Practical 3)Implementation of B+ Tree Indexing for Database query processing

Input-output for Select Query on exact Match, Range Query, Insert , delete Query.

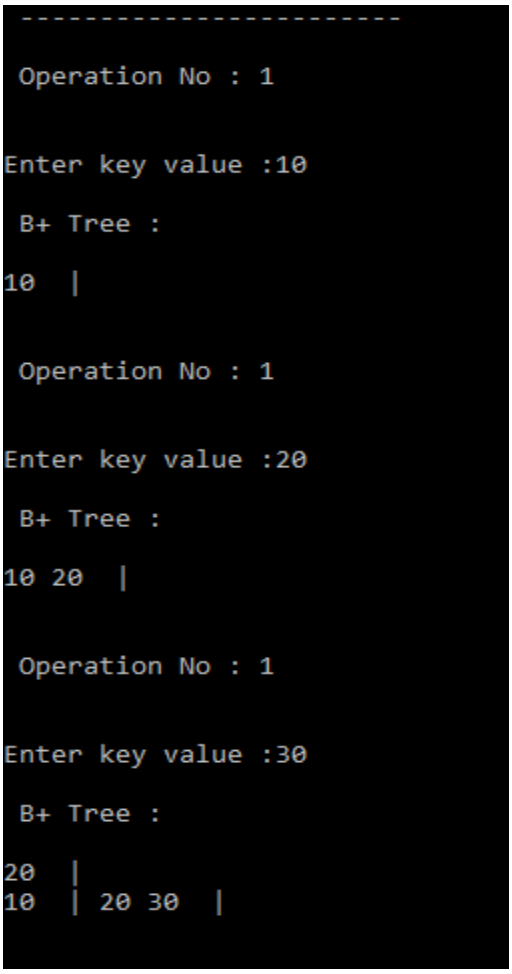
Analysis report

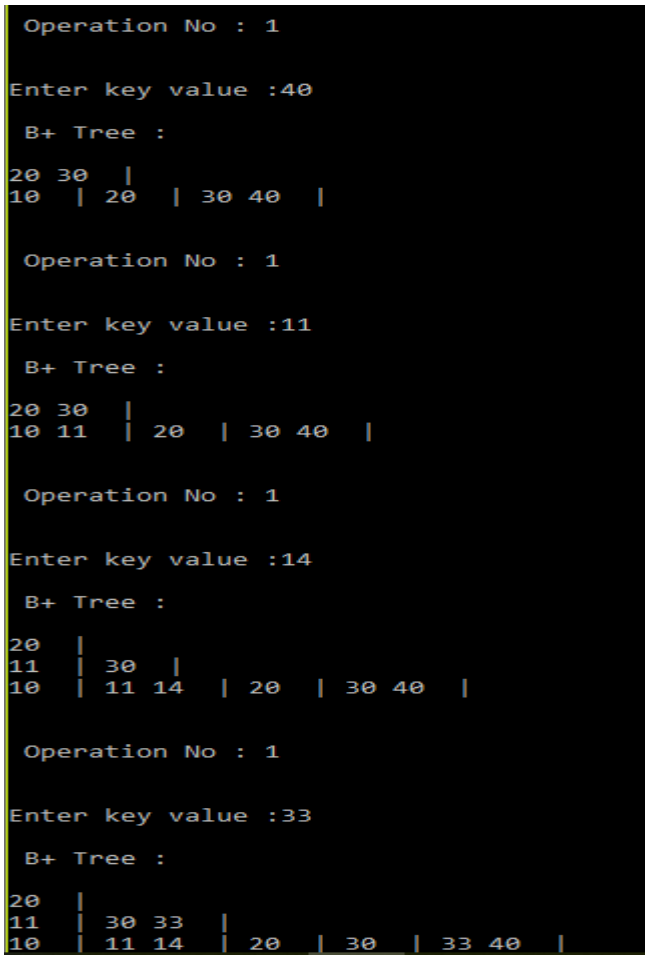
Function Calling :-

```
C:\Users\Vishal\Desktop\bplus_A\b_plus_tree>gcc -o a.exe "B+tree.c"

C:\Users\Vishal\Desktop\bplus_A\b_plus_tree>a.exe
 ***************************
 Functions Calling....

   1   .   Insert

   2   .   Delete

   3   .   Exact MATCH

   4   .   Batch SEARCH

   5   .   RANGE SEARCH


 ........................

 Functions No :
```

Functions Calling:- Insert ,Delete ,Exact Match , Batch Search , Range Search.

