B+ Tree Design

A node is represented as a C struct. There is only 1 struct object for internal and leaf nodes. Each node struct contains the following information:

- capacity: max number of keys to hold.
- nodeType: either ‘leaf’ or ‘node’ (for internal node).
- children: array of pointers to children.
- parentPtr: pointer to parent node.
- leftSisterPtr (and rightSisterPtr): pointers to left and right siblings (for leaf nodes).
- keys: array of keys.
- values: array of values.

Each node contains ‘m’ keys where ***d*** **<=** ***m*** **<=*2d*** and children (fanout) between ***d*** **+** ***1*** and ***2d*** **+** ***1***. Parameter ***d*** is a constant and can be adjusted in ‘main.c’. It is set up to 124 which is equivalent to 1 page size (about 4kb. for a leaf node). Whenever the threshold is reached rebalancing occurs.

FIND algorithm: it receives the root of the tree and the key to be searched. Then recursively navigates the tree top down until finding the leaf. It uses the help of a function (getNextChild) to find the next child and another function to find the the requested value (getValue). If the key is not found in the tree the return value is 0.

- getNextChild: returns child at position *[i]* where *key < keys[i] (*keys and children are always sorted).
- getValue: returns value at position *[i]* where *key == keys[i]*.

INSERTION algorithm: it receives a node and the key and value to be inserted. Recursively navigates the tree top down until finding the leaf (with the help of getNextChild). The key is added using insertion sort with the function addKV. Then, if the leaf has exceeded its key capacity, it is sent to the function splitLeaf. Otherwise it is passed to traverseTreeBottomUp. This last function traverses the tree up to return the root of the tree. The pseudo code would be:

```
if (leaf is over capacity)
    return  traverseTreeBottomUp( splitLeaf ( leaf ) );
else
    return traverseTreeBottomUp( leaf );
```

- splitLeaf: multiples helper functions are used. It creates 2 nodes and distributes the leaf keys and values to those nodes. The parent of those new leaf nodes becomes either a new node (to become the new root), or the parent of the original leaf. The original leaf is “destroyed” (all its pointers and the leaf itself is deallocated) and the new parent is returned. Finally all the sibling pointers are reassigned.

- traverseTreeBottomUp: is calls the parent of each node recursively until finding the root (the node whose parent is NULL). Each node is checked to make sure it is not over capacity. If it is, then it is passed to the splitNode function. The pseudo code would be:

```
if (node is over capacity)
    return traverseTreeBottomUp( splitNode( node ) );
else
     traverseTreeBottomUp( node→parent );
```

- splitNode: similar to splitLeaf but splitting keys and child pointers instead. 1 key is lifted to the parent node and deleted from the new right node. The old node is freed and the new node is returned.

INSERT algorithm takes about 2 times longer to complete than FIND. FIND takes ***LOG-fanout-(N)***, where ***fanout = 248*** and ***N = # of nodes*** (assuming 1 node is equivalent to 1 page size). INSERT would be 2 * FIND given that it has to traverse the tree down to a leaf and back up to the root (we are leaving aside the cost for splitting nodes if the key limit is exceeded).

Range scan is a lot more costly than FIND since it has to access all the pages (nodes) where the values requested exist in the range. Range scan would take 2*O(N) where N is the number of leafs to navigate (or number of pages to read). There are 2 scans, 1 to determine the number of values to retrieve and allocate the memory for the array, the second pass is to actually collect those values.

There are a functions to test the tree by inserting values and displaying the keys and/or multiple statistics about the tree. Examples:

- Inserting 6M key-values sequentially. Keys and values are equal (functions found in ***main.c*** in ***main*** function):

```
insertValues(rootPtr, -1000000, 5000000, 's');
treeInfo(rootPtr);
```

```
antony@discordia:~/Desktop/S165-A1$ make && time ./main
gcc -std=c99 -D_GNU_SOURCE -ggdb3 -W -Wall -Wextra -Werror -O3   -c -o main.o main.c
gcc -std=c99 -D_GNU_SOURCE -ggdb3 -W -Wall -Wextra -Werror -O3 -o main main.o

insertValues (mode s): Values Inserted!


==== TREE INFO: ====

- Is Root: true
- Root type: node
- Hight: 4
   *Direct children 3
- Internal nodes: 391
- Leaf nodes: 48387
- Incorrect nodes (overcapacity): 0
- Total values: 5999999
- Avg. leaf occupancy: 124.000
- Max capacity: 248

real    0m4.881s
user    0m4.827s
sys     0m0.053s
```

- Insert 6M key-values generating random integers:

insertValues(rootPtr, -1000000, 5000000, 'r');
treeInfo(rootPtr);

```
antony@discordia:~/Desktop/S165-A1$ make && time ./main
gcc -std=c99 -D_GNU_SOURCE -ggdb3 -W -Wall -Wextra -Werror -O3   -c -o main.o main.c
gcc -std=c99 -D_GNU_SOURCE -ggdb3 -W -Wall -Wextra -Werror -O3 -o main main.o

insertValues (mode r): Values Inserted!


==== TREE INFO: ====

- Is Root: true
- Root type: node
- Hight: 3
   *Direct children 246
- Internal nodes: 247
- Leaf nodes: 34199
- Incorrect nodes (overcapacity): 0
- Total values: 5994305
- Avg. leaf occupancy: 175.277
- Max capacity: 248

real    0m6.445s
user    0m6.386s
sys     0m0.048s
```

- Insert some values and test range scan:

insertValues(rootPtr, -100, 200, 's');
testRangeScan(rootPtr, -500, 150);

```
antony@discordia:~/Desktop/S165-A1$ make && time ./main
gcc -std=c99 -D_GNU_SOURCE -ggdb3 -W -Wall -Wextra -Werror -O3   -c -o main.o main.c
gcc -std=c99 -D_GNU_SOURCE -ggdb3 -W -Wall -Wextra -Werror -O3 -o main main.o

insertValues (mode s): Values Inserted!

Scan results:
-100
-99
-98
-97
-96
-95
-94
-93
-92
-91
```

(results continue)