl_update(AVLNode *node) {
    node->depth = 1 + max(avl_depth(node->left), avl_depth(node->right));
    node->cnt = 1 + avl_cnt(node->left) + avl_cnt(node->right);
}
```

The node rotation code:

```
static AVLNode *rot_left(AVLNode *node) {
    AVLNode *new_node = node->right;
    if (new_node->left) {
        new_node->left->parent = node;
    }
    node->right = new_node->left;
```

```
    new_node->left = node;
    new_node->parent = node->parent;
    node->parent = new_node;
    avl_update(node);
    avl_update(new_node);
    return new_node;
}

static AVLNode *rot_right(AVLNode *node) {
    // a mirror of the rot_left()
    // code omited...
}
```

A visualization of the **`rot_left`** operation:

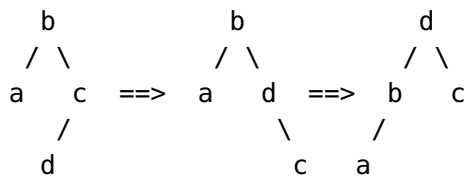
```
  b           d
 / \         /
a   d  ==>  b
   /       / \
  c       a   c
```

The **`avl_fix_left`** and **`avl_fix_right`** are functions for fixing excess height difference:

```
// the left subtree is too deep
static AVLNode *avl_fix_left(AVLNode *root) {
    if (avl_depth(root->left->left) < avl_depth(root->left->right)) {
        root->left = rot_left(root->left);
    }
    return rot_right(root);
}

// the right subtree is too deep
static AVLNode *avl_fix_right(AVLNode *root) {
    if (avl_depth(root->right->right) < avl_depth(root->right->left)) {
        root->right = rot_right(root->right);
    }
    return rot_left(root);
}
```

If the right subtree is too deep, a left rotation will fix it. Before the left rotation, we may need a right rotation on the right subtree to ensure the right subtree is leaning in the correct direction. Here is the visualization:



The **avl_fix** function fixes everything after an insertion/deletion operation. It goes from the initially affected node to the root node. Since the rotation may change the root of the tree, the root node is returned. This is the core of our AVL tree implementation.

```
// fix imbalanced nodes and maintain invariants until the root is reached
static AVLNode *avl_fix(AVLNode *node) {
    while (true) {
        avl_update(node);
        uint32_t l = avl_depth(node->left);
        uint32_t r = avl_depth(node->right);
        AVLNode **from = NULL;
        if (node->parent) {
            from = (node->parent->left == node)
                ? &node->parent->left : &node->parent->right;
        }
        if (l == r + 2) {
            node = avl_fix_left(node);
        } else if (l + 2 == r) {
            node = avl_fix_right(node);
        }
        if (!from) {
            return node;
        }
        *from = node;
        node = node->parent;
    }
}
```

Insertion for binary trees is easy, just walk down from the root until you find an empty subtree and place the new node here, then call up **avl_fix** for maintenance.

Deletion is more complicated. If the target node has no subtree, just remove it straight, if it has one subtree, replace the node with that subtree. The problem arises when the node has both subtrees, we can't remove it straight, instead, we remove its sibling in the right subtree, and swap it with the detached sibling. Here is the function for removing a node:

```
// detach a node and returns the new root of the tree
static AVLNode *avl_del(AVLNode *node) {
    if (node->right == NULL) {
        // no right subtree, replace the node with the left subtree
        // link the left subtree to the parent
        AVLNode *parent = node->parent;
        if (node->left) {
            node->left->parent = parent;
        }
        if (parent) {
            // attach the left subtree to the parent
            (parent->left == node ? parent->left : parent->right) = node->left;
            return avl_fix(parent);
        } else {
            // removing root?
            return node->left;
        }
    } else {
        // swap the node with its next sibling
        AVLNode *victim = node->right;
        while (victim->left) {
            victim = victim->left;
        }
        AVLNode *root = avl_del(victim);

        *victim = *node;
        if (victim->left) {
            victim->left->parent = victim;
        }
        if (victim->right) {
            victim->right->parent = victim;
        }
```

```
        AVLNode *parent = node->parent;
        if (parent) {
            (parent->left == node ? parent->left : parent->right) = victim;
            return root;
        } else {
            // removing root?
            return victim;
        }
    }
}
```

This is the generic function for removing nodes from a binary tree, with the AVL-tree-specific **`avl_fix`**.

Readers with experiences with the RB tree may notice how small and simple the AVL tree implementation is. The maintenance code for RB tree node deletion is significantly more complicated than the insertion; while the AVL tree uses the same function **`avl_fix`** for both insertion and deletion, this symmetry greatly reduces the efforts required to code an AVL tree.

The AVL tree is significantly more complicated than the hashtable we coded before. Thus, we need to invest more time on testing. The testing code also demonstrates the usage of those AVL tree functions.

Here are our testing data types. If you are not familiar with intrusive data structures, read the hashtable chapter.

```
struct Data {
    AVLNode node;
    uint32_t val = 0;
};

struct Container {
    AVLNode *root = NULL;
};
```

The insertion code:

```
static void add(Container &c, uint32_t val) {
    Data *data = new Data();
    avl_init(&data->node);
    data->val = val;

    if (!c.root) {
        c.root = &data->node;
        return;
    }

    AVLNode *cur = c.root;
    while (true) {
        AVLNode **from =
            (val < container_of(cur, Data, node)->val)
            ? &cur->left : &cur->right;
        if (!*from) {
            *from = &data->node;
            data->node.parent = cur;
            c.root = avl_fix(&data->node);
            break;
        }
        cur = *from;
    }
}
```

This demonstrates the deletion of nodes:

```
static bool del(Container &c, uint32_t val) {
    AVLNode *cur = c.root;
    while (cur) {
        uint32_t node_val = container_of(cur, Data, node)->val;
        if (val == node_val) {
            break;
        }
        cur = val < node_val ? cur->left : cur->right;
    }
    if (!cur) {
```

