CPS222 - DATA STRUCTURES AND ALGORITHMS

Project #4 - Due Wednesday, April 13, at the start of class.

Purposes: To give you experience working with B-Trees and disk-based structures

Note: This project must be done in teams of 2. It is desirable that students of similar C++ background work together

Introduction

For this project, you will be implementing a class that stores a collection of keys (character strings) and associated values (also strings) in a disk file structured as a B-Tree. Two operations on this structure are required: insert a new key and its value, and lookup the value given a key. A third operation that can be implemented for significant bonus credit is removing a key and its associated value.

Requirements

1. Copy the files from `/home/cps222/project4` into your ~/cps222/project4 directory. It contains the following files:

`BTree.h`	- header file that declares the class `BTree`. You may add `private` methods to this class, if you wish (but you are not required to do so).
`BTree.cc`	- implementation file for class `BTree`. Several methods are already written; you will implement `insert()`, `lookup()`, and optionally `remove()`, as well as any private methods you add to `BTree.h`.
`BTreeBlock.h`	- header file that declares the class `BTreeBlock`, which is used in the implementation of class `BTree`. (You may add methods to this class, if you wish, but you may not change the methods already present in any way.)
`BTreeBlock.cc`	- prewritten code for the implementation of class `BTreeBlock`. (If you add methods to `BTreeBlock.h` you must implement them here, but you may not change the code already present in any way.)
`BTreeFile.h`	- header file that declares the class `BTreeFile`, which is used in the implementation of class `BTree`. (You may not change this file except to change the declaration of the symbolic constant `DEGREE`.)
`BTreeFile.cc`	- prewritten code for the implementation of class `BTreeFile.h`. (You may not change this file in any way.)
`project4.cc`	- test driver / main program.
`test.tree`	- a file that contains a moderate-size BTree. You can use this for testing `lookup()`, and can also test inserting keys into it.
`onelevel.in`	- a command file that creates a one-level tree from scratch, then prints it.
`twolevel.in`	- a command file that creates a two-level tree from scratch, then prints it.
`threelevel.in`	- a command file that creates a three-level tree from scratch, then prints it.
`Makefile`	- commands for compiling program

   A UML class diagram showing the relationship of the various classes is attached as Appendix A.

2. All of your coding can be done in `BTree.cc`. However, if you wish, you may also modify `BTree.h`, `BTreeBlock.h`, the constant `DEGREE` in `BTreeFile.h` and/or `BTreeBlock.cc` as discussed above. You may not modify class `BTreeFile` other than to change the constant `DEGREE` (Modifying class `BTreeBlock` should only be necessary if you attempt `remove()`),

3. The completion of `insert()` and `lookup()` is required, and will be worth 80 points for correct operation for all cases. (Partial credit for insert possible if it works correctly for some cases but not others.) **You would be wise to implement and test `lookup()` (using the test data file supplied) before attempting `insert()`. This will help you get a good understanding of the tree structure and disk operations. Note, though, that once you get `insert()` working you should perform further tests of `lookup()` to be sure it works correctly with a larger tree.**

4. The implementation of `remove()` is optional for bonus credit. The correct implementation of this method for all cases will result in a one-third letter grade increase in your final grade for the course (3 percent added to your final course average) , with partial credit possible if it works correctly for some cases but not others. **Do not attempt this until you have the required methods working correctly for all cases! (Note: the fact that there is such a large bonus for this option should suggest it is quite difficult. In fact, only three students has ever gotten it correct over a period of many years!)**

5. You can compile your files and link them using the following command:
   ```
   make
   ```

6. Test your program thoroughly. (See discussion of testing procedure below.) You should test your program both working with the test B-Tree you copy from the class directory and with constructing a tree from scratch. If you do the `remove()` option, your testing should include “tearing down” a tree containing many entries until it is totally empty. Note that the sample input files supplied do not test lookup at all - however, since a tree once created “lives” forever in a disk file unless you delete the file, you can use the one of these files to create a tree and then manually perform lookups on the same file - or you can use the example tree provided. (In all cases, if your output is not correct, but you cannot fix the problem, print a copy of it and annotate it to show that you are aware of the problem.)

7. As always, make good use of internal comments and whitespace to document your program and make it readable. Code quality, methodology, documentation, and comments will account for 10 points toward your project grade.

