//Bryan Orville Audric

//2602160750

struct BTreeNode

char \*keys

int degree

struct BTreeNode \*\*child

int numKey

bool leaf

END Struct

//For main root

struct BTree

struct BTreeNode \*root

int degree

END Struct

struct BTree \*BTreeconstruct(int degree):

struct BTree \*bTree = (struct BTree\*)malloc(sizeof(struct BTree))

bTree->root = NULL

bTree->degree = degree

return bTree

END Function

struct BTreeNode \*Nodeconstruct(struct BTree \*bTree, int degree1, bool leaf1):

struct BTreeNode \*node = (struct BTreeNode \*)malloc(sizeof(struct BTreeNode))

node->degree = degree1

node->leaf = leaf1

node->keys = malloc((sizeof(char) \* bTree->degree \* 2) - 1)

node->child = malloc((sizeof(struct BTreeNode) \* bTree->degree \* 2))

node->numKey = 0

return node

END Function

void SplitChild(struct BTreeNode \*nodeRoot, int index, struct BTreeNode \*child, struct BTree \*bTree):

struct BTreeNode \*newNode = Nodeconstruct(bTree, child->degree, child->leaf)

newNode->numKey = nodeRoot->degree - 1

for int j = 0; j < nodeRoot->degree - 1; j++

newNode->keys[j] = child->keys[j + nodeRoot->degree]

END for

if child->leaf is false

for int j = 0; j < nodeRoot->degree; j++

newNode->child[j] = child->child[j + nodeRoot->degree]

END for

END If

child->numKey = nodeRoot->degree - 1

for int j = nodeRoot->numKey; j >= index + 1; j--

nodeRoot->child[j + 1] = nodeRoot->child[j]

END for

nodeRoot->child[index + 1] = newNode

for int j = nodeRoot->numKey - 1; j >= index; j--

nodeRoot->keys[j + 1] = nodeRoot->keys[j]

END for

nodeRoot->keys[index] = child->keys[nodeRoot->degree - 1]

nodeRoot->numKey ++

END Function

void NodeinsertNonFull(struct BTree \*bTree, struct BTreeNode \*nodeRoot, char key):

int i = nodeRoot->numKey - 1

if nodeRoot->leaf is true

while i >= 0 && nodeRoot->keys[i] > key

nodeRoot->keys[i + 1] = nodeRoot->keys[i]

i--

END while

nodeRoot->keys[i + 1] = key

nodeRoot->numKey ++

else

while i >= 0 && nodeRoot->keys[i] > key

i--

END while

if nodeRoot->child[i + 1]->numKey == (2 \* nodeRoot->degree) - 1

SplitChild(nodeRoot, i + 1, nodeRoot->child[i + 1], bTree)

if nodeRoot->keys[i + 1] < key

i++

END if

END if

NodeinsertNonFull(bTree, nodeRoot->child[i + 1], key)

END if

END Function

void Insert(struct BTree \*bTree, char key):

if bTree->root is NULL

bTree->root = Nodeconstruct(bTree, bTree->degree, true)

bTree->root->keys[0] = key

bTree->root->numKey = 1

else

if bTree->root->numKey == (2 \* bTree->degree) - 1

struct BTreeNode \*newNode = Nodeconstruct(bTree, bTree->degree, false)

newNode->child[0] = bTree->root

SplitChild(newNode, 0, bTree->root, bTree)

int i = 0

if newNode->keys[0] < key

i++

END if

NodeinsertNonFull(bTree, newNode->child[i], key)

bTree->root = newNode

else

NodeinsertNonFull(bTree, bTree->root, key)

END if

END if

END Function

void Noderemove(struct BTreeNode \*nodeRoot, char key)

int GetPredecessor(struct BTreeNode \*nodeRoot, int idx):

struct BTreeNode \*Current = nodeRoot->child[idx]

while Current->leaf is false

Current = Current->child[Current->numKey]

END while

return Current->keys[Current->numKey - 1]

END Function

int GetSuccessor(struct BTreeNode \*nodeRoot, int idx):

struct BTreeNode \*Current = nodeRoot->child[idx + 1]

while Current->leaf is false

Current = Current->child[0]

END while

return Current->keys[0]

END Function

void Merge(struct BTreeNode \*nodeRoot, int idx):

struct BTreeNode \*child = nodeRoot->child[idx]

struct BTreeNode \*sibling = nodeRoot->child[idx + 1]

child->keys[nodeRoot->degree - 1] = nodeRoot->keys[idx]

for int i = 0; i < sibling->numKey; i++

child->keys[i + nodeRoot->degree] = sibling->keys[i]

END for

if child->leaf is false

for int i = 0; i <= sibling->numKey; i++

child->child[i + nodeRoot->degree] = sibling->child[i]

END for

END if

for int i = idx + 1; i < nodeRoot->numKey; i++

nodeRoot->keys[i - 1] = nodeRoot->keys[i]

END for

for int i = idx + 2; i <= nodeRoot->numKey; i++

nodeRoot->child[i - 1] = nodeRoot->child[i]

END for

child->numKey += sibling->numKey + 1

nodeRoot->numKey --

free(sibling)

sibling = NULL

return

END Function

void RemoveNonLeaf(struct BTreeNode \*nodeRoot, int idx):

int key = nodeRoot->keys[idx]

if nodeRoot->child[idx]->numKey >= nodeRoot->degree

int Predecessor = GetPredecessor(nodeRoot, idx)

nodeRoot->keys[idx] = Predecessor

Noderemove(nodeRoot->child[idx], Predecessor)

else if nodeRoot->child[idx + 1]->numKey >= nodeRoot->degree

int Successor = GetSuccessor(nodeRoot, idx)

nodeRoot->keys[idx] = Successor

Noderemove(nodeRoot->child[idx + 1], Successor)

else

Merge(nodeRoot, idx)

