CSE216 Programming Abstractions

C Structure (Binary Search Tree)

YoungMin Kwon

Binary Search Tree

- We will implement Binary Search Tree in C
- Download the following files and implement TODOs
  - reci_bintree.h, reci_bintree.c, reci_bintree_test.c
- To compile
  - gcc reci_bintree.c reci_bintree_test.c

```c
/* reci_bintree.h
*/
//offset of m in st
#define offsetof(st, m)          ( (size_t) &(((st *)0)->m) )
//container address when the address of m in st is ptr
#define containerof(ptr, st, m) ((st *) (((char*)(ptr)) - offsetof(st, m)))

typedef struct node node_t;
struct node {
    node_t *left, *right;
};
typedef struct bintree bintree_t;
struct bintree {
    node_t *root;
    void    (*add)       (bintree_t *tree, node_t *n,
                          int (*comp)(node_t *a, node_t *b));
    node_t* (*find)      (bintree_t *tree, void *data,
                          int (*comp)(node_t *a, void* data));
    void    (*preorder)  (bintree_t *tree, void (*visit)(node_t *a));
    void    (*inorder)   (bintree_t *tree, void (*visit)(node_t *a));
    void    (*postorder) (bintree_t *tree, void (*visit)(node_t *a));
};

extern bintree_t *make_bintree();
extern void free_bintree(bintree_t *tree);
```

```
/* reci_bintree.c
*/
//forward definitions
static void tree_add      (bintree_t *tree, node_t *n,
                           int (*comp)(node_t *a, node_t *b));
static node_t* tree_find  (bintree_t *tree, void *data,
                           int (*comp)(node_t *a, void *data));
static void tree_preorder (bintree_t *tree, void (*visit)(node_t *a));
static void tree_inorder  (bintree_t *tree, void (*visit)(node_t *a));
static void tree_postorder(bintree_t *tree, void (*visit)(node_t *a));

/***************************************************
 * make a bin_tree rooted at root
*/
bintree_t *make_bintree() {
    //TODO: allocate memory for tree
    bintree_t *tree =
    //TODO: copy function pointers to tree
    tree->root      = NULL;
    tree->add       =
    tree->find      =
    tree->preorder  =
    tree->inorder   =
    tree->postorder =
    return tree;
}
```

```c
/***************************************************
 * make a bin_tree rooted at root
 * comp(a, b) returns 1 if a > b; 0 if a == b; -1 if a < b
*/
bintree_t *make_bintree() {
    //TODO: allocate memory for tree
    bintree_t *tree = malloc(sizeof(bintree_t));

    //TODO: copy function pointers to tree
    tree->root      = NULL;
    tree->add       = tree_add;
    tree->find      = tree_find;
    tree->preorder  = tree_preorder;
    tree->inorder   = tree_inorder;
    tree->postorder = tree_postorder;
    return tree;
}
```

```
/***************************************************
 * add n to the subtree rooted at root
 * comp(a, b) returns 1 if a > b; 0 if a == b; -1 if a < b
*/
static void add(node_t *root, node_t *n,
                int (*comp)(node_t *a, node_t *b)) {
    //TODO: implement add
}

static void tree_add(bintree_t *tree, node_t *n,
                     int (*comp)(node_t *a, node_t *b)) {
    if(tree->root == NULL)
        tree->root = n;
    else
        add(tree->root, n, comp);
}
```

```
/****************************************************
 * add n to the subtree rooted at root
 * comp(a, b) returns 1 if a > b; 0 if a == b; -1 if a < b
*/
static void add(node_t *root, node_t *n,
                int (*comp)(node_t *a, node_t *b)) {
    //TODO: implement add
    if (comp(root, n) >=0) {
        if(root->left != NULL) {
            add(root->left, n, comp);
        }
        else {
            root->left = n;
        }
    }
    else {
        if(root->right != NULL) {
            add(root->right, n, comp);
        }
        else {
            root->right = n;
        }
    }
}
```

```c
/***************************************************
 * find a node from the subtree rooted at root
 * comp(a, data) returns 1 if a > data; 0 if a == data; -1 if a < data
*/
static node_t* find(node_t *root, void *data,
                    int (*comp)(node_t *a, void* data)) {
    if(root == NULL)
        return NULL;
    int c = comp(root, data);
    //TODO implement find
}

static node_t* tree_find(bintree_t *tree, void *data,
                         int (*comp)(node_t *a, void *data)) {
    return find(tree->root, data, comp);
}
```

```
/***************************************************
 * find a node from the subtree rooted at root
 * comp(a, data) returns 1 if a > data; 0 if a == data; -1 if a < data
*/
static node_t* find(node_t *root, void *data,
                    int (*comp)(node_t *a, void* data)) {
    if(root == NULL)
        return NULL;
