Final Review
Trees

• How to traverse through a Tree?
  • Recursive orders?
  • Level order? (Use a queue)
• Reconstruct a binary tree from traversal results (*)
• Variation of traversal
  • Count nodes
  • Count edges
  • Count leaf nodes
  • Calculate the height of the tree

• Properties of Trees
  • A tree with $n$ nodes is with $n-1$ edges
  • A full binary tree has $2^{h+1}-1$ nodes (if $h$ starts at 0)
Problem: reconstruct a binary tree

- How’s the **full binary tree** look whose **pre-order** traversal is:
  - **U C N L A G E**

- How’s the **full binary tree** look whose **post-order** traversal is:
  - **U C N L A G E**
Problem: Traverse a Regular Tree

- Given a regular tree node definition as below, write the function `int nodeCount(Node *root)` to count number of nodes in the tree.
  - Consider: recursive traversal or non-recursive traversal?

```c
struct Node
{
    int val;
    vector<Node *> children;
};
```
Solution 1: recursive traversal

```c
int nodeCount(Node *root) {
    if (root -> children.empty()) return 1;  //optional
    int num = 1;
    for (int i = 0; i < root -> children.size(); ++i)
        num += nodeCount(root -> children[i]);
    return num;
}
```
Solution 2: level-order traversal (non-recursive)

```c
int nodeCount(Node *root) {
    queue<Node *> q;
    q.push_back(root);
    int num = 0;
    Node *tmp;
    while (!q->empty()) {
        tmp = q.front();
        q.pop_front();
        ++num;
        for (int i=0; i<tmp->children.size(); ++i)
            q.push_back(tmp->children[i]);
    }
    return num;
}
```
Question: Time complexity of nodeCount?

- Recursive traversal?
- Non-recursive traversal?
Question: Edge count?
Problem: Leaf count?

• Write a function `int leafCount(Node *root)` that counts number of leaf nodes in the tree.
  • How do we decide whether a node is a leaf node?
• How do we modify `nodeCount` into `leafCount`?

```c
struct Node {
    int val;
    vector<Node *> children;
};
```
Solution 1: recursive traversal

```cpp
int leafCount(Node *root) {
    if (root -> children.empty()) return 1;
    int num = 0;
    for (int i = 0; i < root -> children.size(); ++i)
        num += leafCount(root -> children[i]);
    return num;
}
```
Solution 2: level-order traversal (non-recursive)

```c++
int leafCount(Node *root) {
    queue<Node *> q;
    q.push_back(root);
    int num = 0;
    Node *tmp;
    while (!q->empty) {
        tmp = q.front();
        q.pop_front();
        if (tmp -> children.empty()) ++num;
        else {
            for (int i=0; i<tmp->children.size(); ++i)
                q.push_back(tmp->children[i]);
        }
    }
    return num;
}
```
Stack & Queue

• Definition of stack and queue:
  • Stack: last-in-first-out (LIFO)
  • Queue: first-in-first-out (FIFO)

• Problems on stack:
  • Record the paths during DFS (e.g. mazes, routes in trees)
  • Infix, postfix expression

• Problems on queue:
  • Sliding window
Problem: find the least common ancestor in a binary tree

- The **least common ancestor (LCA)** of two nodes $n1$ and $n2$ is the closest ancestor node of $n1$ and $n2$ (and it’s the furthest ancestor from the root).
  - E.g. in the example tree, LCA of node 4 and node 3 is node 1; that of node 5 and node 7 is node 3.

