Trees in C

Ying Pei Lin

Fall 2024

Binary Tree

Down below is the code to implement a binary tree in C.

First, we build the structure of the binary tree. The tree is constructed by nodes, each node has a left and right child. At the beginning, the root node is set to NULL, which means the tree is empty. When creating a new node, the left and right child are also set to NULL.

```
tree *construct_tree() {
  tree *tr = (tree*)malloc(sizeof(tree));
  tr->root = NULL;
  return tr;
}

node *construct_node(int val) {
  node *nd = (node*)malloc(sizeof(node));
  nd->value =val;
  nd->left = NULL;
  nd->right = NULL;
  return nd;
}
```

To insert a new value to the tree, we first check if the tree is empty. If the tree is empty, we create a new node and set it as the root node. If the tree is not empty, we traverse the tree from the root node to the leaf node. If the value is already in the tree, we do nothing. If the value is less than the current node, we go to the left child. If the value is greater than the current node, we go to the right child. If the left or right child is NULL, we create a new node and set it as the left or right child.

```
void add(tree *tr, int value) {
  // The tree is empty
  if(tr->root == NULL) {
    tr->root = construct_node(value);
```

```
    return;
  }

  node *current = tr->root;
  while (true) {

    // The value is already in the tree
    if(current->value == value) {
      return;
    }

    // The value is less than the current code
    // Go to the left
    else if (value < current->value) {
      if (current->left == NULL) {
        current->left = construct_node(value);
        return;
      } else {
        current = current->left;
      }
    }

    // The value is greater than the current code
    // Go to the right
    else {
      if (current->right == NULL) {
        current->right = construct_node(value);
        return;
      } else {
        current = current->right;
      }
    }
  }
}
```

Originally, my loop structure was like this, which is incorrect.

```
while (true) {

  // The value is already in the tree
  if(current->value == value) {
    return;
  }
```

```
  // Reach the leaf node
  else if(current == NULL) {
    current = construct_node(value);
    return;
  }

  // The value is less than the current code
  else if (value < current->value) {
    current = current->left;
  }

  // The value is greater than the current code
  else {
    current = current->right;
  }
}
```

After spending a lot of time debugging, I realized that the problem was that `current == NULL` doesn't work, because if the pointer is NULL, then it is not going to create the node at the right place. So we should create the new node immediately after we find the NULL child.

To search for a value in the tree, we traverse the tree from the root node

```
bool lookup(tree *tr, int value) {
  node *current = tr->root;
  while (current != NULL) {

    // Found the value
    if (current->value == value) {
      return true;
    }

    // The value is less than the current code
    // Go to the left
    else if (value < current->value) {
      current = current->left;
    }

    // The value is greater than the current code
    // Go to the right
    else {
      current = current->right;
    }
  }
```

```
  return false;
}
```

Depth first traversal

Depth first traversal is a way to traverse the tree by going as far as possible, we discover into the deepest node before backtracking. There are three types of depth first traversal: pre-order, in-order, and post-order. Down below is the code to implement these three types of depth first traversal.

```
void print_pre_ord(node *nd) {
  if (nd != NULL) {
    printf("%d ", nd->value);
    print_pre_ord(nd->left);
    print_pre_ord(nd->right);
  }
}

void print_in_ord(node *nd) {
  if (nd != NULL) {
    print_in_ord(nd->left);
    printf("%d ", nd->value);
    print_in_ord(nd->right);
  }
}

void print_post_ord(node *nd) {
  if (nd != NULL) {
    print_post_ord(nd->left);
    print_post_ord(nd->right);
    printf("%d ", nd->value);
  }
}

void print_tree(tree *tr) {
  if (tr->root != NULL) {
    print_in_ord(tr->root);
    // print_pre_ord(tr->root);
    // print_post_ord(tr->root);
  }
}
```

Explicit stack

To use a explicit stack to implement the in-order traversal, we first create a stack structure, which is implemented by a linked list. The following code shows how to create a stack, push, pop elements from the stack, and check whether the stack is empty.

```
stack *create_stack() {
  stack *stk = (stack *)malloc(sizeof(stack));
  stk->top = NULL;
  return stk;
}

void push(stack *stk, node *tree_node) {
  stack_node *new_node = (stack_node *)malloc(sizeof(stack_node));
  new_node->tree_node = tree_node;
  new_node->next = stk->top;
  stk->top = new_node;
}

node *pop(stack *stk) {
  if (stk->top == NULL) {
    return NULL;
  }
  stack_node *temp = stk->top;
  node *res = temp->tree_node;
  stk->top = stk->top->next;
  free(temp);
  return res;
}

int is_empty(stack *stk) {
  return stk->top == NULL;
}
```

Then we can use the stack to implement the in-order traversal. The following code shows how to use the stack to print the tree in order.

After entering the while loop, we first check if the current node is NULL and the stack is empty, if so, this means we have traversed all the nodes in the tree, the task is done. If the current node is not NULL, we push all the left nodes to the stack.

After, we pop the top node from the stack, print the value of the node, and move to the right child of the node.

```
void print_in_order(tree *tr) {
  stack *stk = create_stack();
```

```
  node *cur = tr->root;

  while (1) {

    // If the current node is NULL and the stack is empty
    if (cur == NULL && is_empty(stk)) {
      break;
    }

    // Push all left nodes to the stack
    while (cur != NULL) {
      push(stk, cur);
      cur = cur->left;
    }

    // Pop the top node from the stack
    cur = pop(stk);
    printf("%d ", cur->value);

    // Move to the right child
    cur = cur->right;
  }

  free(stk);
}
```