7. B Tree operations

**Aim:**

To perform insertion and display operations on B trees given order of tree is 5.

**Theory:**

* *M – way search trees*

A Multiway Search Tree of order m is a tree in which any node can have a maximum of m – 1 values & a maximum of m children.

To make the processing of m-way trees easier some type of order will be imposed on the keys within each node, resulting in a **multiway search tree of order m** (or an **m-way search tree**). By definition an m-way search tree is an m-way tree in which:

1. Each node has m children and m-1 key fields
2. The keys in each node are in ascending order.
3. The keys in the first i children are smaller than the ith key
4. The keys in the last m-i children are larger than the ith key

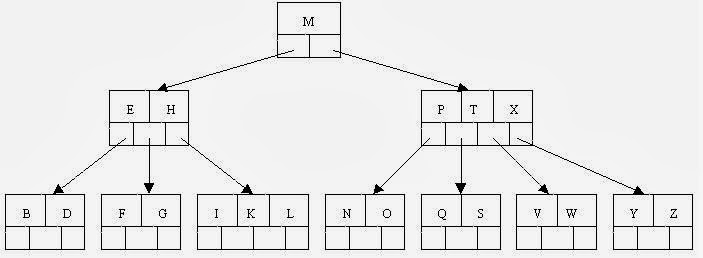


Figure 1 Example of 4 way search tree

M-way search trees give the same advantages to m-way trees that binary search trees gave to binary trees - they provide fast information retrieval and update. However, they also have the same problems that binary search trees had - they can become unbalanced, which means that the construction of the tree becomes of vital importance

B Trees are a special case of Multiway Search Trees.

* *B Trees*

A B Tree of order n is a Multiway Search Tree of order n with the following characteristics:

* All the non - leaf nodes have a max of n child nodes & a min of n/2 child nodes.
* If a root is non leaf node, then it has a max of n non empty child nodes & a min of 2 child nodes.
* If a root node is a leaf node, then it does not have any child node.
* A node with n child nodes has n - 1 values arranged in ascending order.
* All values appearing on the left most child of any node are smaller than the left most value of that node, while all values appearing on the right most child of any node are greater than the right most value of that node.
* If x & y are two adjacent values in a node such that x < y, i.e. they are the *i*th & *(i+1)*th values in the node respectively, then all values in the *(i+1)* th child of that node are > x but < y.
* *Insertion in B tree*
* When inserting an item, first do a search for it in the B-tree. If the item is not already in the B-tree, this unsuccessful search will end at a leaf.
* If there is room in this leaf, just insert the new item here. Note that this may require that some existing keys be moved one to the right to make room for the new item. If instead this leaf node is full so that there is no room to add the new item, then the node must be "split" with about half of the keys going into a new node to the right of this one. The median (middle) key is moved up into the parent node. (Of course, if that node has no room, then it may have to be split as well.)
* Note that when adding to an internal node, not only might we have to move some keys one position to the right, but the associated pointers have to be moved right as well. If the root node is ever split, the median key moves up into a new root node, thus causing the tree to increase in height by one.
* Let’s take an example. Insert the following letters into what is originally an empty B-tree of order 5: C N G A H E K Q M F W L T Z D P R X Y S Order 5 means that a node can have a maximum of 5 children and 4 keys. All nodes other than the root must have a minimum of 2 keys. The first 4 letters get inserted into the same node, resulting in this picture:

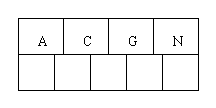


Figure 2 Initial tree after inserting 4 letters

* When we try to insert the H, we find no room in this node, so we split it into 2 nodes, moving the median item G up into a new root node. Note that in practice we just leave the A and C in the current node and place the H and N into a new node to the right of the old one.

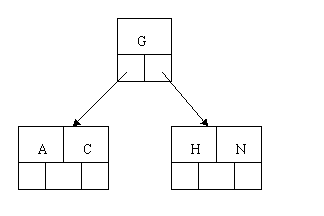


Figure 3 Splitting leaf node after inserting H

* Inserting E, K, and Q proceeds without requiring any splits:

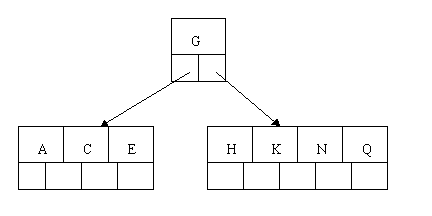


Figure 4 Inserted E,K and Q

* Inserting M requires a split. Note that M happens to be the median key and so is moved up into the parent node.

