CS494 -- Lab 2: A realistic B-Tree Lab

- CS494
- Fall, 2020
- James S. Plank
- This file: **http://web.eecs.utk.edu/~jplank/plank/classes/cs494/Labs/Lab-3-B-Tree**
- Lab Directory: **/home/plank/cs494/Labs/Lab-3-B-Tree**

Please keep this in mind when you are working on the lab. For grading, you need to compile **bin/random_tester_1.cpp** and **bin/random_tester_2.cpp** with your implementation of B-trees. *Double-check youself and make sure that you are doing it.* In multiple semesters, students have copied my **bin/random_tester_1** and **bin/random_tester_2** rather than compile their own. Of course, mine pass the gradescripts. You want to make sure that *yours* are passing the gradescripts.

This is a three-week lab, and it is every bit of a three week lab. It requires attention to detail and disciplined testing. It is the hardest lab that I give, and for that reason, you should allocate time to do it. *Don't think that because you are a better programmer than your friends, that you can do this lab in a day or two. I have seen too many excellent students flail on this lab because they underestimated the time to do it. I mean really good students!*

On the flip side, I think that this is the most rewarding lab that I give. When you finish, and your B-tree is working as it should, it's a really great feeling!

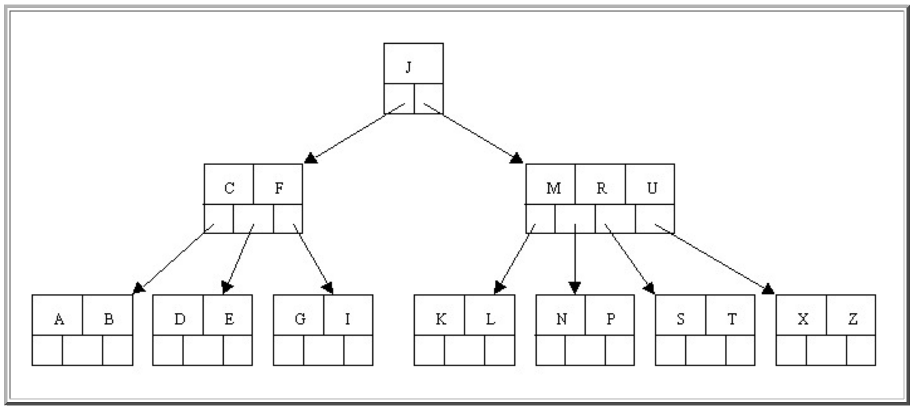
Some completion stats:

- 2015: 15 of 21 students completed (71.4%)
- 2016: 18 of 23 students completed (78.3%)
- 2017: 16 of 24 students completed (66.7%)
- 2018: 17 of 42 students completed (40.5%)
- 2019: 19 of 30 students completed (64.3%)
- 2020: 15 of 18 students completed (83.3%)

I'd *really* like to see some high completion stats this year -- please start on it early!

The reference material that I use is on this web page, by David Carlson and Isidore Minerd, which I think is very well done. I supplement it with my own drawings, which I use when I teach the class. These are in this file.

You are going to implement the "B+ Tree" variant, where internal nodes only hold keys and pointers to other nodes, and external nodes hold keys and pointers to values. One change from the St.Vincent description is that the value of a key in an internal node is going to be held in the largest val pointer in the key's predecessor subtree. Let me show an example, lifted from their notes:



In their example, the val pointer for J is the pointer to the left of K. In our trees, the val pointer for J will be the pointer to the right of I. Similarly:

- The val pointer for F is the pointer to the right of E.
- The val pointer for C is the pointer to the right of B.
- The val pointer for M is the pointer to the right of L.
- The val pointer for R is the pointer to the right of P.
- The val pointer for U is the pointer to the right of T.
- The val pointer to the right of Z is unused.

(As a corollary to this, when you delete an internal node, you swap it with the previous node in the tree, not the subsequent one. Since we are not doing deletion, that doesn't matter.....).

The last page of B-Tree.pdf also has a tree pictured with where its val pointers should be pointing.

Jdisk: A disk emulator

We are using the same disk emulator, **jdisk** as in The FAT lab. Please see that lab writeup for information on **jdisk.c/jdisk.h/jdisk_tester**. Please also see that lab for information on how to view and modify binary files in VI and using **xxd**.

To start out

Please do the following:

```
UNIX> cp -r /home/jplank/cs494/labs/Lab-3-B-Tree/src .
UNIX> cp -r /home/jplank/cs494/labs/Lab-3-B-Tree/include .
UNIX> cp /home/jplank/cs494/labs/Lab-3-B-Tree/makefile .
UNIX> cp /home/jplank/cs494/labs/Lab-3-B-Tree/tree* .
UNIX> mkdir obj
UNIX> mkdir bin
```

You are now ready to start the lab.

What you have to write: src/b_tree.c

Your job is to write **src/b_tree.c**. Its job is to implement the interface in **include/b_tree.h**:

```
#ifndef _BP_TREE_
#define _BP_TREE_

#include "jdisk.h"

void *b_tree_create(char *filename, long size, int key_size);
void *b_tree_attach(char *filename);

unsigned int b_tree_insert(void *b_tree, void *key, void *record);
unsigned int b_tree_find(void *b_tree, void *key);

void *b_tree_disk(void *b_tree);
int b_tree_key_size(void *b_tree);
void b_tree_print_tree(void *b_tree);
#endif
```

What you are going to do is implement B-trees on top of jdisks. The keys will be buffers of exactly **key_size** bytes, which is defined when you create a btree. The vals will be buffers of exactly JDISK_SECTOR_SIZE bytes. Each node of the tree will fit into a sector of the disk.

The jdisks *must* have a specific format. That means that the jdisks that your programs create must be readable by my btree programs and vice versa (so long as they have the same sizes for longs and the same endian-ness). They don't have to be identical to mine. They just have to work. Here is the format:

Sector 0

Sector zero can have anything in it, so long as the first 16 (or 12) bytes contain the following:

- Bytes 0-3: The key size as an integer.
- Bytes 4-7: The LBA of the sector that holds the root node of the B-tree.
- Bytes 8-15 (or 8-11 if longs are 4 bytes): The first free LBA on the jdisk. You are going to allocate sectors consecutively from sector 0, and since you never deallocate a sector (our B-Trees don't allow deletion), you can keep track of the free sectors with a single number.
- Yes, the remaining 1008 bytes are wasted. You can use them, but since your program has to be interoperable with mine and others, their values will be ignored.

