4.2 Abstract Trees

Having introduced the tree data structure, we will step back and consider an Abstract Tree that stores a hierarchical ordering.

4.2.1 Description

An abstract tree stores data that is hierarchically ordered. Operations that may be performed on an abstract tree include:

1. Accessing the root, and
2. Given a reference to any node within the tree:
    a. Get a reference to its predecessor (the node's parent),
    b. Query the number of successors (what is the degree of the node),
    c. Get a reference to a child,
    d. Attach a new sub-tree as a child of the current node, and
    e. Detach the tree rooted at this node from the parent.

In a hierarchical ordering, abstract trees will usually not restrict the degree of a node.

4.2.2 Linked List Implementation

In this elementary implementation of a Tree ADT, we store the children as a linked list of pointers to those sub-trees. A full implementation of this class is found on the ECE 250 web site under `Algorithms/Trees/Simple_trees/`.

```
#include <algorithm>
#include "Single_list.h"

template <typename Type>
class Simple_tree {
    private:
        Type element;
        Simple_tree *parent_node;
        ece250::Single_list<Simple_tree *> children;

    public:
        Simple_tree( Type const &, Simple_tree * = 0 );

        Type retrieve() const;
        Simple_tree *parent() const;

        bool is_root() const;
        bool is_leaf() const;
        int degree() const;

        Simple_tree *child( int n ) const;
        int size() const;
        int height() const;

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

While looking at these member functions, you can consider calling these member functions on the various nodes shown in the simple tree shown in Figure 1.

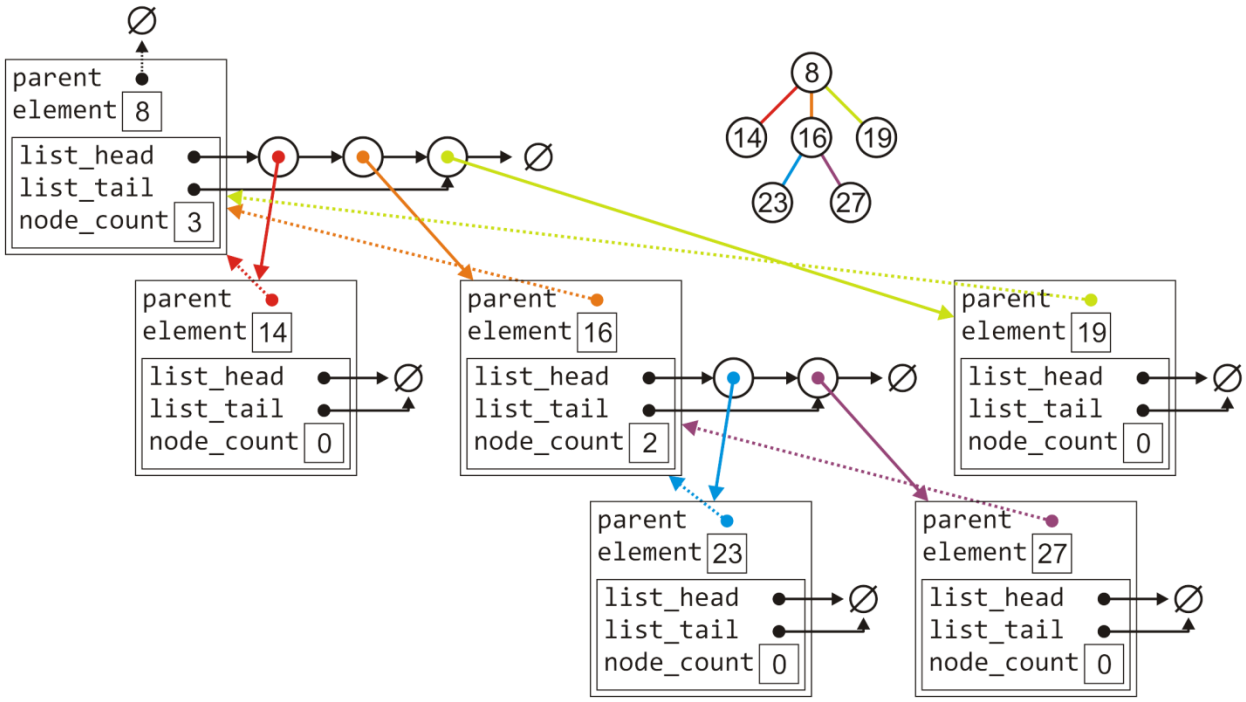

Figure 1. A tree of six nodes stored in the `Simple_tree` data structure.

4.2.2.1 Basic Functionality

The simple accessing and query member functions have implementations that are similar to what you would expect from the `Single_list` and `Single_node` class.

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

template <typename Type>
Type Simple_tree<Type>::retrieve() const {
    return element;
}

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

template <typename Type>
bool Simple_tree<Type>::is_root() const {
    return ( parent() == 0 );
}

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 );
}
```

4.2.2.2 Accesing the n^{th} Child

The user would ask for the n^{th} child. If this is outside the range, we return the zero pointer, otherwise, we step through the linked list until we reach the n^{th} entry.

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

    ece250::Single_node<Simple_tree *> *ptr = children.head();

    for ( int i = 1; i < n; ++i ) {
        ptr = ptr->next();
    }

    return ptr->retrieve();
}
```

4.2.2.3 Attaching and detaching children

If we are attaching a new object into the current node to form a child (suppose we hire a new employee or create a new derived class), we push a new tree onto the back of the linked list:

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

To detach a tree, we first check if it is already the root of a tree—in which case we do nothing. Otherwise, we erase this tree from the children of the parent and set this nodes parent to the zero pointer.

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

    parent()->children.erase( this );
    parent_node = 0;
}
```

If, however, we are attaching an already constructed tree, we must be a little more careful. First, if the tree we are attaching is attached to a different tree, we must detach it from its parent.

```
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 );
}
```

4.2.2.4 A Recursive Size and Height Member Functions

As our first two recursive functions, we will see how we can recursively compute the size (number of nodes in) and height of a tree.

The size of a tree is one plus the sizes of all the children.

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

    for (
        ece250::Single_node<Simple_tree *> *ptr = children.head();
        ptr != 0;
        ptr = ptr->next()
    ) {
        s += ptr->retrieve()->size();
    }

    return s;
}
```

The height of tree with a single node is zero; however, if there are any children, the height is one more than the maximum height of the children.

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

    for (
        ece250::Single_node<Simple_tree *> *ptr = children.head();
        ptr != 0;
        ptr = ptr->next()
    ) {
        h = std::max( h, 1 + ptr->retrieve()->height() );
    }

    return h;
}
```

4.2.3 Array Implementation

An implementation using arrays would be similar to that using a linked list—the implementation, however, would be more complex.

```
template <typename Type>
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
}

template <typename Type>
Simple_tree<Type>::Simple_tree( Type const &obj, Simple_tree *p ):
element( obj ),
parent_node( p ),
child_count( 0 ),
child_capacity( 4 ),
children( new Simple_tree *[child_capacity] ) {
    // Empty constructor
}
```

4.2.4 Locally Defined Orders

The ordering of general trees is usually local:

1. A root node is explicitly defined.
2. A new node is defined as a being a child of a given parent node
3. There is no general definition as to what happens to children when a node is “removed”
4. Given two nodes in a tree, an algorithm must be used to determine any relationship between them based on the ordering within the tree