8. A project quiz will account for 10 points toward your project grade.

Implementation Notes:

1. The B-Tree will reside in a disk file, whose name is specified when the program starts up. The program can either construct a new file from scratch, or access an existing file. Thus, information in the tree is preserved between runs of the program, so if you specify the same file name when you start the tester, you will get the tree as it was at the end of the last testing session. (If you mess up a tree, you can exit the program and delete the file using `rm`, or re-copy `test.tree`, as appropriate, to start over.)

2. Each node of the tree will occupy a 512 byte block. The degree of the tree is specified by the constant `DEGREE` declared in `BTreeFile.h`. A node can contain up to `DEGREE-1` keys and their associated values, and can have up to `DEGREE` children. In the files you copy, `DEGREE` is declared to be 7 - so each node can hold 6 up to key/value pairs and have up to 7 children. Of course, your code *must* use the *symbolic* constant, and *must not* "hardwire" the constant value, or any constants derived from it (e.g. 6) into your code. It should be possible to modify the symbolic constant and recompile all your files to make your program work correctly with a tree of different degree. This can easily be done by including `-D DEGREE=new value` on the command line or by editing the file. (You wouldn't know anyone who might consider doing something like this during testing, especially using an even value of DEGREE, would you? :-).

   (Note: the node size is actually relatively small, and even so the available space in the node is not fully utilized with a degree of 7. This is done deliberately to simplify testing by keeping the branching factor down.)

3. Blocks in the tree are numbered 1, 2, 3 ... There is no block 0, so 0 has the same function for a tree on disk as `NULL` does for a tree in main memory. The data type `BTreeFile::BlockNumber` is defined to use for variables that store a block number. (It's an unsigned int, but use the named type.)

4. Block 1 of the tree is special. It is a header block that is created when a file is created. The header block contains no keys; instead, it stores four values: the degree of the tree, the total size of the file (needed when it becomes necessary to allocate a new block - initially 1 because an empty tree consists only of a header block); the number of the block containing the root of the tree (initially 0 when the tree is empty); and the number of the first block on a linked list of free blocks (initially 0 because the list is empty.) (A block is added to the free list by a `deallocateBlock()` operation when a `remove()` coalesces two blocks into one. Thus, the free list will always be empty unless you do the `remove()` option.)

   (Note: Your code will *not* need to access the header block directly - this is all handled by routines in class `BTreeFile`. Information is provided here about the header so you will know what is going on. Header information is read from disk by the `BTreeFile()` constructor when the program starts up, and is rewritten to the file when any of the `BTreeFile::` methods change it. (E.g. `setRoot()` changes the root block; `allocateBlock()` either changes the free list or the number of the last block used, etc.) You must **not** access the header block directly!

5. Your code will need to make use of five or six of the routines that are declared in class `BTreeFile` (Declared in header file `BTreeFile.h`, and implemented for you in `BTreeFile.cc`):

`getRoot()`	- Get the number of the root block (as recorded in the header.)
`setRoot()`	- Change the number of the root block (in the header.)
`getBlock()`	- Copy a specific block from the file to an in-memory object.
`putBlock()`	- Copy a (possibly modified) in-memory copy of a block back to the file.
`allocateBlock()`	- Get the number of a previously unused block that can now be used as part of the tree structure. (The disk equivalent of new)
`deallocateBlock()`	- Report that a specific block is no longer in use, and can now be returned by a subsequent call to `allocateBlock()`. (The disk equivalent of delete) (Needed only for `remove()`).

   When the tree is initially created, it will have no root. When the first key is created, your code will need to create a root block to contain it, using `allocateBlock()` to assign a block number for it, and `setRoot()` to record this choice. As the program runs, the number of the root block may be changed as a result of splitting the root - use `getRoot()` to find out the number of the current root block, and `setRoot()` to change it when necessary. Any time you need to add a block to the tree, use `allocateBlock()` to get a block number for the new block. Of course, `getBlock()` and `putBlock()` are used to transfer data from/to the file.

6. In addition, you will make use of the class `BTreeBlock` (declared in header file `BTreeBlock.h` and implemented for you in `BTreeBlock.cc`.). A block represents an in-memory copy of information from a block on disk, which is transferred from disk to memory by `BTreeFile::getBlock()` or from memory back to disk by `BTreeFile::putBlock()`. Your code will also need to make use of the routines declared in this class (declared in header file `BTreeBlock.h`, and implemented for you in `BTreeFile.cc`).