Let's stop there. Take a look at a file that is a jdisk for a B-Tree:

```
UNIX> ls -l tree-1.jdisk
-rw-r--r--. 1 jplank jplank 2048000 Sep 18 19:46 tree-1.jdisk
UNIX> xxd -g 1 -len 16 tree-1.jdisk
0000000: 17 00 00 00 29 00 00 00 f1 01 00 00 00 00 00 00  ....)..........  # Remember, you need to view these groupings of bytes as numbers in little endian.
UNIX> xxd -g 4 -e -len 16 tree-1.jdisk                                    # Or if you have -e, that makes life easier.
00000000: 00000017 00000029 000001f1 00000000  ....)..........
UNIX>
```

The file is roughly 2MB, composed of 2,000 sectors. When we look at the first 16 bytes, we see the numbers in little endian format (which is the format of our current Intel processors). The key size is 0x17, or 23 bytes. The LBA of the root node is 0x29, and the first free sector is LBA 0x1f1 = 497.

Nodes

Now, the format of a sector holding a node of the tree is as follows. The first byte is a zero or one, specifying whether the node is external (0) or internal (1). The next byte says how many keys are in the node. How many keys can you fit into a node? The answer is (JDISK_SECTOR_SIZE - 6) / (key_size + 4). You'll see why in a minute. Let's call that value MAXKEY. Keys must be between 4 and 254 bytes (inclusive). Therefore, even if keys are four bytes, you can fit the number of keys into an unsigned char.

The next MAXKEY * key_size bytes are the keys. Then there can be some wasted bytes. The last (MAXKEY+1)*4 bytes are the LBA's, which are the pointers of the B-Trees. If the node is internal, then they are the LBA's of nodes that are pointed to by the node. If the node is external, then they are the LBA's of vals. If there are *nkeys* keys in the node, then there are *nkeys+1* LBA's.

Data (the vals)

Data is stored in a sector. The data *must* be JDISK_SECTOR_SIZE bytes.

A detailed example

Let's explore this a little. Let's try **tree-2.jdisk**:

```
UNIX> ls -l tree-2.jdisk
-rw-r--r--. 1 jplank jplank 20480 Sep 18 19:46 tree-2.jdisk
UNIX> xxd -g 1 -len 16 tree-2.jdisk
0000000: c8 00 00 00 08 00 00 00 0c 00 00 00 00 00 00 00  ................
UNIX> xxd -g 4 -e -len 16 tree-2.jdisk
00000000: 000000c8 00000008 0000000c 00000000  ................
UNIX>
```

This is a file with 20 sectors, of which 12 (0x0c) are currently in use. The key size is 200 (0xc8). The LBA of the root node is 8. Let's take a look at the first 202 bytes of that block:

```
UNIX> echo '8 1024 * p' | dc
8192
```

```
UNIX> xxd -g 1 -s 8192 -len 202 tree-2.jdisk
0002000: 01 01 45 6c 69 00 00 00 00 00 00 00 00 00 00 00  ..Eli...........
0002010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0002020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0002030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0002040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0002050: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0002060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0002070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0002080: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0002090: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00020a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00020b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00020c0: 00 00 00 00 00 00 00 00 00 00                    ..........
UNIX>
```

So, this is an internal node, because the first byte is one. It holds one key, because the second byte is also one. The first key is in the next 200 bytes -- as you see, they are all zeros except for the first three. I know they hold a string (because I created this file), and you can see from the output to **xxd**, that the string is "Eli."

Just a note here about keys that are strings, and the program **bin/jdisk_test**, if you want to use it instead of or in addition to **xxd**. Since the strings are null terminated, **jdisk_test** will print the string, regardless of whether you specify 3 characters or 200. In the example below, **jdisk_test** is reading 200 characters, but since the fourth character is '\0', it only prints out "Eli".

```
UNIX> bin/jdisk_test R tree-2.jdisk string 8194 3
Eli
UNIX> bin/jdisk_test R tree-2.jdisk string 8194 200
Eli
UNIX>
```

So, there is one key, which means that there are two pointers out of the node. Each of these pointers is an LBA of the sector holding the node to which the pointer points. How do we find these LBA's? Well, first, let's figure out what MAXKEY is: (1024 - 6) / (200+4) = 4.99. That means MAXKEY is four (and there is quite a bit of wasted space in our nodes. So it goes). Since a node can hold four keys, it can hold 5 LBA pointers. Those are in the last 5*4=20 bytes of the sector. Let's look at them:

```
UNIX> echo 8192+1024-20 | bc
9196
UNIX> xxd -g 1 -s 9196 -len 20 tree-2.jdisk
00023ec: 01 00 00 00 07 00 00 00 00 00 00 00 00 00 00 00  ................
00023fc: 00 00 00 00                                      ....
UNIX> xxd -g 4 -e -s 9196 -len 20 tree-2.jdisk
000023ec: 00000001 00000007 00000000 00000000  ................
000023fc: 00000000                             ....
UNIX>
```

So, the first pointer is to block 1, and then second is to block 7. Let's look at block 1:

```
UNIX> xxd -g 1 -s 1024 -len 32 tree-2.jdisk
0000400: 00 04 41 6c 65 78 69 73 00 00 00 00 00 00 00 00  ..Alexis........
0000410: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
UNIX>
```

It is an external node that holds four keys. The keys will start at bytes 1026 (1024+2), 1226 (1024+2+200), 1426 (1024+2+400) and 1626 (1024+2+600). You can see from **xxd** that the first key is "Alexsis". Here are the other three keys:

```
UNIX> xxd -g 1 -s 1226 -len 16 tree-2.jdisk
00004ca: 41 6c 6c 69 73 6f 6e 00 00 00 00 00 00 00 00 00  Allison.........
UNIX> xxd -g 1 -s 1426 -len 16 tree-2.jdisk
0000592: 43 61 6c 65 62 00 00 00 00 00 00 00 00 00 00 00  Caleb...........
UNIX> xxd -g 1 -s 1626 -len 16 tree-2.jdisk
000065a: 44 61 6e 69 65 6c 00 00 00 00 00 00 00 00 00 00  Daniel..........
UNIX>
```

Nice -- this is looking like the keys are all string-based (however, they are still 200 characters -- it just so happens that all of the characters after a string are byte 0x0).

