Advanced Database Organization - Spring 2024
CS 525 - 04/05
Programming Assignment IV: B+-Tree
Due: Saturday, May 4th, 2024 by 11:59pm

---

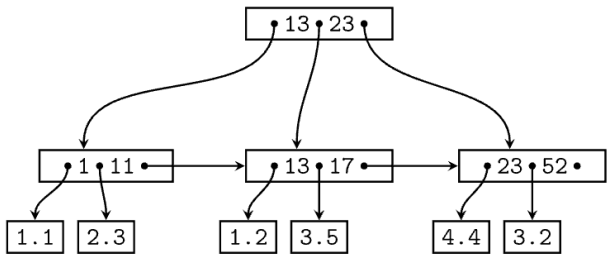
1. Task

The goal of this assignment is to implement a `B+-tree` index. The index should be backed up by a page file and pages of the index should be accessed through your buffer manager. As discussed in the lecture, each node should occupy one page. However, for debugging purposes you should support trees with a smaller fan-out and still let each node occupy a full page. A `B+-tree` stores pointer to records (the `RID` introduced in the last assignment) index by keys of a given datatype. The assignment only requires you to support `DT_INT` (integer) keys (see optional extensions). Pointers to intermediate nodes should be represented by the page number of the page the node is stored in.

To make testing easier your implementation should follow the conventions stated below.

- `Leaf Split`: In case a leaf node need to be split during insertion and `n` is even, the left node should get the extra key. E.g, if `n = 2` and we insert a `key 4` into a node `[1,5]`, then the resulting nodes should be `[1,4]` and `[5]`. For odd values of `n` we can always evenly split the keys between the two nodes. In both cases the value inserted into the parent is the smallest value of the right node.
- `Non-Leaf Split`: In case a non-leaf node needs to be split and `n` is odd, we cannot split the node evenly (one of the new nodes will have one more key). In this case the `middle` value inserted into the parent should be taken from the right node. E.g., if `n = 3` and we have to split a non-leaf node `[1,3,4,5]`, the resulting nodes would be `[1,3]` and `[5]`. The value inserted into the parent would be `4`.
- `Leaf Underflow`: In case of a leaf underflow, your implementation should first try to `redistribute` values from a sibling and only if this fails `merge` the node with one of its siblings. Both approaches should prefer the left sibling. E.g., if we can borrow values from both the left and right sibling, you should borrow from the left one.

You can use the `B+-tree` shown below to sanity check your implementation of inserts. It has been generated by the following sequence of key insertions `1,11,13,17,23,52` for RIDs `1.1,2.3,1.2,3.5,4.4,3.2`.



```c
#ifndef BTREE_MGR_H
#define BTREE_MGR_H

#include "dberror.h"
#include "tables.h"

// structure for accessing btrees
typedef struct BTreeHandle {
  DataType keyType;
  char *idxId;
  void *mgmtData;
} BTreeHandle;

typedef struct BT_ScanHandle {
  BTreeHandle *tree;
  void *mgmtData;
} BT_ScanHandle;

// init and shutdown index manager
extern RC initIndexManager (void *mgmtData);
extern RC shutdownIndexManager ();

// create, destroy, open, and close an btree index
extern RC createBtree (char *idxId, DataType keyType, int n);
extern RC openBtree (BTreeHandle **tree, char *idxId);
extern RC closeBtree (BTreeHandle *tree);
extern RC deleteBtree (char *idxId);

// access information about a b-tree
extern RC getNumNodes (BTreeHandle *tree, int *result);
extern RC getNumEntries (BTreeHandle *tree, int *result);
extern RC getKeyType (BTreeHandle *tree, DataType *result);

// index access
extern RC findKey (BTreeHandle *tree, Value *key, RID *result);
extern RC insertKey (BTreeHandle *tree, Value *key, RID rid);
extern RC deleteKey (BTreeHandle *tree, Value *key);
extern RC openTreeScan (BTreeHandle *tree, BT_ScanHandle **handle);
extern RC nextEntry (BT_ScanHandle *handle, RID *result);
extern RC closeTreeScan (BT_ScanHandle *handle);

// debug and test functions
extern char *printTree (BTreeHandle *tree);

#endif // BTREE_MGR_H
```

2.1. **Index Manager Functions.** These functions are used to initialize the index manager and shut it down, freeing up all acquired resources.

2.2. **B$^+$-tree Functions.** These functions are used to create or delete a b-tree index. The latter should also remove the corresponding page file. Furthermore, before a client can access an b-tree index it has to be opened (`openBtree`). When closing a b-tree (`closeBtree`), the index manager should ensure that all new or modified pages of the index are flushed back to disk (use the buffer manager function for that).

2.3. **Index Manager Functions.** These functions are used to initialize the index manager and shut it down, freeing up all acquired resources.

2.4. **Key Functions.** These functions are used to find, insert, and delete keys in/from a given `B+-tree`.

- `findKey` returns the `RID` for the entry with the search key in the b-tree. If the key does not exist this function should return `RC_IM_KEY_NOT_FOUND` (see `dberror.h`).
- `insertKey` inserts a new key and record pointer pair into the index.

  It should return error code `RC_IM_KEY_ALREADY_EXISTS` if this key is already stored in the b-tree.
- `deleteKey` removes a key (and corresponding record pointer) from the index.

