Wesleyan University, Fall 2023, COMP 211

Homework 10: Graphs, Trees, and Searching

Due by 11:59pm on December 7, 2023

---

1. Written problems (10 points)

Problem 1. *Consider the pseudocode in Program 1. Describe the state of the execution environment right before line 32 is executed and then describe the changes to the environment after line 32 executes. Also give the maximum number of binding tables on the stack as a result of executing line 32. Note that you do not need to describe what happens when the* `insert` *function is called (and the code is not given): you should assume this function inserts keys into the binary search tree and corresponds to the function you implemented in lab.*

Solution: The maximum number of binding tables that are ever on the stack is 5, 3 of which belong to `inorder` function calls, 1 of which belongs to `main`, and 1 of which belongs to `printf`. . The execution semantics of the program are as follows.

(1) After line 31 has executed but before line 32 has executed, there is only one binding table, S^m, on the stack for the main function. Note that a is the address of the array in memory holding the values 7, 2, 13, and 5; b_7 is the address of the bnode in memory holding key value of 7; b_2 is the address of the bnode holding key value of 2, b_{13} is the address of the bnode in memory holding key key value of 13, and b_5 is the address of the bnode in memory holding the key value of 5.

$$S^m = \{t \mapsto b_7, A \mapsto a\}$$

(2) At line 32, the `inorder` function is called. To evaluate the argument $t->root$ being passed to `inorder`, the value of t is obtained from the binding table, b_t, which is the address of the btree struct that was created. Then this address t is dereferenced, the memory found at that address is interpreted as a `btree` struct, and the value of the `root` field (i.e., b_7 which is the address of the bnode containing 7) is passed to the `inorder` function as an argument. Thus a new binding table is pushed on the stack, containing the function parameter n bound to b_7.

$S^m = \{t \mapsto b_t, A \mapsto a\}; S^7 = \{n \mapsto b_7\}$
$S^m = \{t \mapsto b_t, A \mapsto a\}; S^7 = \{n \mapsto b_7\}; S^2 = \{n \mapsto b_2\}$
$S^m = \{t \mapsto b_t, A \mapsto a\}; S^7 = \{n \mapsto b_7\}; S^2 = \{n \mapsto b_2\}; S^N = \{n \mapsto NULL\}$
$S^m = \{t \mapsto b_t, A \mapsto a\}; S^7 = \{n \mapsto b_7\}; S^2 = \{n \mapsto b_2\}$
$S^m = \{t \mapsto b_t, A \mapsto a\}; S^7 = \{n \mapsto b_7\}; S^2 = \{n \mapsto b_2\}; S^p = \{int \mapsto 2\}$
$S^m = \{t \mapsto b_t, A \mapsto a\}; S^7 = \{n \mapsto b_7\}; S^2 = \{n \mapsto b_2\}$
$S^m = \{t \mapsto b_t, A \mapsto a\}; S^7 = \{n \mapsto b_7\}; S^2 = \{n \mapsto b_2\}; S^5 = \{n \mapsto b_5\}$
$S^m = \{t \mapsto b_t, A \mapsto a\}; S^7 = \{n \mapsto b_7\}; S^2 = \{n \mapsto b_2\}; S^5 = \{n \mapsto b_5\}; S^N = \{n \mapsto NULL\}$
$S^m = \{t \mapsto b_t, A \mapsto a\}; S^7 = \{n \mapsto b_7\}; S^2 = \{n \mapsto b_2\}; S^5 = \{n \mapsto b_5\}$
$S^m = \{t \mapsto b_t, A \mapsto a\}; S^7 = \{n \mapsto b_7\}; S^2 = \{n \mapsto b_2\}; S^5 = \{n \mapsto b_5\}; S^p = \{int \mapsto 5\}$

```
1  typedef struct bnode {
2        int key;
3        struct bnode *left;
4        struct bnode *right;
5  } bnode;
6
7  typedef struct btree {
8        bnode *root;
9  } btree;
10
11 btree* empty() {
12       btree *t = malloc(sizeof(btree));
13       t->root = NULL;
14       return t;
15  }
16
17 void inorder(bnode* n) {
18      if (n == NULL) return;
19      inorder(n->left);
20      printf("%d ", n->key);
21      inorder(n->right);
22 }
23
24 void insert(btree *t, int key) {
25     ...
26 }
27
28 int main(void) {
29      btree* t = empty();
30      int A[] = {7, 2, 13, 5};
31      for (int i=0; i<4; i++) insert(t, A[i]);
32      inorder(t->root);
33      return 0;
34 }
```

PROGRAM 1. A program that builds a simple binary search tree and then prints it using inorder traversal.

