C Programming: Data structures

Yipeng Huang

Rutgers University

February 15, 2021

Table of contents

Announcements
- Quizzes and programming assignments
- Reading and class session plan

Understanding pass-by-value and pass-by-reference

Why matMul() is written that way

`bstLevelOrder.c`: Level order traversal of a binary search tree
- Level order traversal of a binary search tree
- Binary search tree: `BSTNode`, `insert()`, `delete()`
- Linked list implementation of a queue: `QueueNode`, `Queue`, `enqueue()`, `dequeue()`
- Tying it together in the `main()` function

Quizzes and programming assignments

Short quiz 3

Now out, due Thursday February 17, 11:59 pm.

Programming assignment 2

- Has been out, due Thursday February 24, 11:59 pm.
- More review of CS 111 and 112 concepts in the C language.
- Stacks. Queues. BSTs. Graph algorithms.

Reading and class session plan

Reading: CS:APP Chapter 2

- Chapter 2: Representing and manipulating information
- First focus on Chapters 2.1-2.3, integers

Class session plan

1. Today: Discussion of PA2. Implementing a stack data structure.
2. Thursday: Representing integers in computer architecture.

Table of contents

Announcements
- Quizzes and programming assignments
- Reading and class session plan

Understanding pass-by-value and pass-by-reference

Why matMul() is written that way

`bstLevelOrder.c`: Level order traversal of a binary search tree
- Level order traversal of a binary search tree
- Binary search tree: `BSTNode`, `insert()`, `delete()`
- Linked list implementation of a queue: `QueueNode`, `Queue`, `enqueue()`, `dequeue()`
- Tying it together in the `main()` function

Understanding pass-by-value and pass-by-reference

In this section, we study the `push()` function for a stack.

The `push()` function needs to make changes to the top of the stack, and return pointers to stack elements such that the elements can later be freed from memory.

We consider four function signatures for `push()` that are incorrect.

1. `void push ( char value, struct stack s );`
2. `void push ( char value, struct stack* s );`
3. `struct stack push ( char value, struct stack s );`
4. `struct stack push ( char value, struct stack* s );`

And we consider two function signatures for `push()` that are correct.

5. `void push ( char value, struct stack** s );`
6. `struct stack* push ( char value, struct stack* s );`

Understanding pass-by-value and pass-by-reference

```
1  void push ( char value, struct stack
        s ) { // bug in signature
2
3      struct stack *bracket = malloc(
           sizeof(struct stack));
4      bracket->data = value;
5      bracket->next = &s;
6
7      s = *bracket;
8
9      return;
10 }
```

```
1 int main () {
2     struct stack s;
3     push( 'S', s );
4     printf ("s.data = %c\n", s.data)
          ;
5 }
```

Version 1. An incorrect function signature for `push()`.

This version of `push()` completely passes-by-value and has no effect on `struct stack s` in `main()`, so `s.data` is uninitialized.

Understanding pass-by-value and pass-by-reference

```
1  void push ( char value, struct stack
       * s ) { // bug in signature
2
3      struct stack *bracket = malloc(
           sizeof(struct stack));
4      bracket->data = value;
5      bracket->next = s;
6
7      s = bracket;
8
9      return;
10 }
```

```
1 int main () {
2     struct stack s;
3     push( 'S', &s );
4     push( 'C', &s );
5     // printf ("s = %p\n", s);
6     struct stack* pointer = &s;
7     printf ("pop: %c\n", pop(&
          pointer));
8     printf ("pop: %c\n", pop(&
          pointer));
9 }
```

Version 2. An incorrect function signature for `push()`.

This version of `push()` also has no effect on `struct stack s` in `main()`.

Understanding pass-by-value and pass-by-reference

```
1  struct stack push ( char value,
       struct stack s ) { // bug in
       signature
2
3      struct stack *bracket = malloc(
           sizeof(struct stack));
4      bracket->data = value;
5      bracket->next = &s;
6
7      s = *bracket;
8
9      return s;
10 }
```

Version 3. An incorrect function signature for `push()`.