Insert Operation





Insert Operations :-

It insert one by one and when its degree+1 key comes it overflow and split it into two parts. By taking the median by degree/2; And adjust the the whole tree and split key also came into the leaf nodes .Left side contain less value then root as follow as right side contain large value.

Exact search : -

```
 Operation No : 3

 Exact Value Search : 14
Found key 14

 Operation No : 3

 Exact Value Search : 33
Found key 33

 Operation No : 3

 Exact Value Search : 32
Not found key 32

 Operation No : 3

 Exact Value Search : 45
Not found key 45

 Operation No :
```

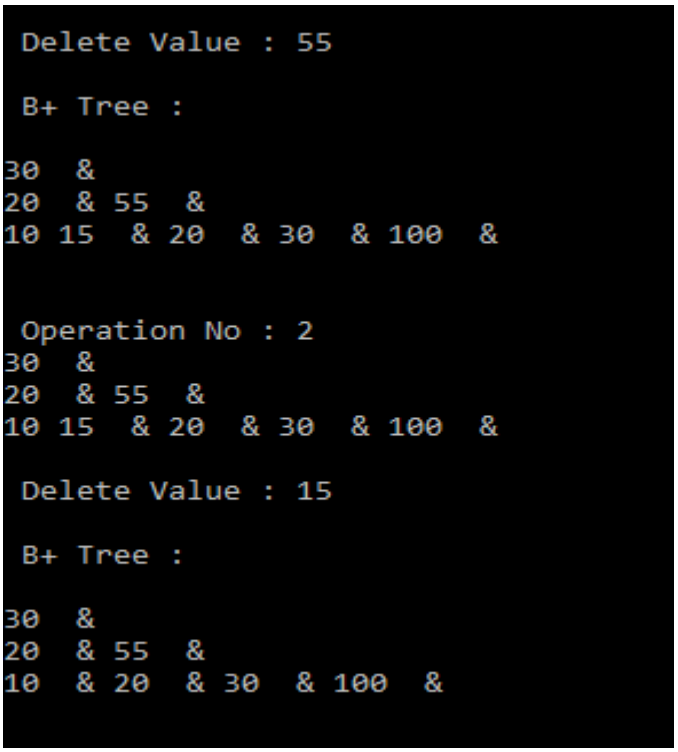
In Exact search it search for leaf node from root to node path. And from the leaf node traverse the neighbor node and compare value.

Range Search:-

```
 Range Search : Enter the Range of value u want to find :Enter2
16
Level Traversed 1
Level Traversed 2
Leaf NODE Traversed Neighbor
10 11 14

 Operation No : 5

 Range Search : Enter the Range of value u want to find :Enter0
100
Level Traversed 1
Level Traversed 2
Leaf NODE Traversed Neighbor
10 11 14 20 30 33 40

 Operation No :
```

In Range query it traverse the left leaf node and from left side it start traverse and go to right side and find value between the min and max ranged.

Delete Operations:-



Analysis :-

It store the data at leaf node ,It posses same level and so efficiently searching time will be reduced.

Insertion Take O(n)

Range query O(logn)

Exact match O(logn)

Very much quick response for exact and range query

Code:-

Main.c

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include "bplus_function.h"

// ------------------- Init() of program ---------------------- //
int main()
{
int c,v,find_key,exact_match,batch_search_value[100],n,i=0,max,min, order = 3;
node *root;
root=NULL;

// printf("\n Order of B+ Tree : ");
// scanf("%d",&order);
printf("\n **************************");
printf("\n Functions Calling....   \n");
printf("\n   1  .  Insert         \n");
printf("\n   2  .  Delete         \n");
printf("\n   3  .  Exact MATCH     \n");
printf("\n   4  .  Batch SEARCH    \n");
printf("\n   5  .  RANGE SEARCH    \n");
printf("\n\n .....................");

do
{
printf("\n\n Functions No : ");
scanf("%d",&c);

switch(c){
case 1:printf("\n\nEnter key value :");
scanf("%d",&v);
root=insert(root,v,v);              // ------------ Insert into node -----------------//
printf("\n B+ Tree : \n\n");
printtree(root);
break;
```

