CS29003 ALGORITHMS LABORATORY
ASSIGNMENT 13
Date: 5 – November – 2019

B-Trees

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL and Red-Black Trees), it is assumed that everything is in main memory. To understand the use of B-Trees, we must think of the huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc) require $O(h)$ disk accesses where h is the height of the tree. B-tree is a fat tree. The height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, a B-Tree node size is kept equal to the disk block size. Since h is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red-Black Tree, ..etc.

A B-tree of order *m* is defined to have the following shape properties:

- ▷ The root is either a leaf or has at least two children.
- ▷ Each internal node, except for the root, has between $\lceil m/2 \rceil$ and m children.
- ▷ All leaves are at the same level in the tree, so the tree is always height balanced.
- ▷ All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.

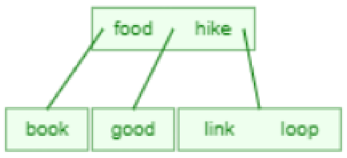


Figure 1: Sample B-Tree with m=3. Any node, therefore, can hold a maximum of 2 keys and pointer to 3 children.

B-Tree Search

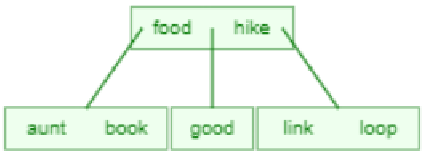
Search is similar to the search in Binary Search Tree. Let the key to be searched be k. We start from the root and recursively traverse down. For every visited non-leaf node, if the node has the key, we simply return the node. Otherwise, we recur down to the appropriate child (The child which is just before the first greater key) of the node. If we reach a leaf node and don't find k in the leaf node, we return NULL. For example, a search for ”book” in the above B-Tree will return ”True” (rather the actual address of the node containing the key), whereas a search for ”boot” will return ”False”.

B-Tree Insertion

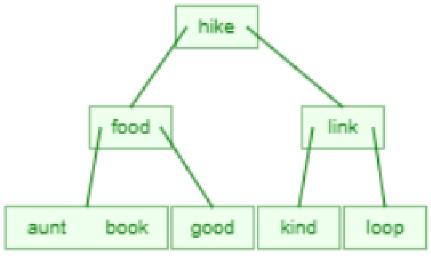
A new key is always inserted at the leaf node. Let the key to be inserted be k. We follow the below steps in order to insert k into the B-Tree:

1. Like Binary Search Trees (BSTs), we start from the root and traverse down till we reach a leaf node.
2. Once we reach a leaf node, unlike BSTs, we first check if the node already contains m keys.
3. If not, we insert the key into that leaf node into its appropriate position.
4. Else, we split the node into two halves, say node_a and node_b. We move the middle element of the node to its parent, update the pointers in the parent node to suitably point to the two new nodes, i.e. node_a and node_b.
5. We repeat Step 4 till the movement of an element from the child to its parent no longer overflows the parent node and results in further splitting.
6. We finally insert the key k in the appropriate leaf node, either node_a or node_b.

In the B-Tree of Figure 1, let us try to insert the key "aunt". Now, alphabetically, "aunt" < "food". Hence, we reach the node pointed to by the first pointer of parent node. We find that the leaf node reached contains a single key "book" and therefore has space for accommodating one more key. Since "aunt" < "book", we insert "aunt" before "book" and obtain the following configuration:



Now, let us try to insert the key "kind". Alphabetically, "kind" > "hike". Hence, we reach the node pointed to by the last pointer of the parent node. Now, this leaf node already contains 2 keys i.e. "link" and "loop" and 3 pointers. Hence, we split the node into two halves and move the middle element "link" to the parent. This, in turn causes the parent node to split as the parent was already full. Hence, "hike" is now moved to a new parent node. First pointer of the newly created parent node points to the node containing "food" and the second pointer points to the node containing "link". We finally insert "kind" to the left of "link" by reaching the leaf node pointed to by the first pointer of the node containing "link". We obtain the following configuration:



Sample Input - ”input1.txt”

```
3
boat
aunt
dine
come
cook
bold
dome
done
fine
find
hood
goon
lift
hack
five
goof
```

