Optiver - UK

2022 Sep10

Question 1

Write a function `DaysBetween` which returns an integer representing the number of days between two dates.

Each date is represented by 3 integers : year, month (1-12) and day (1-31). The first date is guaranteed to occur before the second date. We have also provieded a function `DaysInMonth`, which returns an integer representing the number of days in a month given two integer parameters : month and year. Do not use system provided Date objects. We are testing your implementation, not the system's.

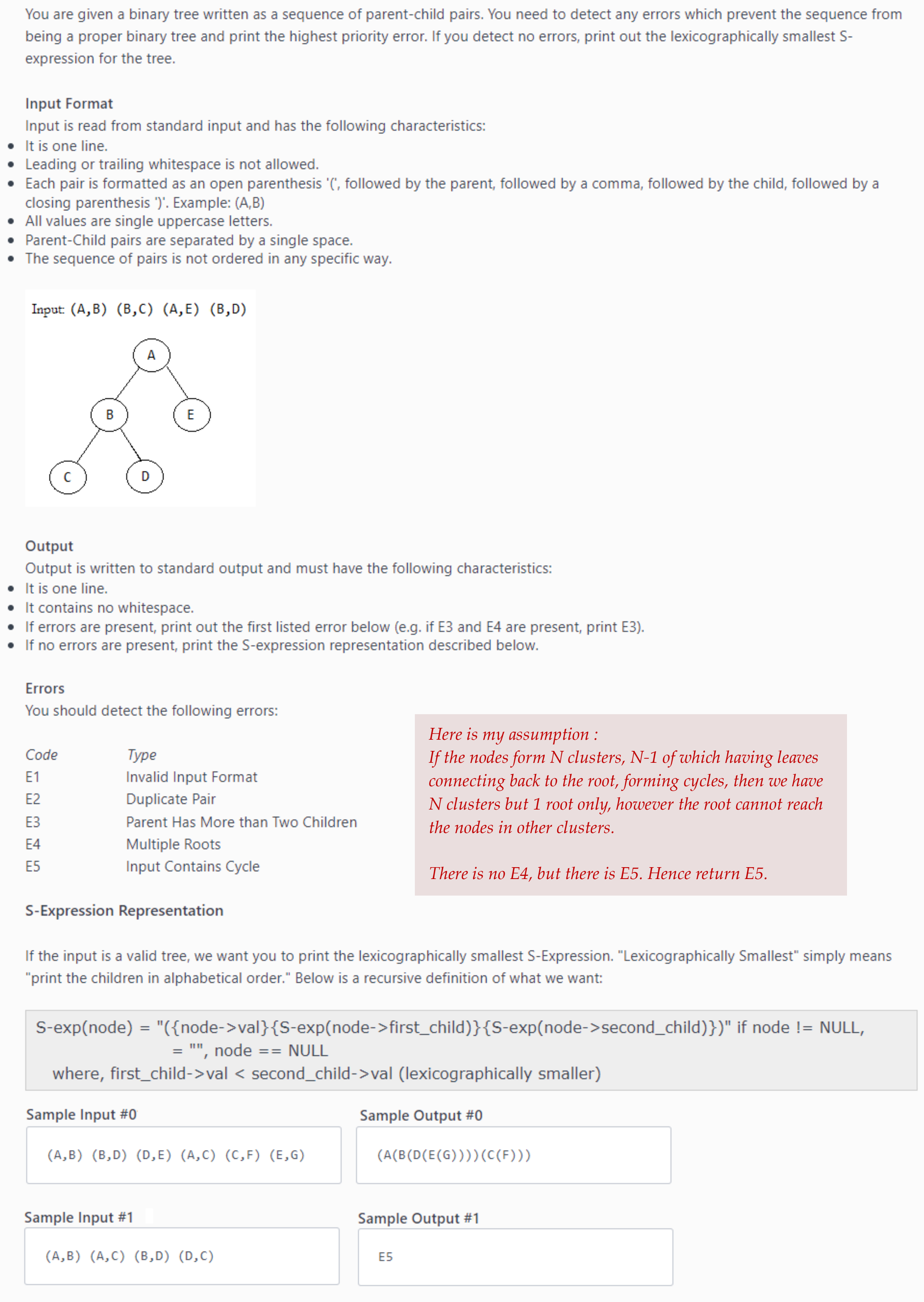
For example : `DaysBetween(2010,5,1,2011,5,1)` returns `365`.