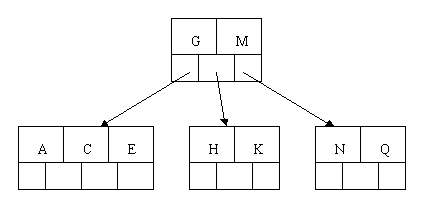


Figure 5 Splitting leaf node after inserting M

* The letters F, W, L, and T are then added without needing any split.

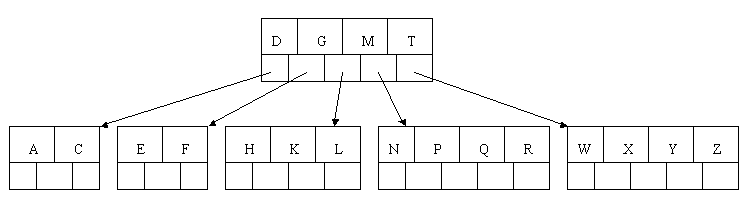
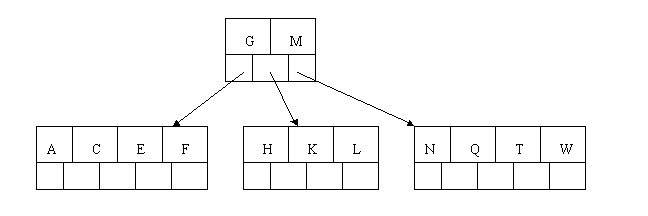


Figure 6 Inserted F, W, L and T

* When Z is added, the rightmost leaf must be split. The median item T is moved up into the parent node. Note that by moving up the median key, the tree is kept fairly balanced, with 2 keys in each of the resulting nodes.

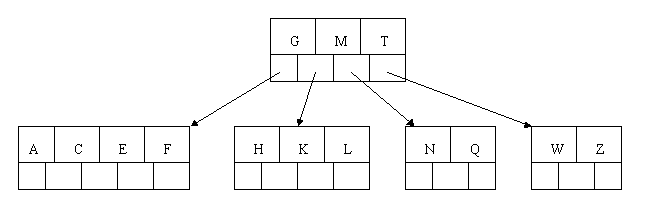


Figure 7 Split rightmost leaf after inserting Z

* The insertion of D causes the leftmost leaf to be split. D happens to be the median key and so is the one moved up into the parent node. The letters P, R, X, and Y are then added without any need of splitting:

Figure 8 Inserted D, P, R, X and Y

* Finally when S is added to the tree, the node with N, P, Q, and R splits, sending the median Q up to the parent. However, the parent node is full, so it splits, sending the median M up to form a new root node. Note how the 3 pointers from the old parent node stay in the revised node that contains D and G.

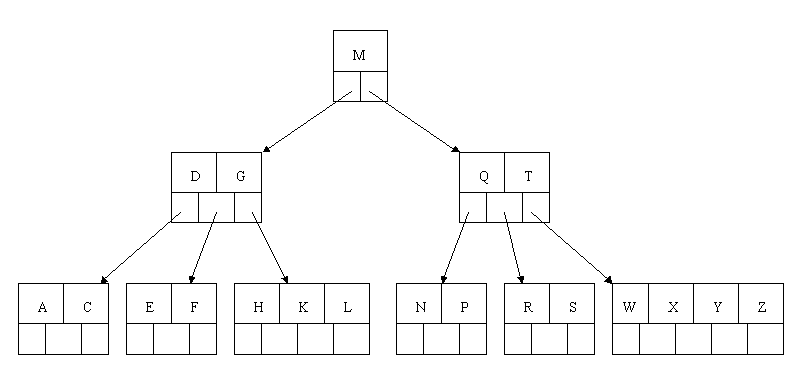


Figure 9 Final B tree

* *Traversal of Btree*

Traversal is also similar to in order traversal of Binary Tree. We start from the leftmost child, recursively print the leftmost child, then repeat the same process for remaining children and keys. In the end, recursively print the rightmost child. The in order traversal of a btree results in the data being traversed in ascending order.