First line of the input file ”input1.txt” contains the order to the B-Tree node, i.e. the no. of child pointers a node ac accommodate. The remaining lines contain the keys that need to be inserted in the B-Tree. The B-Tree after all the insertions should look like:

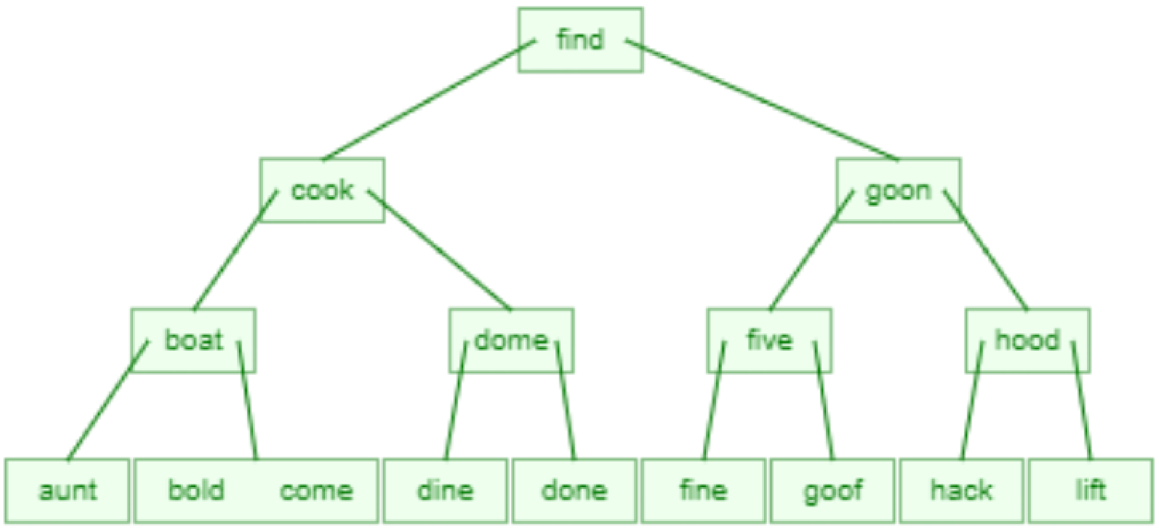


Figure 2: Final B-Tree

First line of the output file ”output1.txt” should print the no. of levels of the obtained B-Tree. Second line should print the keys in the *root node*. Thereafter, no. of nodes at each level and their contents should be printed in subsequent lines as shown.

Sample Output - ”output1.txt”

```
Total Levels: 4
Root node keys: find,
Level 1 nodes: 2
Node 1 keys: cook,
Node 2 keys: goon,
Level 2 nodes: 4
Node 1 keys: boat,
Node 2 keys: dome,
Node 3 keys: five,
Node 4 keys: hood,
Level 3 nodes: 8
Node 1 keys: aunt,
Node 2 keys: bold, come
Node 3 keys: dine,
Node 4 keys: done,
Node 5 keys: fine,
Node 6 keys: goof,
Node 7 keys: hack,
Node 8 keys: lift,
```

Now, ”input2.txt” will contain queries, one query per line, that need to be executed on the obtained B-Tree. A query might contain a single key to be searched in the tree or a range query in which case the line would contain two keys $k1$ and $k2$ separated by comma.

Sample Input - ”input2.txt”

```
bold
dare
deaf, gain
```

For a single key query, you need to print ”true” if the key is found in the B-Tree, else ”false. For a range query, all the keys (strings) occurring in the B-Tree which are $\geqslant k1$ and $\leqslant k2$ should be printed in alphabetically sorted order as shown.

Sample Output - ”output2.txt”

```
true
false
dine, dome, done, find, fine, five
```

File Naming Convention

Please note that the output file names used in this document till now are generic and for explanation purpose only. **Your submissions will not be evaluated unless** you follow the below specified file naming convention for naming your files:

1. Program/Code file Naming Convention :
   <ROLLNO(IN CAPS)>_A<Assign_No>.c/cpp
   Eg: 18CS30004_A13.c / 18CS30004_A13.cpp

2. Output file Naming Convention :
   <ROLLNO(IN CAPS)>_A<Assign_No>_output.txt
   Eg: Part 1: 18CS30004_A13_output1.txt
   Eg: Part 2: 18CS30004_A13_output2.txt