  It should return `RC_IM_KEY_NOT_FOUND` if the key is not in the index. For deletion it is up to the client whether this is handled as an error.
- Furthermore, clients can scan through all entries of a BTree in sort order using the `openTreeScan`, `nextEntry`, and `closeTreeScan` methods.
- The `nextEntry` method should return `RC_IM_NO_MORE_ENTRIES` if there are no more entries to be returned (the scan has gone beyond the last entry of the `B+-tree`).
  Below is an example of how the scan can be used.

```
BT_ScanHandle *sc;
  RID rid;
  int rc;

  startTreeScan(btree, sc, NULL);

  while((rc = nextEntry(sc, &rid)) == RC_OK)
    {
    // do something with rid
    }
    if (rc != RC_IM_NO_MORE_ENTRIES)
       // handle the error
  closeTreeScan(sc);
```

2.5. **Debug functions.** `printTree` is used to create a string representation of a b-tree. It is used in the test cases and can be helpful for debugging. Your code **has to** generate the format described in the following. Each node of a b-tree is represented as one line in the string representation. The order of nodes should be in depth-first pre-order. The following representation should be used for each node `(pos)[pointer,key,pointer, ...]`. Key should be represented using the `serializeValue method` from the last assignment. Pointers to nodes should be shown as the node position according to depth-first pre-order traversal. Thus, the string representations generated by `printTree` is independent on the actual positions of the index nodes on disk. `RID=s should be represented as =PageNumber.Slot`. As an example reconsider the b-tree shown above. Its string representation is shown below.

```
(0)[1,13,2,23,3]
(1)[1.1,1,2.3,11,2]
(2)[1.2,13,3.5,17,3]
(3)[4.4,23,3.2,52]
```

3. Optional Extensions

- `Integrate the B+-tree with the record manager`: Modify your record manager to use the `B+-tree` for search. That is when creating a new relation with the record manager, you should let the user specify which attribute is the key of the table. The record manager should then create and maintain the `B+-tree` for storing the keys from this attribute. When a new record is inserted, the corresponding key should be inserted in the B+-tree. A scan that searches for a record with a given key should use the `B+-tree` instead of scanning through all the records. If you want to support keys with more than one attribute, you are allowed to change the b-tree interface to support that.
- `Allow different data types as keys`: Allow the other data types introduced in `assignment 3` to be used as keys for the `B+-tree`.
- `Implement pointer swizzling`: Once a `B+-tree` node is memory, conceptual pointers to that node (`pagenum`) should be actual memory pointers. Note that B-trees pages are only pointed to by their intermediate parent node. Thus, a simple way to realize swizzling at page load time, is to immediately replace the pointer to a child node with the memory pointer once some code tries to access it. The complicated part is to make sure that memory pointers are replaced with pagenum if the buffer manager evicts a page from the buffer that was pointed to by a memory pointer. A reasonable way to implement this is to create a bookkeeping data structure that stores all places of pointers to a node that have been replaced by memory pointers. The buffer manager has to be extended with a callback function parameter. This callback is called whenever a page is evicted. For this extension the callback would be a clean-up function that accesses the bookkeeping data structure to figure out where it has to replace memory pointers with page numbers. This data structure is also needed when a node gets evicted that has memory pointer to one or more of its child nodes.
- `Support multiple entries for one key`: Instead of enabling only a single `RID` to be stored for each key, allow insertions of multiple entries with the same key. You are allowed to change the `btree_mgr.h` to support accessing all the entries for a key. Please do not change the signature of the `findKey` method, but rather add a new `findAllEntries` method that returns an array/list of entries. The `findKey` should return the first entry only. Instead of `RIDs` the pointers in leave nodes should reference a page storing all entries for a key. If there are more entries for a key, then you can fit onto one page, additional pages should be organized as a linked list. That is each page stores a pointer to the next page.

4. Source Code Structure

Your source code directories should be structured as follows. You should reuse your existing storage manager and buffer manager implementations (also record manager if you plan to implement the record manager integration extension). So before you start to develop, please copy these implementations into the texttt `assign4` folder.

- Put all source files in a folder `assign4` in your git repository
- This folder should contain at least
  - the provided header and `C` files
  - a makefile for building your code `Makefile`.
  - a bunch of `*.c` and `*.h` files implementing the `B+-tree`.
  - `README.txt/README.md`: A markdown or text file with a brief description of your solution

  Example, the structure may look like that:

```
git
   assign4
       Makefile
       buffer_mgr.c
       buffer_mgr.h
       buffer_mgr_stat.c
       buffer_mgr_stat.h
       btree_mgr.c
       btree_mgr.h
       dberror.c
       dberror.h
       dt.h
       expr.c
       expr.h
       record_mgr.c
       record_mgr.h
       rm_serializer.c
       storage_mgr.h
       tables.h
       test_assign4_1.c
       test_expr.c
       test_helper.h
```

5. Test cases

- `test_helper.h`
  - Defines several helper methods for implementing test cases such as `ASSERT_TRUE`.
- `test_expr.c`
  - This file implements several test cases using the `expr.h` interface. Please let your make file generate a `test_expr` binary for this code. You are encouraged to extend it with new test cases or use it as a template to develop your own test files.
- `test_assign4_1.c`
  - This file implements several test cases using the `btree_mgr.h` interface. Please let your make file generate a `test_assign4` binary for this code. You are encouraged to extend it with new test cases or use it as a template to develop your own test files.