NC STATE UNIVERSITY

CSC 541
Assignment 4

B-Trees

Introduction

The goals of this assignment are two-fold:

1. To introduce you to searching data on disk using B-trees.
2. To investigate how changing the order of a B-tree affects its performance.

Index File

During this assignment you will create, search, and manage a binary index file of integer key values. The values stored in the file will be specified by the user. You will structure the file as a B-tree.

Program Execution

Your program will be named `assn_4` and it will run from the command line. Two command line arguments will be specified: the name of the index file, and a B-tree order.

```
assn_4 index-file order
```

For example, executing your program as follows

```
assn_4 index.bin 4
```

would open an index file called `index.bin` that holds integer keys stored in an order-4 B-tree. You can assume `order` will always be ≥ 3. For convenience, we refer to the index file as `index.bin` throughout the remainder of the assignment.

Note. If you are asked open an existing index file, you can assume the B-tree order specified on the command line matches the order that was used when the index file was first created.

B-Tree Nodes

Your program is allowed to hold individual B-tree nodes in memory—but not the entire tree—at any given time. Your B-tree node should have a structure and usage similar to the following.

```
#include <stdlib.h>

int order = 4;    /* B-tree order */

typedef struct {  /* B-tree node */
  int   n;        /* Number of keys in node */
  int  *key;      /* Node's keys */
  long *child;    /* Node's child subtree offsets */
} btree_node;
```

```
btree_node node;  /* Single B-tree node */

node.n = 0;
node.key = (int *) calloc( order - 1, sizeof( int ) );
node.child = (long *) calloc( order, sizeof( long ) );
```

Note. Be careful when you're reading and writing data structures with dynamically allocated memory. For example, trying to write node like this

```
fwrite( &node, sizeof( btree_node ), 1, fp );
```

will write node's key count, the pointer value for its key array, and the pointer value for its child offset array, but **it will not** write the contents of the key and child offset arrays. The arrays' contents and not pointers to their contents need to be written explicitly instead.

```
fwrite( &node.n, sizeof( int ), 1, fp );
fwrite( node.key, sizeof( int ), order - 1, fp );
fwrite( node.child, sizeof( long ), order, fp );
```

Reading node structures from disk would use a similar strategy.

Root Node Offset

In order to manage any tree, you need to locate its root node. Initially the root node will be stored near the front of `index.bin`. If the root node splits, however, a new root will be appended to the end of `index.bin`. The root node's offset will be maintained persistently by storing it at the front of `index.bin` when the file is closed, and reading it when the file is opened.

```
#include <stdio.h>

FILE *fp;    /* Input file stream */
long  root;  /* Offset of B-tree root node */

fp = fopen( "index.bin", "r+b" );

/*  If file doesn't exist, set root offset to unknown and create
 *  file, otherwise read the root offset at the front of the file */

if ( fp == NULL ) {
  root = -1;
  fp = fopen( "index.bin", "w+b" );
  fwrite( &root, sizeof( long ), 1, fp );
} else {
  fread( &root, sizeof( long ), 1, fp );
}
```

User Interface

The user will communicate with your program through a set of commands typed at the keyboard. Your program must support four simple commands:

- `add k`
  Add a new integer key with value k to `index.bin`.
- `find k`
  Find an entry with a key value of k in `index.bin`, if it exists.
- `print`
  Print the contents and the structure of the B-tree on-screen.
- `end`
  Update the root node's offset at the front of the `index.bin`, and close the index file, and end the

program.

Add

Use a standard B-tree algorithm to add a new key *k* to the index file.

1. Search the B-tree for the leaf node *L* responsible for *k*. If *k* is stored in *L*'s key list, print

   ```
   Entry with key=k already exists
   ```

   on-screen and stop, since duplicate keys are not allowed.
2. Create a new key list *K* that contains *L*'s keys, plus *k*, sorted in ascending order.
3. If *L*'s key list is not full, replace it with *K*, update *L*'s child offsets, write *L* back to disk, and stop.
4. Otherwise, split *K* about its median key value k_m into left and right key lists $K_L = (k_0, \dots, k_{m-1})$ and $K_R = (k_{m+1}, \dots, k_{n-1})$. Use ceiling to calculate $m = \lceil ^{(n-1)}/_2 \rceil$. For example, if $n = 3$, $m = 1$. If $n = 4$, $m = 2$.
5. Save K_L and its associated child offsets in *L*, then write *L* back to disk.
6. Save K_R and its associated child offsets in a new node *R*, then append *R* to the end of the index file.
7. Promote k_m , *L*'s offset, and *R*'s offset and insert them in *L*'s parent node. If the parent's key list is full, recursively split its list and promote the median to its parent.
8. If a promotion is made to a root node with a full key list, split and create a new root node holding k_m and offsets to *L* and *R*.

