ECE430.217 Data Structures

# Abstract Trees

## Weiss Book Chapter 4.1

**Byoungyoung Lee**

[https://compsec.snu.ac.kr](https://compsec.snu.ac.kr)

**byoungyoung@snu.ac.kr**

Lecture slides were prepared based on Douglas W. Harder's notes (**dwharder@alumni.uwaterloo.ca).**

# Outline

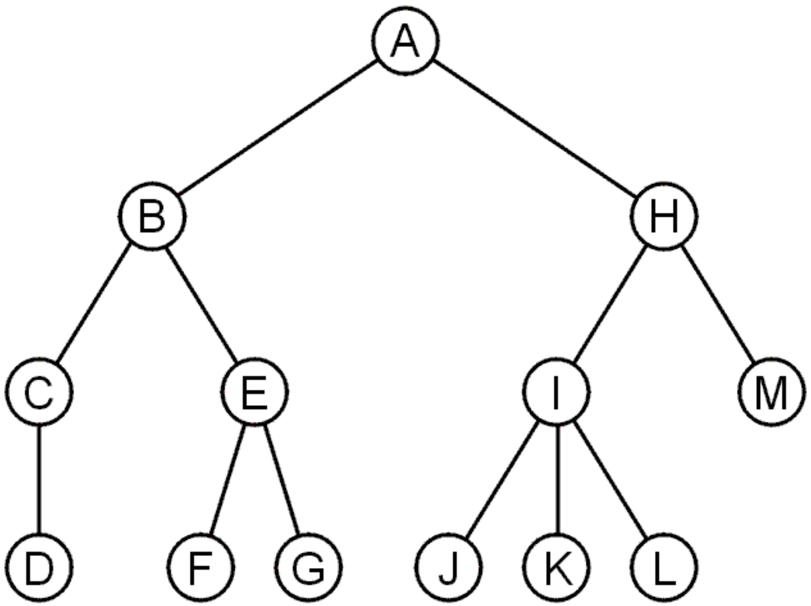
This topic discusses the concept of an abstract tree:

- Hierarchical ordering
- Description of an Abstract Tree/Hierarchy
- Applications
- Implementation
- Local definitions

# Abstract Trees

An abstract tree (or abstract hierarchy) does not restrict the number of nodes

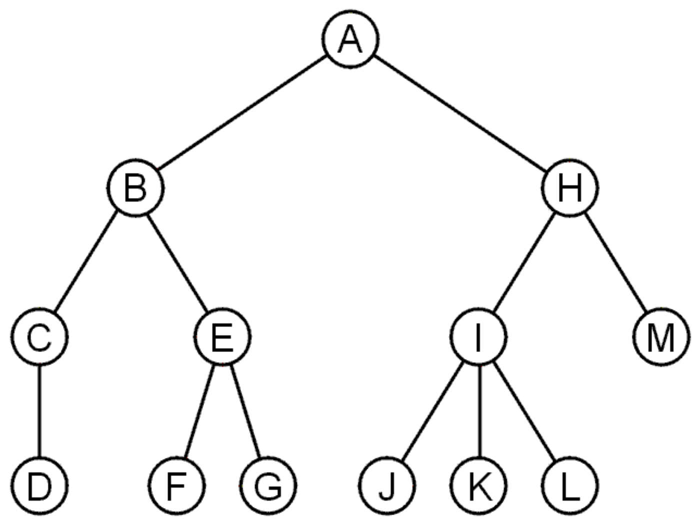
| Degree | Nodes |
|---|---|
| 0 | D, F, G, J, K, L, M |
| 1 | C |
| 2 | A, B, E, H |
| 3 | I |



# Abstract Trees: Design

We implement an abstract tree or hierarchy by using a class that:

- Stores a value
- Stores a parent pointer
- Stores children pointers in a linked-list