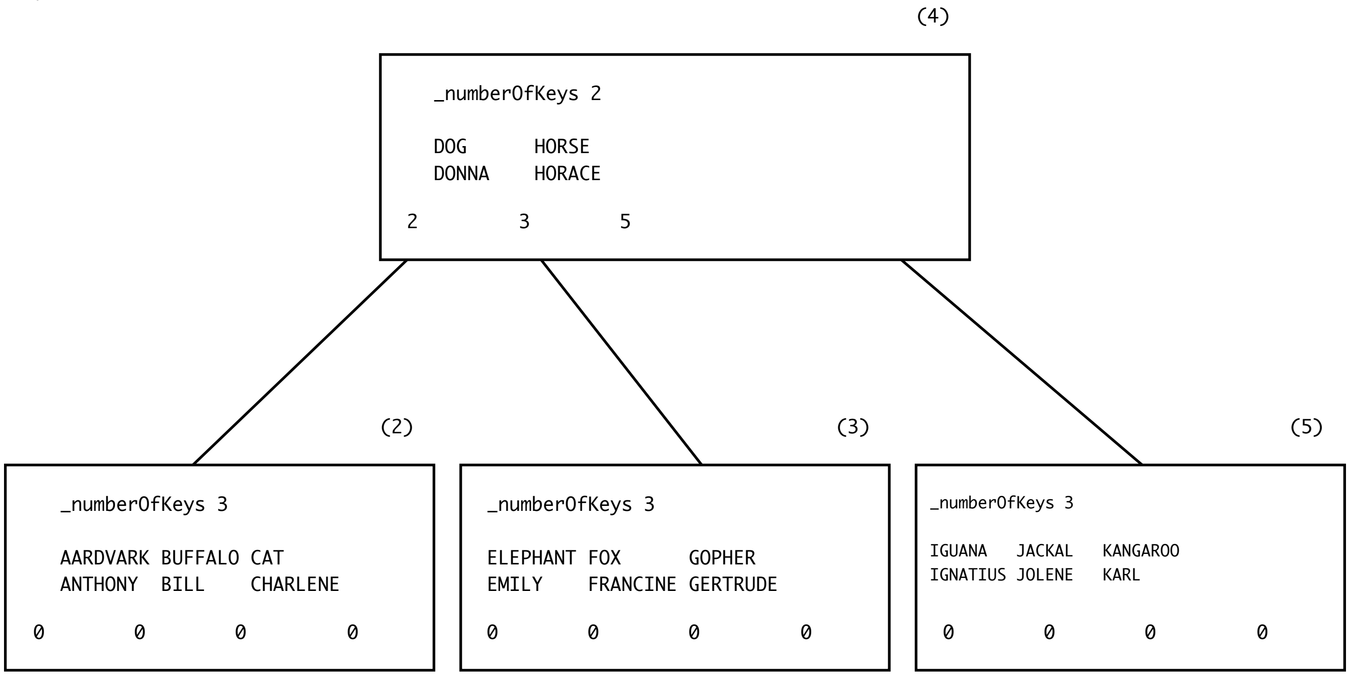
   Please be sure to study this header carefully to see what routines are available to you! Do not reinvent the wheel!

   An important thing to note about this class is that the arrays in it are declared large enough to hold one extra key/value pair and associated child. This simplifies insert - you can put the new information into the `BTreeBlock` object and then see if it has become over-full. Of course, an over-full `BTreeBlock` cannot be transferred to or from disk - it must be split first.

7. It is possible to fulfill this assignment simply by writing two (or three) methods of class `BTree`. You may find it helpful to add `private` methods to this class and/or to add methods to class `BTreeBlock`, but this is not mandatory. How you choose to do the job is up to you, but the quality of your code contributes to 10% of the grade.