```
        return false;
    }

    c.root = avl_del(cur);
    delete container_of(cur, Data, node);
    return true;
}
```

Here is the function for verifying the correctness of the tree structure:

```
static void avl_verify(AVLNode *parent, AVLNode *node) {
    if (!node) {
        return;
    }

    assert(node->parent == parent);
    avl_verify(node, node->left);
    avl_verify(node, node->right);

    assert(node->cnt == 1 + avl_cnt(node->left) + avl_cnt(node->right));

    uint32_t l = avl_depth(node->left);
    uint32_t r = avl_depth(node->right);
    assert(l == r || l + 1 == r || l == r + 1);
    assert(node->depth == 1 + max(l, r));

    uint32_t val = container_of(node, Data, node)->val;
    if (node->left) {
        assert(node->left->parent == node);
        assert(container_of(node->left, Data, node)->val <= val);
    }
    if (node->right) {
        assert(node->right->parent == node);
        assert(container_of(node->right, Data, node)->val >= val);
    }
}
```

Code for comparing the contents of AVL tree with the expected data:

```
static void extract(AVLNode *node, std::multiset<uint32_t> &extracted) {
    if (!node) {
        return;
    }
    extract(node->left, extracted);
    extracted.insert(container_of(node, Data, node)->val);
    extract(node->right, extracted);
}

static void container_verify(
    Container &c, const std::multiset<uint32_t> &ref)
{
    avl_verify(NULL, c.root);
    assert(avl_cnt(c.root) == ref.size());
    std::multiset<uint32_t> extracted;
    extract(c.root, extracted);
    assert(extracted == ref);
}
```

Don't forget to clean up after tests:

```
static void dispose(Container &c) {
    while (c.root) {
        AVLNode *node = c.root;
        c.root = avl_del(c.root);
        delete container_of(node, Data, node);
    }
}
```

Our test cases start with simple things:

```
    Container c;

    // some quick tests
```

```
    container_verify(c, {});
    add(c, 123);
    container_verify(c, {123});
    assert(!del(c, 124));
    assert(del(c, 123));
    container_verify(c, {});

    // sequential insertion
    std::multiset<uint32_t> ref;
    for (uint32_t i = 0; i < 1000; i += 3) {
        add(c, i);
        ref.insert(i);
        container_verify(c, ref);
    }
```

Then we throw in random operations:

```
    // random insertion
    for (uint32_t i = 0; i < 100; i++) {
        uint32_t val = (uint32_t)rand() % 1000;
        add(c, val);
        ref.insert(val);
        container_verify(c, ref);
    }

    // random deletion
    for (uint32_t i = 0; i < 200; i++) {
        uint32_t val = (uint32_t)rand() % 1000;
        auto it = ref.find(val);
        if (it == ref.end()) {
            assert(!del(c, val));
        } else {
            assert(del(c, val));
            ref.erase(it);
        }
        container_verify(c, ref);
    }
```

Some more targeted tests. Given a tree of a certain size, perform insertion/deletion at every possible position.

```
static void test_insert(uint32_t sz) {
    for (uint32_t val = 0; val < sz; ++val) {
        Container c;
        std::multiset<uint32_t> ref;
        for (uint32_t i = 0; i < sz; ++i) {
            if (i == val) {
                continue;
            }
            add(c, i);
            ref.insert(i);
        }
        container_verify(c, ref);

        add(c, val);
        ref.insert(val);
        container_verify(c, ref);
        dispose(c);
    }
}

static void test_remove(uint32_t sz) {
    for (uint32_t val = 0; val < sz; ++val) {
        Container c;
        std::multiset<uint32_t> ref;
        for (uint32_t i = 0; i < sz; ++i) {
            add(c, i);
            ref.insert(i);
        }
        container_verify(c, ref);

        assert(del(c, val));
        ref.erase(val);
        container_verify(c, ref);
        dispose(c);
    }
```

```
}
```

```
    // insertion/deletion at various positions
    for (uint32_t i = 0; i < 200; ++i) {
        test_insert(i);
        test_remove(i);
    }
```

With the help of those test cases, the author did found and fixed a couple of mistakes while writing this chapter.

Exercises:

1. While there is not much code for our AVL tree, this AVL tree implementation is probably not a very efficient one. Our code contains some reductant pointer updates, which might be a source of optimization. Also, we don't need to store the height value for balancing, it is possible to store the height difference instead. Research and explore efficient AVL tree implementations.
2. Can you create more test cases? The test cases presented in this chapter are unlikely to be sufficient.

> - avl.cpp
> - test_avl.cpp

11. The AVL Tree and the Sorted Set

Based on the AVL tree in the last chapter, the sorted set data structure can be easily added. The structure definition:

```
struct ZSet {
    AVLNode *tree = NULL;
    HMap hmap;
};

struct ZNode {
    AVLNode tree;
    HNode hmap;
    double score = 0;
    size_t len = 0;
    char name[0];
};

static ZNode *znode_new(const char *name, size_t len, double score) {
    ZNode *node = (ZNode *)malloc(sizeof(ZNode) + len);
    assert(node);   // not a good idea in real projects
    avl_init(&node->tree);
    node->hmap.next = NULL;
    node->hmap.hcode = str_hash((uint8_t *)name, len);
    node->score = score;
    node->len = len;
    memcpy(&node->name[0], name, len);
    return node;
}
```

The sorted set is a sorted list of pairs of `(score, name)` that supports query or update by the sorting key, or by the `name`. It's a combination of the AVL tree and hashtable, and the pair node belongs to both, which demonstrates the flexibility of intrusive data structures. The `name` string is embedded at the end of the pair node, in the hope of saving up some space overheads.

The function for tree insertion is roughly the same as the testing code seen from the previous chapter:

```
// insert into the AVL tree
static void tree_add(ZSet *zset, ZNode *node) {
    if (!zset->tree) {
        zset->tree = &node->tree;
        return;
    }

    AVLNode *cur = zset->tree;
    while (true) {
        AVLNode **from = zless(&node->tree, cur) ? &cur->left : &cur->right;
        if (!*from) {
            *from = &node->tree;
            node->tree.parent = cur;
            zset->tree = avl_fix(&node->tree);
            break;
        }
        cur = *from;
    }
}
```

zless is the helper function for comparing two pairs:

```
// compare by the (score, name) tuple
static bool zless(
    AVLNode *lhs, double score, const char *name, size_t len)
{
    ZNode *zl = container_of(lhs, ZNode, tree);
    if (zl->score != score) {
        return zl->score < score;
    }
    int rv = memcmp(zl->name, name, min(zl->len, len));
    if (rv != 0) {
        return rv < 0;
    }
```

```
    return zl->len < len;
}

static bool zless(AVLNode *lhs, AVLNode *rhs) {
    ZNode *zr = container_of(rhs, ZNode, tree);
    return zless(lhs, zr->score, zr->name, zr->len);
}
```

The insertion/update subroutines:

```
// update the score of an existing node (AVL tree reinsertion)
static void zset_update(ZSet *zset, ZNode *node, double score) {
    if (node->score == score) {
        return;
    }
    zset->tree = avl_del(&node->tree);
    node->score = score;
    avl_init(&node->tree);
    tree_add(zset, node);
}

// add a new (score, name) tuple, or update the score of the existing tuple
bool zset_add(ZSet *zset, const char *name, size_t len, double score) {
    ZNode *node = zset_lookup(zset, name, len);
    if (node) {
        zset_update(zset, node, score);
        return false;
    } else {
        node = znode_new(name, len, score);
        hm_insert(&zset->hmap, &node->hmap);
        tree_add(zset, node);
        return true;
    }
}

// lookup by name
ZNode *zset_lookup(ZSet *zset, const char *name, size_t len) {
```

```
    // just a hashtable look up
    // code omitted...
}
```

Here is the primary use case of sorted sets: the range query.

```
// find the (score, name) tuple that is greater or equal to the argument,
// then offset relative to it.
ZNode *zset_query(
    ZSet *zset, double score, const char *name, size_t len, int64_t offset)
{
    AVLNode *found = NULL;
    AVLNode *cur = zset->tree;
    while (cur) {
        if (zless(cur, score, name, len)) {
            cur = cur->right;
        } else {
            found = cur;    // candidate
            cur = cur->left;
        }
    }

    if (found) {
        found = avl_offset(found, offset);
    }
    return found ? container_of(found, ZNode, tree) : NULL;
}
```

The range query is just a regular binary tree look-up, followed by an offset operation. The offset operation is what makes the sorted set special, it is not a regular binary tree walk.

Let's review the **`AVLNode`**:

```
struct AVLNode {
    uint32_t depth = 0;
    uint32_t cnt = 0;
    AVLNode *left = NULL;
```

```
    AVLNode *right = NULL;
    AVLNode *parent = NULL;
};
```

It has an extra **cnt** field (the size of the tree), which is not explained in the previous chapter. It is used by the **avl_offset** function:

```
// offset into the succeeding or preceding node.
// note: the worst-case is O(log(n)) regardless of how long the offset is.
AVLNode *avl_offset(AVLNode *node, int64_t offset) {
    int64_t pos = 0;    // relative to the starting node
    while (offset != pos) {
        if (pos < offset && pos + avl_cnt(node->right) >= offset) {
            // the target is inside the right subtree
            node = node->right;
            pos += avl_cnt(node->left) + 1;
        } else if (pos > offset && pos - avl_cnt(node->left) <= offset) {
            // the target is inside the left subtree
            node = node->left;
            pos -= avl_cnt(node->right) + 1;
        } else {
            // go to the parent
            AVLNode *parent = node->parent;
            if (!parent) {
                return NULL;
            }
            if (parent->right == node) {
                pos -= avl_cnt(node->left) + 1;
            } else {
                pos += avl_cnt(node->right) + 1;
            }
            node = parent;
        }
    }
    return node;
}
```

With the size information embedded in the node, we can determine whether the offset target is inside a subtree or not. The offset operation runs in two phases: firstly, it walks up along the tree if the target is not in a subtree, then it walks down the tree, narrowing the distance until the target is met. The worst-case is **`O(log(n))`** regardless of how long the offset is, which is better than offsetting by walking to the succeeding node one by one (best-case of **`O(offset)`**). The real Redis project uses a similar technique for skip lists.

It is a good idea to stop and test the new **`avl_offset`** function now.

```
static void test_case(uint32_t sz) {
    Container c;
    for (uint32_t i = 0; i < sz; ++i) {
        add(c, i);
    }

    AVLNode *min = c.root;
    while (min->left) {
        min = min->left;
    }
    for (uint32_t i = 0; i < sz; ++i) {
        AVLNode *node = avl_offset(min, (int64_t)i);
        assert(container_of(node, Data, node)->val == i);

        for (uint32_t j = 0; j < sz; ++j) {
            int64_t offset = (int64_t)j - (int64_t)i;
            AVLNode *n2 = avl_offset(node, offset);
            assert(container_of(n2, Data, node)->val == j);
        }
        assert(!avl_offset(node, -(int64_t)i - 1));
        assert(!avl_offset(node, sz - i));
    }

    dispose(c.root);
}
```

For now, we have implemented major functionalities of the sorted set. Let's add the sorted set type to our server.

