**BSTdef.h**

#include <stdio.h>

#include <stdlib.h>

typedef struct \_\_bookNode bNode;

typedef struct \_\_bookNode\* node;

typedef struct \_\_book book;

typedef struct \_\_bst\* BST;

typedef struct \_\_bst BinTree;

struct \_\_bst{

int totalBooks;

int height;

node root;

};

struct \_\_book{

int acc; //unique accession number

char \*title;

char \*name;

int price;

};

struct \_\_bookNode

{

book b;

struct \_\_bookNode \*left, \*right;

};

#define TRUE 1

#define FALSE 0

**BSTOps.h**

BST createEmptyBST();

node insertInBST(node tree, book newb);

BST insertBST(BST bt, book b);

node createNode(book b);

void bstPrint(BST bt);

void treePrint(node p);

book findMax(node p, book b);

book findMaxPrice(BST bt);

book findMinPrice(BST bt);

int getHeight(BST bt);

book findLatestBook(BST bt);

book findOldestBook(BST bt);

BST reOrderBST(BST bt);

**BSTOps.c**

#include "BSTdef.h"

#include "BSTOps.h"

BST createEmptyBST()

{

node head=NULL;

BST bt= (BST) malloc(sizeof(BinTree));

bt->totalBooks=0;

bt->height=0;

bt->root=head;

return bt;

}

void deleteNode(node p, book b){

if(findNode(p, b)==0){

printf("The node to be deleted is not in tree.\n");

return;

}

node q = p;

node x, parent, xsucc;

while(q->b.acc != b.acc && q!=NULL){

if(q->b.acc > b.acc){

parent = q;

q = q->left;

}

else{

parent = q;

q = q->right;

}

}

x=q;

/\* if the node to be deleted has two children \*/

if(x->left!=NULL && x->right!=NULL){

parent = x;

xsucc = x->right;

while (xsucc->left != NULL)

{

parent = xsucc ;

xsucc = xsucc->left;

}

x->b = xsucc ->b;

x = xsucc;

}

// /\* if the node to be deleted has no child \*/

if (x->left == NULL && x->right == NULL){

if (parent->right == x)

parent->right = NULL ;

else

parent->left = NULL ;

free (x) ;

return;

}

/\* if the node to be deleted has only right \*/

if (x->left == NULL && x->right != NULL)

{

if (parent->left == x)

parent->left = x->right ;

else

parent->right = x->right ;

free (x) ;

return;

}

/\* if the node to be deleted has only left child \*/

if (x->left != NULL && x->right == NULL)

{

if (parent->left == x)

parent->left = x->left ;

else

parent->right = x->left ;

free (x) ;

return;

}

}

BST delBST(BST bt, book b){

deleteNode(bt->root, b);

return bt;

}

int findNode(node p, book f){

if(p==NULL){

return FALSE;

}

else if (p->b.acc == f.acc){

return TRUE;

}

else if(p->b.acc < f.acc){

findNode(p->right, f);}

else{

findNode(p->left, f);

}

}

int findBST(BST bt, book b){

return findNode(bt->root, b);

}

book newb;

book findMaxP(node p){

if(p!=NULL){

findMaxP(p->left);

//Operations

// printf("Present max price: %d\n",newb.price);

if(p->b.price>newb.price){

// printf(" Pointer price: %d\n",p->b.price);

newb = p->b;

// printf(" New max price: %d\n",newb.price);

}

findMaxP(p->right);

}

return newb;

}

book findMaxPrice(BST bt){

return findMaxP(bt->root);

}

book findMinP(node p){

if(p!=NULL){

// Visit Node

if(p->b.price<newb.price){

newb = p->b;

}

findMinP(p->left);

findMinP(p->right);

}

return newb;

}

book findMinPrice(BST bt){

return findMinP(bt->root);

}

int findHeight(node p){

if(p == NULL)

return -1;

int lefth = findHeight(p->left);

int righth = findHeight(p->right);

if(lefth > righth)

return lefth + 1;

else

return righth +1;

}

int getHeight(BST bt){

return findHeight(bt->root);

}

book findMaxA(node p){

if(p!=NULL){

findMaxA(p->left);