My answer

```
struct year_month
{
    bool operator!=(const year_month& rhs)
    {
        return (y!=rhs.y || m!=rhs.m);
    }

    year_month& operator++()
    {
        ++m;
        if (m==13)
        {
            ++y;
            m = 1;
        }
        return *this;
    }

    int y;
    int m;
};

int DaysBetween(int year1, int month1, int day1, int year2, int month2, int day2)
{
    int day_diff = 0;

    year_month ym1{year1, month1};
    year_month ym2{year2, month2};
    while(ym1!=ym2)
    {
        day_diff += DaysInMonth(ym1.m, ym1.y);
        ++ym1;
    }
    day_diff += day2-day1;
    return day_diff;
}
```

Question 2

This is the same as the interview in DEC 2020. Now we have the precise problem formulation.



You are given a binary tree written as a sequence of parent-child pairs. You need to detect any errors which prevent the sequence from being a proper binary tree and print the highest priority error. If you detect no errors, print out the lexicographically smallest S-expression for the tree.

Input Format

Input is read from standard input and has the following characteristics:

- It is one line.
- Leading or trailing whitespace is not allowed.
- Each pair is formatted as an open parenthesis '(', followed by the parent, followed by a comma, followed by the child, followed by a closing parenthesis ')'. Example: (A,B)
- All values are single uppercase letters.
- Parent-Child pairs are separated by a single space.
- The sequence of pairs is not ordered in any specific way.

Input: `(A,B) (B,C) (A,E) (B,D)`

Output

Output is written to standard output and must have the following characteristics:

- It is one line.
- It contains no whitespace.
- If errors are present, print out the first listed error below (e.g. if E3 and E4 are present, print E3).
- If no errors are present, print the S-expression representation described below.

Errors

You should detect the following errors:

Code	*Type*
E1	Invalid Input Format
E2	Duplicate Pair
E3	Parent Has More than Two Children
E4	Multiple Roots
E5	Input Contains Cycle

Here is my assumption :
If the nodes form N clusters, N-1 of which having leaves connecting back to the root, forming cycles, then we have N clusters but 1 root only, however the root cannot reach the nodes in other clusters.

There is no E4, but there is E5. Hence return E5.

S-Expression Representation

If the input is a valid tree, we want you to print the lexicographically smallest S-Expression. "Lexicographically Smallest" simply means "print the children in alphabetical order." Below is a recursive definition of what we want:

```
S-exp(node) = "({node->val}{S-exp(node->first_child)}{S-exp(node->second_child)})" if node != NULL,
                        = "", node == NULL
   where, first_child->val < second_child->val (lexicographically smaller)
```

