SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

DEPARTMENT OF INFORMATION TECHNOLOGY

COURSE CODE: DJ19ITL502 **DATE:4/12/2022**

COURSE NAME: Advanced Data Structure Laboratory **CLASS: T.Y. BTech-A3**

EXPERIMENT NO. 06

AIM / OBJECTIVE: Implement B Tree

Theory:

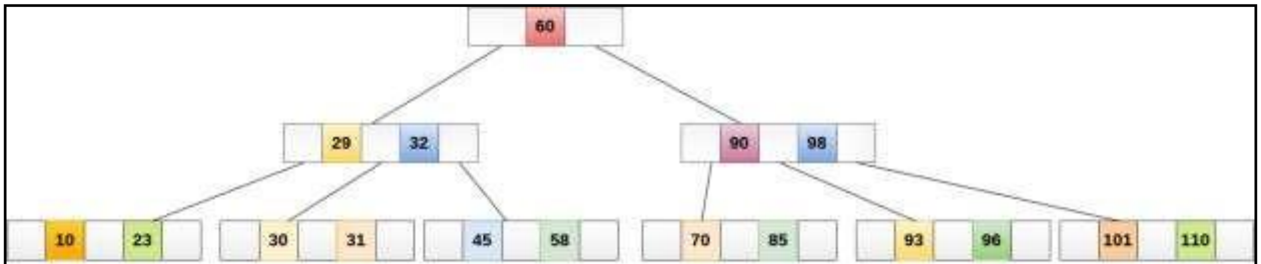
B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reasons of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

1. Every node in a B-Tree contains at most m children.
2. Every node in a B-Tree except the root node and the leaf node contain at least m/2 children.
3. The root nodes must have at least 2 nodes.
4. All leaf nodes must be at the same level.

It is not necessary that, all the nodes contain the same number of children but, each node must have m/2 number of nodes.

A B tree of order 4 is shown in the following image.



Operations on a B-tree

(1). Searching an element in a B-tree

Searching for an element in a B-tree is the generalized form of searching an element in a Binary Search Tree. The following steps are followed.

1. Starting from the root node, compare k with the first key of the node.
   If k = the first key of the node, return the node and the index.
2. If k.leaf = true, return NULL (i.e. not found).
3. If k < the first key of the root node, search the left child of this key recursively.
4. If there is more than one key in the current node and k > the first key, compare k with the next key in the node.
   If k < next key, search the left child of this key (ie. k lies in between the first and the second keys).
   Else, search the right child of the key.
5. Repeat steps 1 to 4 until the leaf is reached.

(2). Insertion Operation

1. If the tree is empty, allocate a root node and insert the key.
2. Update the allowed number of keys in the node.
3. Search the appropriate node for insertion.
4. If the node is full, follow the steps below.
5. Insert the elements in increasing order.
6. Now, there are elements greater than its limit. So, split at the median.
7. Push the median key upwards and make the left keys as a left child and the right keys as a right child.
8. If the node is not full, follow the steps below.
9. Insert the node in increasing order.

ALGORITHM:

Algorithm For Searching an Element:

```
BtreeSearch(x, k)
 i = 1
 while i ≤ n[x] and k ≥ keyi[x]        // n[x] means number of keys in x node
   do i = i + 1
if i n[x] and k = keyi[x]
   then return (x, i)
if leaf [x]
   then return NIL
else
```

```
    return BtreeSearch(ci[x], k)
```

Algorithm For Inserting an Element:

```
BreeInsertion(T, k)
r  root[T]
if n[r] = 2t - 1
    s = AllocateNode()
    root[T] = s
    leaf[s] = FALSE
    n[s] <- 0
    c1[s] <- r
    BtreeSplitChild(s, 1, r)
    BtreeInsertNonFull(s, k)
else BtreeInsertNonFull(r, k)
BtreeInsertNonFull(x, k)
i = n[x]
if leaf[x]
    while i ≥ 1 and k < keyi[x]
        keyi+1 [x] = keyi[x]
        i = i - 1
    keyi+1[x] = k
    n[x] = n[x] + 1
else while i ≥ 1 and k < keyi[x]
        i = i - 1
    i = i + 1
    if n[ci[x]] == 2t - 1
        BtreeSplitChild(x, i, ci[x])
        if k &rt; keyi[x]
            i = i + 1
    BtreeInsertNonFull(ci[x], k)
BtreeSplitChild(x, i)
BtreeSplitChild(x, i, y)
z = AllocateNode()
leaf[z] = leaf[y]
n[z] = t - 1
for j = 1 to t - 1
    keyj[z] = keyj+t[y]
if not leaf [y]
```

```
   for j = 1 to t
      cj[z] = cj + t[y]
n[y] = t - 1
for j = n[x] + 1 to i + 1
   cj+1[x] = cj[x]
ci+1[x] = z
for j = n[x] to i
   keyj+1[x] = keyj[x]
keyi[x] = keyt[y]
n[x] = n[x] + 1
```

SOURCE CODE:

```c
// Searching a key on a B-tree in C

#include <stdio.h>
#include <stdlib.h>

#define MAX 3
#define MIN 2

struct BTreeNode {
  int val[MAX + 1], count;
  struct BTreeNode *link[MAX + 1];
};

struct BTreeNode *root;

// Create a 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;
}

// Insert node
```

```c
void insertNode(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 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;
  (*newNode)->count = MAX - median;

  if (pos <= MIN) {
    insertNode(val, pos, node, child);
  } else {
    insertNode(val, pos - median, *newNode, child);
  }
  *pval = node->val[node->count];
  (*newNode)->link[0] = node->link[node->count];
  node->count--;
```

```
}

// Set the value
int setValue(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 are not permitted\n");
      return 0;
    }
  }
  if (setValue(val, pval, node->link[pos], child)) {
    if (node->count < MAX) {
      insertNode(*pval, pos, node, *child);
    } else {
      splitNode(*pval, pval, pos, node, *child, child);
      return 1;
    }
  }
  return 0;
}

// Insert the value
void insert(int val) {
int flag, i;
  struct BTreeNode *child;

  flag = setValue(val, &i, root, &child);
  if (flag)
```

```
    root = createNode(i, child);
}

// Search node
void search(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("%d is found", val);
      return;
    }
  }
  search(val, pos, myNode->link[*pos]);

  return;
}

// Traverse then nodes
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;

  insert(8);
```

```
 insert(9);
 insert(10);
 insert(11);
 insert(15);
 insert(16);
 insert(17);
 insert(18);
 insert(20);
 insert(23);

 traversal(root);

 printf("\n");
 search(11, &ch, root);
}
```

OUTPUT:

```
8 9 10 11 15 16 17 18 20 23
11 is found

...Program finished with exit code 0
Press ENTER to exit console.
```

ANALYSIS:

Worst case Time complexity: Θ(log n)

Average case Time complexity: Θ(log n)

Best case Time complexity: Θ(log n)

CONCLUSION:

Hence, we successfully implemented B Tree.

BOOKS AND WEB RESOURCES:

- https://sd.blackball.lv/library/Introduction_to_Algorithms_Third_Edition_(2009).pdf
- https://www.programiz.com/dsa/b-tree