**Program:**

#include <stdio.h>

#include <stdlib.h>

#define MAX 4

#define MIN 2

struct btreeNode {

int val[MAX + 1], count;

struct btreeNode \*link[MAX + 1];

};

struct btreeNode \*root;

/\* creating new node \*/

struct btreeNode \* createNode(int val, struct btreeNode \*child) {

struct btreeNode \*newNode;

newNode = (struct btreeNode \*)malloc(sizeof(struct btreeNode));

newNode->val[1] = val;

newNode->count = 1;

newNode->link[0] = root;

newNode->link[1] = child;

return newNode;

}

/\* Places the value in appropriate position \*/

void addValToNode(int val, int pos, struct btreeNode \*node,

struct btreeNode \*child) {

int j = node->count;

while (j > pos) {

node->val[j + 1] = node->val[j];

node->link[j + 1] = node->link[j];

j--;

}

node->val[j + 1] = val;

node->link[j + 1] = child;

node->count++;

}

/\* split the node \*/

void splitNode (int val, int \*pval, int pos, struct btreeNode \*node,

struct btreeNode \*child, struct btreeNode \*\*newNode) {

int median, j;

if (pos > MIN)

median = MIN + 1;

else

median = MIN;

\*newNode = (struct btreeNode \*)malloc(sizeof(struct btreeNode));

j = median + 1;

while (j <= MAX) {

(\*newNode)->val[j - median] = node->val[j];

(\*newNode)->link[j - median] = node->link[j];

j++;

}

node->count = median;

printf("\nMedian value is: %d",node->val[median]);

(\*newNode)->count = MAX - median;

if (pos <= MIN) {

addValToNode(val, pos, node, child);

} else {

addValToNode(val, pos - median, \*newNode, child);

}

\*pval = node->val[node->count];

(\*newNode)->link[0] = node->link[node->count];

node->count--;

}

/\* sets the value val in the node \*/

int setValueInNode(int val, int \*pval,

struct btreeNode \*node, struct btreeNode \*\*child) {

int pos;

if (!node) {

\*pval = val;

\*child = NULL;

return 1;

}

if (val < node->val[1]) {

pos = 0;

} else {

for (pos = node->count;

(val < node->val[pos] && pos > 1); pos--);

if (val == node->val[pos]) {

printf("Duplicates not allowed\n");

return 0;

}

}

if (setValueInNode(val, pval, node->link[pos], child)) {

if (node->count < MAX) {

addValToNode(\*pval, pos, node, \*child);

} else {

printf("\nSplitting node because of overflow...");

splitNode(\*pval, pval, pos, node, \*child, child);

return 1;

}

}

return 0;

}

/\* insert val in B-Tree \*/

void insertion(int val) {

int flag, i;

struct btreeNode \*child;

printf("\nInserting %d in btree...",val);

flag = setValueInNode(val, &i, root, &child);

if (flag)

root = createNode(i, child);

}

/\* search val in B-Tree \*/

void searching(int val, int \*pos, struct btreeNode \*myNode) {

if (!myNode) {

return;

}

if (val < myNode->val[1]) {

\*pos = 0;

} else {

for (\*pos = myNode->count;

(val < myNode->val[\*pos] && \*pos > 1); (\*pos)--);

if (val == myNode->val[\*pos]) {

printf("Given data %d is present in B-Tree", val);

return;

}

}

searching(val, pos, myNode->link[\*pos]);

return;

}

/\* B-Tree Traversal \*/

void traversal(struct btreeNode \*myNode) {

int i;

if (myNode) {

for (i = 0; i < myNode->count; i++) {

traversal(myNode->link[i]);

printf("%d ", myNode->val[i + 1]);

}

traversal(myNode->link[i]);

}

}

int main() {

int val, ch;

while (1) {

printf("1. Insertion\n");

printf("2. Traversal\n");

printf("3. Exit\nEnter your choice:");

scanf("%d", &ch);

switch (ch) {

case 1:

printf("Enter your input:");

scanf("%d", &val);

insertion(val);

break;

case 2:

traversal(root);

break;

case 3:

exit(0);

default:

printf("You have entered wrong option!!\n");

break;

