Justin Ciocoi

CSCI 373 Textbook Notes

Chapter 7: Trees

7.1: General Trees

- **7.1.1: Tree Definitions and Properties**
    - A *tree* is an abstract data type which stores elements hierarchically
    - Each element, with the exception of the top node has a parent element and zero or more children elements
    - The top node of a tree is generally called the *root*
    - Formally, a tree T can be defined as a set of nodes storing elements in a parent-child relationship with the following properties:
        - If T is non-empty, it has a special node, called its root, which has no parent
        - each node v of T different from the root has a unique parent node w; every node with parent w is a child of w
    - An *edge* of tree T is a pair of nodes (u, v) such that u is the parent of v or vice-versa
    - A *path* of T is a sequence of nodes such that any two consecutive nodes in the sequence form an edge
    - A tree is *ordered* if there is a linear ordering defined for the children of each node
- **7.1.2: Tree Functions**
    - The tree ADT stores elements at the nodes of the tree
    - Since nodes are internal aspects of our implementations, we don't allow direct access to them, opting instead to associate each node with a position object which provides public access to nodes
    - It is useful to overload the dereference operator, *, in order to return the element of node p when `*p` is called

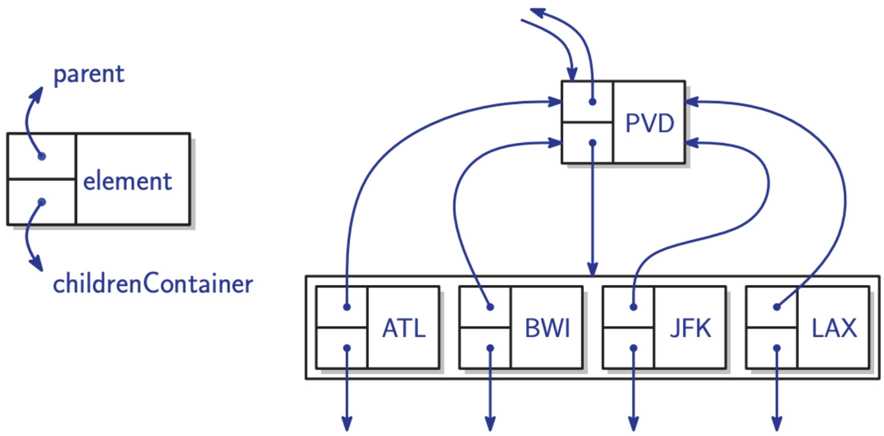
        - Given a position p of tree T, we can define the following:
            - `p.parent()` : Return the parent of p; error occurs if p is the root
            - `p.children()` : Return a position list containing the children of node p
            - `p.isRoot()` : Return true if p is the root and false otherwise
            - `p.isExternal()` : Return true if p is external and false otherwise
    - If a tree T is ordered, then the list provided by `p.children()` provides access to the children of node p in order
    - If p is external, then `p.children()` returns an empty list
    - For the tree ADT, we can also define the following functions
        - `size()` : Returns the number of nodes in the tree
        - `empty()` : Returns true if the tree is empty and false otherwise
        - `root()` : Returns a position for the tree's root; error occurs if the tree is empty
        - `positions()` : Returns a position list of all the nodes of the tree
- **7.1.3: A C++ Tree Interface**
    - First, let us present an interface for a position class which will represent a position in a tree

```
template <typename E>
class Position<E>
{
 public:
        E& operator*();
        Position parent() const;
        PositionList children() const;
        bool isRoot() const;
        bool isExternal() const;
}
```

    - Next, let us look at the C++ interface for a tree

```
template <typename E>
class Tree<E>
{
    public:
        class Position;
        class PositionList;
    public:
        int size() const;
        bool empty() const;
        Position root() const;
        PositionList positions() const;
}
```

- **7.1.4: A Linked Structure for General Trees**
  - A natural way to realize a tree T is to use a linked structure where we represent each node by a reference to its parent, its element, and its children
  - The fundamental idea of this can be seen in the below diagram where arrows function as pointers between data values



  - Now we can summarize the time complexity of the associated functions outlined above

Operation	***Time***
isRoot, isExternal	$O(1)$
parent	$O(1)$
children(p)	$O(c_p)$
size, empty	$O(1)$
root	$O(1)$
positions	$O(n)$

