Project 3

Due: Friday, May 9th 11:59pm

Description

In this project, you will write an interactive program that creates and manages index files. The index files will contain a b-tree. The user can create, insert, and search such index files. You may choose between implementing in C, C++, Java, and Python.

Details

The program will be a command-line program that allows the user to give various commands as command-line arguments. These commands will perform various operations on an index file. You should handle any errors in user input.

The index file will represent a B-Tree. Your implementation should never have more than 3 nodes in memory at a time.

Commands

All commands should be lowercase.

create Create a new index file. The first argument after "create" is assumed to be the name of the index file. If that file already exists, fail with an error message. The file should remain unmodified.

Example: `project3 create test.idx`

insert The first argument after "insert" is assumed to be the name of the index file. If the file does not exist or if the file is not a valid index file then exit with an error. The next two arguments are the key and value, respectively. These should be converted into unsigned integers (signed integers are also fine in Java), and then inserted into the B-Tree.

Example: `project3 insert test.idx 15 100`

search The first argument after "search" is assumed to be the name of the index file. If the file does not exist or if the file is not a valid index file then exit with an error. The next argument is assumed to be a key. It should be converted into an unsigned integer (signed integer is also fine in Java). Search the index for the key. If found, print the key/value pair. Otherwise, print an error message.

Example: `project3 search test.idx 15`

load The first argument after the "load" is assumed to be the name of the index file. If the file does not exist or if the file is not a valid index file then exit with an error. The next argument is assumed to be the name of a csv file. If the file does not exist, the exit with an error message. Each line of the file is a comma separated key/value pair. Read the file, inserting each pair as above with the insert command.

Example: `project3 load test.idx input.csv`

print The first argument after the "print" is assumed to be the name of the index file. If the file does not exist or if the file is not a valid index file then exit with an error. Print every key/value pair in the index to standard output.

Example: `project3 print test.idx`

extract The first argument after the "extract" is assumed to be the name of the index file. If the file does not exist or if the file is not a valid index file then exit with an error. The next argument is a filename. If the file exists, exit with an error message. The file should remain unmodified. Save every key/value pair in the index as comma separated pairs to the file.

Example: `project3 extract test.idx output.csv`

The Index File

The index file will be divided into blocks of 512 bytes. Each node of the btree will fit in one 512 byte block, and the file header will use the entire first block. New nodes will be appended to the end of the file. Since, we do not have a delete operation, we do not need to worry about deleting nodes.

As will be seen below, the size of the header information and a node will actually be smaller than a block size. This is okay. The remaining space in the block will remain unused.

All numbers stored in the file should be stored as 8-byte integers with the big endian byte order. This is the way Java stores all integers. Python integers have a `to_bytes` method that takes as a parameter the number of bytes to convert the integer into and the byte order. So, `n.to_bytes(8, 'big')` will convert the integer `n` to a sequence of 8 bytes in big-endian order. C and C++ do not have a simple way of doing this. You need to detect the byte order of the system and then convert to big endian if needed. Here are two C functions that can do this.

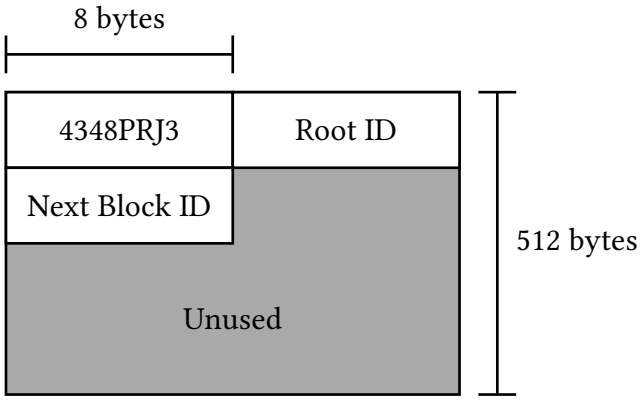
```
#include <stdint.h>

int is_bigendian() {
    int x = 1;
    return ((uint8_t *)&x)[0] != 1;
}

uint64_t reverse_bytes(uint64_t x) {
    uint8_t dest[sizeof(uint64_t)];
    uint8_t *source = (uint8_t*)&x;
    for(int c = 0; c < sizeof(uint64_t); c++)
        dest[c] = source[sizeof(uint64_t)-c-1];
    return *(uint64t_t *)dest);
}
```

Each block will have a block id determined by the order in the file, starting at zero.

Header Format

The header can be maintained in memory, but needs to be in sync with the file. The header will have the following fields, in the order presented.

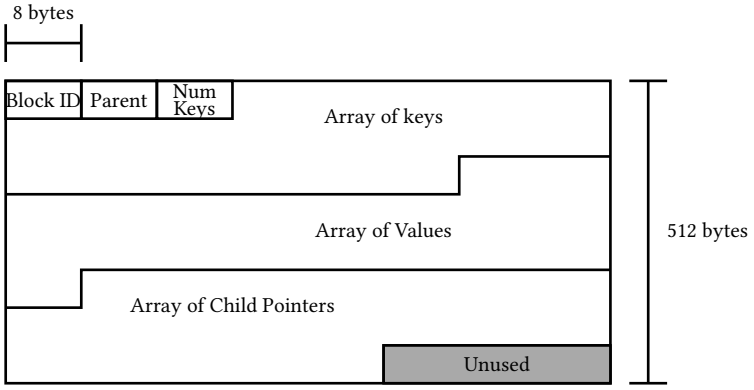
- 8-bytes: The magic number "4348PRJ3" (as a sequence of ASCII values).
- 8-bytes: The id of the block containing the root node. This field is zero if the tree is empty.
- 8-bytes: The id of the next block to be added to the file. This is the next location for a new node.
- The remaining bytes are unused.



The B-Tree

The b-tree should have minimal degree 10. This will give 19 key/value pairs, and 20 child pointers. Each node will be stored in a single block with some header information. Below is the node block fields in order.

- 8-bytes: The block id this node is stored in.
- 8-bytes: The block id this nodes parent is located. If this node is the root, then this field is zero.
- 8-bytes: Number of key/value pairs currently in this node.
- 152-bytes: A sequence of 19 64-bit keys
- 152-bytes: A sequence of 19 64-bit values
- 160-bytes: A sequence of 20 64-bit offsets. These block ids are the child pointers for this node. If a child is a leaf node, the corresponding id will be zero.
- Remaining bytes are unused.



The sequence of keys, values, and child pointers correspond to each other. So, the first key (key 0) corresponds to the first value. The first child pointer is the pointer to the subtree containing all entries with a key less than the first key. The second child pointer is the pointer to the subtree containing all entries with a key greater than the first key, but less than the second.