Here, we try returning an updated stack data structure via the return type of `push()`. Lines 3, 7, and 9 will lead to a memory leak (pointer is lost). Line 5 assigns the `next` pointer to an address `&s` which will be out of scope in `main()`.

Understanding pass-by-value and pass-by-reference

```
1  struct stack push ( char value,
       struct stack* s ) { // bug in
       signature
2
3      struct stack *bracket = malloc(
           sizeof(struct stack));
4      bracket->data = value;
5      bracket->next = s;
6
7      s = bracket;
8
9      return *s;
10 }
```

```
1 int main () {
2     struct stack s;
3     s = push( 'S', &s );
4     printf ("s.data = %c\n", s.data)
          ;
5     s = push( 'C', &s );
6     printf ("s.data = %c\n", s.data)
          ;
7 }
```

Version 4. An incorrect function signature for `push()`.

Here, we again try returning an updated stack data structure via the return type of `push()`. Lines 3, 7, and 9 will still lead to a memory leak (pointer is lost).

Understanding pass-by-value and pass-by-reference

```
1 void push ( char value, struct stack
      ** s ) {
2
3     struct stack *bracket = malloc(
          sizeof(struct stack));
4     bracket->data = value;
5     bracket->next = *s;
6
7     *s = bracket;
8
9     return;
10 }
```

```
1 int main () {
2     struct stack* s;
3     push( 'S', &s );
4     push( 'C', &s );
5     printf ("pop: %c\n", pop(&s));
6     printf ("pop: %c\n", pop(&s));
7 }
```

Version 5. A correct function signature for `push()`.

`struct stack* s` in `main()` updates by passing the `struct stack *` parameter via pass-by-reference, leading to the `push()` signature that you see here. This matches the signature that you see for the `pop()` function.

Understanding pass-by-value and pass-by-reference

```
1  struct stack* push ( char value,
       struct stack* s ) {
2
3      struct stack *bracket = malloc(
           sizeof(struct stack));
4      bracket->data = value;
5      bracket->next = s;
6
7      s = bracket;
8
9      return s;
10 }
```

```
1 int main () {
2     struct stack* s;
3     s = push( 'S', s );
4     s = push( 'C', s );
5     printf ("pop: %c\n", pop(&s));
6     printf ("pop: %c\n", pop(&s));
7 }
```

Version 6. A correct function signature for `push()`.

`struct stack* s` updates via the return type of `push()` in `main()`, lines 3 and 4. Side note, this is similar to the function signature `BSTNode* insert (BSTNode* root, int key)` shown in class on 2/4. Side note, `pop()` needs to return the character data, so `pop()` cannot have a similar function signature.

Table of contents

Announcements
- Quizzes and programming assignments
- Reading and class session plan

Understanding pass-by-value and pass-by-reference

Why matMul() is written that way

`bstLevelOrder.c`: Level order traversal of a binary search tree
- Level order traversal of a binary search tree
- Binary search tree: `BSTNode`, `insert()`, `delete()`
- Linked list implementation of a queue: `QueueNode`, `Queue`, `enqueue()`, `dequeue()`
- Tying it together in the `main()` function

Why matMul() is written that way

The matMul function signature in the provided example code.

```
1 void matMul (
2      unsigned int l,
3      unsigned int m,
4      unsigned int n,
5      int** matrix_a,
6      int** matrix_b,
7      int** matMulProduct
8 );
```

A more "natural" function signature with return. How to implement?

```
1 int** matMul (
2      unsigned int l,
3      unsigned int m,
4      unsigned int n,
5      int** matrix_a,
6      int** matrix_b
7 );
```

Why matMul() is written that way

The matMul function signature in the provided example code. Caller of matMul allocates memory.

```
1 void matMul (
2     unsigned int l,
3     unsigned int m,
4     unsigned int n,
5     int** matrix_a,
6     int** matrix_b,
7     int** matMulProduct
8 );
```

