CSE 203: Abstract Tree Implementation

Dr. Mohammed Eunus Ali

Professor

CSE, BUET

Slides are from Douglas Wilhelm Harder, **dwharder@alumni.uwaterloo.ca**

Outline

A hierarchical ordering of a finite number of objects may be stored in a tree data structure

Operations on a hierarchically stored container include:

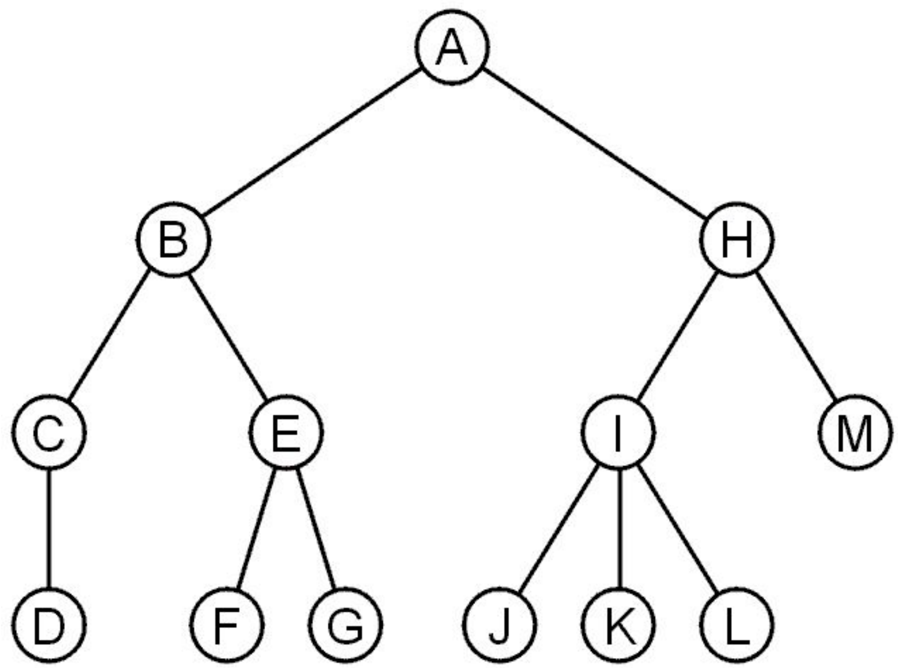
- Accessing the root:
- Given an object in the container:
  - Access the parent of the current object
  - Find the degree of the current object
  - Get a reference to a child,
  - Attach a new sub-tree to the current object
  - Detach this tree from its parent

Abstract Trees

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

- In this tree, the degrees vary:

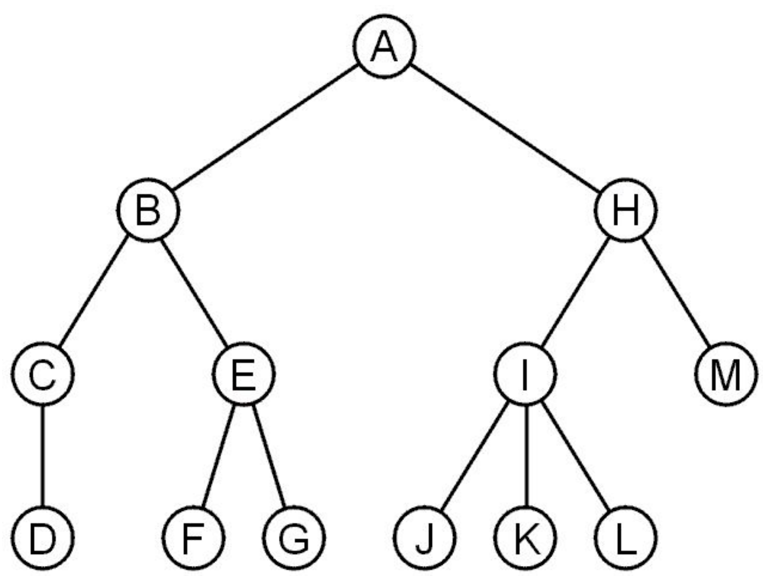
Degree	**Nodes**
0	D, F, G, J, K, L, M
1	C
2	A, B, E, H
3	I