Now, let's look at the LBA's, which will start 20 bytes from the end of the sector:

```
UNIX> xxd -g 1 -s 2028 -len 20 tree-2.jdisk
00007ec: 04 00 00 00 0b 00 00 00 0a 00 00 00 02 00 00 00  ................
00007fc: 06 00 00 00                                      ....
UNIX>
```

These are the vals:

- Alexis' val is the sector at LBA 4.
- Allison's val is the sector at LBA 11.
- Caleb's val is the sector at LBA 10.
- Daniel's val is the sector at LBA 2.
- And Eli's val is the sector at LBA 6. Remember how vals work for internal nodes!

If you examine these LBA's, you'll see that they are also strings, where all of the bytes after the strings are zero. Here, I'll prove it for LBA 4, and then we'll look at the strings in those five sectors:

```
UNIX> xxd -g 1 -s 4096 -len 1024 tree-2.jdisk
0001000: 47 79 72 6f 73 63 6f 70 65 00 00 00 00 00 00 00  Gyroscope.......
0001010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0001020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
# .....  Lots of lines of zeros
00013f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
UNIX> echo '1024 * 11' | bc
11264
UNIX> xxd -s 11264 -len 16 -g 1 tree-2.jdisk
0002c00: 45 6d 62 65 6c 6c 69 73 68 00 00 00 00 00 00 00  Embellish.......
UNIX> xxd -s 10240 -len 16 -g 1 tree-2.jdisk
0002800: 53 75 64 61 6e 65 73 65 00 00 00 00 00 00 00 00  Sudanese........
UNIX> xxd -s 2048 -len 16 -g 1 tree-2.jdisk
0000800: 53 68 61 64 6f 77 79 00 00 00 00 00 00 00 00 00  Shadowy.........
```

```
UNIX> echo '1024 * 6' | bc
6144
UNIX> xxd -s 6144 -len 16 -g 1 tree-2.jdisk
0001800: 45 69 64 65 72 00 00 00 00 00 00 00 00 00 00 00  Eider...........
UNIX>
```

So, now we know:

- Key: Alexis - Val: Gyroscope
- Key: Allison - Val: Embellish
- Key: Caleb - Val: Sudanese
- Key: Daniel - Val: Shadowy
- Key: Eli - Val: Eider

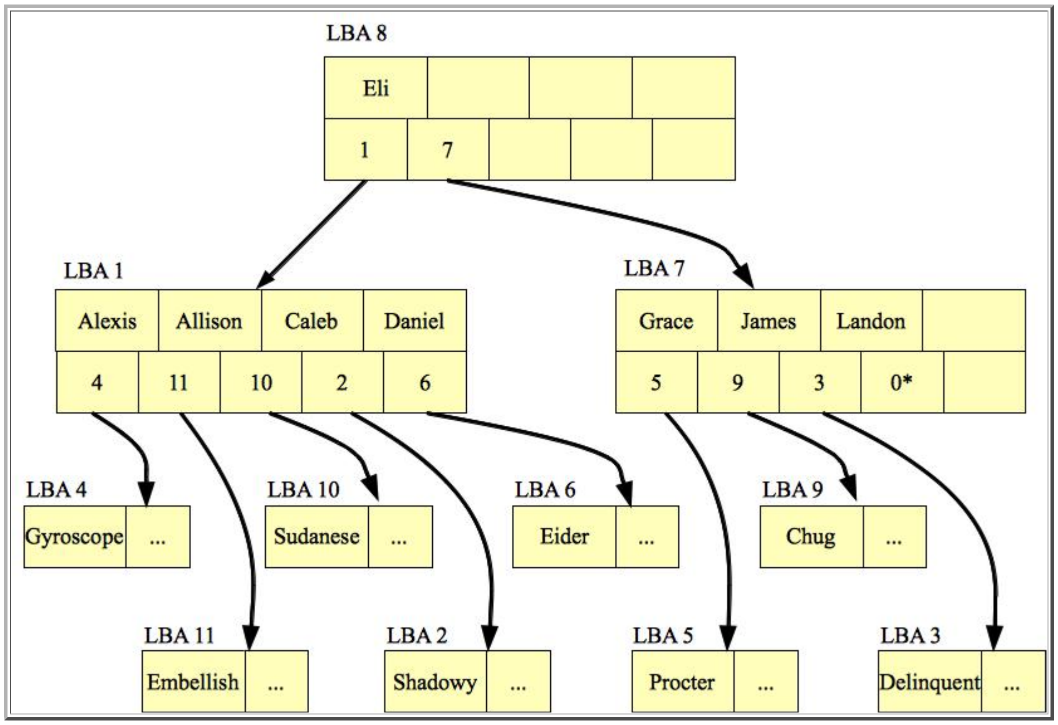
Let's look at LBA 7 to see the keys greater than "Eli" -- as you can see, this block is an external node with three keys:

```
UNIX> echo '7*1024' | bc
7168
UNIX> xxd -g 1 -s 7168 -len 16 tree-2.jdisk
0001c00: 00 03 47 72 61 63 65 00 00 00 00 00 00 00 00 00  ..Grace.........
UNIX> xxd -g 4 -e -s 8172 -len 20 tree-2.jdisk
00001fec: 00000005 00000009 00000003 00000000  ................
00001ffc: 00000000                             ....
UNIX> xxd -g 1 -s 7370 -len 16 tree-2.jdisk
0001cca: 4a 61 6d 65 73 00 00 00 00 00 00 00 00 00 00 00  James...........
UNIX> xxd -g 1 -s 7570 -len 16 tree-2.jdisk
0001d92: 4c 61 6e 64 6f 6e 00 00 00 00 00 00 00 00 00 00  Landon..........
UNIX>
```

Let's see the vals, which start 20 bytes from the end of sector 7:

```
UNIX> echo 7168+1024-20 | bc
8172
UNIX> xxd -g 1 -s 8172 -len 20 tree-2.jdisk
0001fec: 05 00 00 00 09 00 00 00 03 00 00 00 00 00 00 00  ................
0001ffc: 00 00 00 00                                      ....
UNIX> echo '5*1024' | bc
5120
UNIX> xxd -g 1 -s 5120 -len 16 tree-2.jdisk
0001400: 50 72 6f 63 74 65 72 00 00 00 00 00 00 00 00 00  Procter.........
UNIX> echo '9*1024' | bc
9216
UNIX> xxd -g 1 -s 9216 -len 16 tree-2.jdisk
0002400: 43 68 75 67 00 00 00 00 00 00 00 00 00 00 00 00  Chug............
UNIX> echo '3*1024' | bc
3072
UNIX> xxd -g 1 -s 3072 -len 16 tree-2.jdisk
0000c00: 44 65 6c 69 6e 71 75 65 6e 74 00 00 00 00 00 00  Delinquent......
UNIX>
```

So, armed with that information, we can draw our B-Tree as follows:



I have starred that last LBA, because it is unused.

