client.c -->

treePublic.h, treePrivate.h, tree.c -->

dataInt.h dataInt.c OR dataStr.h dataStr.c

###

### client.c

###

#include "treePublic.h"

#define OK 1

#define KO 0

int

main (

void

)

{

node\_t \*root, \*tmp;

data\_t d;

int retValue, end = 0;

char row[MAXC];

FILE \*fp = NULL;

root = NULL;

root = createEmptyTree();

while (!end) {

fprintf (stdout, "User selection\n");

fprintf (stdout, "\t1) Insert one single node\n\t2) Cancel one single node\n\t3) Search a node\n");

fprintf (stdout, "\t4) Print BST on stdout\n\t5) Write BST on file\n\t6) Read BST from file\n");

fprintf (stdout, "\t7) Compute min and max\n\t8) End\n\t> ");

scanf ("%1s", row);

switch (atoi(row)) {

case 1:

fprintf (stdout, "data: ");

retValue = readData (stdin, &d);

if (retValue==0) {

fprintf (stderr, "Input erro.\n");

exit (0);

}

root = insert (root, d);

break;

case 2:

fprintf (stdout, "data: ");

retValue = readData (stdin, &d);

if (retValue==0) {

fprintf (stderr, "Input erro.\n");

exit (0);

}

root = delete (root, d);

break;

case 3:

fprintf (stdout, "data: ");

retValue = readData (stdin, &d);

if (retValue==0) {

fprintf (stderr, "Input error.\n");

exit (0);

}

tmp = searchI (root, d);

if (tmp != NULL) {

fprintf (stdout, "Iterative Search: Found -> ");

writeData (stdout, getData(tmp));

} else {

fprintf (stdout, "Iterative Search: Not Found\n");

}

tmp = searchR (root, d);

if (tmp != NULL) {

fprintf (stdout, "Recursive Search: Found -> ");

writeData (stdout, getData(tmp));

} else {

fprintf (stdout, "Recursive Search: Not Found\n");

}

break;

case 4:

writeTree (stdout, root, INORDER);

break;

case 5:

fprintf (stdout, "File Name: ");

scanf ("%s", row);

if (strcmp (row, "stdout") == 0) {

fp = stdout;

} else {

fp = fopen (row, "w");

}

if (fp == NULL) {

fprintf (stderr, "Error Opening File %s\n", row);

break;

}

writeTree (fp, root, PREORDER);

if (strcmp (row, "stdout") != 0) {

fclose(fp);

}

break;

case 6:

fprintf (stdout, "File Name: ");

scanf ("%s", row);

fp = fopen (row, "r");

if (fp == NULL) {

fprintf (stderr, "Error Opening File %s\n", row);

} else {

freeTree (root);

root = readTree (fp);

}

break;

case 7:

fprintf (stdout, "Tree minimum iterative: ");

tmp = treeMinI (root);

writeData (stdout, getData(tmp));

fprintf (stdout, "Tree minimum recursive: ");

tmp = treeMinR (root);

writeData (stdout, getData(tmp));

fprintf (stdout, "Tree maximum iterative: ");

tmp = treeMaxI (root);

writeData (stdout, getData(tmp));

fprintf (stdout, "Tree maximum recursive: ");

tmp = treeMaxR (root);

writeData (stdout, getData(tmp));

break;

case 8:

end = 1;

break;

default:

fprintf (stderr, "Unknown Option.\n");

break;

}

}

freeTree(root);

return (OK);

}

###

### treePublic.h

###

#ifndef \_TREE\_PUBLIC\_INCLUDED

#define \_TREE\_PUBLIC\_INCLUDED

#include "data.h"