$S^m = \{t \mapsto b_t, A \mapsto a\}; S^7 = \{n \mapsto b_7\}; S^2 = \{n \mapsto b_2\}; S^5 = \{n \mapsto b_5\}$
$S^m = \{t \mapsto b_t, A \mapsto a\}; S^7 = \{n \mapsto b_7\}; S^2 = \{n \mapsto b_2\}; S^5 = \{n \mapsto b_5\}; S^N = \{n \mapsto NULL\}$
$S^m = \{t \mapsto b_t, A \mapsto a\}; S^7 = \{n \mapsto b_7\}; S^2 = \{n \mapsto b_2\}; S^5 = \{n \mapsto b_5\}$
$S^m = \{t \mapsto b_t, A \mapsto a\}; S^7 = \{n \mapsto b_7\}; S^2 = \{n \mapsto b_2\}$
$S^m = \{t \mapsto b_t, A \mapsto a\}; S^7 = \{n \mapsto b_7\};$
$S^m = \{t \mapsto b_t, A \mapsto a\}; S^7 = \{n \mapsto b_7\}; S^p = \{int \mapsto 7\}$
$S^m = \{t \mapsto b_t, A \mapsto a\}; S^7 = \{n \mapsto b_7\};$
$S^m = \{t \mapsto b_t, A \mapsto a\}; S^7 = \{n \mapsto b_7\}; S^{13} = \{n \mapsto b_{13}\}$
$S^m = \{t \mapsto b_t, A \mapsto a\}; S^7 = \{n \mapsto b_7\}; S^{13} = \{n \mapsto b_{13}\}; S^N = \{n \mapsto NULL\}$

$S^m = \{t \mapsto b_t, A \mapsto a\}; S^7 = \{n \mapsto b_7\}; S^{13} = \{n \mapsto b_{13}\}$
$S^m = \{t \mapsto b_t, A \mapsto a\}; S^7 = \{n \mapsto b_7\}; S^{13} = \{n \mapsto b_{13}\}; S^p = \{int \mapsto 13\}$
$S^m = \{t \mapsto b_t, A \mapsto a\}; S^7 = \{n \mapsto b_7\}; S^{13} = \{n \mapsto b_{13}\}$
$S^m = \{t \mapsto b_t, A \mapsto a\}; S^7 = \{n \mapsto b_7\}; S^{13} = \{n \mapsto b_{13}\}; S^N = \{n \mapsto NULL\}$
$S^m = \{t \mapsto b_t, A \mapsto a\}; S^7 = \{n \mapsto b_7\}; S^{13} = \{n \mapsto b_{13}\}$
$S^m = \{t \mapsto b_t, A \mapsto a\}; S^7 = \{n \mapsto b_7\}$
$S^m = \{t \mapsto b_t, A \mapsto a\}$

2. Programming problems (10 points)

Problem 2. *In this problem, you will use an adjacency list representation to implement a graph data structure and then implement various functions to perform operations on a graph. You should assume an undirected graph.*

Recall that in the adjacency list representation of a graph that an array of lists is used, one list for each node of the graph where the lists contains the neighbors of the associated node. More specifically, the following struct and create function have been provided in `hw10.c` *and* `hw10.h`*, along with a function to print the adjacency lists of the graph.*

```
typedef struct graph {
      list *A; // array of lists
      int N; // number of nodes
} graph;

graph* graph_create(int N)
{
      graph *g = malloc(sizeof(graph));
      g->A = malloc(sizeof(list*) * N);
      g->N = N;

      for (int i=0; i<N; i++) {
          g->A[i] = list_create();
      }
      return g;
  }
```