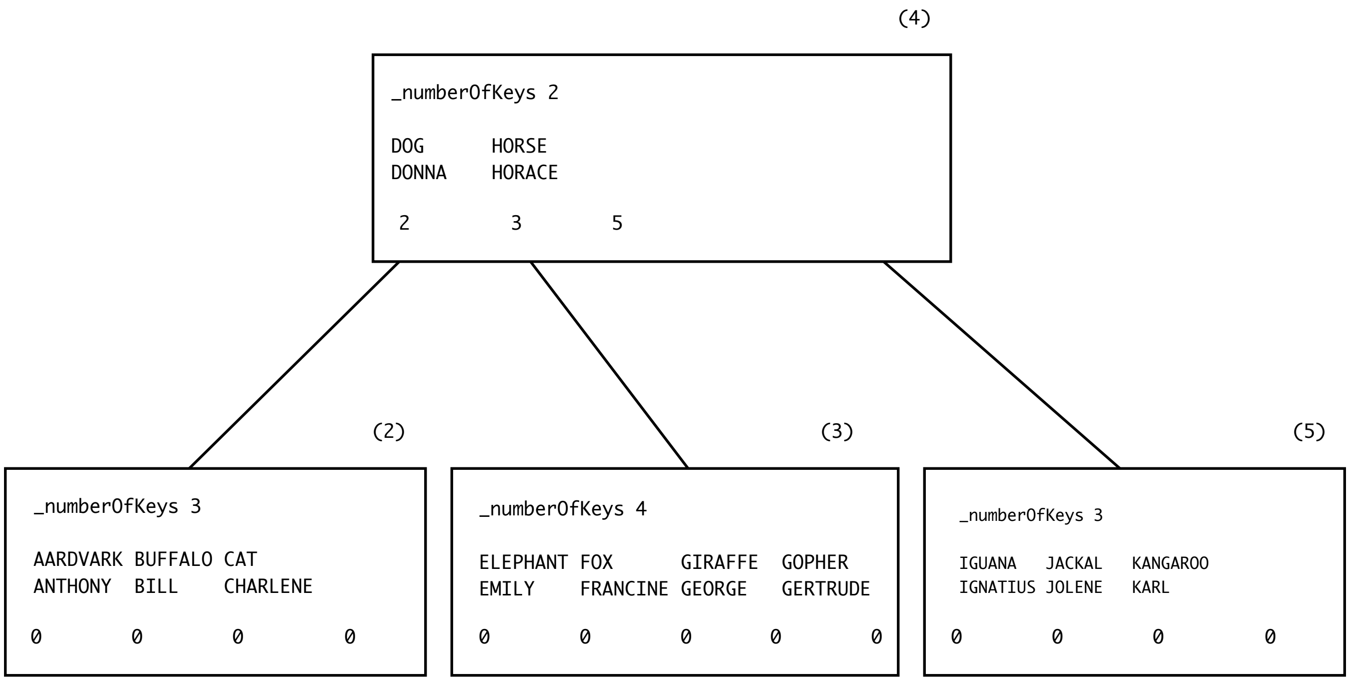
Example:

Consider the following B-Tree: (Where the number in the upper-right hand corner of each block is the block number - e.g. the root is stored in block 4; its children are blocks 2, 3 and 5. The keys are `DOG`, `HORSE`, etc. and the associated values are `DONNA`, `HORACE`, etc.)

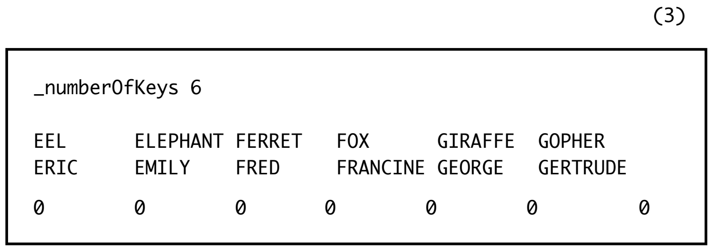


Suppose we now want to insert the key `GIRAFFE` (with associated value `GEORGE`) into the tree. The following sequence of operations would occur:

1. `getRoot()` would be called, and would return that the root is in block 4.
2. `getBlock(4 ...)` would be called to make an in-memory copy of the contents of block 4.
3. `getPosition(GIRAFFE)` invoked on this block would report that this key belongs at position 1. Since this block is not a leaf, `getChild(1)` sends us to block 3.
4. `getBlock(3 ...)` would be called to make an in-memory copy of the contents of block 3.
5. `getPosition(GIRAFFE)` invoked on this block would report that this key belongs at position 2. Since this block is a leaf, we know that the key/value pair actually goes here, not in a subtree.
6. `insert(2, GIRAFFE ...)` invoked on this block would insert the new key/value pair into the block, causing the information to the right of it (here `GOPHER/GERTRUDE` and the child to the *right*) to be slid over one space, and would increment the number of keys in the block.
7. Since the block is not over-full (`splitNeeded()` returns false), `putBlock(3 ...)` would be called to write the modified block back to block 3 on disk.
8. As a result, the tree would now look like this:



Again, consider what would happen if we did additional insertions until block (3) became full - i.e. so it looks like this:


(3)
_numberOfKeys 6
EEL ELEPHANT FERRET FOX GIRAFFE GOPHER
ERIC EMILY FRED FRANCINE GEORGE GERTRUDE
0 0 0 0 0 0 0

Now suppose we want to insert `EMU` (with value `EMIL`) into this block.