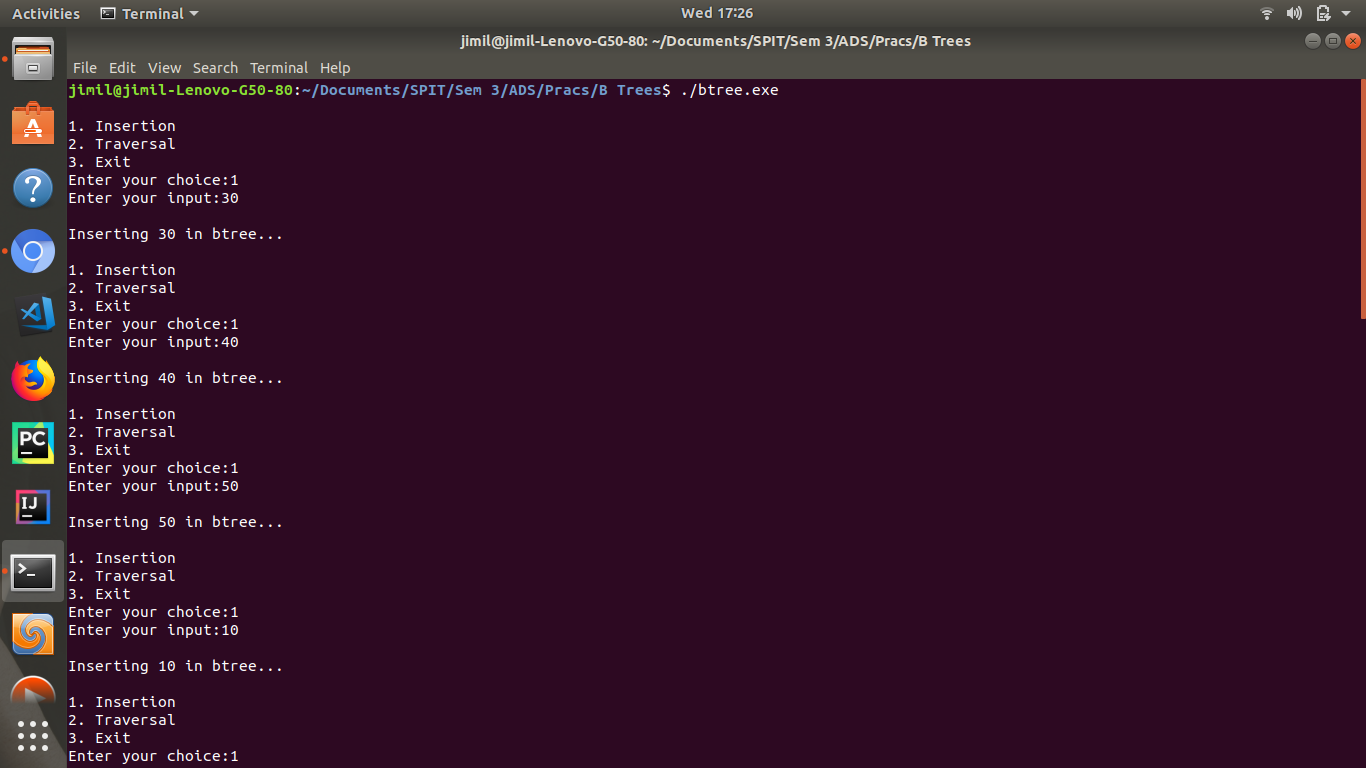
}