```
enum {
    T_STR = 0,
    T_ZSET = 1,
};

// the structure for the key
struct Entry {
    struct HNode node;
    std::string key;
    std::string val;
    uint32_t type = 0;
    ZSet *zset = NULL;
};
```

The rest of the code is considered trivial, which will be omitted in the code listing.

Adding a Python script for testing new commands:

```
CASES = r'''
$ ./client zscore asdf n1
(nil)
$ ./client zquery xxx 1 asdf 1 10
(arr) len=0
(arr) end
# more cases...
'''

import shlex
import subprocess

cmds = []
outputs = []
lines = CASES.splitlines()
for x in lines:
    x = x.strip()
    if not x:
        continue
    if x.startswith('$ '):
        cmds.append(x[2:])
        outputs.append('')
    else:
        outputs[-1] = outputs[-1] + x + '\n'

assert len(cmds) == len(outputs)
for cmd, expect in zip(cmds, outputs):
    out = subprocess.check_output(shlex.split(cmd)).decode('utf-8')
    assert out == expect, f'cmd:{cmd} out:{out}'
```

Exercises:

1. The **`avl_offset`** function gives us the ability to query sorted set by rank, now do the reverse, given a node in an AVL tree, find its rank, with a worst-case of **`O(log(n))`**. (This is the **`zrank`** command.)
2. Another sorted set application: count the number of elements within a range. (also with a worst-case of **`O(log(n))`**.)
3. The **`11_server.cpp`** file already contains some sorted set commands, try adding more.

- 11_client.cpp
- 11_server.cpp
- avl.cpp
- avl.h
- common.h
- hashtable.cpp
- hashtable.h
- test_cmds.py
- test_offset.cpp
- zset.cpp
- zset.h

12. The Event Loop and Timers

There is one major thing missing in our server: timeouts. Every networked application needs to handle timeouts since the other side of the network can just disappear. Not only do ongoing IO operations like read/write need timeouts, but it is also a good idea to kick out idle TCP connections. To implement timeouts, the event loop must be modified since the **`poll`** is the only thing that is blocking.

Looking at our existing event loop code:

```
int rv = poll(poll_args.data(), (nfds_t)poll_args.size(), 1000);
```

The **`poll`** syscall takes a timeout argument, which imposes an upper bound of time spent on the **`poll`** syscall. The timeout value is currently an arbitrary value of 1000 ms. If we set the timeout value according to the timer, **`poll`** should wake up at the time it expires, or before that; then we have a chance to fire the timer in due time.

The problem is that we might have more than one timer, the timeout value of **`poll`** should be the timeout value of the nearest timer. Some data structure is needed for finding the nearest timer. The heap data structure is a popular choice for finding the min/max value and is often used for such purpose. Also, any data structure for sorting can be used. For example, we can use the AVL tree to order timers and possibly augment the tree to keep track of the minimum value.

Let's start by adding timers to kick out idle TCP connections. For each connection there is a timer, set to a fixed timeout into the future, every time there are IO activities on the connection, the timer is renewed to a fixed timeout. Notice that when we renew a timer, it becomes the most distant one; therefore, we can exploit this fact to simplify the data structure; a simple linked list is sufficient to keep the order of timers: the new or updated timer simply goes to the end of the list, and the list maintains sorted order. Also, operations on linked lists are **`O(1)`**, which is better than sorting data structures.

Defining the linked list is a trivial task:

```
struct DList {
    DList *prev = NULL;
    DList *next = NULL;
```

```
};

inline void dlist_init(DList *node) {
    node->prev = node->next = node;
}

inline bool dlist_empty(DList *node) {
    return node->next == node;
}

inline void dlist_detach(DList *node) {
    DList *prev = node->prev;
    DList *next = node->next;
    prev->next = next;
    next->prev = prev;
}

inline void dlist_insert_before(DList *target, DList *rookie) {
    DList *prev = target->prev;
    prev->next = rookie;
    rookie->prev = prev;
    rookie->next = target;
    target->prev = rookie;
}
```

get_monotonic_usec is the function for getting the time. Note that the timestamp must be monotonic. Timestamp jumping backward can cause all sorts of troubles in computer systems.

```
static uint64_t get_monotonic_usec() {
    timespec tv = {0, 0};
    clock_gettime(CLOCK_MONOTONIC, &tv);
    return uint64_t(tv.tv_sec) * 1000000 + tv.tv_nsec / 1000;
}
```

The next step is adding the list to the server and the connection struct.

```
// global variables
static struct {
    HMap db;
    // a map of all client connections, keyed by fd
    std::vector<Conn *> fd2conn;
    // timers for idle connections
    DList idle_list;
} g_data;
```

```
struct Conn {
    int fd = -1;
    uint32_t state = 0;     // either STATE_REQ or STATE_RES
    // buffer for reading
    size_t rbuf_size = 0;
    uint8_t rbuf[4 + k_max_msg];
    // buffer for writing
    size_t wbuf_size = 0;
    size_t wbuf_sent = 0;
    uint8_t wbuf[4 + k_max_msg];
    uint64_t idle_start = 0;
    // timer
    DList idle_list;
};
```

An overview of the modified event loop:

```
int main() {
    // some initializations
    dlist_init(&g_data.idle_list);

    int fd = socket(AF_INET, SOCK_STREAM, 0);
    // bind, listen & other miscs
    // code omitted...
```

```
    // the event loop
    std::vector<struct pollfd> poll_args;
    while (true) {
        // prepare the arguments of the poll()
        // code omitted...