```
case 2:
printtree(root);
printf("\n Delete Value : ");
scanf("%d", &v);
root = deletee(root, v);              // ----------Incomplete Delete node in some test cases.....  //
printf("\n B+ Tree : \n\n");
printtree(root);
break;
case 3:
printf("\n Exact Value Search : ");
scanf("%d", &v);
exact_match=exact_search(root,v);          // ---------- exact search ----------//
if (exact_match)
{
printf("Found key %d",v);
}else{
printf("Not found key %d",v);
}
break;
case 4:
printf("\n Batch Search : ");               //  ---------- batch search ----------//
printf("Enter the number of value u want to find :");
scanf("%d",&n);
for(i=0;i<n;i++){
scanf("%d",&batch_search_value[i]);
}
for (int i = 0; i < n; ++i)
{
exact_match=exact_search(root,batch_search_value[i]);
if (exact_match)
{
printf("Found key %d  \n",batch_search_value[i]);
}else{
printf("Not found key %d \n",batch_search_value[i]);
```

```
}
}
break;
case 5:
printf("\n Range Search : ");
printf("Enter the Range of value u want to find :");

// for (int i = min; i < max; ++i)
// {
//              exact_match=0;
exact_match=range_search(root);             //  ----------------- Range search ----------//
//                                          if (exact_match)
// {
//              printf("Found key %d  \n",i);
//                                          }
// }
break;
case 6:
return 0;
default:printf("\n..... Try once more .....");
}
}while(c!=4);