The procedures in b_tree.h

Here is a description of the procedures that you have to write.

- **void *b_tree_create(char *filename, long size, int key_size);** This creates an empty btree with the given file size, key_size and filename. The empty btree will have a root node which is external and has zero keys. It returns a handle to the btree in a void *.
- **void *b_tree_attach(char *filename);** This opens the given btree file, which should have been created previously with **b_tree_create()**. Again, it returns a handle to the btree.
- **unsigned int b_tree_insert(void *b_tree, void *key, void *record);** In this procedure, **key** is a pointer to **key_size** bytes, and **record** is a pointer to JDISK_SECTOR_SIZE bytes. If the key is in the btree, then the procedure replaces the val with **record**, and returns the LBA of the val. If the key is not in the btree, then it is inserted, and the val for that key is set to **record**. In either case, the LBA of the val is returned. It will return 0 if our file is out of room. When this returns, the btree file is in the proper shape (in other words, **jdisk_write()** calls need to be made for all of the sectors that have been added or changed).

  I want to stress here that even though our examples above used null-terminated strings as keys, our btrees can take *any* keys that are **key_size** bytes. Use **memcmp()** for key comparison. (And use **memcpy()** to copy keys and vals to their respective homes if need be).
- **unsigned int b_tree_find(void *b_tree, void *key);** This finds the given key, and returns the LBA of the val. If the key is not in the tree, this returns 0.
- **void *b_tree_disk(void *b_tree);** This returns the jdisk pointer for the btree.
- **int key_size(void *b_tree);** This returns the key size.
- **void *b_tree_print_tree(void *b_tree);** This prints the tree -- see my examples for format. I'm *not* going to grade you on this. This can be a very useful procedure for debugging.

Useful program #1: b_tree_test

The program src/b_tree_test.c is a nice program to help you debug. It is called as follows:

```
bin/b_tree_test file [CREATE file_size key_size]
```

If you call it with "CREATE", then it creates a btree file with the given size and key size. If you don't call it with "CREATE", then it attaches to the preexisting btree file.

Once the file is created, it accepts three commands:

- "I key val" -- **Key** must be a string less than or equal to the key size, and **val** must be a string less than or equal to JDISK_SECTOR_SIZE. The key is padded to be the key size with zeros, as is the val, to JDISK_SECTOR_SIZE. Then, **b_tree_insert()** is called with this key and value. It prints the LBA of the inserted val.
- "F key" -- Finds the key and returns the LBA.
- "P" calls **b_tree_print_tree()**. If the key strings equal the key_size, this will do some ugly stuff.

So, let's create the tree above:

```
UNIX> rm tree-2.jdisk
UNIX> bin/b_tree_test tree-2.jdisk CREATE 20480 200
I Daniel Shadowy
Insert return value: 2
I Landon Delinquent
Insert return value: 3
I Alexis Gyroscope
Insert return value: 4
I Grace Procter
Insert return value: 5
P
LBA 0x00000001.  Internal: 0
  Entry 0: Key: Alexis                          LBA: 0x00000004
  Entry 1: Key: Daniel                          LBA: 0x00000002
  Entry 2: Key: Grace                           LBA: 0x00000005
  Entry 3: Key: Landon                          LBA: 0x00000003
  Entry 4:                                      LBA: 0x00000000

I Eli Eider
Insert return value: 6
P
LBA 0x00000008.  Internal: 1
  Entry 0: Key: Eli                             LBA: 0x00000001
  Entry 1:                                      LBA: 0x00000007

LBA 0x00000001.  Internal: 0
  Entry 0: Key: Alexis                          LBA: 0x00000004
  Entry 1: Key: Daniel                          LBA: 0x00000002
  Entry 2:                                      LBA: 0x00000006

LBA 0x00000007.  Internal: 0
  Entry 0: Key: Grace                           LBA: 0x00000005
  Entry 1: Key: Landon                          LBA: 0x00000003
  Entry 2:                                      LBA: 0x00000000

I James Chug
Insert return value: 9
I Caleb Sudanese
Insert return value: 10
I Allison Embellish
Insert return value: 11
P
LBA 0x00000008.  Internal: 1
  Entry 0: Key: Eli                             LBA: 0x00000001
  Entry 1:                                      LBA: 0x00000007

LBA 0x00000001.  Internal: 0
```

```
  Entry 0: Key: Alexis                          LBA: 0x00000004
  Entry 1: Key: Allison                         LBA: 0x0000000b
  Entry 2: Key: Caleb                           LBA: 0x0000000a
  Entry 3: Key: Daniel                          LBA: 0x00000002
  Entry 4:                                      LBA: 0x00000006

LBA 0x00000007.  Internal: 0
  Entry 0: Key: Grace                           LBA: 0x00000005
  Entry 1: Key: James                           LBA: 0x00000009
  Entry 2: Key: Landon                          LBA: 0x00000003
  Entry 3:                                      LBA: 0x00000000

<CNTL-D>
Reads: 18
Writes: 28
UNIX>
```

Useful program #2: random_tester_1

This is a pretty heavyweight testing program:

```
bin/random_tester_1 seed nevents key_size val_size tree_file input_file output_file
```

Here are the parameters:

- **seed** is a seed for **srand48()** (It's a long).
- **nevents** is the number of events that you want to generate.
- **key_size** is the size of the key.
- **val_size** is the number of characters that you want in the vals.
- **tree_file** is the file name. If it doesn't exist, then it will be created with **key_size**, and a size of 2*nevents*JDISK_SECTOR_SIZE.
- **input_file** is an optional file that tells you what is already in the tree. Use "-" to omit.
- **output_file** is an optional file that stores what's in the tree at the end of the program. You use this file as input for subsequent calls.

