/\*Program of insertion and deletion in B tree\*/  
#include  
#include  
#define M 5  
  
struct node{  
int n; /\* n < M No. of keys in node will always less than order of B  
tree \*/  
int keys[M-1]; /\*array of keys\*/  
struct node \*p[M]; /\* (n+1 pointers will be in use) \*/  
}\*root=NULL;  
  
enum KeyStatus { Duplicate,SearchFailure,Success,InsertIt,LessKeys };  
  
void insert(int key);  
void display(struct node \*root,int);  
void DelNode(int x);  
void search(int x);  
enum KeyStatus ins(struct node \*r, int x, int\* y, struct node\*\* u);  
int searchPos(int x,int \*key\_arr, int n);  
enum KeyStatus del(struct node \*r, int x);  
  
int main()  
{  
int key;  
int choice;  
printf("Creation of B tree for node %d\n",M);  
while(1)  
{  
printf("1.Insert\n");  
printf("2.Delete\n");  
printf("3.Search\n");  
printf("4.Display\n");  
printf("5.Quit\n");  
printf("Enter your choice : ");  
scanf("%d",&choice);  
  
switch(choice)  
{  
case 1:  
printf("Enter the key : ");  
scanf("%d",&key);  
insert(key);  
break;  
case 2:  
printf("Enter the key : ");  
scanf("%d",&key);  
DelNode(key);  
break;  
case 3:  
printf("Enter the key : ");  
scanf("%d",&key);  
search(key);  
break;  
case 4:  
printf("Btree is :\n");  
display(root,0);  
break;  
case 5:  
exit(1);  
default:  
printf("Wrong choice\n");  
break;  
}/\*End of switch\*/  
}/\*End of while\*/  
return 0;  
}/\*End of main()\*/  
  
void insert(int key)  
{  
struct node \*newnode;  
int upKey;  
enum KeyStatus value;  
value = ins(root, key, &upKey, &newnode);  
if (value == Duplicate)  
printf("Key already available\n");  
if (value == InsertIt)  
{  
struct node \*uproot = root;  
root=malloc(sizeof(struct node));  
root->n = 1;  
root->keys[0] = upKey;  
root->p[0] = uproot;  
root->p[1] = newnode;  
}/\*End of if \*/  
}/\*End of insert()\*/  
  
enum KeyStatus ins(struct node \*ptr, int key, int \*upKey,struct node  
\*\*newnode)  
{  
struct node \*newPtr, \*lastPtr;  
int pos, i, n,splitPos;  
int newKey, lastKey;  
enum KeyStatus value;  
if (ptr == NULL)  
{  
\*newnode = NULL;  
\*upKey = key;  
return InsertIt;  
}  
n = ptr->n;  
pos = searchPos(key, ptr->keys, n);  
if (pos < n && key == ptr->keys[pos])  
return Duplicate;  
value = ins(ptr->p[pos], key, &newKey, &newPtr);  
if (value != InsertIt)  
return value;  
/\*If keys in node is less than M-1 where M is order of B tree\*/  
if (n < M - 1)  
{  
pos = searchPos(newKey, ptr->keys, n);  
/\*Shifting the key and pointer right for inserting the new key\*/  
for (i=n; i>pos; i--)  
{  
ptr->keys[i] = ptr->keys[i-1];  
ptr->p[i+1] = ptr->p[i];  
}  
/\*Key is inserted at exact location\*/  
ptr->keys[pos] = newKey;  
ptr->p[pos+1] = newPtr;  
++ptr->n; /\*incrementing the number of keys in node\*/  
return Success;  
}/\*End of if \*/  
/\*If keys in nodes are maximum and position of node to be inserted is  
last\*/  
if (pos == M - 1)  
{  
lastKey = newKey;  
lastPtr = newPtr;  
}  
else /\*If keys in node are maximum and position of node to be inserted  
is not last\*/  
{  
lastKey = ptr->keys[M-2];  
lastPtr = ptr->p[M-1];  
for (i=M-2; i>pos; i--)  
{  
ptr->keys[i] = ptr->keys[i-1];  
ptr->p[i+1] = ptr->p[i];  
}  
ptr->keys[pos] = newKey;  
ptr->p[pos+1] = newPtr;  
}  
splitPos = (M - 1)/2;  
(\*upKey) = ptr->keys[splitPos];  
  
(\*newnode)=malloc(sizeof(struct node));/\*Right node after split\*/  
ptr->n = splitPos; /\*No. of keys for left splitted node\*/  
(\*newnode)->n = M-1-splitPos;/\*No. of keys for right splitted node\*/  
for (i=0; i < (\*newnode)->n; i++)  
{  
(\*newnode)->p[i] = ptr->p[i + splitPos + 1];  
if(i < (\*newnode)->n - 1)  
(\*newnode)->keys[i] = ptr->keys[i + splitPos + 1];  
else  
(\*newnode)->keys[i] = lastKey;  
}  
(\*newnode)->p[(\*newnode)->n] = lastPtr;  
return InsertIt;  
}/\*End of ins()\*/  
  
void display(struct node \*ptr, int blanks)  
{  
if (ptr)  
{  
int i;  
for(i=1;i<=blanks;i++)  
printf(" ");  
for (i=0; i < ptr->n; i++)  
printf("%d ",ptr->keys[i]);  
printf("\n");  
for (i=0; i <= ptr->n; i++)  
display(ptr->p[i], blanks+10);  
}/\*End of if\*/  
}/\*End of display()\*/  
  
