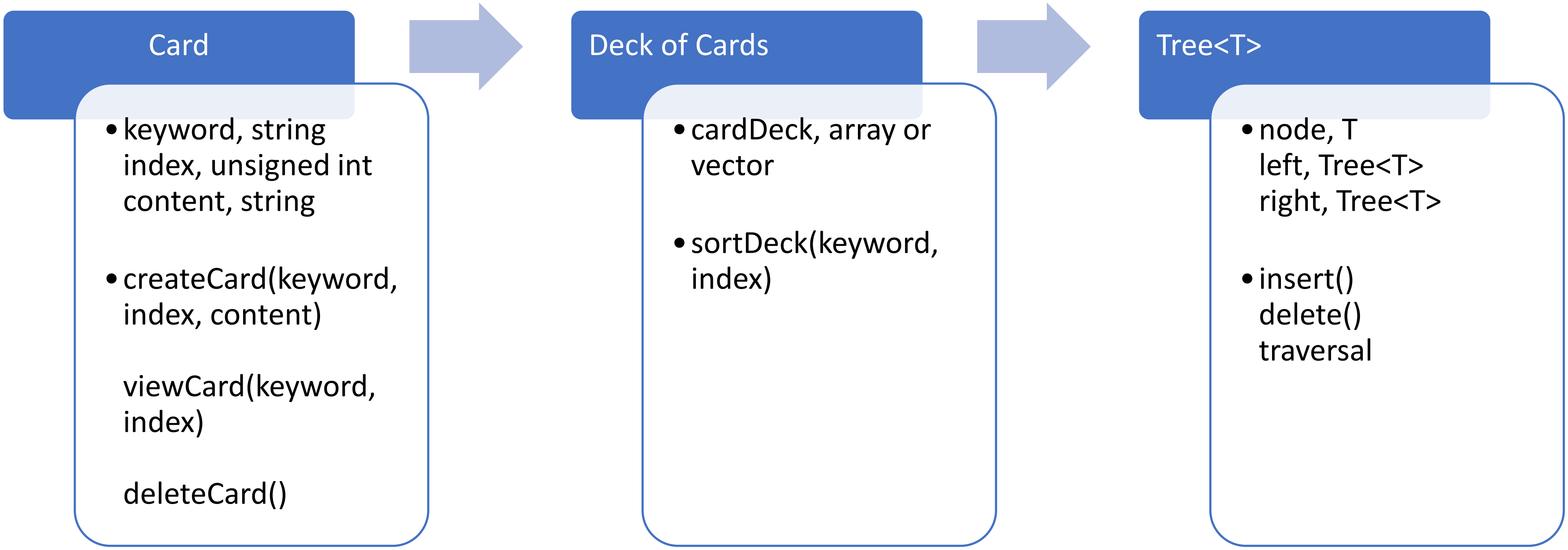

Card

- keyword, string
  index, unsigned int
  content, string
- createCard(keyword, index, content)

  viewCard(keyword, index)

  deleteCard()

Deck of Cards

- cardDeck, array or vector
- sortDeck(keyword, index)

