Data Structures
Assignment 3: Tables and Indices

Pablo Cuesta Sierra and Álvaro Zamanillo Sáez

Contents

1 Goal of the practice

In this assignment we implement a simplified database that stores the title of a book and a primary key that is associated with each book. For this implementation, we use two files per database: the data file (where all the records are stored), and the index file (where we implement an BTree, with one element per node, in order to search easily for the keys). This way, we do not store the data of our database in RAM memory, but in files that can be accessed simultaneously by different programs.

1.1 Functions to be implemented

In this assignment we are required to implement the following functions which are based on several binary tree algorithms:

1. `printTree`: traversal of the index file in order to print all of the primary keys as well as their position in the binary tree.
2. `findKey`: search algorithm in a Binary Search Tree (stored in a .idx file).
3. `addTableEntry`: this function inserts a record in the data file (.dat).
4. `addIndexEntry`: this function inserts a node in the index file (.idx).

We also have to code `createIndex` and `createTable`, but they are very simple functions that only check whether a file exists, and if not, it is created and the initialization headers are written.

2 Explanation of each algorithm

2.1 Binary search

Binary search is a recursive algorithm that looks for a key in a binary tree (for all subtree T' of T it is satisfied that info(left(T')) is smaller than T' and info(right(T')), greater). The algorithm starts at the root and compares the info(Root) with the key that is being searched. If it is not found the algorithm calls itself recursively on the left or right subtree depending on whether the info of the node was lower or higher than info(Root). The algorithm stops when the key is found or when the child that has to be visited next does not exist, which means the key is not in the tree. The node where this algorithm stops is where the node with the key that was being searched should be inserted (in case it is not in the tree).

Example of search:

Figure 1: Example BST.

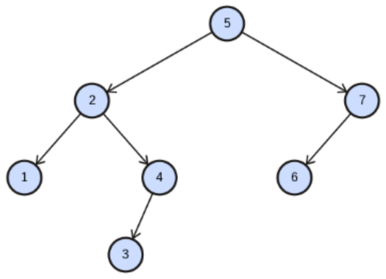


Figure 2: Search 6.

Current node	Comparison	Next step
5 (Root)	6>5	->Right child
7	6<7	->Left child
6	6=6	return FOUND

Figure 3: Search 8.

Current node	Comparison	Next step
5 (Root)	8>5	->Right child
7	8>7	->Right child
-1		return NOT FOUND

2.2 Insert

Insertion starts with a binary search. If the key is found it is not inserted due to the fact that primary keys must be unique. If it is not found, we append a new node (a child) to the last node visited in the search operation. Because we must keep the tree being a binary search tree, the child will be appended in the left if its info is smaller than its parent's info, or to the right otherwise.

The following is an example of insertion of the node '9' in the tree as in Figure 1.

Figure 4: Initial tree.

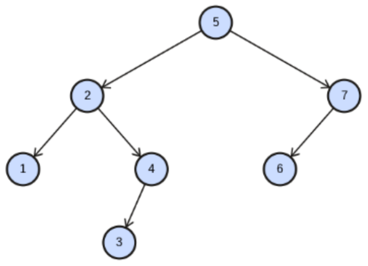
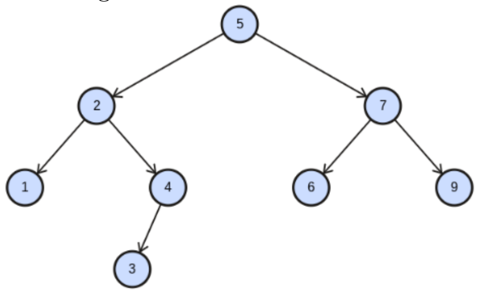


Figure 5: Search 9.

Current node	Comparison	Next step
5	9>5	->Right child
7	9>7	->Right child
-1 (the key was not in the tree, we can insert it as a child of 7)		Insert

Figure 6: Tree after insertion.

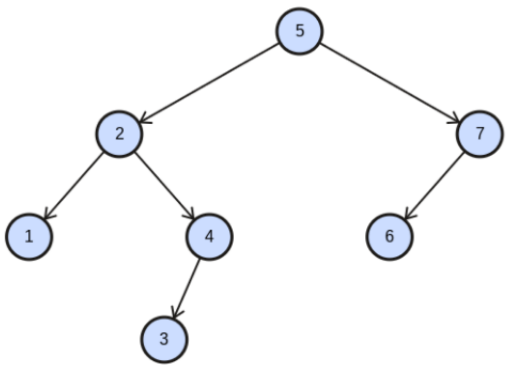


2.3 Preorder traversal

The goal here is to visit all the nodes of the tree exactly once. For every subtree, this algorithm visits the root and then, recursively, its left subtree and right subtree (in that order).