7.2: Tree Traversal Algorithms

- **7.2.1: Depth and Height**
  - Let p be a node of tree T
  - The *depth* of p is the number of ancestors of p excluding p itself
    - If p is the root, then the depth of p is 0
    - Otherwise, the depth is one plus the depth of the parent of p
  - Thus, we can achieve this with the following recursive function:

```
int depth(T, p)
{
    if(p.isRoot())
    {
         return 0;
    }
    else
    {
        return 1+depth(T,p.parent())
    }
}
```

    - The running time of this algorithm is $O(d_p)$, where d_p denotes the depth of the node p in the tree T
  - We can use a similar definition with the height of a tree, and also define the height of a node p recursively
    - If p is external, then the height of p is 0
    - Otherwise, the height of p is one plus the maximum height of a child of p

```
height1(T)
{
    for(p in T.positions)
    {
        if (p.isExternal())
            h=max(h,depth(T,p))
    }
    return h;
}
```

  - Which has a C++ implementation as follows

```
int height1(const Tree& T)
{
    int h = 0;
    PositionList nodes = T.positions();
    for(Iterator q = nodes.begin(); q != nodes.end;++q)
    {
        if(q->isExternal())
            h = max(h, depth(T, *q))
    }
    return h;
}
```

- However, this is an inefficient method and there exists a different C++ implementation that improves efficiency

```
int height2(const Tree& T, const Position &p)
{
    if(p.isExternal())
        return 0;
    int h = 0;
    PositionList ch = p.children();
    for(Iterator q = ch.begin();q != ch.end(); ++q)
        h = max(h, height2(T, *q);
    return 1+ h;
}
```

- **7.2.2: Pre-Order Traversal**
  - There are various methods for traversing across a tree structure, one of which is the pre-order traversal
  - This means that the root is processed, then the root's left child, and finally the root's right child
  - This process is done recursively down the entire tree until every node in the tree has been processed by the traversal algorithm
  - We can use the following C++ implementation of a pre-order traversal:

```
void preorderPrint(const Tree &T, const Position &p)
{
    cout<< *p;
    PositionList ch = p.children()
    for(Iterator q = ch.begin(); q != ch.end(); ++q)
    {
        cout<<" ";
        preorderPrint(T, *q);
    }
}
```

- **7.2.3: Post-Order Traversal**
    - In a post-order traversal, the left subtree is processed, then the right subtree, and finally, the root

```
void postorderPrint(const Tree &T, const Position &p)
{
    PositionList ch = p.children()
    for(Iterator q = ch.begin(); q != ch.end(); ++q)
    {
        cout<<" ";
        preorderPrint(T, *q);
    }
    cout<< *p;
}
```

7.3: Binary Trees

- A *binary tree* is an ordered tree in which every node has at most two children
- A binary tree must adhere to the following properties
    1. Every node has at most two children
    2. Each child node is labeled as being either a left child or a right child
    3. A left child precedes a right child in the ordering of children of a node
- A binary tree is proper if each node has either zero or two children
- A binary tree can be also be defined in a recursive manner:
    - A node r called the root of T and storing an element

- A binary tree, called the left subtree of T
- A binary tree, called the right subtree of T

- **7.3.1: The Binary Tree ADT**
  - Each node of the binary tree stores an element and is associated with a position object, which provides public access to nodes
  - By overloading the dereferencing operator, an element associated with position p can be accessed using $*p$
  - A position object, p, supports the following operations

    `p.left()` : Returns left child of p; Error if p is external

    `p.right()` : Returns right child of p; Error if p is external

    `p,parent()` : Returns parent of p; Error if p is root

    `p.isRoot()` : Returns true if p is the root and false otherwise

    `p.isExternal()` : Returns true if p is external and false otherwise
  - The binary tree ADT supports the same operations as the general tree, namely:
    - `size()` : Returns the number of nodes in the tree
    - `empty()` : Returns true if the tree is empty and false otherwise
    - `root()` : Returns a position for the tree's root; error occurs if the tree is empty
    - `positions()` : Returns a position list of all the nodes of the tree
- **7.3.2: A C++ Binary Tree Interface**
  - First, we can define the interface for the position objects which will be used by the class

```
template <typename E>
class Position <E>
{
 public:
 E& operator*();
 Position left() const;
 Position right() const;
 Position parent() const;
 bool isRoot() const;
 bool isExternal() const;
};
```