Node* LCA(Node *root, Node *n1, Node *n2)

<table>
<thead>
<tr>
<th>Hint: what’s common on the two paths from the root to n1 or n2? Where does LCA appear on those two routes</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct Node</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>int val;</td>
</tr>
<tr>
<td>Node *left;</td>
</tr>
<tr>
<td>Node *right;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>
Problem: find the least common ancestor in a binary tree

- The paths from the root to $n1$ and $n2$ always have the same prefix. LCA always appear at the last of the prefix.
- We use two stacks, either of them records the path from the root to $n1$ or $n2$. Then we pop the two stacks towards the LCA.
Node *LCA(Node *root, Node *n1, Node *n2) {
    stack<Node *> route1, route2;
    if (!DFS_search(root, n1, route1)) return NULL; //route1: root ==> n1
    if (!DFS_search(root, n2, route2)) return NULL; //route2: root ==> n2
    while (route1.size() > route2.size() ) route1.pop();
    while (route1.size() < route2.size() ) route2.pop();
    while (route1.top() != route2.top() ) {route1.pop(); route2.pop(); }
    return route1.top();
}

bool DFS_search(Node *current, Node *n, stack<Node *> &S) {
    if (current == NULL) return false;
    S.push_back(current);
    if (current == n) return true; //found
    if (DFS_search(current->left, n, S) == true) return true; //found in left
    else if (DFS_search(current->right, n, S)==true) return true; //found in right
    else { //current node is not in the route towards n, pop it and trace back
        S.pop_back();
        return false;
    }
}
Problem: Infix to postfix

- Given infix expressions as below, what are their postfix expressions:
  - $4 + 13 / 5$
  - $3 + 6 * 7 * 8 - 3$
  - $(3 + 5) * (4 + 3 / 2) - 5$
Problem: evaluate a postfix expression

1. Start with the left-most token.
2. If the token is a **number**:
   a. Push it onto the stack
3. If the token is an **operator**:
   a. Pop the top two #s and calculate them, then push the result back
4. Repeat until all tokens are pushed into the stack, then return the top of the stack.
int evalRPN(vector<string>& tokens) {
    stack<int> nums;
    int a;
    for (auto &s:tokens){
        if (s=="+"){
            a=nums.top();nums.pop();
            nums.top()+=a;
        }else if (s=="-"                                                                             
        a=nums.top();nums.pop();
        nums.top()=a;
        }else if (s=="/"                                                                             
        a=nums.top();nums.pop();
        nums.top() /=a;
        }else if (s=="*"                                                                             
        a=nums.top();nums.pop();
        nums.top() *=a;
        }else{
            nums.push(stoi(s));
        }
    }
    return nums.top();
}
Valid brackets

Given a mathematical expression containing parentheses "(, ""), curly braces "{, "}"", and square brackets "[, ""]". Decide if three kinds of brackets are closed validly.

The brackets must close in the correct order, "()" and "()[]{}" are all valid but "[]" and "([])" are not.

All brackets must close.

More examples:

(2 + 4) * 6  valid
[(2 + 4) * {15 - 20}] valid
([{12+30}] not valid
(({{[[<<<_*_>>>]]}}))) valid
(((()((((((())))))))) not valid
Solution

• Use a stack to record brackets.

• Each time read a character from the expression.
  • If it’s any of ‘(’, ‘[’, or ‘{’, push it into the stack.
  • If it’s any of ‘)’, ‘]’, or ‘}’, remove the top from the stack and check if these two closes validly.
  • If it’s any other character, let it pass.

• The expression is valid if the stack is empty in the end, all characters are processed, and no invalid pair of brackets are found.
bool isValid(const string &exp) {
    stack<char> S;
    for (int i=0; i<exp.size(); ++i) {
        char ch = exp[i];
        if (ch == '(' || ch == '{' || ch == '[') S.push_back(ch);
        else if (ch == ')' || ch == '}' || ch == ']') {
            if (S.empty()) return false; // Extra closed bracket
            if (ch == ')') && S.top() == '(' || ch == '}' && S.top() == '{' || ch == ']' && S.top() == '[')
                S.pop_back();
            else return false; // Miss match
        }
    }
    return S.empty(); // return whether no bracket is left in the stack
}
Hash Table

• Given hash function, insert several items to hash table, how will it look?
  • Close addressing? (Separate chaining)
  • Open addressing?
    • Linear Probing
    • Quadratic Probing