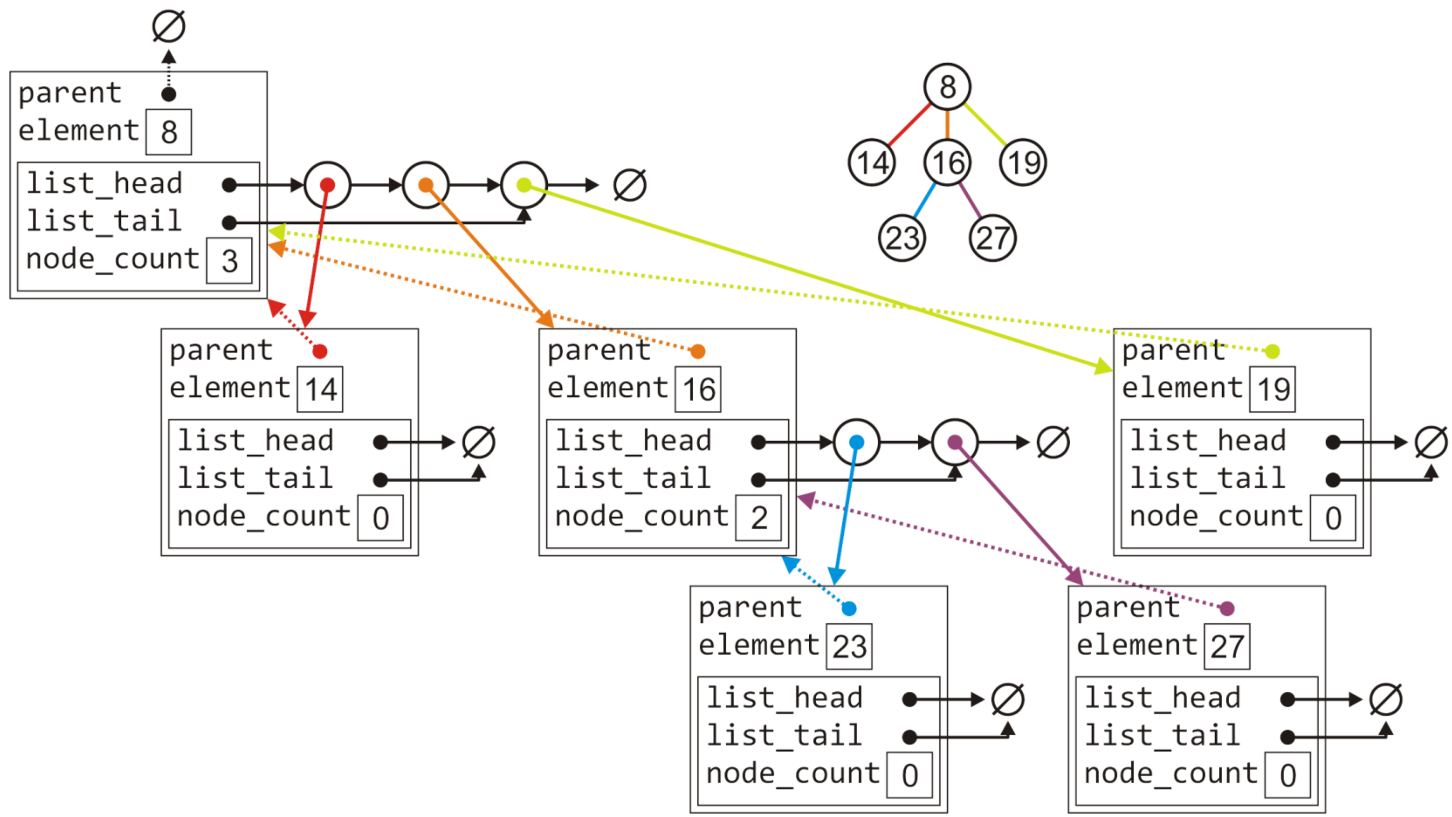
# Implementation

The class definition would be:

```
template <typename Type>
class SimpleTree {
private:
    Type value;
    SimpleTree *parent;
    List<SimpleTree *> children;

public:
    SimpleTree(Type const & = Type(), SimpleTree * = nullptr );

    Type get_value() const;
    SimpleTree<Type> *get_parent() const;
    int get_degree() const;
    bool is_root() const;
    bool is_leaf() const;
    SimpleTree<Type> *get_child(int n) const;
    SimpleTree<Type> *attach(Type const &obj);

    void attach_subtree(SimpleTree *);
    void detach_from_parent();

    int size() const;
    int height() const;
};
```

# Implementation

The tree with six nodes would be stored as follows:



# Implementation

Much of the functionality is similar to that of the `List` class:

```
template <typename Type>
SimpleTree<Type>::SimpleTree(Type const &obj, SimpleTree *p)
    : value(obj), parent(p) {
    // Empty constructor
}

template <typename Type>
Type SimpleTree<Type>::get_value() const {
    return value;
}

template <typename Type>
SimpleTree<Type> *SimpleTree<Type>::get_parent() const {
    return parent;
}
```

# Implementation

Much of the functionality is similar to that of the `List` class:

```
template <typename Type>
bool SimpleTree<Type>::is_root() const {
    return get_parent() == nullptr;
}

template <typename Type>
int SimpleTree<Type>::get_degree() const {
    return children.size();
}

template <typename Type>
bool SimpleTree<Type>::is_leaf() const {
    return get_degree() == 0;
}
```

# Implementation

**Accessing the $n^{th}$ child** requires a for loop ($\Theta(n)$):

```
template <typename Type>
SimpleTree<Type> *SimpleTree<Type>::get_child(int n) const {
    if (n < 0 || n >= get_degree()) {
        return nullptr;
    }

    auto it = children.begin();

    // Skip the first n - 1 children
    for (int i = 1; i <= n; ++i) {
        ++it;
    }
    return *it;
}
```

# Implementation

**Attaching a new object** to become a child is similar to a linked list:

```
template <typename Type>
SimpleTree<Type> *SimpleTree<Type>::attach(Type const &obj) {
    auto child = new SimpleTree(obj, this);
    children.push_back(child);
    return child;
}
```

# Implementation

Suppose we want to find **the size of a tree**:

- An empty tree has size 0, a tree with no children has size 1
- Otherwise, the size is one plus the size of all the children

```
template <typename Type>
int SimpleTree<Type>::size() const {
    int tree_size = 1;

    for (auto child = children.begin(); child != children.end(); child++) {
        tree_size += (*child)->size();
    }

    return tree_size;
}
```

# Implementation

Suppose we want to find **the height of a tree**:

- An empty tree has height –1 and a tree with no children is height 0
- Otherwise, the height is one plus the maximum height of any sub tree

```
template <typename Type>
int SimpleTree<Type>::height() const {
    int tree_height = 0;

    for (auto child = children.begin(); child != children.end(); child++) {
        tree_height = std::max(tree_height, 1 + (*child)->height());
    }

    return tree_height;
}
```

# Questions

- Any inefficient implementation point?
- If you see such points, how would you improve?

# Summary

In this topic, we have looked at one implementation of a general tree:

- Store the value of each node
- Store all the children in a linked list
- Size and height of a tree

# References

[1] Donald E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd Ed., Addison Wesley, 1997, §2.2.1, p.238.