Name: Rahul S Mayuranath
SRN: 01FB15ECS225

B-Tree

A B-tree is a self balancing search tree. It's usually implemented when all the nodes of the tree do not fit in memory. It's widely used for database implementations. It has a minimum degree t; Every node except the root have at least t-1 records and at most 2t-1 records. Hence, number of records is limited by 2t.

Citation:
The insert process is an implementation of the pseudo code present in the CLRS textbook.
For delete, the following geeksforgeeks link was used as a reference.
http://www.geeksforgeeks.org/b-tree-set-3delete/

Implementing B-tree as an array:

To implement b-tree as an array, we have a structure for a b-tree node that stores the following information:

1. Size of the node
2. Position of the node in the array
3. A boolean value indication whether the node is a leaf
4. Array of type Record with 2t-1 nodes.
5. An int array of size 2t containing the locations of the children in the array.

A structure for a b-tree is maintained holding the following information:

1. The root of the tree.
2. The next free position we can insert a node.
3. An array of type bTreeNode that can store a million different records.

The tree is allocated the size of the bTree. Root and next position to insert are initialized to 0.

Implementing Search:

Implementing search is pretty straight forward. We initially pass the root to the search function, if the record is not present in the root, we find the appropriate child and recurse the subtree rooted at the child. Eventually if we reach a leaf node, we return NULL indicating that the search record was not found.
If it is found, we return the node where the record was found.

Implementing Insert:

A new record is always inserted only in a leaf node. Before inserting a node, we’ve to make sure the number of records in a node is bounded by 2t-1.
In all the insert operations, a one pass mechanism is employed. While traversing down the tree to find the appropriate position to insert the record if a node contains 2t-1 nodes, it is split using the splitChild function right there so that later if we push a record up to a parent node, we don’t have to worry about the

parent being full. This is also a key optimisations for disk operations since you don’t want to keep swapping nodes to and from the disk.
If this is the first record being inserted into the tree, We create a new root node and insert it into it.
Now, 2 different cases arise.
If the root node is full, we call the split child function.
In split child, we create an empty node z and copy over the first t-1 nodes. The middle node is pushed up to the parent and node y on who the split child was called has the remaining t-1 nodes. If y is not a leaf node, the children are also copied onto z. Y and z are set as children of the new parent node. Finally, we write into the array.
If the root child is not full, we call the insert_nonfull function.
In non full, if the node is a leaf node, we insert it there and return. Else we split the node and call insert_nonfull recursively on the child node until we reach an appropriate leaf node.

Implementing Delete:

Delete is a complicated procedure on a btree. To implement delete, we have to worry about if it is a leaf node or not and have to make sure no node except the root node has a degree lesser than t-1.
There are basically 3 possible cases for delete:
Case 1:

Name: Rahul S Mayuranath
SRN: 01FB15ECS225

If the record is in node x and x is leaf such that once we delete it still has at least t-1 records, we delete the node and write it to the array.
Case 2a:
If the child y that precedes k in node x has at least t records, then find the predecessor k0 of k in the sub-tree rooted at y. Recursively delete k0, and replace k by k0 in x.
Case 2b:
If y has fewer than t records, then, symmetrically, examine the child z that follows k in node x. If z has at least t records, then find the successor k0 of k in the subtree rooted at z. Recursively delete k0, and replace k by k0 in x.
Case 2c:
If both y and z have only t-1 records, merge k and all of z into y, so that x loses both k and the pointer to z, and y now contains 2t-1 records. Then free z and recursively delete k from y.
Case 3: If the record is not present in the internal node, we find the appropriate subtree rooted at one of the children and will result in one of the following cases:
Case 3a:
If there is a sibling with at least t keys, we move a record from the parent to the node and push an appropriate node to the parent from one of siblings.
Case 3b:
If the siblings only have t-1 keys, we merge the child with one of the siblings.

The following helper functions are used to implement delete:

Name: Rahul S Mayuranath
SRN: 01FB15ECS225

1. deleteNode: A wrapper function used to remove record k sub rooted at this node.
2. removeFromLeaf: Function to remove the record present in the idx th position in this node which is a leaf.
3. removeFromNonLeaf:Remove record from node when it is not a leaf.
4. getPred: Finds the predecessor of the record when the record is present in the idx th position.
5. getSucc: Finds the successor of the record where the record is present in the idx th position.
6. Fill: Fills the child node if deletion results in less than t-1 keys.
7. borrowFromPrev: Borrows a record from a previous(sibling) node and places it in the current node.
8. borrowFromNext: Borrows a record from the next(sibling) node and places it in the current node.
9. Merge: Merges the child and one of its sibling.
10. Findkey: Returns the index of the first key that is equal to or greater than the record.

The code is compiled with CFLAGS= -c Dt=3 indicating that a t value of 3 is passed to the program at compile time.
#define TOTAL 100000 is used in the test.c function indicating we want to read a million records.

Name: Rahul S Mayuranath
SRN: 01FB15ECS225

Instructions to run the code:

Execute the run.sh script(./run.sh). This runs the Makefile and produces the required output.
The value of t can be manipulated in the Makefile.
Sample output:

```
------ Node position is 33002 ------------
keys
999882 RSA 999913 USA
links
-1 -1 -1 -1 -1 -1

------ Node position is 45003 ------------
keys
999917 RSA 999919 EGY 999935 GBR 999949 EGY 999954 IND
links
-1 -1 -1 -1 -1 -1

------ Node position is 22020 ------------
keys
999971 EGY 999976 GBR 999986 PAK 999987 AFG
links
-1 -1 -1 -1 -1 -1
Record found in Node 6429
Record has been deleted or is not found
rahul@rahul-HP-Spectre-x360-Convertible-13:~/B-Tree$ |
```

Name: Rahul S Mayuranath
SRN: 01FB15ECS225

```
rahul@rahul-HP-Spectre-x360-Convertible-13:~/B-Tree$ ./run.sh
rm -rf *.o
gcc -c -Dt=3  lib/bTree.c
gcc -c -Dt=3  lib/node.c
gcc -c -Dt=3  test.c
gcc -c -Dt=3  lib/helper.c
gcc bTree.o node.o test.o helper.o

------ Node position is 13 ------------
keys
574878 IND
links
2 15 -1 -1 -1 -1

------ Node position is 2 ------------
keys
132486 IRQ 335197 IND
links
0 12 4 10 8 6

------ Node position is 0 ------------
keys
32497 RSA 44587 USA 65933 EGY
```