• Is a hash function good?
  • Does it always separate keys into different buckets with the same probability?
  • Does it require very little time to calculate the hash value?
Problem: Chaining

```c
int hashFunc(int x)
{
    return (x * 2) % HASH_SIZE;
}

Assume HASH_SIZE = 10. Here is the hash table’s insert function:

void insert(int key)
{
    int index = hashFunc(key);
    hash_array[index].push_back(key);
}

where hash_array is an array of list<int>'s.

insert(7);
insert(1);
insert(23);
insert(14);
insert(19);
insert(53);
insert(37);
insert(83);
```
Inheritance & Polymorphism

• Inheritance:
  • Be careful about orders of construction and destruction
  • class B : A
    • construct B <= construct A then construct B
    • delete B <= destruct B then destruct A

• Polymorphism
  • We have both member functions msg() in A and B
  • A *ptr = new B();
  • If msg() in A is not virtual
    • ptr -> msg() runs the msg() defined in A
  • If msg() in A is virtual
    • ptr -> msg() runs the msg() defined in B
class A
{
    public:
        A() : m_msg("Apple") {}
        A(string msg) : m_msg(msg) {}
        virtual ~A() { message(); } // virtual destructor
        void message() const
        { cout << m_msg << endl; }
    private:
        string m_msg;
};

int main()
{
    A *b1 = new B;
    B *b2 = new B;
    A *b3 = new B("Apple");
    b1->message();
    b2->message();
    b3->message();
    delete b1;
    delete b2;
    delete b3;
}

class B : public A
{
    public:
        B() : A("Orange") {}
        B(string msg) : A(msg), m_a(msg) {}
        void message() const
        { m_a.message(); }
    private:
        A m_a;
};

(a) How many **Apple** and **Orange** do you see in the output?
(b) If we change message() in A into a virtual function: `virtual void message() const;

How many **Apple** and **Orange** do you see in the output?
If `A::message()` is not virtual, you get:

Orange  // b1->message()
Apple    // b2->message()
Apple    // b3->message()
Apple    // destroying m_a of b1
Orange    // destroying b1
Apple    // destroying m_a of b2
Orange    // destroying b2
Apple    // destroying m_a of b3
Apple    // destroying b3

6 Apple’s and 3 Orange’s

If `A::message()` is virtual, you get:

Apple
Apple
Apple
Apple
Apple
Orange
Apple

7 Apple’s and 2 Orange’s
Other topics

- Big-O notation
- Binary Search Tree
- How does each sorting algorithm work?
- Complexity of all the algorithms you have learnt in CS32
### Order of Complexity

<table>
<thead>
<tr>
<th>Big O</th>
<th>Name</th>
<th>$n = 128$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>constant</td>
<td>1</td>
</tr>
<tr>
<td>$O(\log n)$</td>
<td>logarithmic</td>
<td>7</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>linear</td>
<td>128</td>
</tr>
<tr>
<td>$O(n \log n)$</td>
<td>“n log n”</td>
<td>896</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>quadratic</td>
<td>16192</td>
</tr>
<tr>
<td>$O(n^k)$, $k \geq 1$</td>
<td>polynomial</td>
<td></td>
</tr>
<tr>
<td>$O(2^n)$</td>
<td>exponential</td>
<td>$10^{40}$</td>
</tr>
<tr>
<td>$O(n!)$</td>
<td>factorial</td>
<td>$10^{214}$</td>
</tr>
</tbody>
</table>
## Sorting

<table>
<thead>
<tr>
<th>Sorting Algorithm</th>
<th>Time Complexity</th>
<th>Space Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best Case</td>
<td>Average Case</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>$\Omega(N)$</td>
<td>$\Theta(N^2)$</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>$\Omega(N^2)$</td>
<td>$\Theta(N^2)$</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$\Omega(N)$</td>
<td>$\Theta(N^2)$</td>
</tr>
<tr>
<td>Merge Sort</td>
<td>$\Omega(N \log N)$</td>
<td>$\Theta(N \log N)$</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>$\Omega(N \log N)$</td>
<td>$\Theta(N \log N)$</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>$\Omega(N \log N)$</td>
<td>$\Theta(N \log N)$</td>
</tr>
</tbody>
</table>
Good Luck!