#define PREORDER -1

#define INORDER 0

#define POSTORDER 1

typedef struct node node\_t;

data\_t getData (node\_t \*);

node\_t \*createEmptyTree ();

node\_t \*readTree(FILE \*);

node\_t \*searchI (node\_t \*, data\_t);

node\_t \*searchR (node\_t \*, data\_t);

node\_t \*treeMinI (node\_t \*);

node\_t \*treeMinR (node\_t \*);

node\_t \*treeMaxI (node\_t \*);

node\_t \*treeMaxR (node\_t \*);

node\_t \*insert(node\_t \*, data\_t);

node\_t \*delete(node\_t \*, data\_t);

void writeTree(FILE \*, node\_t \*, int);

void freeTree(node\_t \*);

#endif

###

### treePrivate.h

###

#ifndef \_TREE\_PRIVATE\_INCLUDED

#define \_TREE\_PRIVATEINCLUDED

#include "treePublic.h"

struct node {

data\_t val;

struct node \*left;

struct node \*right;

};

#endif

###

### tree.c

###

#include "treePrivate.h"

#define FIND 0

static node\_t \*myAlloc (void);

#if FIND

static data\_t findDeleteMax1 (node\_t \*\*);

#endif

#if !FIND

static node\_t \*findDeleteMax2 (data\_t \*, node\_t \*);

#endif

data\_t

getData (

node\_t \*node

)

{

return (node->val);

}

node\_t \*

createEmptyTree (

void

)

{

return (NULL);

}

node\_t \*

treeMinI (

node\_t \*rp

)

{

if (rp == NULL)

return (rp);

while (rp->left != NULL) {

rp = rp->left;

}

return (rp);

}

node\_t \*

treeMinR (

node\_t \*rp

)

{

if (rp == NULL || rp->left==NULL)

return (rp);

return (treeMinR (rp->left));

}

node\_t \*

treeMaxI (

node\_t \*rp

)

{

if (rp == NULL)

return (rp);

while (rp->right != NULL) {

rp = rp->right;

}

return (rp);

}

node\_t \*

treeMaxR (

node\_t \*rp

)

{

if (rp == NULL || rp->right==NULL)

return (rp);

return (treeMaxR (rp->right));

}

node\_t \*

searchI (

node\_t \*rp,

data\_t data

)

{

while (rp != NULL) {

if (compare(rp->val, data) == 0)

return (rp);

if (compare(data, rp->val) < 0)

rp = rp->left;

else

rp = rp->right;

}

return (NULL);

}

node\_t \*

searchR (

node\_t \*rp,

data\_t data

)

{

if (rp==NULL || compare(rp->val, data)==0)

return (rp);

if (compare(data, rp->val) < 0)

return (searchR (rp->left, data));

else

return (searchR (rp->right, data));

}

node\_t \*

insert (

node\_t \*rp,

data\_t data

)

{

node\_t \*p;

/\* Empty Tree: Found Position \*/

if (rp == NULL) {

p = myAlloc();

p->val = data;

p->left = p->right = NULL;

return (p);

}

/\* Duplicated Element \*/

if (compare(data, rp->val) == 0) {

return (rp);

}

if (compare(data, rp->val) < 0) {

/\* Insert on the left \*/

rp->left = insert (rp->left, data);

} else {

/\* Insert on the right \*/

rp->right = insert (rp->right, data);

}

return (rp);

}

node\_t \*

readTree (

FILE \*fp

)

{

node\_t \*rp;

data\_t d;

rp = createEmptyTree ();

while (readData (fp, &d) != EOF) {

rp = insert (rp, d);

}

return (rp);

}

void

freeTree (

node\_t \*rp

)

{

if (rp == NULL) {

return;

}

freeTree (rp->left);

freeTree (rp->right);

free (rp);

return;

}

void

writeTree (

FILE \*fp,

node\_t \*rp,

int modo

)

{

if (rp == NULL) {

return;

}

if (modo == PREORDER) {

writeData (fp, rp->val);

}

writeTree (fp, rp->left, modo);

if (modo == INORDER) {

writeData(fp, rp->val);

}

writeTree (fp, rp->right, modo);

if (modo == POSTORDER) {

writeData(fp, rp->val);

}

return;

}

node\_t \*

delete (

node\_t \*rp,

data\_t data

)

{

node\_t \*p;