So, **random_tester_1** randomly generates **b_tree_insert()** and **b_tree_find()** calls with a 50/50 probability. It stores its results internally, and double-checks the LBA's and vals of **b_tree_find()** calls. It generates keys which are random strings whose sizes are less than **key_size**. After the strings are null characters. The vals are random strings whose sizes are less than or equal **val_size**. All of the characters after the string (up to JDISK_SECTOR_SIZE) are null characters. Thus, these guys aren't too disgusting when you print them.

At the end, it prints the number of reads and writes to the jdisk. Here are some examples:

```
UNIX> bin/random_tester_1 0 100 50 30 tree-3.jdisk - tree-3.txt > tmp-output.txt
```

Take a look at the program's output, in tmp-output.txt. Here are the first few lines:

```
UNIX> head -n 5 tmp-output.txt
I mjeglqyuapnuiutrwhuvvjmvgglhhapmuclvaynkuh ajujjsdaaeuuuzhqroq
I xcedrqfnxavvqfguowkwpx jlkrkbpjgahf
I lgltigobrvwkgathopicmh sxstblhiqfyowhyvefbweptotgp
I bkcz ybsmyalyricsxgmptjelqds
F xcedrqfnxavvqfguowkwpx
UNIX>
```

It inserted four keys/vals, and then it performed a find on "xcedrqfnxavvqfguowkwpx". After that find, it double-checked that the LBA matched the original insert, and that the bytes are "jlkrkbpjgahf". We can double-check that with **b_tree_test** and **jdisk_test**:

```
UNIX> echo F xcedrqfnxavvqfguowkwpx | bin/b_tree_test tree-3.jdisk
Attached to tree-3.jdisk.  FS: 10240000  -  KS: 50
Find return value: 3
Reads: 3
Writes: 0
UNIX> echo '3*1024' | bc
3072
UNIX> xxd -s 3072 -len 16 -g 1 tree-3.jdisk
0000c00: 6a 6c 6b 72 6b 62 70 6a 67 61 68 66 00 00 00 00  jlkrkbpjgahf....
UNIX>
UNIX>
```

The file tree-3.txt has the output from **random_tester_1**. I can use it to make another call to **random_tester_1** that attaches to the tree that it just created. The input file tells me what's in the tree (keys, vals and LBA's). Now it generates new events (using the old keys and the new keys for finding):

```
UNIX> bin/random_tester_1 1 10 50 30 tree-3.jdisk tree-3.txt -
I ndbqtrkuwgsgmilthoqwkhsvhxerjetjbcxzakw ctvztvyrhnbig
I hwe delscbm
F auaerduagfi
F dmkaaoptzhmyszbbybrfzfqyyfshbq
F yh
I murekjom mxv
I epunbavuprsytivhufivhxhpt j
F jblrqvbhhtssacoxqeksrxosrhnhpeiqmxjjaxrfiue
F riosrojnkzqcyqgkasvfwybfiqgpvfzacsfbcodp
I wdxszkdgjxqyakiqlweuah lt
Reads: 26
Writes: 17
UNIX>
```

You'll note, it called **b_tree_find()** on "yh", which was in the old tree, and made sure that the LBA and val for that key is what it was when we inserted it on the first call to **random_tester_1**.

random_tester_1 will be a good way for you to test that your btree programs and mine are interoperable. For example, you can call it once with my version of **random_tester_1**, and then again on your version with the input generated from the first call. They should both work together!

Life without a net: random_tester_2

The parameters to **bin/random_tester_2** are similar to **random_tester_1**:

```
random_tester_2 seed nevents key_size tree_file input_file output_file
```

The only parameter that is missing is **val_size**. **random_tester_2** now generates random keys that are exactly **key_size** bytes, and they can be *any* bytes. No longer are they strings. It also generates random vals that are exactly JDISK_SECTOR_SIZE bytes. This is the ultimate test for your programs, because you can't debug with nice strings. You'll have to debug with a little moxy.

The input and output files are now binary.

---

b_tree_dcs

I have a program called **b_tree_dcs**, which stands for "B-Tree Double-Checker and Serializer." It takes a B-Tree file as its command line argument, and the first thing that it does is double-check that it is a valid B-Tree. You may find that useful for debugging (I did). After that, it prints all the keys out in sorted order, and then all the vals in the same order as the keys. If a key or val is a string followed by all null characters to fill out the buffer, then it prints the key or val with "S" and a string. Otherwise, it prints "H" and all of the bytes in hex. In that way, you can actually look at the trees of **random_tester_2** without resorting completely to **jdisk_test** or VI:

```
UNIX> bin/b_tree_dcs tree-2.jdisk
Key_Size: 200
Key     0: S Alexis
Key     1: S Allison
Key     2: S Caleb
Key     3: S Daniel
Key     4: S Eli
Key     5: S Grace
Key     6: S James
Key     7: S Landon
Val     0: S Gyroscope
Val     1: S Embellish
Val     2: S Sudanese
Val     3: S Shadowy
Val     4: S Eider
Val     5: S Procter
Val     6: S Chug
Val     7: S Delinquent
UNIX> rm -f tmp.jdisk; bin/random_tester_2 0 2 25 tmp.jdisk - - > /dev/null
UNIX> bin/b_tree_dcs tmp.jdisk
Key_Size: 25
Key     0: H A95E509709556B8137127CE6160C538C53E4488F4B083ADA9A
Key     1: H C00A91D0F269AEBAB4483B0B5CEA668381C81552B9297D3A7D
Val     0: H 7824B141A5586A73755C3CFC645E2F4FC454332A25B44DE8CFCDEC636CC ...
Val     1: H 5AD463298B1EBB204624070D8C848B448EE15246053A1B632E59E07A666 ...
UNIX>
```

The Grading Script, in grader.sh

You asked for a grading script, so I have provided. In **grader.sh**, you specify two command line arguments: A number and Y|N. The second argument says whether the script will delete its files or not. The first number can be any number. The script will create two jdisk files: **test-answer.jdisk** and **test-$USER.jdisk**. The first one is created by the programs in the lab directory. The second one is created using the programs in the current directory. When the jdisk files are created, they are compared with **b_tree_dcs**, and if the outputs match, the problem is correct.