- Next, we define the interface for the actual tree class itself

```
template <typename E>
class BinaryTree <E>
{
    public:
        class Position;
        class PositionList;
    public:
        int size() const;
        bool empty() const;
        Position root() const;
        PositionList positions() const;
};
```

- **7.3.3: Properties of Binary Trees**
    - Let T be a non-empty binary tree
    - Let n denote the number of nodes
    - Let n_E denote the number of external nodes
    - Let n_I denote the number of internal nodes
    - Let h denote the height of the tree
    - Then, a binary property will adhere to the following properties:
        1. $h + 1 \leq n \leq 2^{h+1} - 1$
        2. $1 \leq n_E \leq 2^h$

3. $h \leq n_I \leq 2^h - 1$

4. $log(n+1) - 1 \leq h \leq n - 1$

- If a tree is proper, then the following conditions will also be satisfied

  1. $2h + 1 \leq n \leq 2^{h+1} - 1$

  2. $h + 1 \leq n_E \leq 2^h$

  3. $h \leq n_I \leq 2^h - 1$

  4. $log(n+1) - 1 \leq h \leq (n-1)/2$

- We also have the relationship that in a non-empty proper binary tree, the number of external nodes is exactly one more than the number of internal nodes

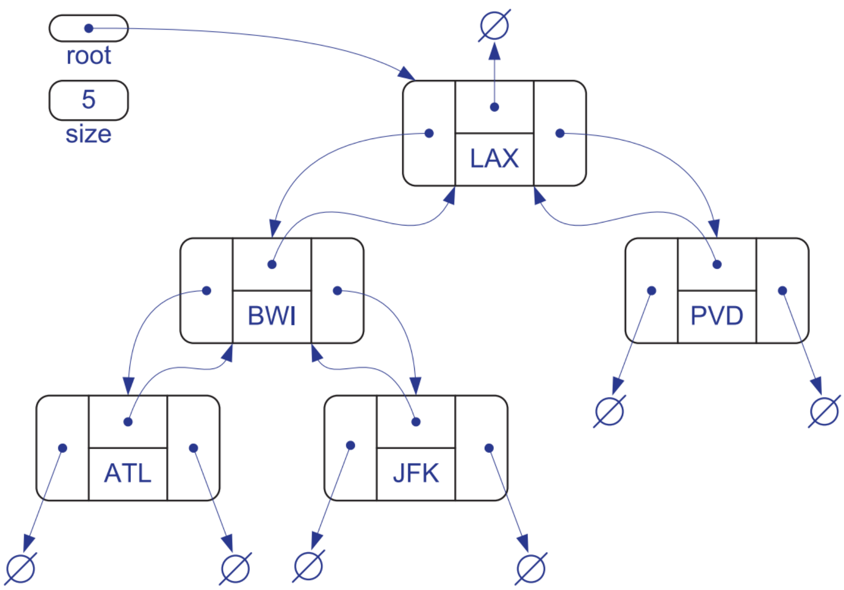
- **7.3.4: A Linked Structure for Binary Trees**

  - This structure will define each node of a binary tree T as a part of a linked structure where each node has 4 components

    - First is the element of the node

    - Next is the parent pointer which points to the node's parent

    - Finally, two more pointers left and right point to the nodes left and right children respectively



  - Here is the implementation for the node class

```cpp
struct Node
{
    Elem elt;
    Node* parent;
    Node* left;
    Node* right;
    Node() : elt(), parent(NULL), left(NULL), right(NULL)
};
```

- Next we can define the position class

```cpp
class Position
{
    private:
        Node* v;
    public:
        Position(Node* _v = NULL) : v(_v) {}
        Elem& operator*()
        {
            return v->elt;
        }
        Position left() const
        {
            return Position(v->left);
        }
        Position right() const
        {
            return Position(v->right);
        }
        Position parent() const
        {
            return Position(v->parent);
        }
        bool isRoot() const
        {
            return v->parent == NULL
        }
        bool isExternal() const
        {
            return v->left == NULL && v->right == NULL
        };
    typedef std::list<Position> PositionList;
}
```