An example:

Figure 7: Example of preorder.



Order of visits when applying preorder: 5, 2, 1, 4, 3, 7, 6.

3 Explanation of the functions

Note on deleted records.

We have seen on the test functions provided for this assignment (in particular, in `createTestDataFile`), that the deleted records in the data files have: offset (in the PK field) of the next deleted record and size of the title that could fit in that record. Therefore, we have assumed that is the structure of those records. So we have defined in utils.h the following structure to treat them in a more abstract way when handling them.

```
typedef struct deleted_record {
    int next;  /* offset of the next deleted
                  record: -1 if it is the last*/
    int title_len;  /* title length */
} Deleted_record;
```

3.1 FindKey

This function is a binary search in the index file. As mentioned in 2.1, binary search starts at the root, therefore, we need to read the header of the index file to know which node is the root of our tree. Once in the root, we start the recursive algorithm described before. To go from one node to another (like its child) we just have to move the cursor in the binary file by using fseek(). The number of bytes to move the cursor is given by the nodeId: because all nodes have the same length, we just have to multiply the id by the size of the node and sum the header size (if we are counting from the beginning of the file). The code implementation is without tail recursion. Instead, it uses a loop where the "base case" (the next node to visit does not exist) is checked in every iteration.

3.2 AddIndexEntry

This time, the algorithm behind this function is the insertion in a binary tree. The search step mentioned in 2.2 is done by calling the function `FindKey`. The last visited node is stored in a variable which is passed to FindKey, thus we know where to add the node in the tree (if the key was not already in it). Regarding the implementation with files, we distinguish two cases: there are no deleted nodes, or there are. If there are no deleted nodes we just write the entry at the end of the file whereas, in the other case, we write it in the first deleted node. In the index file all nodes have the same length (fixed-length records), thus we just insert it at the first deleted record and updated the header with the new first deleted record.

3.3 AddTableEntry

This function requires once more to do a (binary) search in the index file. If we are to add an entry, we have to take into account the possible deleted records. However, in the data file the records have variable length so we must traverse the linked list of deleted records until we find one big enough for our entry (first-fit). In the event there was no record that could fit the new entry, we would add it at the end of the file just like when there are no deleted records. Once the new book is inserted in the data file, we have to update the index file.

Furthermore, when inserting in a deleted record we try to reuse the extra space (when there is more than a minimum for the title which is defined in a macro[1]). So, for instance, if we have a deleted record of size 20 bytes (for the title) and we insert a book of length 4, we reuse that 16 extra bytes: 4 bytes for the primary key (which is the offset of the next deleted record), 4 bytes for the length of this new deleted record (8 bytes in this case) and the 8 bytes for the title.

3.4 printTree

Finally to print the tree we have coded two functions: `printTree` and `printnode`. The first one is basically a wrapper where we call `printnode` from the root. This first function receives also as an argument the maximum level that is to

[1]This macro is called `MIN_TITLE` and it is defined as 1, so as to run our test `checkAddTableEntry`.

be printed of the tree. The function `printnode` is the recursive function that follows the algorithm of pre-order traversal described above (2.3), until the maximum depth has been reached, or there are no deeper nodes. For every node, it prints the primary key of the node, whether it is the left or right child of its parent node, its index in the index file, and the offset of its record in the data file. The indentation of each node (proportional to its depth) is managed trough an argument which indicates the depth of the node which is being printed.

The following is an example of the result of this function when printing the tree used by the test functions:

```
MAR2 (0): 4
    l MAR0 (2): 37
        l BAR0 (5): 88
            r CAR5 (9): 157
        r MAR1 (6): 105
    r VAR1 (3): 53
        l PAR2 (7): 121
            r RAR2 (10): 174
        r WAR3 (1): 21
```

4 About checkAddTableEntry

For the test of this function we have contemplated several cases from a given table (created with `createTestFiles`). Firstly, trying to insert an existing key should result in a warning because primary keys must be unique. Next case is when there are no deleted records, so the insertion is made at the end of the file. On the other hand if there are deleted records we have to assure that the linked list of deleted records is preserved and the header updated. Furthermore, if there is enough extra memory for a new deleted record, the test checks that this new deleted record and its fields are handled correctly.