//Visit Node

if(p->b.acc>newb.acc){

newb = p->b;

}

findMaxA(p->right);

}

return newb;

}

book findLatestBook(BST bt){

return findMaxA(bt->root);

}

book findMinA(node p){

if(p!=NULL){

// Visit Node

if(p->b.acc<newb.acc){

newb = p->b;

}

findMinA(p->left);

findMinA(p->right);

}

return newb;

}

book findOldestBook(BST bt){

return findMinA(bt->root);

}

node createNode(book b){

node newNode = (node)malloc(sizeof(bNode));

newNode->b = b;

newNode->left = newNode->right = NULL;

return newNode;

}

node insertInBST(node tree, book newb){

if(tree==NULL){

tree = createNode(newb);

tree->left = tree->right = NULL;

}

else if(tree->b.acc < newb.acc){

tree->right = insertInBST(tree->right, newb);

}

else

tree->left = insertInBST(tree->left, newb);

return tree;

}

BST insertBST(BST bt, book b){

bt->root=insertInBST(bt->root, b);

bt->totalBooks++;

return bt;

}

void treePrint(node p){

if(p!=NULL){

treePrint(p->left);

printf("%d-> ", p->b.acc);

treePrint(p->right);

}

}

void bstPrint(BST bt){

treePrint(bt->root);

}

node insertPrice(node tree, book newb){

// printf("newb %s\n", newb.title);

if(tree==NULL){

tree = createNode(newb);

tree->left = tree->right = NULL;

// printf("Inserting %d with price %d\n", newb.acc, newb.price);

}

else if(tree->b.price < newb.price){

tree->right = insertPrice(tree->right, newb);

// printf("Inserting %d with price %d\n", newb.acc, newb.price);

}

else{

tree->left = insertPrice(tree->left, newb);

// printf("Inserting %d with price %d\n", newb.acc, newb.price);

}

return tree;

}

node reOrd(node root, node tree){

if(root!=NULL){

tree = reOrd(root->left, tree);

if(tree==NULL){

tree = createNode(root->b);

tree->left = tree->right = NULL;

}

else{

tree = insertPrice(tree, root->b);

}

tree = reOrd(root->right, tree);

}

return tree;

}

BST reOrderBST(BST bt){

BST btp = createEmptyBST();

btp->root = reOrd(bt->root, btp->root);

return btp;

}

**AVLTree.h**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef struct avlnode \*node;

typedef struct pos \*posNode;

struct avlnode{

char word[100];

struct avlnode \*left;

struct avlnode \*right;

int height;

int count;

posNode root;

};

struct pos{

long pos;

struct pos \* next;

};

**treeOps.c – Complete with linked List in every node**

# include "tree.h"

posNode createNode(long pos){

posNode newNode=(posNode)malloc(sizeof(struct pos));

newNode->pos = pos;

newNode->next = NULL;

}

posNode insertNode(posNode root, long pos){

struct pos \*temp = root;

if(temp==NULL){

temp = createNode(pos);

return root;

}

if ( temp != NULL ) {

while ( temp->next != NULL)

{

temp = temp->next;

}

}

temp->next = createNode(pos);

return root;

}

void print(struct pos \* root){

// root = root->next;

while(root!=NULL){

printf("%ld\t", root->pos);

root = root->next;

}

}

node newNode(char word[], long pos){

node newNode = (node)malloc(sizeof(struct avlnode));

strcpy(newNode->word,word);

newNode->left = newNode->right = NULL;

newNode->height = 1;

newNode->count = 1;

newNode->root = (struct pos\*)malloc(sizeof(struct pos));

newNode->root->next = NULL;

newNode->root->pos = pos;

return newNode;

}

int max(int a, int b){

return (a > b)? a : b;

}

int height(node N){

if (N == NULL)

return 0;

return N->height;

}

int findNode(node p, char word[]){

if(p==NULL){

return 0;

}

else if (strcmp(p->word, word)==0){

return 1; //if found, return 1

}

else if(strcmp(p->word, word)<0){

findNode(p->right, word);}

else{

findNode(p->left, word);