/\* Empty Tree \*/

if (rp == NULL) {

printf("Error: Unknown Data\n");

return (rp);

}

if (compare (data, rp->val) < 0) {

/\* Delete on the left sub-treee Recursively \*/

rp->left = delete (rp->left, data);

return (rp);

}

if (compare(data, rp->val)> 0) {

/\* Delete on the rigth sub-treee Recursively \*/

rp->right = delete (rp->right, data);

return (rp);

}

/\* Delete Current Note rp \*/

p = rp;

if (rp->right == NULL) {

/\* Empty Right Sub-Tree: Return Left Sub-Tree \*/

rp = rp->left;

free (p);

return (rp);

}

if (rp->left == NULL) {

/\* Empty Left Sub-Tree: Return Right Sub-Tree \*/

rp = rp->right;

free (p);

return rp;

}

/\* Find Predecessor and Substitute \*/

#if FIND

rp->val = findDeleteMax1 (&(rp->left));

#endif

#if !FIND

{

data\_t val;

rp->left = findDeleteMax2 (&val, rp->left);

rp->val = val;

}

#endif

return (rp);

}

static node\_t \*

myAlloc (

void

)

{

node\_t \*p;

p = (node\_t \*)malloc(sizeof(node\_t));

if (p == NULL) {

printf ("Allocation Error.\n");

exit (1);

}

return (p);

}

#if FIND

static data\_t

findDeleteMax1 (

node\_t \*\*rpp

)

{

node\_t \*p;

data\_t d;

/\* Find The Rigth-Most Node (max value) \*/

while ((\*rpp)->right != NULL)

rpp = &((\*rpp)->right);

p = \*rpp;

d = p->val;

\*rpp = (\*rpp)->left;

free (p);

return (d);

}

#endif

#if !FIND

static node\_t \*

findDeleteMax2 (

data\_t \*d,

node\_t \*rp

)

{

node\_t \*tmp;

if (rp->right == NULL) {

\*d = rp->val;

tmp = rp->left;

free (rp);

return (tmp);

}

rp->right = findDeleteMax2 (d, rp->right);

return (rp);

}

#endif

dataInt.h

#ifndef \_DATA\_INCLUDED

#define \_DATA\_INCLUDED

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAXC 20

typedef int data\_t;

int readData (FILE \*, data\_t \*);

void writeData (FILE \*, data\_t);

int compare (data\_t, data\_t);

#endif

dataInt.c

#include "data.h"

int

readData (

FILE \*fp,

data\_t \*data

)

{

int retValue;

retValue = fscanf (fp, "%d", data);

return (retValue);

}

void

writeData (

FILE \*fp,

data\_t data

)

{

fprintf(fp, "%d\n", data);

return;

}

int

compare (

data\_t d1,

data\_t d2

)

{

if (d1 < d2) {

return (-1);

} else {

if (d1 == d2) {

return (0);

} else {

return (1);

}

}

}

dataStr.h

#ifndef \_DATA\_INCLUDED

#define \_DATA\_INCLUDED

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#define MAXC 60

typedef char \*data\_t;

int readData (FILE \*, data\_t \*);

void writeData (FILE \*, data\_t);

int compare (data\_t, data\_t);

#endif

dataStr.c

#include "data.h"

int

readData (

FILE \*fp,

data\_t \*data

)

{

char \*tmpData, row[MAXC];

int retValue;

retValue = fscanf (fp, "%s", row);

tmpData = (char \*) malloc ((strlen(row)+1) \* sizeof (char));

if (tmpData == NULL) {

fprintf (stderr, "Allocation Error.\n");

exit (1);

}

/\* Get Rid of New-Line \*/

strcpy (tmpData, row);

\*data = (data\_t) tmpData;

return (retValue);

}

void

writeData (

FILE \*fp,

data\_t data

)

{

fprintf(fp, "%s\n", data);

return;

}

int

compare (

data\_t d1,

data\_t d2

)

{

return (strcmp(d1, d2));

}