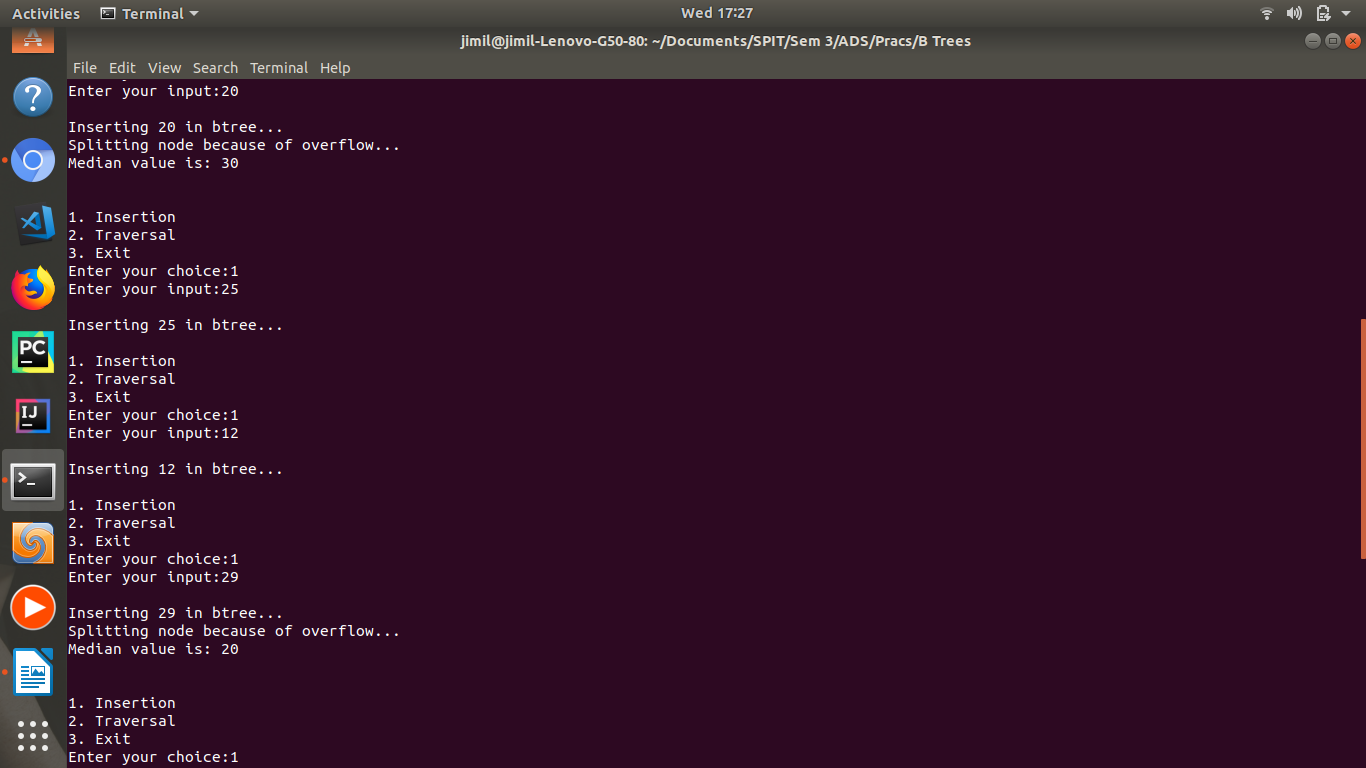
printf("\n");