1. We would follow a process similar to the earlier example to work our way down to block 3, eventually inserting `EMU` at position 2, and sliding four remaining keys, values, and children over.
2. This results in a block that contains seven keys, seven values, and eight children - which is over-full by one (`splitNeeded()` returns true)
3. At this point, we would create another `BTreeBlock` object (in memory) invoke the `split()` method on the over-full block produce the results shown below - with the key `FERRET` and associated value `FRED` being returned to be "promoted" to the root. (Note that the parameters of this method are reference parameters representing values returned to the caller.)

```
                          (3)                                              (not yet assigned)
 ---------------------------------                        ---------------------------------
|  _numberOfKeys 3                |  (Returned to promote |  _numberOfKeys 3                |
|                                 |  to parent:           |                                 |
|  EEL      ELEPHANT EMU          |                       |  FOX      GIRAFFE  GOPHER       |
|  ERIC     EMILY    EMIL         |  FERRET               |  FRANCINE GEORGE   GERTRUDE     |
|                                 |  FRED  )              |                                 |
| 0       0        0        0     |                       | 0       0        0        0     |
 ---------------------------------                         ---------------------------------
```

4. `putBlock(3 ...)` would be used to write the modified "left" block to disk (now containing just the three keys `EEL, ELEPHANT` and `EMU` plus associated values.)
5. `allocateBlock()` would be called to get the number of an available block on disk - say 6.
6. `putBlock(6 ...)` would be used to write the newly created "right" block to disk (containing the three keys `FOX, GIRAFFE`, and `GOPHER` plus associated values.)
7. At this point, if you were not using a recursive approach to insert, you would retrieve the contents of block 4 again (using `getBlock()`), then use `getPosition(FERRET)` to find where it belongs. (If using recursion, these values would be "remembered" in the local variables of the parent call.)
8. In either case, using `insert(1, FERRET, FRED, 6)` on this block would put the promoted key and value into it along with the new child.
9. `putBlock(4 ...)` would be used to write this modified block to disk.

The root block would now look like this:

```
                                          (4)
 ---------------------------------------------
|  _numberOfKeys 3                            |
|                                             |
|  DOG      FERRET   HORSE                    |
|  DONNA    FRED     HORACE                   |
|                                             |
| 2        3        6        5                |
 ---------------------------------------------
```

Testing procedure:

1. When you run the test driver, it will ask you for the name of a disk file containing the tree.

   ```
   Name of file containing the B-Tree:
   ```

   If a file of this name exists, it will be opened and the tree contained in it will be used; if not, a new file containing an empty tree will be created. If an existing file is opened, a message like the following will be displayed:

   ```
   Opened file foo.file of degree 7 using 12 blocks. Root at 12, first free 0
   ```

   The information reported is read from the header block of the file.

   If a new file is created, a message like the following will be displayed:

   ```
   Created file foo.file of degree 7 using 1 blocks. Root at 0, first free 0
   ```

2. The test driver accepts the following commands. (In each case, you enter just the first letter of the command; the portion listed in parentheses is just to explain what the command means.)

   ```
   I(nsert) key value
   L(ookup) key
   R(emove) key
   P(rint)
   D(ump) start [finish]
   Q(uit)
   ```

   Where key is a single word, preceded and followed by a space, but containing no spaces and value is the remainder of the line after the key (including additional spaces). (Of course, only insert specifies a value; lookup specifies a key and returns the value, and remove specifies only a key.)

   Example: The following command

   ```
   I Aardvark Anthony A.
   ```

   would be interpreted as a command to insert the key "`Aardvark`" with value "`Anthony A.`" in the tree.

3. There are two commands available to print the tree, or portions of it, for verifying correctness and/or debugging:

   `P` will print out the *entire tree* using a preorder traversal. Indentation is used to highlight the structure of the tree. The following is an example of what the Print command will produce for a small tree. (Note: all the code needed to support this has been written for you.)