Noderemove(nodeRoot->child[idx], key)

END if

return

END Function

void RemoveLeaf(struct BTreeNode \*nodeRoot, int idx):

for int i = idx + 1; i < nodeRoot->numKey; i++

nodeRoot->keys[i - 1] = nodeRoot->keys[i]

END for

nodeRoot->numKey -= 1

return

END Function

void BorrowPrev(struct BTreeNode \*nodeRoot, int idx):

struct BTreeNode \*child = nodeRoot->child[idx]

struct BTreeNode \*sibling = nodeRoot->child[idx - 1]

for int i = child->numKey - 1; i >= 0; i--

child->keys[i + 1] = child->keys[i]

END for

if child->leaf is false

for int i = child->numKey; i >= 0; i--

child->child[i + 1] = child->child[i]

END for

END if

child->keys[0] = nodeRoot->keys[idx - 1]

if child->leaf is false

child->child[0] = sibling->child[sibling->numKey]

END if

nodeRoot->keys[idx - 1] = sibling->keys[sibling->numKey - 1]

child->numKey ++

sibling->numKey --

return

END Function

void BorrowNext(struct BTreeNode \*nodeRoot, int idx):

struct BTreeNode \*child = nodeRoot->child[idx]

struct BTreeNode \*sibling = nodeRoot->child[idx + 1]

child->keys[(child->numKey)] = nodeRoot->keys[idx]

if child->leaf is false

child->child[(child->numKey) + 1] = sibling->child[0]

END if

nodeRoot->keys[idx] = sibling->keys[0]

for int i = 1; i < sibling->numKey; i++

sibling->keys[i - 1] = sibling->keys[i]

END for

if sibling->leaf is false

for int i = 1; i <= sibling->numKey; i++

sibling->child[i - 1] = sibling->child[i]

END for

END if

child->numKey ++

sibling->numKey --

return

END Function

void Nodefill(struct BTreeNode \*nodeRoot, int idx):

if idx != 0 && nodeRoot->child[idx - 1]->numKey >= nodeRoot->degree

BorrowPrev(nodeRoot, idx)

else if idx != nodeRoot->numKey && nodeRoot->child[idx + 1]->numKey >= nodeRoot->degree

BorrowNext(nodeRoot, idx)

else

if idx is not nodeRoot->numKey

Merge(nodeRoot, idx)

else

Merge(nodeRoot, idx - 1)

END if

END if

return

END Function

int FindKey(struct BTreeNode \*nodeRoot, char key):

int idx = 0

while idx < nodeRoot->numKey && nodeRoot->keys[idx] < key

idx++

END while

return idx

END Function

void Noderemove(struct BTreeNode \*nodeRoot, char key):

int idx = FindKey(nodeRoot, key)

if idx < nodeRoot->numKey && nodeRoot->keys[idx] == key

if nodeRoot->leaf is true

RemoveLeaf(nodeRoot, idx)

else

RemoveNonLeaf(nodeRoot, idx)

END If

else

bool flag

if nodeRoot->leaf is true

Display"%c does not exist in the tree\n", key

return

END if

If idx is nodeRoot->numKey

flag = true

else

flag = false

END If

if nodeRoot->child[idx]->numKey < nodeRoot->degree

Nodefill(nodeRoot, idx)

END if

if flag is true && idx > nodeRoot->numKey

Noderemove(nodeRoot->child[idx - 1], key)

else

Noderemove(nodeRoot->child[idx], key)

END if

END If

return

END Function

void BTreeremove(struct BTree \*bTree, char key):

if bTree->root is NULL

Display"There is no data\n"

return

END if

Noderemove(bTree->root, key)

if bTree->root->numKey is 0

struct BTreeNode \*temp = bTree->root

if bTree->root->leaf is true

bTree->root = NULL

else

bTree->root = bTree->root->child[0]

END if

free(temp)

temp = NULL

END if

return

END Function

void Print(struct BTreeNode \*nodeRoot):

int i

for i = 0; i < nodeRoot->numKey; i++

if nodeRoot->leaf is false

Print(nodeRoot->child[i])

END if

Display "%c ", nodeRoot->keys[i]

END for

if nodeRoot->leaf is false

Print(nodeRoot->child[i])

END if

END Function

void BTreeCheck(struct BTree \*root):

if root->root is not NULL

Print(root->root)

END if

END Function

int main():

int menu

char data

//ORDER = 4

struct BTree \*root=BTreeconstruct(2)

Insert(root, 'A')

Insert(root, 'C')

Insert(root, 'D')

Insert(root, 'K')

Insert(root, 'M')

Insert(root, 'O')

Insert(root, 'P')

Insert(root, 'R')

Insert(root, 'S')

Insert(root, 'T')

Insert(root, 'U')

do

Display "1. Insert\n"

Display "2. Delete\n"

Display "3. Print\n"

Display "4. Exit\n"

Display ">> "

Input menu

switch(menu)

case 1:

Insert(root, 'E')

Insert(root, 'I')

Insert(root, 'L')

Insert(root, 'G')

Insert(root, 'X')

Display "[E,I,L,G,X] Inserted \n"

system("pause")

system("cls")

break

case 2:

Display "Insert data to delete: "

Input data

data = getchar()

BTreeremove(root,data)

system("pause")

system("cls")

break

case 3:

BTreeCheck(root)

Display "\n"

system("pause")

system("cls")

break

case 4:

return 0

END switch

While menu is not 4

END do while

return 0

END Function