return 0;
}
```

Bplusfunction.h

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

#define defaultorder 4
#define bool char
#define false 0
#define true 1
```

```
//----------------Structure for Data-Key Record----------------//
typedef struct record {
int value;
} record;

//----------------Structure for Node ----------------------//
//----------  Contains leaf is there or not ---- ,no_of_values --> Counter ------- ** pointers --> for parent head....-------//
//-------------  Keys ---> Data Value... ----------//
//----------    Next --> is for only leaf node child of neighbor  ---------------------//

typedef struct node {
char leaf_or_not;
int number_of_keys;
void ** ptrs;
int * keys;
struct node * parent;
struct node * next;
} node;

//
int order ;
node * delete_entry( node *, node *, int, void * );           //-----------  Delete Entry -------------//
node* insert_into_parent(node *, node *, int, node *);        //-----------  Insert into parent from traversal root to level node to leaf -----------//
int exact_search(node * root, int key);           //-----------  For Exact Search from leaf traversed --------------------//
int range_search(node *root);      // ---------- For Range Search from leaf travered _min_ , _max_ ---------- //
// int batch_search(node * root,int n, int key);

//  ---------- New structure ----------------//
record* newrecord(int value) {
record * new_record = (record *)malloc(sizeof(record));
if (new_record == NULL) {
perror("Record creation.");
exit(EXIT_FAILURE);
}
```

```
else {
new_record->value = value;
}
return new_record;
}

//-------------------------------- _FINDING_LEAF_Node_VALUE-------------------------//
node* find_leaf( node * root, int key) {
int i = 0;
node *c = root;
if (c == NULL) {
return c;
}
while (!c->leaf_or_not)
{
i = 0;
while (i < c->number_of_keys) {
if (key >= c->keys[i]) i++;
else break;
}
c = (node *)c->ptrs[i];
}
return c;
}

record *find(node * root, int key) {
int i = 0;
node *c = find_leaf( root, key);
if (c == NULL) return NULL;
for (i = 0; i<c->number_of_keys; i++)
if (c->keys[i] == key) break;
if (i == c->number_of_keys)
return NULL;
else
return (record *)c->ptrs[i];
```

```
}



node * newnode( void ) {

node * new_node;

new_node = (node*)malloc(sizeof(node));


new_node->keys = (int*)malloc( (order - 1) * sizeof(int) );


new_node->ptrs = (void**)malloc( order * sizeof(void *) );


new_node->leaf_or_not = 0;

new_node->number_of_keys = 0;

new_node->parent = NULL;

new_node->next = NULL;

return new_node;

}


node * insert_at_leaf( node * leaf, int key, record * pointer ) {


int i, insertpoint;


insertpoint = 0;

while (insertpoint < leaf->number_of_keys && leaf->keys[insertpoint] < key)

insertpoint++;


for (i = leaf->number_of_keys; i > insertpoint; i--) {

leaf->keys[i] = leaf->keys[i - 1];

leaf->ptrs[i] = leaf->ptrs[i - 1];

}

leaf->keys[insertpoint] = key;

leaf->ptrs[insertpoint] = pointer;

leaf->number_of_keys++;

return leaf;

}
```

```
node * insert_into_node(node * root, node * n,int left_index, int key, node * right) {
int i;

for (i = n->number_of_keys; i > left_index; i--) {
n->ptrs[i + 1] = n->ptrs[i];
n->keys[i] = n->keys[i - 1];
}
n->ptrs[left_index + 1] = right;
n->keys[left_index] = key;
n->number_of_keys++;
return root;
}

node * insert_into_node_after_splitting(node * root, node * old_node, int left_index,
int key, node * right) {

int i, j, s, k_prime;
node * new_node, * child;
int * temp_keys;
node ** temp_ptrs;

temp_ptrs =(node**) malloc( (order + 1) * sizeof(node *) );

temp_keys = (int*)malloc( order * sizeof(int) );

for (i = 0, j = 0; i < old_node->number_of_keys + 1; i++, j++) {
if (j == left_index + 1) j++;
temp_ptrs[j] = (node*)old_node->ptrs[i];
}

for (i = 0, j = 0; i < old_node->number_of_keys; i++, j++) {
if (j == left_index) j++;
temp_keys[j] = old_node->keys[i];
}
```

```
temp_ptrs[left_index + 1] = right;
temp_keys[left_index] = key;

if(order%2==0)
s=order/2;
else
s=order/2+1;

new_node = newnode();

old_node->number_of_keys = 0;
for (i = 0; i < s - 1; i++) {
old_node->ptrs[i] = temp_ptrs[i];
old_node->keys[i] = temp_keys[i];
old_node->number_of_keys++;
}
old_node->ptrs[i] = temp_ptrs[i];
k_prime = temp_keys[s - 1];
for (++i, j = 0; i < order; i++, j++) {
new_node->ptrs[j] = temp_ptrs[i];
new_node->keys[j] = temp_keys[i];
new_node->number_of_keys++;
}
new_node->ptrs[j] = temp_ptrs[i];

new_node->parent = old_node->parent;
for (i = 0; i <= new_node->number_of_keys; i++) {
child =(node*) new_node->ptrs[i];
child->parent = new_node;
}


return insert_into_parent(root, old_node, k_prime, new_node);
}
```

```
node* insert_into_parent(node * root, node * left, int key, node * right)
{

int left_index;
node * parent;

parent = left->parent;

if (parent == NULL)
{
node * r = newnode();
r->keys[0] = key;
r->ptrs[0] = left;
r->ptrs[1] = right;
r->number_of_keys++;