For example, gradescript number 1 uses **b_tree_test** to insert one key into the B-tree file:

```
UNIX> sh /home/jplank/www-home/cs494/labs/Lab-3-B-Tree/grader.sh 1 N
Problem 1 Correct.

tree-answer.jdisk and tree-jplank.jdisk created as follows:
-----------------------------------------------------------
rm -f tree-jplank.jdisk tree-answer.jdisk
/home/jplank/cs494/labs/Lab-3-B-Tree/bin/b_tree_test tree-answer.jdisk CREATE 51200 26 < input.txt > /dev/null
./bin/b_tree_test tree-jplank.jdisk CREATE 51200 26 < input.txt > /dev/null
UNIX> cat input.txt
I Mackenzie Malignant
UNIX> bin/b_tree_dcs tree-answer.jdisk
Key_Size: 26
Key     0: S Mackenzie
Val     0: S Malignant
UNIX> bin/b_tree_dcs tree-jplank.jdisk
Key_Size: 26
Key     0: S Mackenzie
Val     0: S Malignant
UNIX>
```

Gradescript number 2 is a little more complex -- it creates an input file with 161 random insertions (some of which will replace keys). It calls the **b_disk_test** program in the lab directory to create **tree-answer.jdisk**, from the first 137 entries of the input file. Then it copies **tree-answer.jdisk** to **tree-$USER.jdisk**, and processes the remaining 24 insertions using the lab program and your program:

```
UNIX> sh /home/jplank/www-home/cs494/labs/Lab-3-B-Tree/grader.sh 2 N
Problem 2 Correct.

tree-answer.jdisk and tree-jplank.jdisk created as follows:
-----------------------------------------------------------
rm -f tree-jplank.jdisk tree-answer.jdisk
head -n 137 input.txt | /home/jplank/cs494/labs/Lab-3-B-Tree/bin/b_tree_test tree-answer.jdisk CREATE 614400 129 > /dev/null
cp tree-answer.jdisk tree-jplank.jdisk
sed  1,137d input.txt | /home/jplank/cs494/labs/Lab-3-B-Tree/bin/b_tree_test tree-answer.jdisk > /dev/null
sed  1,137d input.txt | ./bin/b_tree_test tree-jplank.jdisk > /dev/null
```

```
UNIX> wc input.txt
 161  483 2807 input.txt
UNIX> head input.txt
I Blake Shelter
I Mackenzie Effluvium
I Brianna Reluctant
I Evelyn Vigilantism
I Mason Drawn
I Anthony Singlet
I Anna Microjoule
I Kate Howell
I Paige Tinder
I Xavier Roulette
UNIX> tail input.txt
I Peyton Tammany
I Caleb Abe
I Landon Knick
I Makayla Fortescue
I Brody Repeat
I Nathan Swizzle
I Madison Nonetheless
I Ava Astronautic
I Zoey Indebted
I Chloe Smirk
UNIX> bin/b_tree_dcs tree-answer.jdisk | head
Key_Size: 129
Key     0: S Aaron
Key     1: S Abigail
Key     2: S Addison
Key     3: S Aidan
Key     4: S Aiden
Key     5: S Alex
Key     6: S Alexander
Key     7: S Alexis
Key     8: S Allison
UNIX>
```

The tests performed by the gradescript are based on the number given mod ten. The tests are as follows:

- **Number mod 10 = 1**: 1 to 15 random insertions using **b_tree_test**. Key size is between 15 and 50.
- **Number mod 10 = 2**: 1 to 150 random insertions using the lab directory's **b_tree_test**. Key size is between 15 and 214. Then 1 to 150 more random insertions using **b_tree_test**.
- **Number mod 10 = 3**: Calling **random_tester_1** with the gradescript number as the seed. 1 to 20 events; Key size between 15 and 214; Val size between 1 and 200.
- **Number mod 10 = 4**: Calling **random_tester_1** with the gradescript number as the seed. 1 to 500 events; Key size between 15 and 252; Val size between 1 and 1023.
- **Number mod 10 = 5**: Calling **random_tester_2** with the gradescript number as the seed. 1 to 500 events; Key size between 15 and 252.
- **Number mod 10 = 6**: Calling **random_tester_2** with the gradescript number as the seed. 1 to 5000 events; Key size between 4 and 10.
- **Number mod 10 = 7**: Calling **random_tester_2** with the gradescript number as the seed. 1 to 5000 events; Key size = 254.
- **Number mod 10 = 8**: Calling **b_tree_test** to create a b-tree of appropriate size. Calling **random_tester_2** in the lab directory to with up to 5000 events, key size = 254. The resulting tree is then copied to **tree-$USER.jdisk**, and your **random_tester_2** is called for another 1 to 5000 events.
- **Number mod 10 = 9**: Same as test 8, only now the key size is between 4 and 10.
- **Number mod 10 = 0**: Same as test 9, only now the key size is between 4 and 254.

Each of these can take a few seconds, because the files can be in the 10's of megabytes.

Some help

You don't need this section, but you probably should read it. My internal btree struct was as follows:

```
typedef struct {
  int key_size;                 /* These are the first 16/12 bytes in sector 0 */
  unsigned int root_lba;
  unsigned long first_free_block;

  void *disk;                   /* The jdisk */
  unsigned long size;           /* The jdisk's size */
  unsigned long num_lbas;       /* size/JDISK_SECTOR_SIZE */
  int keys_per_block;           /* MAXKEY */
  int lbas_per_block;           /* MAXKEY+1 */
  Tree_Node *free_list;         /* Free list of nodes */

  Tree_Node *tmp_e;             /* When find() fails, this is a pointer to the external node */
  int tmp_e_index;              /* and the index where the key should have gone */

  int flush;                    /* Should I flush sector[0] to disk after b_tree_insert() */
} B_Tree;
```

Note how the first three fields are such that you can write them to sector 0 (and read them from sector 0). As it turns out, I write the whole struct to sector 0, but the remaining bytes are ignored.