What you will implement are the following functions. The functions are specified in `hw10.h`*, and you will implement them in* `hw10.c`*.*

- `graph_add` *function: add an edge to the graph. Since the graph is undirected, for an edge* (x, y) *you will want to be sure to add* x *to* y*'s adjacency list and* y *to* x*'s adjacency list.*

- `graph_del` *function: remove an edge from the graph.*

- `graph_dfs` *function: perform an iterative depth first search (DFS) on a graph starting from a given node. Note that your implementation must be iterative not recursive. A review of DFS is provided for you below.*

- `graph_connected` *function: use* `graph_dfs` *to check whether graph is connected. More specifically, if every node is visited during an execution of* `graph_dfs` *then, assuming a correct implementation, the graph is connected.*

One of the goals of this last homework is to tie together a number of different concepts you have seen throughout the course. In particular, to implement the adjacency list representation of a graph, you will need to use a list abstract data type, and to perform the iterative depth-first search on the graph, you will need to use a stack abstract data type. Both a list implementation and a stack implementation have been provided for you, see `list.c` *and* `stack.c`*. But now you will be a user of the list and stack abstract data types, rather than implementing them yourselves.*

Iterative depth-first search. *From Sedgewick's Algorithms in C book: "To visit a vertex, we mark it as having been visited, then (recursively) visit all the vertices that are adjacent to it and that have not yet been marked." This gives rise to the following recursive implementation of depth-first search, where the* `visited` *array is used to keep track of which nodes have been visited and* `idx` *is the current node from which the search is being performed. Initially, during the first call to* `graph_dfs_r`*, this is the starting node of the search which without loss of generality can be node 0.*

```
void graph_dfs_r(graph *g, int* visited, int idx)
{
    visited[idx] = 1;
    for (node *nbr=g->A[idx].head->next; nbr!=NULL; nbr=nbr->next) {
        if (visited[nbr->data] == 0) {
            graph_dfs_r(g, visited, nbr->data);
        }
    }
}
```

Because you will be implementing an iterative version of depth-first search, you will not be able to take advantage of the implicit recursion stack, and will instead need to use an explicit stack. Specifically, you will want to push and pop integers, in the form of node indices, on and off the stack abstract data type provided for you.

3. Code distribution and submission

As usual, the code distribution contains a driver program (`make driver`) and a tests program (`make tests`). Figure 1 shows a sample session using the driver. You will want to add more tests to the tests program.

Submit your written work as `hw10.pdf` and your code as `hw10.c` to the Google Drive directory I have created for you named `comp211-f23-USERNAME/hw10/`. You should replace `USERNAME` with your Wesleyan username.

Do not forget that your written work must be submitted as a PDF! And make sure that at the top of each file you have put your name! Do not, however, change the names of the files.

```
$ ./driver
(0) Exit
(1) Create empty graph
(2) Add edge
(3) Delete edge
(4) Do DFS
(5) Check connected
(6) Print adjacency lists
Enter choice: 1
Enter number of nodes: 2
(0) Exit
(1) Create empty graph
(2) Add edge
(3) Delete edge
(4) Do DFS
(5) Check connected
(6) Print adjacency lists
Enter choice: 2
Enter first node id: 0
Enter second node id: 1
(0) Exit
(1) Create empty graph
(2) Add edge
(3) Delete edge
(4) Do DFS
(5) Check connected
(6) Print adjacency lists
Enter choice: 4
visited[0]: 1
visited[1]: 1
(0) Exit
(1) Create empty graph
(2) Add edge
(3) Delete edge
(4) Do DFS
(5) Check connected
(6) Print adjacency lists
Enter choice: 6
Node 0: 1
Node 1: 0
(0) Exit
(1) Create empty graph
(2) Add edge
(3) Delete edge
(4) Do DFS
(5) Check connected
(6) Print adjacency lists
```

FIGURE 1. A sample session using the driver program.