r->parent = NULL;
left->parent = r;
right->parent = r;
return r;
}

left_index=0;

while (left_index <= parent->number_of_keys &&parent->ptrs[left_index] != left)
{left_index++;}

if (parent->number_of_keys < (order - 1))
return insert_into_node(root, parent, left_index, key, right);

return insert_into_node_after_splitting(root, parent, left_index, key, right);
}

node * split(node * root, node * leaf, int key, record * pointer)
{
```

```
node * leaf_s;
int * newkeys;
void ** newptrs;
int insertindex, s, new_key, i, j;

leaf_s = newnode();
leaf_s->leaf_or_not=1;

newkeys = (int*)malloc( order * sizeof(int) );

newptrs = (void**)malloc( order * sizeof(void *) );

insertindex = 0;
while (insertindex < order - 1 && leaf->keys[insertindex] < key)
insertindex++;

for (i = 0, j = 0; i < leaf->number_of_keys; i++, j++) {
if (j == insertindex) j++;
newkeys[j] = leaf->keys[i];
newptrs[j] = leaf->ptrs[i];
}

newkeys[insertindex] = key;
newptrs[insertindex] = pointer;

leaf->number_of_keys = 0;

if((order-1)%2==0)
s = (order - 1)/2;
else
s = ((order - 1)/2)+1;

for (i = 0; i < s; i++) {
leaf->ptrs[i] = newptrs[i];
leaf->keys[i] = newkeys[i];
```

```
leaf->number_of_keys++;
}

for (i = s, j = 0; i < order; i++, j++) {
leaf_s->ptrs[j] = newptrs[i];
leaf_s->keys[j] = newkeys[i];
leaf_s->number_of_keys++;
}


leaf_s->ptrs[order - 1] = leaf->ptrs[order - 1];//node pointed by last pointer now should be pointed by new node
leaf->ptrs[order - 1] = leaf_s;//new node should be now pointed by previous node

for (i = leaf->number_of_keys; i < order - 1; i++)//key holes in a node
leaf->ptrs[i] = NULL;
for (i = leaf_s->number_of_keys; i < order - 1; i++)
leaf_s->ptrs[i] = NULL;//pointer holes in a node

leaf_s->parent = leaf->parent;
new_key = leaf_s->keys[0];

return insert_into_parent(root, leaf, new_key, leaf_s);
}

node *insert( node * root, int key, int value ) {
record *pointer;
node *leaf;

if (root == NULL)
{
node * l = newnode();

l->leaf_or_not = 1;
root = l;
root->keys[0] = key;
```

```
root->ptrs[0] = pointer;
root->ptrs[order - 1] = NULL;
root->parent = NULL;
root->number_of_keys++;
//printf("\nroot key[0] = %d",root->keys[0]);
return root;
}

if (find(root, key) != NULL)
return root;
pointer = newrecord(value);

leaf = find_leaf(root, key);

if (leaf->number_of_keys < order - 1) {
leaf = insert_at_leaf(leaf, key, pointer);
return root;
}

return split(root, leaf, key, pointer);
}

int path_to_root( node * root, node * child ) {
int length = 0;
node * c = child;
while (c != root) {
c = c->parent;
length++;
}
return length;
}

node *queue=NULL;

void enq( node * new_node ) {
```

```
node * c;
if (queue == NULL) {
queue = new_node;
queue->next = NULL;
}
else {
c = queue;
while(c->next != NULL) {
c = c->next;
}
c->next = new_node;
new_node->next = NULL;
}
}

node * dq( void ) {
node * n = queue;
queue = queue->next;
n->next = NULL;
return n;
}
void printtree( node * root ) {

node * n = NULL;
int i = 0;
int rank = 0;
int new_rank = 0;

if (root == NULL) {
printf("\n Empty tree.\n");
return;
}

queue = NULL;
enq(root);
```

```
while( queue != NULL )
{
n = dq();
if (n->parent != NULL && n == n->parent->ptrs[0])
{
new_rank = path_to_root( root, n );
if (new_rank != rank)
{
rank = new_rank;
printf("\n");
}
}

for (i = 0; i < n->number_of_keys; i++)
{
printf("%d ", n->keys[i]);
}
if (!n->leaf_or_not){
for (i = 0; i <= n->number_of_keys; i++)
enq((node*)n->ptrs[i]);
}
printf(" & ");
}
printf("\n");
}

int cut( int length ) {
if (length % 2 == 0)
return length/2;
else
return length/2 + 1;
}

int exact_search(node * root, int key){
int i = 0,exact_match_flag=0;
```

```
//-------------------------first find in leaf node is the key is found.---------
node *c = find_leaf( root, key);
if (c == NULL) {
exact_match_flag=0;    //not found ;
}
for (i = 0; i<c->number_of_keys; i++)
{
if (c->keys[i] == key){
exact_match_flag=1;                    //Found KEY ;
break;
}
}
return exact_match_flag;
}
int range_search(node *root){
printf("Enter");

int max,min,flag=1;
node * n = NULL;
int i = 0;
int rank = 0;
int new_rank = 0;
int exact_match_flag=0;
scanf("%d", &min);
scanf("%d", &max);
//-----------------------------------
queue = NULL;
enq(root);
while( queue != NULL )
{
n = dq();
if (n->parent != NULL && n == n->parent->ptrs[0])
{
new_rank = path_to_root( root, n );
if (new_rank != rank)
```

```
{
rank = new_rank;
printf("Level Traversed %d", rank);
printf("\n");
}
}

for (i = 0; i < n->number_of_keys; i++)
{
if (n->leaf_or_not && n->keys[i]>=min && n->keys[i]<=max)
{
if(flag){ printf("Leaf NODE Traversed Neighbor\n"); flag=0;}
printf("%d ",n->keys[i]);
}
}
if (!n->leaf_or_not){
for (i = 0; i <= n->number_of_keys; i++)
enq((node*)n->ptrs[i]);
}
// if (n->leaf_or_not && n->keys[i]>=min && n->keys[i]<=max)
//              {
//              printf(" | ");
//              }
}
return 0;
}
```