A **Tree_Node** is the internal representation of a node. Here's its struct:

```
typedef struct tnode {
  unsigned char bytes[JDISK_SECTOR_SIZE+256]; /* This holds the sector for reading and writing.
                                                 It has extra room because your internal representation
                                                 will hold an extra key. */
  unsigned char nkeys;                       /* Number of keys in the node */
  unsigned char flush;                       /* Should I flush this to disk at the end of b_tree_insert()? */
  unsigned char internal;                    /* Internal or external node */
  unsigned int lba;                          /* LBA when the node is flushed */
  unsigned char **keys;                      /* Pointers to the keys.  Size = MAXKEY+1 */
  unsigned int *lbas;                        /* Pointer to the array of LBA's.  Size = MAXKEY+2 */
  struct tnode *parent;                      /* Pointer to my parent -- useful for splitting */
  int parent_index;                          /* My index in my parent */
  struct tnode *ptr;                         /* Free list link */
} Tree_Node;
```

I maintain my own free list of **Tree_Node** structs. When I allocate one, I do three **malloc()** calls. One is for the **Tree_Node** itself, one is for **keys** and one is for **lbas**. I set the values of **keys** right after I allocate them, since they point to fixed locations in **bytes**. For example, **keys[0]** will point to **bytes+2**. **keys[1]** will point to **bytes+2+key_size**. Etc. The size of **keys** is **MAXKEY+1**. Moreover, the size of **lbas** is **MAXKEY+2**. The reason is that in my internal representation of a B-Tree node, I am allowed to store an extra key and val. This simplifies the implementation of splitting in an *enormous* way. If you don't believe me, ask anyone in the 2015 version of CS494 who tried to implement B-Trees without it. When I'm done with a **Tree_Node**, I put it onto my free list. That way, if I need a new **Tree_Node** later, and there's one on the free list, I don't have to do any **malloc()** calls -- I simply grab it from the free list. **Keys** and **lbas** are already allocated, and the values of **keys** are already set correctly. That's convenient.

I have a procedure which reads a **Tree_Node** from a jdisk. It reads it into **bytes**, and then it copies the LBA's from the end of the sector into **lbas** (using **memcpy**). It has to do this, because it uses the end of the sector for that extra key. It also reads **nkeys** and **internal** from the sector. A convenient thing is that the **keys** pointers are already pointing to the correct place, so you don't need to do anything special with the keys.

I set the **flush** field whenever I modify a B-Tree node, and the modified node needs to be flushed to disk. I do the final flushing at the end of **b_tree_insert()**. When I flush a node, I need to create the sector. I do this by copying **nkeys** and **internal** into their proper place in **bytes**. I then copy **lbas** to their proper place in **bytes**. The keys are already in their proper place. I then write **bytes** using **jdisk_write()**.

What I did for splitting was as follows -- I'd go ahead and insert the key/lba. Since there's room for an extra key and lba in the **Tree_Node**, this works even when the node is full. I then checked to see if the node needed to be split, and if so, I called a procedure called **split_node**. This procedure has to be recursive, BTW.

When I call **find()**, I have it read in all of the nodes on the path to the external node, setting their parent fields, but setting **flush** to 0. The **flush** field is set by **b_tree_insert()** when the node changes. The parent fields are used on splitting. When I'm done with **b_tree_insert()** or **b_tree_find()**, I call **free_and_flush()** on every node from the external node up to the root. This flushes the node if **flush** is set, and puts the node onto the free list. I found **free_and_flush** super-helpful.

---

b_tree_instrument.o: Instrumented code.

In **b_tree_instrument.o**, I have added code that prints my **Tree_Node** just after reading, and just before flushing. I have compiled it with **b_tree_test** to make the executable **b_tree_test_inst**. This may help you figure out how my pointers work. Here's an example, where I insert a new node into **tree-2.jdisk**. First, here's what I'm doing:

```
UNIX> cp tree-2.jdisk tmp.jdisk
UNIX> echo P | bin/b_tree_test tmp.jdisk
Attached to tmp.jdisk.  FS: 20480  -  KS: 200
LBA 0x00000008.  Internal: 1
  Entry 0: Key: Eli                             LBA: 0x00000001
  Entry 1:                                      LBA: 0x00000007

LBA 0x00000001.  Internal: 0
  Entry 0: Key: Alexis                          LBA: 0x00000004
  Entry 1: Key: Allison                         LBA: 0x0000000b
  Entry 2: Key: Caleb                           LBA: 0x0000000a
  Entry 3: Key: Daniel                          LBA: 0x00000002
  Entry 4:                                      LBA: 0x00000006

LBA 0x00000007.  Internal: 0
  Entry 0: Key: Grace                           LBA: 0x00000005
  Entry 1: Key: James                           LBA: 0x00000009
  Entry 2: Key: Landon                          LBA: 0x00000003
  Entry 3:                                      LBA: 0x00000000

Reads: 4
Writes: 0
UNIX> echo I A-Aron Stoae | bin/b_tree_test tmp.jdisk
Attached to tmp.jdisk.  FS: 20480  -  KS: 200
Insert return value: 12
Reads: 3
Writes: 5
UNIX> echo P | bin/b_tree_test tmp.jdisk
Attached to tmp.jdisk.  FS: 20480  -  KS: 200
LBA 0x00000008.  Internal: 1
  Entry 0: Key: Allison                         LBA: 0x00000001
  Entry 1: Key: Eli                             LBA: 0x0000000d
  Entry 2:                                      LBA: 0x00000007

LBA 0x00000001.  Internal: 0
  Entry 0: Key: A-Aron                          LBA: 0x0000000c
  Entry 1: Key: Alexis                          LBA: 0x00000004
  Entry 2:                                      LBA: 0x0000000b

LBA 0x0000000d.  Internal: 0
  Entry 0: Key: Caleb                           LBA: 0x0000000a
  Entry 1: Key: Daniel                          LBA: 0x00000002
  Entry 2:                                      LBA: 0x00000006

LBA 0x00000007.  Internal: 0
  Entry 0: Key: Grace                           LBA: 0x00000005
  Entry 1: Key: James                           LBA: 0x00000009
  Entry 2: Key: Landon                          LBA: 0x00000003
  Entry 3:                                      LBA: 0x00000000
```

```
Reads: 5
Writes: 0
UNIX>
```