        // poll for active fds
        int timeout_ms = (int)next_timer_ms();
        int rv = poll(poll_args.data(), (nfds_t)poll_args.size(), timeout_ms);
        if (rv < 0) {
            die("poll");
        }

        // process active connections
        for (size_t i = 1; i < poll_args.size(); ++i) {
            if (poll_args[i].revents) {
                Conn *conn = g_data.fd2conn[poll_args[i].fd];
                connection_io(conn);
                if (conn->state == STATE_END) {
                    // client closed normally, or something bad happened.
                    // destroy this connection
                    conn_done(conn);
                }
            }
        }

        // handle timers
        process_timers();

        // try to accept a new connection if the listening fd is active
        if (poll_args[0].revents) {
            (void)accept_new_conn(fd);
        }
    }

    return 0;
}
```

A couple of things were modified:

1. The timeout argument of **`poll`** is calculated by the **`next_timer_ms`** function.
2. The code for destroying a connection was moved to the **`conn_done`** function.
3. Added the **`process_timers`** function for firing timers.
4. Timers are updated in **`connection_io`** and initialized in **`accept_new_conn`**.

The **`next_timer_ms`** function takes the first (nearest) timer from the list and uses it the calculate the timeout value of **`poll`**.

```
const uint64_t k_idle_timeout_ms = 5 * 1000;

static uint32_t next_timer_ms() {
    if (dlist_empty(&g_data.idle_list)) {
        return 10000;   // no timer, the value doesn't matter
    }

    uint64_t now_us = get_monotonic_usec();
    Conn *next = container_of(g_data.idle_list.next, Conn, idle_list);
    uint64_t next_us = next->idle_start + k_idle_timeout_ms * 1000;
    if (next_us <= now_us) {
        // missed?
        return 0;
    }

    return (uint32_t)((next_us - now_us) / 1000);
}
```

At each iteration of the event loop, the list is checked in order to fire timers in due time.

```
static void process_timers() {
    uint64_t now_us = get_monotonic_usec();
    while (!dlist_empty(&g_data.idle_list)) {
        Conn *next = container_of(g_data.idle_list.next, Conn, idle_list);
        uint64_t next_us = next->idle_start + k_idle_timeout_ms * 1000;
        if (next_us >= now_us + 1000) {
            // not ready, the extra 1000us is for the ms resolution of poll()
            break;
```

```
        }

        printf("removing idle connection: %d\n", next->fd);
        conn_done(next);
    }
}
```

Timers are updated in the **`connection_io`** function:

```
static void connection_io(Conn *conn) {
    // waked up by poll, update the idle timer
    // by moving conn to the end of the list.
    conn->idle_start = get_monotonic_usec();
    dlist_detach(&conn->idle_list);
    dlist_insert_before(&g_data.idle_list, &conn->idle_list);

    // do the work
    if (conn->state == STATE_REQ) {
        state_req(conn);
    } else if (conn->state == STATE_RES) {
        state_res(conn);
    } else {
        assert(0);  // not expected
    }
}
```

Timers are initialized in the **`accept_new_conn`** function:

```
static int32_t accept_new_conn(int fd) {
    // code omitted...

    // creating the struct Conn
    struct Conn *conn = (struct Conn *)malloc(sizeof(struct Conn));
    if (!conn) {
        close(connfd);
        return -1;
```

```
    }
    conn->fd = connfd;
    conn->state = STATE_REQ;
    conn->rbuf_size = 0;
    conn->wbuf_size = 0;
    conn->wbuf_sent = 0;
    conn->idle_start = get_monotonic_usec();
    dlist_insert_before(&g_data.idle_list, &conn->idle_list);
    conn_put(g_data.fd2conn, conn);
    return 0;
}
```

Don't forget to remove the connection from the list when done:

```
static void conn_done(Conn *conn) {
    g_data.fd2conn[conn->fd] = NULL;
    (void)close(conn->fd);
    dlist_detach(&conn->idle_list);
    free(conn);
}
```

We can test the idle timeouts using the **nc** or **socat** command:

```
$ ./server
removing idle connection: 4
```

```
$ socat tcp:127.0.0.1:1234 -
```

The server should close the connection by 5s.

Exercises:

1. Add timeouts to IO operations (read & write).
2. Try to implement more generic timers using sorting data structures.

- 12_server.cpp
- avl.cpp
- avl.h
- common.h
- hashtable.cpp
- hashtable.h
- list.h
- zset.cpp
- zset.h

13. The Heap Data Structure and the TTL

The primary use of Redis is as cache servers, and one way to manage the size of the cache is through explicitly setting TTLs (time to live). TTLs can be implemented using timers. Unfortunately, timers in the last chapter are of fixed value (using linked lists); thus, a sorting data structure is needed for implementing arbitrary and mutable timeouts; and the heap data structure is a popular choice. Compared with the AVL tree we used before, the heap data structure has the advantage of using less space.

A quick review of the heap data structure:

1. A heap is a binary tree, packed into an array; and the layout of the tree is fixed. The parent-child relationship is implicit, pointers are not included in heap elements.
2. The only constraint on the tree is that parents are no bigger than their kids.
3. The value of an element can be updated. If the value changes:
    - Its value is bigger than before: it may be bigger than its kids, and if so, swap it with the smallest kid, so that the parent-child constraint is satisfied again. Now that one of the kids is bigger than before, continue this process until reaching a leave.
    - Its value is smaller: likewise, swap it with its parent until reaching the root.
4. New elements are added to the end of the array as leaves. Maintain the constraint as above.
5. When removing an element from a heap, replace it with the last element in the array, then maintain the constraint as if its value was updated.

The code listing begins:

```
struct HeapItem {
    uint64_t val = 0;
    size_t *ref = NULL;
};

// the structure for the key
struct Entry {
    struct HNode node;
    std::string key;
```

```
    std::string val;
    uint32_t type = 0;
    ZSet *zset = NULL;
    // for TTLs
    size_t heap_idx = -1;
};
```

The heap is used to order the timestamps, and the **`Entry`** is mutually linked with the timestamp. The **`heap_idx`** is the index of the corresponding **`HeapItem`**, and the **`ref`** points to the **`Entry`**. We are using the intrusive data structure again; the **`ref`** pointer points to the **`heap_idx`** field.

The parent-child relationship is fixed:

```
static size_t heap_parent(size_t i) {
    return (i + 1) / 2 - 1;
}

static size_t heap_left(size_t i) {
    return i * 2 + 1;
}

static size_t heap_right(size_t i) {
    return i * 2 + 2;
}
```

Swap with the parent when a kid is smaller than its parent. Note the **`heap_idx`** is updated through the **`ref`** pointer while swapping.

