The Tree ADT

- **Non-linear** collection: not a linear sequence of 1st entry, 2nd entry, etc.
- Terminology
  - **parent**: parent of a node is the single node linked directly above it
  - **child**: child of a node is a node linked directly below it
  - **sibling**: 2 nodes are siblings if they have the same parent
  - **root**: special node that has no parent
  - **leaf**: node that has no children
  - **ancestor**: any node from which this node descends directly or indirectly
  - **descendant**: any node that descends from this node directly or indirectly
  - **subtree**: smaller tree 'rooted' by some particular node in the tree
  - **depth of node**: # edges in directed path from this node to the root (depth of root is 0)
  - **depth of tree**: maximum depth of any leaf node
  - **height of tree**: depth of tree + 1
  - **binary tree**: tree where each node can have maximum 2 children
  - **full binary tree**: every leaf has same depth and every non-leaf has 2 children
  - **complete binary tree**: every level (except deepest) must contain as many nodes as possible, and, at deepest level, all nodes are as far left as possible
- Functionality:
  - initialize, insert, delete, find, management (copy, clear, depth, etc.)

Class Invariant

- Explicit statement of how the data structure is used to represent the ADT; ***each function depends on class invariant being valid when function called***

- Binary tree class implemented with array

  (1) Nodes in tree stored in partially filled array called *data*
  (2) Root node is in *data[0]*
  (3) For node in *data[i]*, its left child is in *data[2i + 1]* and its right child is in *data[2i + 2]*
  (4) For node in *data[i]*, its parent is in *data[(i – 1)/2]*
  (5) *Used* and *capacity* of array are maintained

- Binary tree class implemented with (pointer-based) linked list

  (1) Entire tree is represented as a pointer to the *root* node
  (2) Each node contains *data*, *left child pointer*, and *right child pointer*
  (3) For leaf node, *left* and *right child pointers* are NULL

// BNODE.h

```
#ifndef BNODE_H
#define BNODE_H

#include <iostream>
using namespace std;

class BNODE {
public:
  typedef int valueType;

private:
  // Member variables
  valueType data;
  BNODE *left, *right;

public:
  // Constructor
  BNODE (const valueType initData = valueType( ),
              BNODE* leftLink = NULL, BNODE* rightLink = NULL) {
    setData(initData);
    setLeft(leftLink);
    setRight(rightLink);
  }

  // Accessors
  valueType getData( ) const { return(data); }
  valueType& getDataForUpdate( ) { return(data); }

  BNODE* getLeft() { return(left); }
  BNODE*& getLeftForUpdate() { return(left); }

  BNODE* getRight() { return(right); }
  BNODE*& getRightForUpdate( ) { return(right); }

  // Mutators
  void setData(const valueType& x) { data = x; }
  void setLeft(BNODE* leftLink) { left = leftLink; }
  void setRight(BNODE* rightLink) { right = rightLink; }

  // Other functions
  bool isLeaf() const { return((left == NULL) && (right == NULL)); }
};

#endif  // BNODE_H
```

// BINARYTREE.h

```
#ifndef BINARYTREE_H
#define BINARYTREE_H

#include <iostream>
#include <cstdlib>
#include "BNODE.h"
using namespace std;

class BINARYTREE {
private:
  // Member variables
  BNODE *root;

public:
  // Constructor
  BINARYTREE( ) { setRoot(NULL); }

  // Destructor
  ~BINARYTREE( ) { clear(root); }

  // Accessors
  BNODE* getRoot( ) { return(root); }
  BNODE*& getRootForUpdate( ) { return(root); }

  // Mutators
  void setRoot(BNODE* newRoot) { root = newRoot; }

  // Other functions
  void clear(BNODE*& rootPtr);
};

#endif  // BINARYTREE_H
```

// BINARYTREE.cpp

```
#include "BINARYTREE.h"
using namespace std;

void BINARYTREE::clear(BNODE*& rootPtr) {
  if (rootPtr != NULL) {
    clear(rootPtr->getLeftForUpdate( ));      // clear left subtree
    clear(rootPtr->getRightForUpdate( ));    // clear right subtree
    delete rootPtr;                          // delete root
    rootPtr = NULL;                          // don't leave it pointing to garbage
  }
}
```