Suppose we want matMul() to be in charge of allocating memory. How to implement?

```
1 void matMul (
2     unsigned int l,
3     unsigned int m,
4     unsigned int n,
5     int** matrix_a,
6     int** matrix_b,
7     int*** matMulProduct
8 );
```

Table of contents

Announcements
- Quizzes and programming assignments
- Reading and class session plan

Understanding pass-by-value and pass-by-reference

Why matMul() is written that way

`bstLevelOrder.c`: Level order traversal of a binary search tree
- Level order traversal of a binary search tree
- Binary search tree: `BSTNode`, `insert()`, `delete()`
- Linked list implementation of a queue: `QueueNode`, `Queue`, `enqueue()`, `dequeue()`
- Tying it together in the `main()` function

Binary search tree

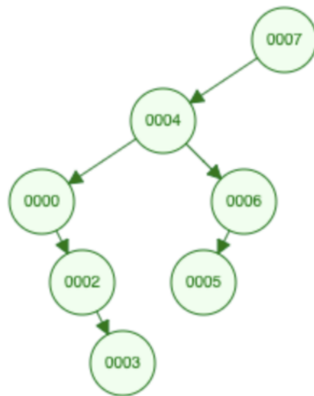

Figure: BST with input sequence 7, 4, 7, 0, 6, 5, 2, 3. Duplicates ignored.

Binary search tree level order traversal

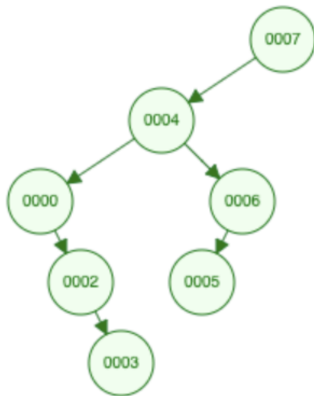

Figure: Level order, left-to-right traversal would return 7, 4, 0, 6, 2, 5, 3.

Binary search tree traversal orders

Breadth-first

- For example: level-order.
- Needs a queue (first in first out).
- Today in class we will build a BST and a Queue.

Depth-first

- For example: in-order traversal, reverse-order traversal.
- Needs a stack (first in last out).
- DEEP question: where is the stack in your recursive implementation in `bstReverseOrder.c`?

typedef

Why types are important

- Natural language has nouns, verbs, adjectives, adverbs.
- Type safety.
- Interpretation vs. compilation.

struct

arrays vs structs

- Arrays group data of the same type. The `[]` operator accesses array elements.
- Structs group data of different type. The `.` operator accesses struct elements.

These are equivalent; the latter is shorthand:

```
BSTNode* root;
```

- `(*root).key = key;`
- `root->key = key;`

When structs are passed to functions, they are passed BY VALUE.

BSTNode

```
typedef struct BSTNode BSTNode;
struct BSTNode {
    int key;
    BSTNode* l_child; // nodes with smaller key will be in left s
    BSTNode* r_child; // nodes with larger key will be in right s
};
```

Let's implement `insert()` and `delete()`

- Recursive implementations for `insert()` and `delete()`.
- Note the matching `malloc()` in `insert()` and `free()` in `delete()`.
- Tricky part: knowing what to pass as parameters and to return.
- Think: where should the data live, and how long should it persist?

QueueNode, Queue

```
// queue needed for level order traversal
typedef struct QueueNode QueueNode;
struct QueueNode {
    BSTNode* data;
    QueueNode* next; // pointer to next node in linked list
};
typedef struct Queue {
    QueueNode* front; // front (head) of the queue
    QueueNode* back; // back (tail) of the queue
} Queue;
```

Let's implement `enqueue()`

`https://visualgo.net/en/queue`

- First, consider if queue is empty.
- Then, consider if queue is not empty. Only need to touch back (tail) of the queue.

Let's implement `dequeue()`

`https://visualgo.net/en/queue`

- First, consider if queue will become empty.
- Then, consider if queue will not not empty. Only need to touch front (head) of the queue.

Subtle point: why are the function signatures (return, parameters) of `enqueue()` and `dequeue()` the way they are?

Tying it together in the `main()` function