Now, here are the tree nodes as they are read in the beginning of the insertion, and as they are flushed after the insertion:

```
UNIX> cp tree-2.jdisk tmp.jdisk
UNIX> echo I A-Aron Stoae | bin/b_tree_test_inst tmp.jdisk
Attached to tmp.jdisk.  FS: 20480  -  KS: 200
Find(): Read -- Tree Node 0x1e840c0
  Bytes:          0x1e840c0
  Nkeys Address:  0x1e845c0
  Nkeys Value:    1
  Flush Value:    0
  Internal Value: 1
  LBA Value:      8
  Keys:           0x1e84600         keys is allocated with malloc().
  Keys[0]         0x1e840c2 (Eli)    This is bytes+2
  Keys[1]         0x1e8418a (Landon) This is bytes+2+key_size.  Since nkeys equals 1, the contents here are
                                    meaningless.  They happen to hold "Landon" from a previous time,
                                    when "Landon" was the second key.  You have to ignore this, because
                                    Nkeys is one.
  Keys[2]         0x1e84252 ()       This is bytes+2+key_size*2
  Keys[3]         0x1e8431a ()
  Keys[4]         0x1e843e2 ()       This is the extra key.
  Lbas:           0x1e84630         lbas is allocated with malloc().
  Lbas[0]         0x1               Lba of the internal node before "Eli".
  Lbas[1]         0x7               Lba of the internal node after "Eli".
  Lbas[2]         0x0               The value of these are meaningless.  They happen to be zero.
  Lbas[3]         0x0
  Lbas[4]         0x0
  Lbas[5]         0x0               This is the extra lba
  Parent          0x0
  Parent_index    -1
  &ptr            0x1e845e8         This is meaningless, because the tree_node isn't on the free list.
Find(): Read -- Tree Node 0x1e84650
  Bytes:          0x1e84650
  Nkeys Address:  0x1e84b50
  Nkeys Value:    4
  Flush Value:    0
  Internal Value: 0
  LBA Value:      1
  Keys:           0x1e84b90
  Keys[0]         0x1e84652 (Alexis)
  Keys[1]         0x1e8471a (Allison)
  Keys[2]         0x1e847e2 (Caleb)
  Keys[3]         0x1e848aa (Daniel)
  Keys[4]         0x1e84972 ()
  Lbas:           0x1e84bc0
  Lbas[0]         0x4
  Lbas[1]         0xb
  Lbas[2]         0xa
  Lbas[3]         0x2
  Lbas[4]         0x6
  Lbas[5]         0x0
  Parent          0x1e840c0
  Parent_index    0
  &ptr            0x1e84b78
Free_and_flush(): Write -- Tree Node 0x1e84be0
  Bytes:          0x1e84be0
  Nkeys Address:  0x1e850e0
  Nkeys Value:    2
  Flush Value:    1
  Internal Value: 0
  LBA Value:      13
  Keys:           0x1e85120
  Keys[0]         0x1e84be2 (Caleb)
  Keys[1]         0x1e84caa (Daniel)
  Keys[2]         0x1e84d72 ()
  Keys[3]         0x1e84e3a ()
  Keys[4]         0x1e84f02 ()
  Lbas:           0x1e85150
  Lbas[0]         0xa
  Lbas[1]         0x2
  Lbas[2]         0x6
  Lbas[3]         0x0
  Lbas[4]         0x0
  Lbas[5]         0x0
  Parent          0x0
  Parent_index    0
  &ptr            0x1e85108
Free_and_flush(): Write -- Tree Node 0x1e84650
  Bytes:          0x1e84650
  Nkeys Address:  0x1e84b50
  Nkeys Value:    2
  Flush Value:    1
  Internal Value: 0
  LBA Value:      1
  Keys:           0x1e84b90
  Keys[0]         0x1e84652 (A-Aron)
  Keys[1]         0x1e8471a (Alexis)
  Keys[2]         0x1e847e2 (Allison) These are leftover from before the split.  nkeys is 2,
  Keys[3]         0x1e848aa (Caleb)   so they are ignored, but they are written to disk.
  Keys[4]         0x1e84972 (Daniel)  As you can see there is room for 5 keys, which makes splitting easier.
  Lbas:           0x1e84bc0
  Lbas[0]         0xc
  Lbas[1]         0x4
  Lbas[2]         0xb
  Lbas[3]         0xa                Same with these LBA's.
  Lbas[4]         0x2
  Lbas[5]         0x6
  Parent          0x1e840c0
  Parent_index    0
  &ptr            0x1e84b78
Free_and_flush(): Write -- Tree Node 0x1e840c0
  Bytes:          0x1e840c0
  Nkeys Address:  0x1e845c0
  Nkeys Value:    2
  Flush Value:    1
  Internal Value: 1
  LBA Value:      8
  Keys:           0x1e84600
  Keys[0]         0x1e840c2 (Allison)
  Keys[1]         0x1e8418a (Eli)
  Keys[2]         0x1e84252 ()
  Keys[3]         0x1e8431a ()
  Keys[4]         0x1e843e2 ()
  Lbas:           0x1e84630
  Lbas[0]         0x1
  Lbas[1]         0xd
  Lbas[2]         0x7
  Lbas[3]         0x0
  Lbas[4]         0x0
  Lbas[5]         0x0
  Parent          0x0
  Parent_index    -1
  &ptr            0x1e845e8
Insert return value: 12
Reads: 3
Writes: 5
UNIX>
```

Btree Haikus

```
How hard can it be?
Inserting keys at random
Segmentation fault
```

David C. - 2015

```
Time to start split node.
I saved the best part for last.
What day is it now?
```

Tyler M. - 2016

```
Five hundred lines long
One hundred problems correct
Please, now can I sleep?
```

David C. - 2015

```
Seg fault on case 5...
I guess I should try Valgrind.
Seg fault on case 6...
```

Tyler M. - 2016

```
Passes the gradescript
Allocates 10 million bytes
Pattern not found: free()
```

Elliot G. - 2016

```
My data structure
Does not include a spare key.
No spring break for me.
```

Dr. Plank - 2015

```
Seven hundred lines?
I guess I should have used find..
Insert can find too!
```

Tyler M. - 2016