Sample Input #0

```
(A,B) (B,D) (D,E) (A,C) (C,F) (E,G)
```

Sample Output #0

```
(A(B(D(E(G))))(C(F)))
```

Sample Input #1

```
(A,B) (A,C) (B,D) (D,C)
```

Sample Output #1

```
E5
```

```
struct detail
{
    std::set<char> parents;
    char lhs_child;
    char rhs_child;
};

class tree_verifier
{
public:
    tree_verifier() : root(empty){}

    std::string run(const std::string& s)
    {
        std::set<std::pair<char,char>> edges;

        auto E12 = get_edges(s, edges);
        if (!E12.first)  return "E1";
        if (!E12.second) return "E2";

        auto E3 = construct_tree(edges);
        if (!E3) return "E3";

        auto E45 = find_root();
        if (!E45.first)  return "E4";
        if (!E45.second) return "E5";
        return s_expression(root);
    }

private:
    bool parse_an_edge(const std::string& s, std::uint64_t& n, char& x, char& y)
    {
        if (n!=0)
        {
            if (s[n]!=' ')      return false;
            if (++n>=s.size())  return false;
        }
        if (s[n]!='(')          return false;
        if (++n>=s.size())      return false;
        if (s[n]<'A'||s[n]>'Z') return false;
        x = s[n];
        if (++n>=s.size())      return false;
        if (s[n]!=',')          return false;
        if (++n>=s.size())      return false;
        if (s[n]<'A'||s[n]>'Z') return false;
        y = s[n];
        if (++n>=s.size())      return false;
        if (s[n]!=')')          return false;
        ++n;
        return true;
    }

    std::pair<bool,bool> get_edges(const std::string& s, std::set<std::pair<char,char>>& edges) // return error E1 E2
    {
    //  bool E1=true;
        bool E2=true;

        std::uint64_t n=0;
        while(n<s.size())
        {
            char x,y;
            if (!parse_an_edge(s,n,x,y)) return std::make_pair(false,E2);

            auto edge = std::make_pair(x,y);
            if (edges.find(edge) == edges.end())         edges.insert(edge);
            else                                         E2 = false;
        }
        return std::make_pair(true,E2);
    }

    bool construct_tree(const std::set<std::pair<char,char>>& edges) // return true for error E3
    {
        for(const auto& edge:edges)
        {
            // update children
            auto iter=impl.find(edge.first);
            if (iter==impl.end())                        impl[edge.first] = detail{{}, edge.second, empty};
            else if (iter->second.lhs_child == empty)    iter->second.lhs_child = edge.second;
            else if (iter->second.rhs_child == empty)    iter->second.rhs_child = edge.second;
            else return false;

            // update parents
            auto iter2=impl.find(edge.second);
            if (iter2==impl.end())                       impl[edge.second] = detail{{edge.first}, empty, empty};
            else                                         iter2->second.parents.insert(edge.first);
        }
        return true;
    }
    std::pair<bool,bool> find_root() // return true for error E4 E5
    {
        bool E4=true;
        bool E5=true;

        std::uint32_t num_of_root = 0;
        for(const auto& x:impl)
        {
            if (x.second.parents.empty())
            {
                root = x.first;
                ++num_of_root;
            }
            else if (x.second.parents.size()>1)
            {
                  E5 = false;
            }
        }

        if (num_of_root >1) E4 = false;
        if (num_of_root==0) E5 = false; // (A,B) (B,A) is considered as cycle
        if (root != empty)
        {
            if (tree_size(root)!=impl.size()) E5 = false; // (A,B) (C,C) is considered as cycle (just one root A)
        }
        return std::make_pair(E4,E5);
    }

    std::uint64_t tree_size(char this_node) const
    {
        std::uint64_t size = 1;

        auto iter = impl.find(this_node);
        assert(iter!=impl.end());
        if (iter->second.lhs_child != empty) size += tree_size(iter->second.lhs_child);
        if (iter->second.rhs_child != empty) size += tree_size(iter->second.rhs_child);
        return size;
    }

    std::string s_expression(char this_node) const
    {
        auto iter = impl.find(this_node);
        assert(iter!=impl.end());

        std::stringstream ss;
        ss << "(" << this_node;
        if (iter->second.lhs_child != empty &&
            iter->second.rhs_child != empty)
        {
            if (iter->second.lhs_child < iter->second.rhs_child)
            {
                ss << s_expression(iter->second.lhs_child);
                ss << s_expression(iter->second.rhs_child);
            }
            else
            {
                ss << s_expression(iter->second.rhs_child);
                ss << s_expression(iter->second.lhs_child);
            }
        }
        else if (iter->second.lhs_child != empty)
        {
            ss << s_expression(iter->second.lhs_child);
        }
        ss << ")";
        return ss.str();
    }

private:
    char root;
    std::unordered_map<char, detail> impl;
    static const char empty = '0';
};

int main()
{
    std::string s;
    std::getline(std::cin, s);

    tree_verifier v;
    std::cout << v.run(s);
    return 0;
}
```