- Finally, we will look at the Linked Binary Tree class itself as well as the implementations of its member functions

```
typedef int Elem;
class LinkedBinaryTree
{
    protected:
        // node declaration
    public:
        // position declaration
    public:
        LinkedBinaryTree();
        int size() const;
        bool empty() const;
        Position root() const;
        PositionList positions() const;
        void addRoot();
        void expandExternal(const Position& p);
        Position removeAboveExternal(conts Position& p);
    protected:
        void preorder(Node *v, PositionList& pl) const;
    private:
        Node* _root;
        int n;
};
```

- Here n is the total number of nodes in the tree

- Next we can see the implementations of some of the above member functions

```cpp
LinkedBinaryTree::LinkedBinaryTree()
: _root(NULL), n(0) {}

int LinkedBinaryTree::size() const
{
    return n;
}

bool LinkedBinaryTree::empty() const
{
    return size() == 0;
}

LinkedBinaryTree::Position LinkedBinaryTree::root() const
{
    return Position(_root);
}

void LinkedBinaryTree::addRoot()
{
    _root = new Node;
    n=1;
}
```

- Binary tree updating functions
  - `expandExternal(p)` will transform p from an external node to an internal node by creating two new external nodes as left and right children of p respectively; an error occurs if p is an internal node
  - `removeAboveExternal(p)` will remove the external node p along with its parent, q and then replace q with the sibling of p; an error occurs either if p is internal or if it is the root
- Now lets look at a C++ implementation of the `expandExternal()` function

```cpp
void LinkedBinaryTree::expandExternal(const Possition& p)
{
    Node* v = p.v;
    v->left = new Node;
    v->left->parent = v;
    v->right = new Node;
    v->right->parent = v;
    n += 2;
}
```

- And now the `removeAboveExternal()` C++ implementation

```cpp
LinkedBinaryTree::Position
LinkedBinaryTree::removeAboveExternal(const Position& p)
{
    Node* w = p.v;
    Node* v = w->parent;
    Node* sib = (w == v->left ? v->right : v->left);
    if(v == _root}
    {
        _root = sib;
        sib->parent = NULL;
    }
    else
    {
        Node* gpar = v->parent;
        if(v==gpar->left)
            gpar->left = sib;
        else
            gpar->right = sib;
        sib->par = gpar;
    }
    delete w;
    delete v;
    n -= 2;
    return Position(sib);
}
```

- Let us snow also look at the `positions()` function

```
LinkedBinaryTree::PositionList LinkedBinaryTree::positions() const
{
    PositionList pl;
    preorder(_root, pl);
}
```

- Which utilizes the following `preorder()` function for pre-order traversal of trees

```
void LinkedBinaryTree::preorder(Node* v, PositionList& pl) const
{
    pl.push_back(Position(v));
    if(v->left != NULL)
        preorder(v->left, pl);

    if(v->right != NULL)
        preorder(v->right, pl);
}
```

- And finally, we can look at the various time complexities for these various functions in the below table

Operation	***Time***
left, right, parent, isExternal, isRoot	$O(1)$
size, empty	$O(1)$
root	$O(1)$
expandExternal, removeAboveExternal	$O(1)$
positions	$O(n)$

- **7.3.5: A Vector-Based Structure for Binary Trees**
  - A fairly simple structure for representing a binary tree T is based on a way of numbering of the tree's nodes
  - For any node v of tree T, left $f(v)$ be the integer defined as follows
    - If v is the root of T, then $f(v) = 1$
    - If v is the left child of node u, then $f(v) = 2f(u)$
    - If v is the right child of node u, then $f(v) = 2f(u) + 1$
  - You must omit the 0^{th} index from the vector in order for the above rules to work

    - In this binary tree implementation, time complexities of the functions is the same as the linked-structure based binary tree

- **7.3.6: Traversals of a Binary Tree**
    - *Pre-order traversal of a binary tree*
        - The pre-order traversal processes the root, followed by recursive processing of the left subtree, and then the right subtree
    - *Post-order traversal of a binary tree*
        - The post-order traversal will first recursively process the left subtree, then the right subtree, and finishes by processing the root
    - *In-order traversal of a binary tree*
        - The in-order traversal will recursively process the left subtree, then the root, and will finish by processing the right subtree