    int c = comp(root, data);
    //TODO implement find
    return  c > 0 ? find(root->left,  data, comp)
        :   c < 0 ? find(root->right, data, comp)
        :   root
        ;
}
```

```c
/***************************************************
 * preorder traverse the subtree rooted at root
 * call visit when visiting a node
*/
static void preorder(node_t *root, void (*visit)(node_t *a)) {
    //TODO: implement preorder
}

/***************************************************
 * inorder traverse the subtree rooted at root
 * call visit when visiting a node
*/
static void inorder(node_t *root, void (*visit)(node_t *a)) {
    //TODO: implement order
}

/***************************************************
 * postorder traverse the subtree rooted at root
 * call visit when visiting a node
*/
static void postorder(node_t *root, void (*visit)(node_t *a)) {
    //TODO: implement postorder
}
```

```
static void preorder(node_t *root, void (*visit)(node_t *a)) {
    //TODO: implement preorder
    visit(root);
    if(root->left != NULL)
        preorder(root->left, visit);
    if(root->right != NULL)
        preorder(root->right, visit);
}

static void inorder(node_t *root, void (*visit)(node_t *a)) {
    //TODO: implement order
    if(root->left != NULL)
        inorder(root->left, visit);
    visit(root);
    if(root->right != NULL)
        inorder(root->right, visit);
}

static void postorder(node_t *root, void (*visit)(node_t *a)) {
    //TODO: implement postorder
    if(root->left != NULL)
        postorder(root->left, visit);
    if(root->right != NULL)
        postorder(root->right, visit);
    visit(root);
}
```

```
/* reci_bintree_test.c
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "reci_bintree.h"

/*****************************************************
 * struct word
*/
typedef struct word word_t;
struct word {
    char *str;
    node_t node;
};
word_t* make_word(char *str) {
    word_t *word = malloc(sizeof(word_t));
    word->str = strdup(str); //strdup will allocate memory in heap
    word->node.left = word->node.right = NULL;
}
```

```
/***************************************************
 * build a binary tree of words
*/
int comp(node_t *a, node_t *b) {
    //TODO: get the container of a and b and
    //      compare their str using strcmp
}

bintree_t *build(char **words) {
    bintree_t *tree = make_bintree();
    for(int i = 0; words[i] != NULL; i++) {
        word_t *word = make_word(words[i]);
        tree->add(tree, &word->node, comp);
    }
    return tree;
}
```

```c
int comp(node_t *a, node_t *b) {
    //TODO: get the container of a and b and
    //      compare their str using strcmp
    word_t *wa = containerof(a, word_t, node);
    word_t *wb = containerof(b, word_t, node);
    return strcmp(wa->str, wb->str);
}
```

```c
/***************************************************
 * find word from tree
*/
int comp_data(node_t *a, void *data) {
    //TODO: get the container of a and cast data to char*
    //      compare str and data using strcmp
}

void find(bintree_t *tree, char *str) {
    node_t *node = tree->find(tree, str, comp_data);
    word_t *word = containerof(node, word_t, node);
    printf("** find moe: %s\n", word->str);
}
```

```c
int comp_data(node_t *a, void *data) {
    //TODO: get the container of a and cast data to char*
    //      compare str and data using strcmp
    word_t *wa = containerof(a, word_t, node);
    char *str = (char*) data;
    return strcmp(wa->str, str);
}
```

```
/***************************************************
 * sort
*/
void visit(node_t *node) {
    word_t *word = containerof(node, word_t, node);
    printf("%s\n", word->str);
}
void print_sorted(bintree_t *tree) {
    printf("** sorted\n");
    //TODO: inorder traverse tree with visit
}

/***************************************************
 * destroy tree
*/
void visit_free(node_t *node) {
    word_t *word = containerof(node, word_t, node);
    free(word->str);
    free(word);
}
void free_tree(bintree_t *tree) {
    //TODO: postorder traverse tree with visit_free

    //TODO: free tree using free_bintree
}
```

```c
void print_sorted(bintree_t *tree) {
    printf("** sorted\n");
    //TODO: inorder traverse tree with visit
    tree->inorder(tree, visit);
}

void free_tree(bintree_t *tree) {
    //TODO: postorder traverse tree with visit_free
    tree->postorder(tree, visit_free);

    //TODO: free tree using free_bintree
    free_bintree(tree);
}
```

Expected result

```
** find moe: moe
** sorted
Catch
Eeny
Eeny
If
a
by
go
he
him
hollers
let
meeny
meeny
miny
miny
moe
moe
the
tiger
toe
```