Find

To find key value *k* in the index file, search the root node for *k*. If *k* is found, the search succeeds. Otherwise, determine the child subtree *S* that is responsible for *k*, then recursively search *S*. If *k* is found during the recursive search, print

```
Entry with key=k exists
```

on-screen. If at any point in the recursion *S* does not exist, print

```
Entry with key=k does not exist
```

on-screen.

Print

This command prints the contents of the B-tree on-screen, level by level. Begin by considering a single B-tree node. To print the contents of the node on-screen, print its key values separated by commas.

```
int         i;      /* Loop counter */
btree_node  node;   /* Node to print */
long        off;    /* Node's offset */

for( i = 0; i < node.n - 1; i++ ) {
  printf( "%d,", node.key[ i ] );
}
printf( "%d", node.key[ node.n - 1 ] );
```

To print the entire tree, start by printing the root node. Next, print the root node's children on a new line, separating each child node's output by a space character. Then, print their children on a new line, and so on until all the nodes in the tree are printed. This approach prints the nodes on each level of the B-tree left-to-right on a common line.

For example, inserting the integers 1 through 13 inclusive into an order-4 B-tree would produce the following output.

```
1: 9
2: 3,6 12
3: 1,2 4,5 7,8 10,11 13
```

Hint. To process nodes left-to-right level-by-level, do not use recursion. Instead, create a queue containing the root node's offset. Remove the offset at the front of the queue (initially the root's offset) and read the corresponding node from disk. Append the node's non-empty subtree offsets to the end of the queue, then print the node's key values. Continue until the queue is empty.

End

This command ends the program by writing the root node's offset to the front of `index.bin`, then closing the index file.

Programming Environment

All programs must be written in C, and compiled to run on the `remote.eos.ncsu.edu` Linux server. Any ssh client can be used to access your Unity account and AFS storage space on this machine.

Your assignment will be run automatically, and the output it produces will be compared to known, correct output using `diff`. Because of this, **your output must conform to the print command's description**. If it doesn't, `diff` will report your output as incorrect, and it will be marked accordingly.

Supplemental Material

In order to help you test your program, we provide example input and output files.

- `input-01.txt`, an input file of commands applied to an initially empty index file saved as an order-4 B-tree, and
- `input-02.txt`, an input file of commands applied to the index file produced by `input-01.txt`.

The output files show what your program should print after each input file is processed.

- `output-01.txt`, the output your program should produce after it processes `input-01.txt`.
- `output-02.txt`, the output your program should produce after it processes `input-02.txt`.

To test your program, you would issue the following commands:

```
% rm index.bin
% assn_4 index.bin 4 < input-01.txt > my-output-01.txt
% assn_4 index.bin 4 < input-02.txt > my-output-02.txt
```

You can use `diff` to compare output from your program to our output files. If your program is running properly and your output is formatted correctly, your program should produce output identical to what is in these files.

Please remember, the files we're providing here are meant to serve as examples only. Apart from holding valid commands, **you cannot make any assumptions** about the size or the content of the input files we will use to test your program.

Hand-In Requirements

Use Moodle (the online assignment submission software) to submit the following files:

- `assn_4`, a Linux executable of your finished assignment, and
- all associated source code files (these can be called anything you want).

There are four important requirements that your assignment must satisfy.

1. Your executable file must be named exactly as shown above. The program will be run and marked electronically using a script file, so using a different name means the executable will not be found, and subsequently will not be marked.
2. Your program must be compiled to run on `remote.eos.ncsu.edu`. If we cannot run your program, we will not be able to mark it, and we will be forced to assign you a grade of 0.
3. Your program must produce output that exactly matches the format described in the print command section of this assignment. If it doesn't, it will not pass our automatic comparison to known, correct output.
4. You must submit your source code with your executable prior to the submission deadline. If you do not submit your source code, we cannot MOSS it to check for code similarity. Because of this,

any assignment that does not include source code will be assigned a grade of 0.

Updated 01-Nov-17