void search(int key)  
{  
int pos, i, n;  
struct node \*ptr = root;  
printf("Search path:\n");  
while (ptr)  
{  
n = ptr->n;  
for (i=0; i < ptr->n; i++)  
printf(" %d",ptr->keys[i]);  
printf("\n");  
pos = searchPos(key, ptr->keys, n);  
if (pos < n && key == ptr->keys[pos])  
{  
printf("Key %d found in position %d of last dispalyed  
node\n",key,i);  
return;  
}  
ptr = ptr->p[pos];  
}  
printf("Key %d is not available\n",key);  
}/\*End of search()\*/  
  
int searchPos(int key, int \*key\_arr, int n)  
{  
int pos=0;  
while (pos < n && key > key\_arr[pos])  
pos++;  
return pos;  
}/\*End of searchPos()\*/  
  
void DelNode(int key)  
{  
struct node \*uproot;  
enum KeyStatus value;  
value = del(root,key);  
switch (value)  
{  
case SearchFailure:  
printf("Key %d is not available\n",key);  
break;  
case LessKeys:  
uproot = root;  
root = root->p[0];  
free(uproot);  
break;  
}/\*End of switch\*/  
}/\*End of delnode()\*/  
  
enum KeyStatus del(struct node \*ptr, int key)  
{  
int pos, i, pivot, n ,min;  
int \*key\_arr;  
enum KeyStatus value;  
struct node \*\*p,\*lptr,\*rptr;  
  
if (ptr == NULL)  
return SearchFailure;  
/\*Assigns values of node\*/  
n=ptr->n;  
key\_arr = ptr->keys;  
p = ptr->p;  
min = (M - 1)/2;/\*Minimum number of keys\*/  
  
pos = searchPos(key, key\_arr, n);  
if (p[0] == NULL)  
{  
if (pos == n || key < key\_arr[pos])  
return SearchFailure;  
/\*Shift keys and pointers left\*/  
for (i=pos+1; i < n; i++)  
{  
key\_arr[i-1] = key\_arr[i];  
p[i] = p[i+1];  
}  
return --ptr->n >= (ptr==root ? 1 : min) ? Success : LessKeys;  
}/\*End of if \*/  
  
if (pos < n && key == key\_arr[pos])  
{  
struct node \*qp = p[pos], \*qp1;  
int nkey;  
while(1)  
{  
nkey = qp->n;  
qp1 = qp->p[nkey];  
if (qp1 == NULL)  
break;  
qp = qp1;  
}/\*End of while\*/  
key\_arr[pos] = qp->keys[nkey-1];  
qp->keys[nkey - 1] = key;  
}/\*End of if \*/  
value = del(p[pos], key);  
if (value != LessKeys)  
return value;  
  
if (pos > 0 && p[pos-1]->n > min)  
{  
pivot = pos - 1; /\*pivot for left and right node\*/  
lptr = p[pivot];  
rptr = p[pos];  
/\*Assigns values for right node\*/  
rptr->p[rptr->n + 1] = rptr->p[rptr->n];  
for (i=rptr->n; i>0; i--)  
{  
rptr->keys[i] = rptr->keys[i-1];  
rptr->p[i] = rptr->p[i-1];  
}  
rptr->n++;  
rptr->keys[0] = key\_arr[pivot];  
rptr->p[0] = lptr->p[lptr->n];  
key\_arr[pivot] = lptr->keys[--lptr->n];  
return Success;  
}/\*End of if \*/  
if (posn > min)  
{  
pivot = pos; /\*pivot for left and right node\*/  
lptr = p[pivot];  
rptr = p[pivot+1];  
/\*Assigns values for left node\*/  
lptr->keys[lptr->n] = key\_arr[pivot];  
lptr->p[lptr->n + 1] = rptr->p[0];  
key\_arr[pivot] = rptr->keys[0];  
lptr->n++;  
rptr->n--;  
for (i=0; i < rptr->n; i++)  
{  
rptr->keys[i] = rptr->keys[i+1];  
rptr->p[i] = rptr->p[i+1];  
}/\*End of for\*/  
rptr->p[rptr->n] = rptr->p[rptr->n + 1];  
return Success;  
}/\*End of if \*/  
  
if(pos == n)  
pivot = pos-1;  
else  
pivot = pos;  
  
lptr = p[pivot];  
rptr = p[pivot+1];  
/\*merge right node with left node\*/  
lptr->keys[lptr->n] = key\_arr[pivot];  
lptr->p[lptr->n + 1] = rptr->p[0];  
for (i=0; i < rptr->n; i++)  
{  
lptr->keys[lptr->n + 1 + i] = rptr->keys[i];  
lptr->p[lptr->n + 2 + i] = rptr->p[i+1];  
}  
lptr->n = lptr->n + rptr->n +1;  
free(rptr); /\*Remove right node\*/  
for (i=pos+1; i < n; i++)  
{  
key\_arr[i-1] = key\_arr[i];  
p[i] = p[i+1];  
}  
return --ptr->n >= (ptr == root ? 1 : min) ? Success : LessKeys;  
}/\*End of del()\*/ 