Traversal of Binary Trees

1. Introduction

Knowledge of data structures is an essential skill that each programmer must have. And traversal of a binary tree is probably one of the most frequently asked questions on coding interviews. An important thing to note is that a tree can be explored breadth-first and depth-first. To traverse a tree breadth-first a queue is used as a data structure and to traverse a tree depth-first a stack must be used. An example for depth-first traversal is: Inorder, Postorder and Preorder and an example for breadth-first traversal is Level Order.

2. Depth-First Traversal

When traversing a binary tree in depth it can be done recursively and iteratively. This essay will be explaining only the iterative approach. To implement the traversing function a stack must be used as a data structure which follows the LIFO methodology (Last In First Out).

- **Preorder Traversal**

The order of preorder tree traversal is as follows: root, left, right. The following algorithm uses 1 stack to traverse the tree in preorder. The algorithm is of O(n) time complexity and the space complexity is the height of the tree.

```
void preOrder(Node* root) {
  stack<Node*> st;
  st.push(root);

  while(!st.empty()) {
      Node* cur = st.top();
      st.pop();
      cout << cur->data << " ";

      if(cur->right != NULL) {
          st.push(cur->right);
      }

      if(cur->left != NULL) {
          st.push(cur->left);
      }
  }
}
```

We create one empty stack and push the root into it. Until the stack is empty, we pop one element from the top of the stack, we print it, and we push its right child and left child into the stack.

- **Postorder Traversal**

  The order of postorder tree traversal is as follows: left, right, root. The following algorithm uses 2 stacks to traverse the tree in postorder. The algorithm is of O(n) time complexity and the space complexity is the height of the tree.

```
void postOrder(Node* root) {
  stack<Node*> st1;
  stack<Node*> st2;

  st1.push(root);

  while(!st1.empty()) {
      Node* cur = st1.top();

      st1.pop();
      st2.push(cur);

      if(cur->left != NULL) {
          st1.push(cur->left);
      }

      if(cur->right != NULL) {
          st1.push(cur->right);
      }

  }

  while(!st2.empty()) {
      cout << st2.top()->data << " ";
      st2.pop();
  }

}
```

We create two empty stacks, and we push the root into the first stack. Until the first stack is empty, we pop one element from the top of the stack, we push it into the second stack, and we push the left child and the right child of the current element into the first stack. At the end we print the contents of the second stack.

- **Inorder Traversal**

  The order of inorder tree traversal is as follows: left, root, right. The following algorithm uses 1 stack and a pointer to the current element of the stack to traverse the tree in inorder. The algorithm is of O(n) time complexity and the space complexity is the height of the tree.

```
void inOrder(Node *root) {
  stack<Node*> st;
  Node* cur = root;

  while(!st.empty() || cur != NULL) {

      if(cur != NULL) {
          st.push(cur);
          cur = cur->left;
      } else {
          cur = st.top();
          st.pop();
          cout << cur->data << " ";

          cur = cur->right;
      }

  }

}
```

  First, we create an empty stack and a node that points to the root of the tree. Until the stack is empty, and the current element is not null we do the following algorithm:

  1. If the current element is not NULL, we push it into the stack and set the current node to its left child.

  2. If the current element is NULL, we pop one element from the top of the stack, we print it and we set the current element to its right child.

3. **Breadth-First Traversal**

When traversing a tree in breadth it can be done only iteratively. To implement the traversing function a queue must be used as a data structure which follows the FIFO methodology (First In First Out).

- **Level Order Travesal**

This method traverses a tree level by level and prints the elements from each level from left to right. The following algorithm uses 2 queues to traverse the tree by levels. The algorithm is of O(n) time complexity and the space complexity is the height of the tree.

```
void levelOrder(Node* root) {
  queue<Node*> queue1;
  queue<Node*> queue2;

  queue1.push(root);

  while(!queue1.empty()) {
      Node* cur = queue1.front();

      queue1.pop();
      queue2.push(cur);

      if(cur->left != NULL) {
          queue1.push(cur->left);
      }

      if(cur->right != NULL) {
          queue1.push(cur->right);
      }

  }

  while(!queue2.empty()) {
      cout << queue2.front()->data << " ";
      queue2.pop();
  }

}
```

First, we create two empty queues, and we push the root into the first queue. Until the first queue is empty, we pop the front element from the queue, and we push it into the second queue. We also push the left child and the right child of the current element into the first queue. At the end we just print the contents of the second queue.

4. **Detecting a Cycle in a Linked List**

Although it is not on the topic of binary trees, I want to discuss the linked list as a data structure as well. Another very popular question in coding interviews is how to detect a cycle in a linked list. To achieve that we use the so-called method of the hare and the tortoise. The idea behind this method is to have two pointers travelling with different speeds. If there is a cycle within the list then the pointers will meet at some point, if there is not a cycle then they will simply reach the end of the list. This is the algorithm itself:

```
bool has_cycle(SinglyLinkedListNode* head) {
    if(head == NULL) return false;

    SinglyLinkedListNode* slow = head;
    SinglyLinkedListNode* fast = head->next;

    while(fast != NULL && fast->next != NULL && slow != NULL) {

        if(slow == fast) {
            return true;
        }

        slow = head->next;
        fast = head->next->next;
    }

    return true;
}
```

As we can see we have two pointers: one slow and one fast, starting at different locations in the list. The slow pointer advances with one space while the fast pointer advances with two spaces and if they meet at some point then a cycle has occurred. The tricky part is the while condition, we want to do the operations until the fast and the slow pointers are not NULL meaning, they have not reached the end of the list. But because we are jumping two spaces in the fast pointer we also need to check if the next element is also not NULL. The algorithm is of O(n) time complexity.