Tree<T>

- node, T
  left, Tree<T>
  right, Tree<T>
- insert()
  delete()
  traversal

```
// IndexCardTree.cpp
// Includes source code referenced from Gaddis Starting out with C++ 8th edition

#include "stdafx.h"
#include <iostream>
using namespace std;
// Specification file for the BinaryTree class
// Needs to go back and change to TemplatedBinaryTree class
#ifndef INTBINARYTREE_H
#define INTBINARYTREE_H

template <class T>
class BinaryTree
{
private:
    struct TreeNode
    {
        T value;
        TreeNode *left;
        TreeNode *right;
    };

    TreeNode *root;

    void insert(TreeNode *&, TreeNode *&);  // *& poTer being passed by reference
    void destroySubtree(TreeNode *);
    void deleteNode(T, TreeNode *&);
    void makeDeletion(TreeNode *&);
    void displayInOrder(TreeNode *) const;
    void displayPreOrder(TreeNode *) const;
    void displayPostOrder(TreeNode *) const;

public:
    // Constructor
    BinaryTree()
    {
        root = nullptr;
    }

    // Destructor
 /*    ~BinaryTree()
    {
        destroySubtree(root);
    }*/

    //Binary tree operations
    void insertNode(T);
    bool searchNode(T);
    void removeNode(T);
    void displayInOrder() const
    {
        displayInOrder(root);
    }
    void displayPreOrder() const
    {
        displayPreOrder(root);
    }
    void displayPostOrder() const
    {
        displayPostOrder(root);
    }
};

#endif

template <class T>
void BinaryTree<T>::insertNode(T num)
{
    TreeNode *newNode = nullptr;

    // Create a new node and store num in it
    newNode = new TreeNode;
    newNode->value = num;
    newNode->left = newNode->right = nullptr;

    // Insert the node
    insert(root, newNode);
}

template <class T>
void BinaryTree<T>::insert(TreeNode *&nodePtr, TreeNode *&newNode)
{
    if (nodePtr == nullptr)
        nodePtr = newNode;
    else if (newNode->value < nodePtr->value)
        insert(nodePtr->left, newNode); // insert left branch
    else
        insert(nodePtr->right, newNode); // insert right branch
}

// The displayInOrder member function displays the values
// in the subtree pointed to by nodePtr, via inorder traversal.
// left, root, right
template <class T>
void BinaryTree<T>::displayInOrder(TreeNode *nodePtr) const
{
    if (nodePtr)
    {
        displayInOrder(nodePtr->left);
        cout << nodePtr->value << endl;
        displayInOrder(nodePtr->right);
    }
}

// The displayPreOrder member function displays the values
// in the subtree pointed to by nodePtr, via preorder traversal.
// root, left, right
template <class T>
void BinaryTree<T>::displayPreOrder(TreeNode *nodePtr) const
{
    if (nodePtr)
    {
        cout << nodePtr->value << endl;
        displayPreOrder(nodePtr->left);
        displayPreOrder(nodePtr->right);
    }
}

// The displayPostOrder member function displays the values
// in the subtree pointed to by nodePtr, via postorder traversal.
// left, right, root
template <class T>
void BinaryTree<T>::displayPostOrder(TreeNode *nodePtr) const
{
    if (nodePtr)
    {
        displayPostOrder(nodePtr->left);
        displayPostOrder(nodePtr->right);
        cout << nodePtr->value << endl;
    }
}

template <class T>
bool BinaryTree<T>::searchNode(T num)
{
    TreeNode *nodePtr = root;

    while (nodePtr)
    {
        if (nodePtr->value == num)
            return true;
        else if (num < nodePtr->value)
            nodePtr = nodePtr->left;
        else
            nodePtr = nodePtr->right;
    }

    return false;
}

template <class T>
void BinaryTree<T>::removeNode(T num)
{
    deleteNode(num, root);
}

template <class T>
void BinaryTree<T>::deleteNode(T num, TreeNode *&nodePtr)
{
    if (num < nodePtr->value)
        deleteNode(num, nodePtr->left);
    else if (num > nodePtr->value)
        deleteNode(num, nodePtr->right);
    else
        makeDeletion(nodePtr);
}

// makeDeletion member function deletes node from tree and
// reattach the deleted node's subtrees
template <class T>
void BinaryTree<T>::makeDeletion(TreeNode *&nodePtr)
{
    // Define a temp pointer to use in reattaching the left subtree
    TreeNode *tempNodePtr = nullptr;

    if (nodePtr == nullptr)
        cout << "Cannot delete empty node. \n";
    else if (nodePtr->right == nullptr)
    {
        tempNodePtr = nodePtr;
        nodePtr = nodePtr->left; // Reattach left child
        delete tempNodePtr;
    }
    else if (nodePtr->left == nullptr)
    {
        tempNodePtr = nodePtr;
        nodePtr = nodePtr->right; // Reattach right child
        delete tempNodePtr;
    }
    // If the node has two children
    else
    {
        // Move one node to the right
        tempNodePtr = nodePtr->right;

        // Go to the end of left node
        while (tempNodePtr->left)
            tempNodePtr = tempNodePtr->left;

        // Reattach the left subtree
        tempNodePtr->left = nodePtr->left;
        tempNodePtr = nodePtr;

        //Reattach the right subtree
        nodePtr = nodePtr->right;
        delete tempNodePtr;
    }
}


```