Abstract Trees: Design

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

- Stores a value
- Stores the children in a linked-list



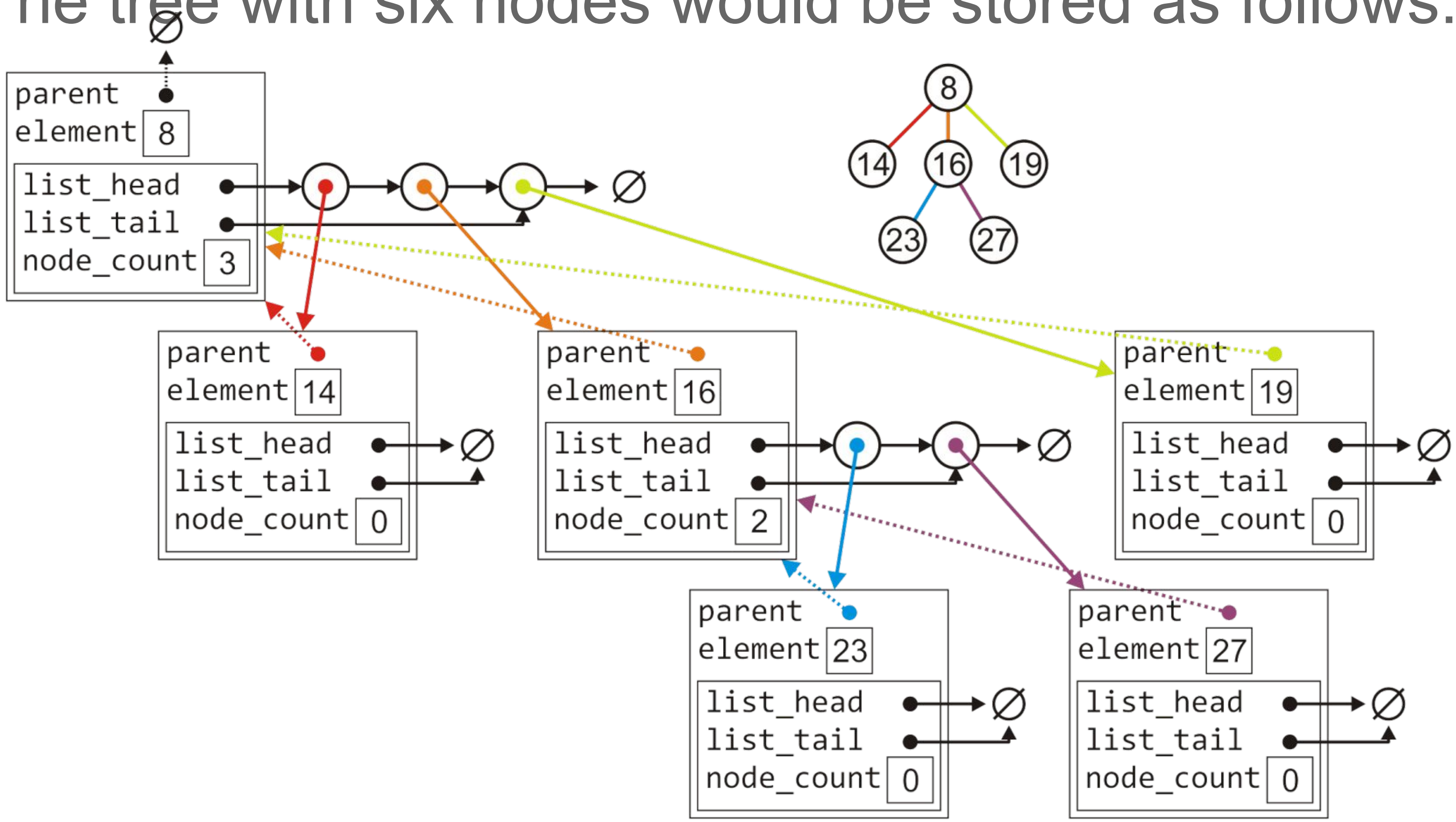
Implementation

The class definition would be:

```
template <typename Type>
class Simple_tree {
    private:
        Type node_value;
        Simple_tree *parent_node;
        Single_list<Simple_tree *> children;

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

        Type value() const;
        Simple_tree *parent() const;
        int degree() const;
        bool is_root() const;
        bool is_leaf() const;
        Simple_tree *child( int n ) const;
        int height() const;

        void insert( Type const & );
        void attach( Simple_tree * );
        void detach();
};
```

Implementation

The tree with six nodes would be stored as follows:



Single Linke List

```
class Single_node {
      private:
       Type element;
       Single_node *next_node;
     public:
       Type value ();
       Single_node next();
}
class Single_list {
       private:
              Single_node<Type> *list_head;
              Single_node<Type> *list_tail;
              int list_size;

       public:
              Single_list();
              Single_list( Single_list const & );
```

Implementation

Much of the functionality is similar to that of the `linked list`:

```
template <typename Type>
Simple_tree<Type>::Simple_tree( Type const &obj, Simple_tree *p ):
node_value( obj ),
parent_node( p ) {
    // Empty constructor
}

template <typename Type>
Type Simple_tree<Type>::value() const {
    return node_value;
}

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

Implementation

Much of the functionality is similar to that of the `Linked list`:

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

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

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

Implementation

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

```
template <typename Type>
Simple_tree<Type> *Simple_tree<Type>::child( int n ) const {
    if ( n < 0 || n >= degree() ) {
        return nullptr;
    }

    auto *child = children.head();

    // Skip the first n - 1 children
    for ( int i = 1; i < n; ++i ) {