```
static void heap_up(HeapItem *a, size_t pos) {
    HeapItem t = a[pos];
    while (pos > 0 && a[heap_parent(pos)].val > t.val) {
        // swap with the parent
        a[pos] = a[heap_parent(pos)];
        *a[pos].ref = pos;
        pos = heap_parent(pos);
```

```
    }
    a[pos] = t;
    *a[pos].ref = pos;
}
```

Swapping with the smallest kid is similar.

```
static void heap_down(HeapItem *a, size_t pos, size_t len) {
    HeapItem t = a[pos];
    while (true) {
        // find the smallest one among the parent and their kids
        size_t l = heap_left(pos);
        size_t r = heap_right(pos);
        size_t min_pos = -1;
        size_t min_val = t.val;
        if (l < len && a[l].val < min_val) {
            min_pos = l;
            min_val = a[l].val;
        }
        if (r < len && a[r].val < min_val) {
            min_pos = r;
        }
        if (min_pos == (size_t)-1) {
            break;
        }
        // swap with the kid
        a[pos] = a[min_pos];
        *a[pos].ref = pos;
        pos = min_pos;
    }
    a[pos] = t;
    *a[pos].ref = pos;
}
```

The **`heap_update`** is the heap function for updating a position. It is used for updating, inserting, and deleting.

```
void heap_update(HeapItem *a, size_t pos, size_t len) {
    if (pos > 0 && a[heap_parent(pos)].val > a[pos].val) {
        heap_up(a, pos);
    } else {
        heap_down(a, pos, len);
    }
}
```

Add the heap to our server:

```
// global variables
static struct {
    HMap db;
    // a map of all client connections, keyed by fd
    std::vector<Conn *> fd2conn;
    // timers for idle connections
    DList idle_list;
    // timers for TTLs
    std::vector<HeapItem> heap;
} g_data;
```

Updating, adding, and removing a timer to the heap. Just call the **`heap_update`** after updating an element of the array.

```
// set or remove the TTL
static void entry_set_ttl(Entry *ent, int64_t ttl_ms) {
    if (ttl_ms < 0 && ent->heap_idx != (size_t)-1) {
        // erase an item from the heap
        // by replacing it with the last item in the array.
        size_t pos = ent->heap_idx;
        g_data.heap[pos] = g_data.heap.back();
        g_data.heap.pop_back();
        if (pos < g_data.heap.size()) {
            heap_update(g_data.heap.data(), pos, g_data.heap.size());
        }
        ent->heap_idx = -1;
```

```
    } else if (ttl_ms >= 0) {
        size_t pos = ent->heap_idx;
        if (pos == (size_t)-1) {
            // add an new item to the heap
            HeapItem item;
            item.ref = &ent->heap_idx;
            g_data.heap.push_back(item);
            pos = g_data.heap.size() - 1;
        }
        g_data.heap[pos].val = get_monotonic_usec() + (uint64_t)ttl_ms * 1000;
        heap_update(g_data.heap.data(), pos, g_data.heap.size());
    }
}
```

Removing the possible TTL timer when deleting an **`Entry`**:

```
static void entry_del(Entry *ent) {
    switch (ent->type) {
    case T_ZSET:
        zset_dispose(ent->zset);
        delete ent->zset;
        break;
    }
    entry_set_ttl(ent, -1);
    delete ent;
}
```

The **`next_timer_ms`** function is modified to use both idle timers and TTL timers.

```
static uint32_t next_timer_ms() {
    uint64_t now_us = get_monotonic_usec();
    uint64_t next_us = (uint64_t)-1;

    // idle timers
    if (!dlist_empty(&g_data.idle_list)) {
        Conn *next = container_of(g_data.idle_list.next, Conn, idle_list);
```

```
        next_us = next->idle_start + k_idle_timeout_ms * 1000;
    }

    // ttl timers
    if (!g_data.heap.empty() && g_data.heap[0].val < next_us) {
        next_us = g_data.heap[0].val;
    }

    if (next_us == (uint64_t)-1) {
        return 10000;   // no timer, the value doesn't matter
    }

    if (next_us <= now_us) {
        // missed?
        return 0;
    }
    return (uint32_t)((next_us - now_us) / 1000);
}
```

Adding TTL timers to the **process_timers** function:

```
static void process_timers() {
    // the extra 1000us is for the ms resolution of poll()
    uint64_t now_us = get_monotonic_usec() + 1000;

    // idle timers
    while (!dlist_empty(&g_data.idle_list)) {
        // code omitted...
    }

    // TTL timers
    const size_t k_max_works = 2000;
    size_t nworks = 0;
    while (!g_data.heap.empty() && g_data.heap[0].val < now_us) {
        Entry *ent = container_of(g_data.heap[0].ref, Entry, heap_idx);
        HNode *node = hm_pop(&g_data.db, &ent->node, &hnode_same);
        assert(node == &ent->node);
```

```
        entry_del(ent);
        if (nworks++ >= k_max_works) {
            // don't stall the server if too many keys are expiring at once
            break;
        }
    }
}
```

This is just checking the minimal value of the heap and removing keys. Note that we put a limit on the number of keys expired per event loop iteration; the limit is needed to prevent the server from stalling should there are too many keys expiring at once.

The command for updating and querying TTLs is straightforward to add:

```
static void do_expire(std::vector<std::string> &cmd, std::string &out) {
    int64_t ttl_ms = 0;
    if (!str2int(cmd[2], ttl_ms)) {
        return out_err(out, ERR_ARG, "expect int64");
    }

    Entry key;
    key.key.swap(cmd[1]);
    key.node.hcode = str_hash((uint8_t *)key.key.data(), key.key.size());