}

return 0;

}

node retNode(node p, char word[]){

if(p==NULL){

return NULL;

}

else if (strcmp(p->word, word)==0){

return p;

}

else if(strcmp(p->word, word)<0){

retNode(p->right, word);}

else{

retNode(p->left, word);

}

}

node rightRotate(node y){

node x = y->left;

node T2 = x->right;

// Perform rotation

x->right = y;

y->left = T2;

// Update heights

y->height = max(height(y->left), height(y->right))+1;

x->height = max(height(x->left), height(x->right))+1;

// Return new root

return x;

}

node leftRotate(node x){

node y = x->right;

node T2 = y->left;

// Perform rotation

y->left = x;

x->right = T2;

// Update heights

x->height = max(height(x->left), height(x->right))+1;

y->height = max(height(y->left), height(y->right))+1;

// Return new root

return y;

}

int getBalance(node N){

if (N == NULL)

return 0;

return height(N->left) - height(N->right);

}

node insert(node node, char word[], long pos){

if (node == NULL)

return(newNode(word,pos));

int x = strcmp(word, node->word);

if (x<0)

node->left = insert(node->left, word, pos);

else

node->right = insert(node->right, word, pos);

/\* 2. Update height of this ancestor node \*/

node->height = max(height(node->left), height(node->right)) + 1;

/\* 3. Get the balance factor of this ancestor node to check whether

this node became unbalanced \*/

int balance = getBalance(node);

// If this node becomes unbalanced, then there are 4 cases

// Left Left Case

if (balance > 1 && (strcmp(word, node->left->word)<0))

return rightRotate(node);

// Right Right Case

if (balance < -1 && (strcmp(word, node->right->word)>0))

return leftRotate(node);

// Left Right Case

if (balance > 1 && (strcmp(word, node->left->word)>0)) {

node->left = leftRotate(node->left);

return rightRotate(node);

}

// Right Left Case

if (balance < -1 && (strcmp(word, node->right->word)<0)) {

node->right = rightRotate(node->right);

return leftRotate(node);

}

/\* return the (unchanged) node pointer \*/

return node;

}

void inOrder(node root){

if(root != NULL){

inOrder(root->left);

printf("Word: %s,\tNo of Instances: %d,\tPosition from beginning:", root->word, root->count);

print(root->root);

printf("\n");

inOrder(root->right);

}

}

node asliInsert(node node, char word[], long pos){

// int z = findNode(node, word);

struct avlnode\* up = retNode(node, word);

if(up!=NULL){

up->count = up->count+1;

// printf("%s\n", up->word);

up->root = insertNode(up->root, pos);

}

else

node = insert(node, word, pos);

return node;

}

int main(){

char word[100];

node root = NULL;

FILE \*input = fopen("input.txt", "r");//text file for reading into a hash table

if(input==NULL){

exit(1);

}

while(fscanf(input, "%s ", word)!=EOF){

// root = newNode(word);

long pos = ftell(input);

root = asliInsert(root, word, pos);

// printf("%ld\t",ftell(input));

}

inOrder(root);

}

**basicLL.c**

struct pos \* newNode(long pos){

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

newNode->pos = pos;

newNode->next = NULL;

return newNode;

}

struct pos \* insert(struct pos\* root, long pos){

struct pos \*temp = root;

if(temp==NULL){

temp = newNode(pos);

return root;

}

if ( temp != NULL ) {

while ( temp->next != NULL)

{

temp = temp->next;

}

}

temp->next = newNode(pos);

return root;

}

void print(struct pos \* root){

root = root->next;

while(root!=NULL){

printf("%ld\t", root->pos);

root = root->next;

}

}

int main(){

struct pos \*root = (struct pos\*)malloc(sizeof(struct pos));

root->next = NULL;

root->pos = 0;

// root = (struct pos\*)malloc(sizeof(struct pos));

// root->next = NULL;

// root->pos = 0;

root = insert(root, 5);

root = insert(root, 15);

root = insert(root, 15);

root = insert(root, 15);

root = insert(root, 15);

root = insert(root, 55);

print(root);

return 0;