        child = child->next();
    }

    return child->value(); //this is linked_list node value (not
the tree value)
}
```

Implementation

Inserting a new object to become a child is similar to a linked list:

```
template <typename Type>
void Simple_tree<Type>::insert( Type const
 &obj ) {
    children.push_back( new Simple_tree(
 obj, this ) );
}
```

Implementation

To detach a tree from its parent:

- If it is already a root, do nothing
- Otherwise, erase this object from the parent's list of children and set the parent pointer to zero

```
template <typename Type>
void Simple_tree<Type>::detach() {
    if ( is_root() ) {
        return;
    }

    parent()->children.erase( this );
    parent_node = nullptr;
}
```

Implementation

Attaching an entirely new tree as a sub-tree, however, first requires us to check if the tree is not already a sub-tree of another node:

- If so, we must detach it first and only then can we add it

```
template <typename Type>
void Simple_tree<Type>::attach( Simple_tree<Type>
  *tree ) {
    if ( !tree->is_root() ) {
        tree->detach();
    }

    tree->parent_node = this;
    children.push_back( tree );
}
```

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 Simple_tree<Type>::size() const {
    if ( this == nullptr ) {
        return 0;
    }

    int tree_size = 1;

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

```
#include <algorithm>

template <typename Type>
int Simple_tree<Type>::height() const {
    if ( this == nullptr ) {
        return -1;
    }

    int tree_height = 0;

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

    return tree_height;
}
```

Implementation

Implementing a tree by storing the children in an array is similar, however, we must deal with the full structure

```
class Simple_tree
{
    private:
        Type element;
        Simple_tree *parent_node;
        int child_count;
        int child_capacity;
        Simple_tree *children;
        // Everything else is similar to above
}
```

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
- not an easy ($\Theta(1)$) way to access children
- if we use an array, different problems...

References

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