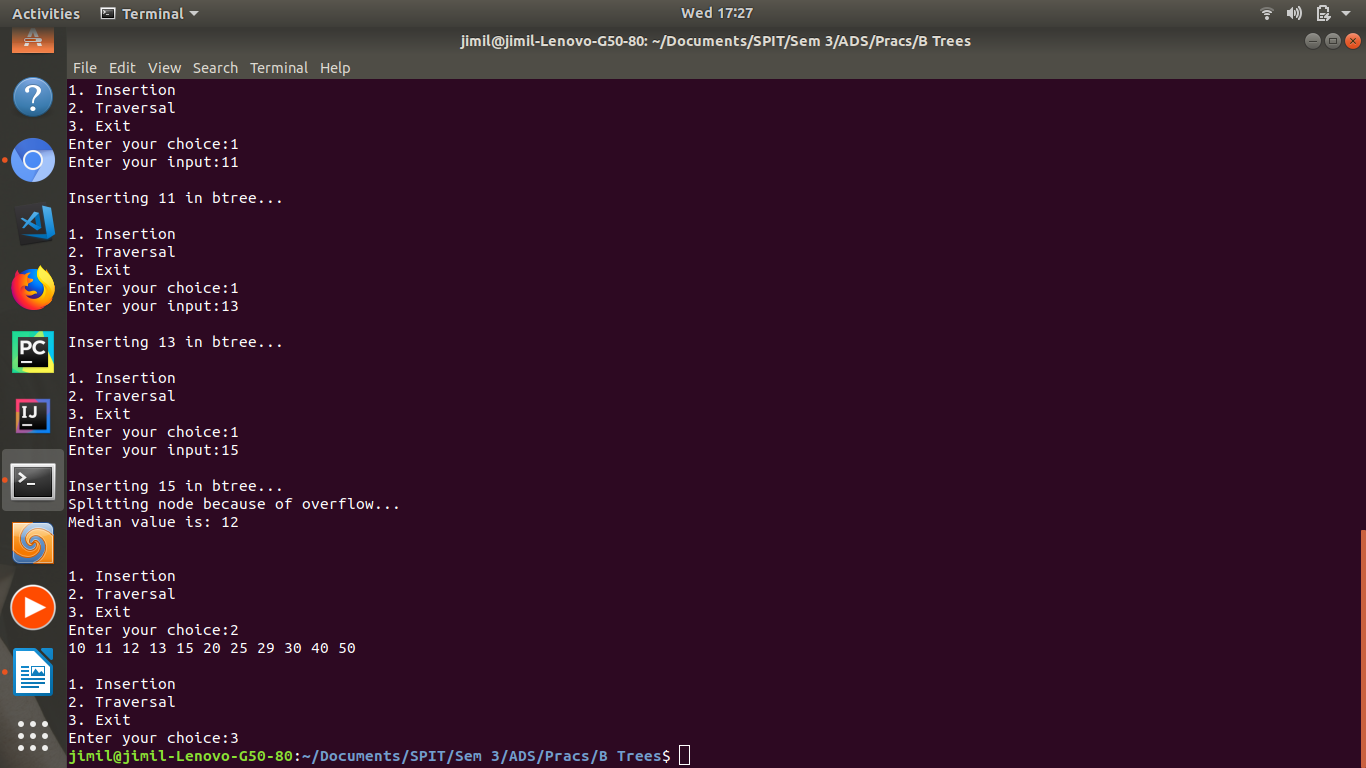
}

}

**Output:**







**Result:**

The B tree makes the index fast and is therefore used for disk access.

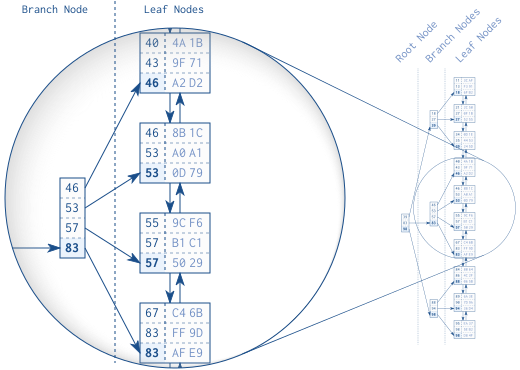


Figure 10 shows an example index with 30 entries. The doubly linked list establishes the logical order between the leaf nodes. The root and branch nodes support quick searching among the leaf nodes.

The tree traversal starts at the root node on the left-hand side. Each entry is processed in ascending order until a value is greater than or equal to (>=) the search term (57). In the figure it is the entry 83. The database follows the reference to the corresponding branch node and repeats the procedure until the tree traversal reaches a leaf node. **The B-tree enables the database to find a leaf node quickly.**

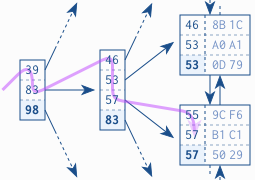


Figure 11 shows an index fragment to illustrate a search for the key “57”

* The tree traversal is a very efficient operation primarily because of the tree balance, which allows accessing all elements with the same number of steps, and secondly because of the logarithmic growth of the tree depth. That means that the tree depth grows very slowly compared to the number of leaf nodes.
* The logarithmic growth enables the example index to search a million records with ten tree levels, but a real world index is even more efficient. The main factor that affects the tree depth, and therefore the lookup performance, is the number of entries in each tree node. This number corresponds to—mathematically speaking—the basis of the loga­rithm. The higher the basis, the shallower the tree and faster the traversal.
* Databases exploit this concept to a maximum extent and put as many entries as possible into each node—often hundreds. That means that every new index level supports a hundred times more entries.

**Conclusion:**

Thus through this experiment we have understood and implemented a special case of m – way search trees called B trees. The worst case complexity of performing insert, delete and search operations on B trees is O (log n). We also understood how B trees make indexing faster thereby improving performance of disk access because of tree balance and logarithmic growth of tree depth.