}

**basicTries.c**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define ARRAY\_SIZE(a) sizeof(a)/sizeof(a[0])

// Alphabet size (# of symbols)

#define ALPHABET\_SIZE (26)

#define INDEX(c) ((int)c - (int)'a')

#define FREE(p) \

free(p); \

p = NULL;

// forward declration

typedef struct trie\_node trie\_node\_t;

// trie node

struct trie\_node

{

int value; // non zero if leaf

trie\_node\_t \*children[ALPHABET\_SIZE];

};

// trie ADT

typedef struct trie trie\_t;

struct trie

{

trie\_node\_t \*root;

int count;

};

trie\_node\_t \*getNode(void)

{

trie\_node\_t \*pNode = NULL;

pNode = (trie\_node\_t \*)malloc(sizeof(trie\_node\_t));

if( pNode )

{

int i;

pNode->value = 0;

for(i = 0; i < ALPHABET\_SIZE; i++)

{

pNode->children[i] = NULL;

}

}

return pNode;

}

void initialize(trie\_t \*pTrie)

{

pTrie->root = getNode();

pTrie->count = 0;

}

void insert(trie\_t \*pTrie, char key[])

{

int level;

int length = strlen(key);

int index;

trie\_node\_t \*pCrawl;

pTrie->count++;

pCrawl = pTrie->root;

for( level = 0; level < length; level++ )

{

index = INDEX(key[level]);

if( pCrawl->children[index] )

{

// Skip current node

pCrawl = pCrawl->children[index];

}

else

{

// Add new node

pCrawl->children[index] = getNode();

pCrawl = pCrawl->children[index];

}

}

// mark last node as leaf (non zero)

pCrawl->value = pTrie->count;

}

int search(trie\_t \*pTrie, char key[])

{

int level;

int length = strlen(key);

int index;

trie\_node\_t \*pCrawl;

pCrawl = pTrie->root;

for( level = 0; level < length; level++ )

{

index = INDEX(key[level]);

if( !pCrawl->children[index] )

{

return 0;

}

pCrawl = pCrawl->children[index];

}

return (0 != pCrawl && pCrawl->value);

}

int leafNode(trie\_node\_t \*pNode)

{

return (pNode->value != 0);

}

int isItFreeNode(trie\_node\_t \*pNode)

{

int i;

for(i = 0; i < ALPHABET\_SIZE; i++)

{

if( pNode->children[i] )

return 0;// return false, if any children exist

}

return 1;//else return true if it a free node

}

bool deleteHelper(trie\_node\_t \*pNode, char key[], int level, int len)

{

if( pNode )

{

// Base case

if( level == len )

{

if( pNode->value )

{

// Unmark leaf node

pNode->value = 0;

// If empty, node to be deleted

if( isItFreeNode(pNode) )

{

return true;

}

return false;

}

}

else // Recursive case

{

int index = INDEX(key[level]);

if( deleteHelper(pNode->children[index], key, level+1, len) )

{

// last node marked, delete it

FREE(pNode->children[index]);

// recursively climb up, and delete eligible nodes

return ( !leafNode(pNode) && isItFreeNode(pNode) );

}

}

}

return false;

}

void deleteKey(trie\_t \*pTrie, char key[])

{

int len = strlen(key);

if( len > 0 ) {

deleteHelper(pTrie->root, key, 0, len);

}

}

int main(){

char keys[][8] = {"she", "sells", "sea", "shore", "the", "by", "sheer"};

trie\_t trie;

initialize(&trie);

for(int i = 0; i < ARRAY\_SIZE(keys); i++){

insert(&trie, keys[i]);

}

deleteKey(&trie, keys[0]);

printf("%s %s\n", "she", search(&trie, "she") ? "Present in trie" : "Not present in trie");

return 0;}

**B Trees**

#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;

(\*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) {//NULL =0; if null enter this condition

\*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 {

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;

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

if (flag)

root = createNode(i, child);

}

/\* copy successor for the value to be deleted \*/

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

struct btreeNode \*dummy;

dummy = myNode->link[pos];

for (;dummy->link[0] != NULL;)

dummy = dummy->link[0];

myNode->val[pos] = dummy->val[1];