    HNode *node = hm_lookup(&g_data.db, &key.node, &entry_eq);
    if (node) {
        Entry *ent = container_of(node, Entry, node);
        entry_set_ttl(ent, ttl_ms);
    }
    return out_int(out, node ? 1: 0);
}
```

```
static void do_ttl(std::vector<std::string> &cmd, std::string &out) {
    Entry key;
    key.key.swap(cmd[1]);
```

```
    key.node.hcode = str_hash((uint8_t *)key.key.data(), key.key.size());

    HNode *node = hm_lookup(&g_data.db, &key.node, &entry_eq);
    if (!node) {
        return out_int(out, -2);
    }

    Entry *ent = container_of(node, Entry, node);
    if (ent->heap_idx == (size_t)-1) {
        return out_int(out, -1);
    }

    uint64_t expire_at = g_data.heap[ent->heap_idx].val;
    uint64_t now_us = get_monotonic_usec();
    return out_int(out, expire_at > now_us ? (expire_at - now_us) / 1000 : 0);
}
```

Exercises:

1. The heap-based timer adds **`O(log(n))`** operations to the server, which might be a bottleneck for a sufficiently large number of keys. Can you think of optimizations for a large number of timers?
2. The real Redis does not use sorting for expiration, find out how it is done, and list the pros and cons of both approaches.

> - 13_server.cpp
> - avl.cpp
> - avl.h
> - common.h
> - hashtable.cpp
> - hashtable.h
> - heap.cpp
> - heap.h
> - list.h
> - test_heap.cpp
> - zset.cpp

> - zset.h

14. The Thread Pool & Asynchronous Tasks

There is a flaw in our server since the introduction of the sorted set data type: the deletion of keys. If the size of a sorted set is huge, it can take a long time to free its nodes and the server is stalled during the destruction of the key. This can be easily fixed by using multi-threading to move the destructor away from the main thread.

Firstly, we introduce the "thread pool", which is literally a pool of threads. The thread from the pool consumes tasks from a queue and executes them. It is trivial to code a multi-producer multi-consumer queue using **`pthread`** APIs. (Although there is only a single producer in our case.)

The relevant **`pthread`** primitives are **`pthread_mutex_t`** and **`pthread_cond_t`**; they are called the mutex and the condition variable respectively. If you are unfamiliar with them, it is advised to get some education on multi-threading after reading this chapter. (Such as manpages of the **`pthread`** APIs, textbooks on operating systems, online courses, etc.)

Here is a really short introduction to the two **`pthread`** primitives:

- The queue is accessed by multiple threads (both the producer and consumers), so it needs the protection of a mutex, obviously.
- The consumer threads should be sleeping when idle, and only be waken up when the queue is not empty, this is the job of the condition variable.

The thread pool data type is defined as follows:

```
struct Work {
    void (*f)(void *) = NULL;
    void *arg = NULL;
};

struct TheadPool {
    std::vector<pthread_t> threads;
    std::deque<Work> queue;
    pthread_mutex_t mu;
    pthread_cond_t not_empty;
};
```

The **thread_pool_init** is for initialization and starting threads. **pthread** types are initialized by **pthread_xxx_init** functions and the **pthread_create** starts a thread with the target function **worker**.

```
void thread_pool_init(TheadPool *tp, size_t num_threads) {
    assert(num_threads > 0);

    int rv = pthread_mutex_init(&tp->mu, NULL);
    assert(rv == 0);
    rv = pthread_cond_init(&tp->not_empty, NULL);
    assert(rv == 0);

    tp->threads.resize(num_threads);
    for (size_t i = 0; i < num_threads; ++i) {
        int rv = pthread_create(&tp->threads[i], NULL, &worker, tp);
        assert(rv == 0);
    }
}
```

The consumer code:

```
static void *worker(void *arg) {
    TheadPool *tp = (TheadPool *)arg;
    while (true) {
        pthread_mutex_lock(&tp->mu);
        // wait for the condition: a non-empty queue
        while (tp->queue.empty()) {
            pthread_cond_wait(&tp->not_empty, &tp->mu);
        }

        // got the job
        Work w = tp->queue.front();
        tp->queue.pop_front();
        pthread_mutex_unlock(&tp->mu);

        // do the work
        w.f(w.arg);
```

```
    }
    return NULL;
}
```

The producer code:

```
void thread_pool_queue(TheadPool *tp, void (*f)(void *), void *arg) {
    Work w;
    w.f = f;
    w.arg = arg;

    pthread_mutex_lock(&tp->mu);
    tp->queue.push_back(w);
    pthread_cond_signal(&tp->not_empty);
    pthread_mutex_unlock(&tp->mu);
}
```

The explanation:

1. For both the producer and consumers, the queue access code is surrounded by the **`pthread_mutex_lock`** and the **`pthread_mutex_unlock`**, only one thread can access the queue at once.
2. After a consumer acquired the mutex, check the queue:
    - If the queue is not empty, grab a job from the queue, release the mutex and do the work.
    - Otherwise, release the mutex and go to sleep, the sleep can be wakened later by the condition variable. This is accomplished via a single **`pthread_cond_wait`** call.
3. After the producer puts a job into the queue, the producer calls the **`pthread_cond_signal`** to wake up a potentially sleeping consumer.
4. After a consumer woken up from the **`pthread_cond_wait`**, the mutex is held again automatically. The consumer must check for the condition *again* after waking up, if the condition (a non-empty queue) is not satisfied, go back to sleep.

The use of the condition variable needs some more explanations: The **`pthread_cond_wait`** function is *always* inside a loop checking for the condition. This is because the condition

could be changed by other consumers before the wakening consumer grabs the mutex; the mutex is not transferred from the signaler to the to-be-waked consumer! It is probably a mistake if you see a condition variable used without a loop.

A concrete sequence to help you understand the use of condition variables:

1. The producer signals.
2. The producer releases the mutex.
3. Some consumer grabs the mutex and empties the queue.
4. A consumer wakes up from the producer's signal and grabs the mutex, but the queue is empty!

Note that the **`pthread_cond_signal`** doesn't need to be protected by the mutex, singaling after releasing the mutex is also correct.

The thread pool is done. Let's add that to our server:

```
// global variables
static struct {
    HMap db;
    // a map of all client connections, keyed by fd
    std::vector<Conn *> fd2conn;
    // timers for idle connections
    DList idle_list;
    // timers for TTLs
    std::vector<HeapItem> heap;
    // the thread pool
    TheadPool tp;
} g_data;
```

Inside the **`main`** function:

```
    // some initializations
    dlist_init(&g_data.idle_list);
    thread_pool_init(&g_data.tp, 4);
```

The **`entry_del`** function is modified: It will put the destruction of large sorted sets into the thread pool. And the thread pool is only for the large ones since multi-threading has some overheads too.

```
// deallocate the key immediately
static void entry_destroy(Entry *ent) {
    switch (ent->type) {
    case T_ZSET:
        zset_dispose(ent->zset);
        delete ent->zset;
        break;
    }
    delete ent;
}

static void entry_del_async(void *arg) {
    entry_destroy((Entry *)arg);
}

// dispose the entry after it got detached from the key space
static void entry_del(Entry *ent) {
    entry_set_ttl(ent, -1);

    const size_t k_large_container_size = 10000;
    bool too_big = false;
    switch (ent->type) {
    case T_ZSET:
        too_big = hm_size(&ent->zset->hmap) > k_large_container_size;
        break;
    }

    if (too_big) {
        thread_pool_queue(&g_data.tp, &entry_del_async, ent);
    } else {
        entry_destroy(ent);
    }
}
```

Exercises:

1. The semaphore is often introduced as a multi-threading primitive instead of the condition variable and the mutex. Try to implement the thread pool using the

semaphore.

2. Some fun exercises to help you understand these primitives further:
    1. Implement the mutex using the semaphore. (Trivial)
    2. Implement the semaphore using the condition variable. (Easy)
    3. Implement the condition variable using only mutexes. (Intermediate)
    4. Now that you know these primitives are somewhat equivalent, why should you prefer one to another?

> - 14_server.cpp
> - avl.cpp
> - avl.h
> - common.h
> - hashtable.cpp
> - hashtable.h
> - heap.cpp
> - heap.h
> - list.h
> - thread_pool.cpp
> - thread_pool.h
> - zset.cpp
> - zset.h

APPENDIXES

A1: Hints to Exercises

08. Data Structure: Hashtables

> Q: Our hashtable triggers resizing when the load factor is too high, should we also shrink the hashtable when the load factor is too low? Can the shrinking be performed automatically?

Hints:

Hashtable shrinking is not done automatically in practice. Many real-world usage patterns are periodic, shrinking is not always clearly beneficial. Besides, shrinking does not always return the memory to OS, this is dependent on many factors such as the malloc implementation and the level of memory fragmentation; the outcome of shrinking is not easily predictable.

10. The AVL Tree: Implementation & Testing

> Q: Can you create more test cases? The test cases presented in this chapter are unlikely to be sufficient.

Hints:

Our existing test cases enumerate AVL trees of various sizes. However, given a tree of a particular size, there are many possible configurations, we can go further by enumerating tree configurations too.

Also, for more complicated code, it is helpful to use profiling tools to check whether the test cases give full coverage of the target code. Non-full coverage indicates bugs in test cases or target code.

Another technique that is worth mentioning is Fuzz Testing.

11. The AVL Tree and the Sorted Set

> Q: The `avl_offset` function gives us the ability to query sorted set by rank, now do the reverse, given a node in an AVL tree, find its rank, with a worst-case of `O(log(n))`. (This is the `zrank` command.)

Hints:

The rank of a node is related to the rank of its parent. And the rank of the root is obvious.

> Q: Another sorted set application: count the number of elements within a range. (also with a worst-case of `O(log(n))`.)

Hints:

Use the rank of the node.

13. The Heap Data Structure and the TTL

> Q: The heap-based timer adds `O(log(n))` operations to the server, which might be a bottleneck for a sufficiently large number of keys. Can you think of optimizations for a large number of timers?

Hints:

We can make the heap more cache friendly by using the n-ary tree instead of the binary tree. Some real-world project uses the quadtree which fits in the 64-byte cache line.

Also, in our case, the TTL timers don't have to be fired at the exact time. We can use a very coarse timestamp (such as round up to 1min resolution) for TTL timers, and keys with the same timestamp can share the same timer. This reduces the number of timers, but the timers are delayed so we need to check the real expiration time when accessing the key.

> Q: The real Redis does not use sorting for expiration, find out how it is done, and list the pros and cons of both approaches.

Hints:

Taking the idea that keys don't need to be expired at the exact time, the read Redis samples the key space at random to find dead keys. The higher the ratio of dead keys, the easier to find and eliminate them.

The pros of this approach are:

1. It doesn't require extra space.
2. The concept is simple, and the implementation is easy.

The cons:

1. It requires that keys with a TTL should not be mixed with keys without a TTL, otherwise, the non-TTL keys interfere with the sampling, making it harder to find dead keys. This can be a source of surprise for operators.
2. While the concept is simple, the implementation uses some heuristics to determine the rate of the sampling. If the heuristic is not tuned properly, in a worse-case, the server might not be removing dead keys fast enough, leading to excessive memory usage, which may frustrate the operator.

14. The Thread Pool & Asynchronous Tasks

> Q: Implement the condition variable using only mutexes. (Intermediate)

Hints:

You need to figure out how to sleep and wake up using mutex first. Then you need to keep track of a list of sleepers in the condition variable so that you can wake up them later.