   ```
   BTree in file demo.tree of degree 7 using 4 blocks. Root at 4, first free 0

   Block 4 at level 1 contains 1 key(s)
      2               DOG                               DONNA
      3

     Block 2 at level 2 contains 3 key(s)
        0          AARDVARK                              ANTHONY
        0           BUFFALO                                 BILL
        0               CAT                             CHARLENE
        0

     Block 3 at level 2 contains 3 key(s)
        0          ELEPHANT                                EMILY
        0               FOX                             FRANCINE
        0            GOPHER                             GERTRUDE
        0
   ```

   `D` will print out one or more *specific blocks*. If D is entered followed by a single integer, that block number will be printed; if it is followed by two integers, all the blocks in that range will be printed. (e.g. in the above example, `D 3 4` would print out the block containing `ELEPHANT`.., then the block containing `DOG`, without level numbers or indentation.)

4. When the program terminates normally, the destructor will print a final message recording how many blocks were read and written in that run of the program, (including reads/writes to the header and/or free blocks done by the supplied code for class File.)

   ```
   Closing file demo.tree of degree 7 using 4 blocks.  Root at 4, first free 0
   Total gets done: 9 (0 header, 0 free blocks).
   Total puts done: 15 (6 header, 0 free blocks).
   ```

   (e.g. the above resulted from a session that created the above tree from scratch. Inserting each key after the first resulted in a get for the block into which it was placed, and three gets were done while printing the tree. Inserting each of the seven keys resulted in one put for the block in which it was placed; the last key resulted in two additional puts because a block was split and a new root was created. The six header puts resulted from creating the header initially, updating the header after each of the three new blocks was allocated, and changing the root twice - once when the first key was entered, and once when it was split.

5. You can use the `test.tree` supplied file for testing `lookup()`, and also for some testing of `insert()`.

6. To simplify more thorough testing of `insert()`, you can use a file as input, rather than the keyboard. Simply put all your commands (including the specification of the tree name) into a file and redirect standard input as follows:

   ```
   ./project4 < demo.in
   ```

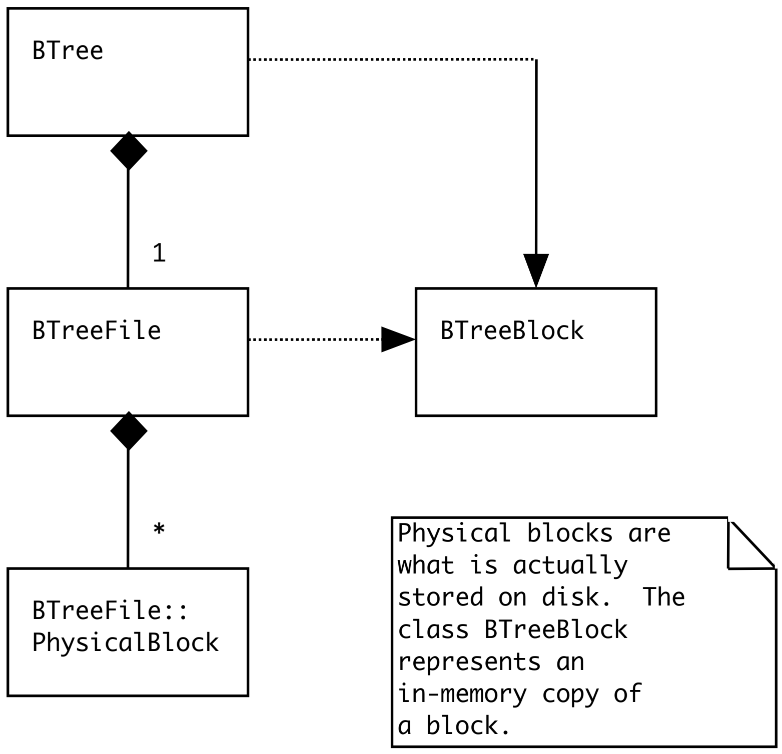
   Three files of test data (`onelevel.in`, `twolevel.in`, `threelevel.in`) are provided to facilitate testing of your operation, but you may want to create your own test data files - and will certainly need to do so if you implement `remove()`. Of course, you will need to verify the correctness of the printed results by inspection. Also, if you use a command file more than once, be sure to delete the tree created between runs to avoid putting the same information in the tree twice.

Turn in:
1. Source code for any files you actually changed.
2. Test data and results of testing.

Also leave your project folder on the file server for testing - and indicate which partner's account it is in.

Appendix A

UML Diagram for Classes Used in this Project


BTree
1
BTreeFile
BTreeBlock
*
BTreeFile::
PhysicalBlock
Physical blocks are what is actually stored on disk. The class BTreeBlock represents an in-memory copy of a block.