}

/\* removes the value from the given node and rearrange values \*/

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

int i = pos + 1;

while (i <= myNode->count) {

myNode->val[i - 1] = myNode->val[i];

myNode->link[i - 1] = myNode->link[i];

i++;

}

myNode->count--;

}

/\* shifts value from parent to right child \*/

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

struct btreeNode \*x = myNode->link[pos];

int j = x->count;

while (j > 0) {

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

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

}

x->val[1] = myNode->val[pos];

x->link[1] = x->link[0];

x->count++;

x = myNode->link[pos - 1];

myNode->val[pos] = x->val[x->count];

myNode->link[pos] = x->link[x->count];

x->count--;

return;

}

/\* shifts value from parent to left child \*/

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

int j = 1;

struct btreeNode \*x = myNode->link[pos - 1];

x->count++;

x->val[x->count] = myNode->val[pos];

x->link[x->count] = myNode->link[pos]->link[0];

x = myNode->link[pos];

myNode->val[pos] = x->val[1];

x->link[0] = x->link[1];

x->count--;

while (j <= x->count) {

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

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

j++;

}

return;

}

/\* merge nodes \*/

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

int j = 1;

struct btreeNode \*x1 = myNode->link[pos], \*x2 = myNode->link[pos - 1];

x2->count++;

x2->val[x2->count] = myNode->val[pos];

x2->link[x2->count] = myNode->link[0];

while (j <= x1->count) {

x2->count++;

x2->val[x2->count] = x1->val[j];

x2->link[x2->count] = x1->link[j];

j++;

}

j = pos;

while (j < myNode->count) {

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

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

j++;

}

myNode->count--;

free(x1);

}

/\* adjusts the given node \*/

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

if (!pos) {

if (myNode->link[1]->count > MIN) {

doLeftShift(myNode, 1);

} else {

mergeNodes(myNode, 1);

}

} else {

if (myNode->count != pos) {

if(myNode->link[pos - 1]->count > MIN) {

doRightShift(myNode, pos);

} else {

if (myNode->link[pos + 1]->count > MIN) {

doLeftShift(myNode, pos + 1);

} else {

mergeNodes(myNode, pos);

}

}

} else {

if (myNode->link[pos - 1]->count > MIN)

doRightShift(myNode, pos);

else

mergeNodes(myNode, pos);

}

}

}

/\* delete val from the node \*/

int delValFromNode(int val, struct btreeNode \*myNode) {

int pos, flag = 0;

if (myNode) {

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

pos = 0;

flag = 0;

} else {

for (pos = myNode->count;

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

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

flag = 1;

} else {

flag = 0;

}

}

if (flag) {

if (myNode->link[pos - 1]) {

copySuccessor(myNode, pos);

flag = delValFromNode(myNode->val[pos], myNode->link[pos]);

if (flag == 0) {

printf("Given data is not present in B-Tree\n");

}

} else {

removeVal(myNode, pos);

}

} else {

flag = delValFromNode(val, myNode->link[pos]);

}

if (myNode->link[pos]) {

if (myNode->link[pos]->count < MIN)

adjustNode(myNode, pos);

}

}

return flag;

}

/\* delete val from B-tree \*/

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

struct btreeNode \*tmp;

if (!delValFromNode(val, myNode)) {

printf("Given value is not present in B-Tree\n");

return;

} else {

if (myNode->count == 0) {

tmp = myNode;

myNode = myNode->link[0];

free(tmp);

}

}

root = myNode;

return;

}

/\* 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\t2. Deletion\n");

printf("3. Searching\t4. Traversal\n");

printf("5. Exit\nEnter your choice:");

scanf("%d", &ch);

switch (ch) {

case 1:

printf("Enter your input:");

scanf("%d", &val);

insertion(val);

break;

case 2:

printf("Enter the element to delete:");

scanf("%d", &val);

deletion(val, root);

break;

case 3:

printf("Enter the element to search:");

scanf("%d", &val);

searching(val, &ch, root);

break;

case 4:

traversal(root);

break;

case 5:

exit(0);

default:

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

break;

}

printf("\n");

}

}