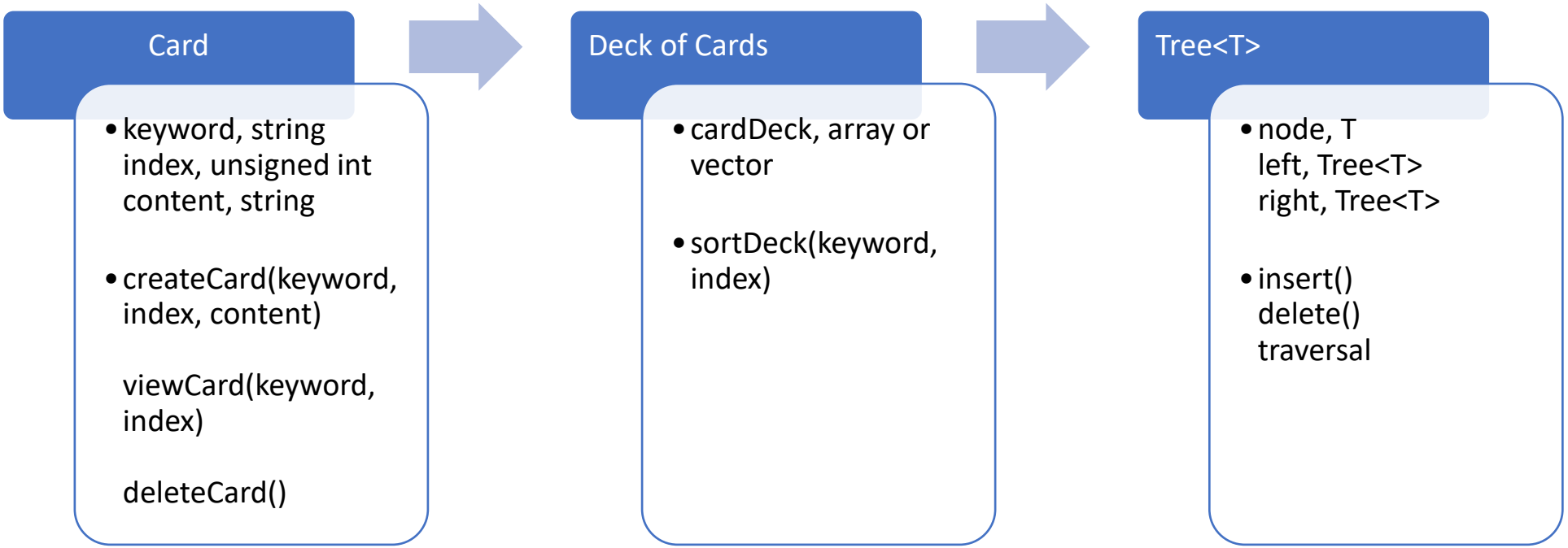
Card
• keyword, string
index, unsigned int
content, string
• createCard(keyword,
index, content)
viewCard(keyword,
index)
deleteCard()
Deck of Cards
• cardDeck, array or
vector
• sortDeck(keyword,
index)
Tree<T>
• node, T
left, Tree<T>
right, Tree<T>
• insert()
delete()
traversal

```
1  // IndexCardTree.cpp
2  // Includes source code referenced from Gaddis Starting out with C++ 8th
     edition
3
4  #include "stdafx.h"
5  #include <iostream>
6  using namespace std;
7  // Specification file for the BinaryTree class
8  // Needs to go back and change to TemplatedBinaryTree class
9  #ifndef INTBINARYTREE_H
10 #define INTBINARYTREE_H
11
12 template <class T>
13 class BinaryTree
14 {
15 private:
16      struct TreeNode
17      {
18           T value;
19           TreeNode *left;
20           TreeNode *right;
21      };
22
23      TreeNode *root;
24
25      void insert(TreeNode *&, TreeNode *&);  // *& poTer being passed by
          reference
26      void destroySubtree(TreeNode *);
27      void deleteNode(T, TreeNode *&);
28      void makeDeletion(TreeNode *&);
29      void displayInOrder(TreeNode *) const;
30      void displayPreOrder(TreeNode *) const;
31      void displayPostOrder(TreeNode *) const;
32
33 public:
34      // Constructor
35      BinaryTree()
36      {
37           root = nullptr;
38      }
39
40      // Destructor
41  /*    ~BinaryTree()
42      {
43           destroySubtree(root);
44      }*/
45
46      //Binary tree operations
47      void insertNode(T);
```

```
48        bool searchNode(T);
49        void removeNode(T);
50        void displayInOrder() const
51        {
52             displayInOrder(root);
53        }
54        void displayPreOrder() const
55        {
56             displayPreOrder(root);
57        }
58        void displayPostOrder() const
59        {
60             displayPostOrder(root);
61        }
62   };
63
64   #endif
65
66   template <class T>
67   void BinaryTree<T>::insertNode(T num)
68   {
69        TreeNode *newNode = nullptr;
70
71        // Create a new node and store num in it
72        newNode = new TreeNode;
73        newNode->value = num;
74        newNode->left = newNode->right = nullptr;
75
76        // Insert the node
77        insert(root, newNode);
78   }
79
80   template <class T>
81   void BinaryTree<T>::insert(TreeNode *&nodePtr, TreeNode *&newNode)
82   {
83        if (nodePtr == nullptr)
84             nodePtr = newNode;
85        else if (newNode->value < nodePtr->value)
86             insert(nodePtr->left, newNode); // insert left branch
87        else
88             insert(nodePtr->right, newNode); // insert right branch
89   }
90
91   // The displayInOrder member function displays the values
92   // in the subtree pointed to by nodePtr, via inorder traversal.
93   // left, root, right
94   template <class T>
95   void BinaryTree<T>::displayInOrder(TreeNode *nodePtr) const
96   {
```

```
 97      if (nodePtr)
 98      {
 99           displayInOrder(nodePtr->left);
100           cout << nodePtr->value << endl;
101           displayInOrder(nodePtr->right);
102      }
103 }
104
105 // The displayPreOrder member function displays the values
106 // in the subtree pointed to by nodePtr, via preorder traversal.
107 // root, left, right
108 template <class T>
109 void BinaryTree<T>::displayPreOrder(TreeNode *nodePtr) const
110 {
111      if (nodePtr)
112      {
113           cout << nodePtr->value << endl;
114           displayPreOrder(nodePtr->left);
115           displayPreOrder(nodePtr->right);
116      }
117 }
118
119 // The displayPostOrder member function displays the values
120 // in the subtree pointed to by nodePtr, via postorder traversal.
121 // left, right, root
122 template <class T>
123 void BinaryTree<T>::displayPostOrder(TreeNode *nodePtr) const
124 {
125      if (nodePtr)
126      {
127           displayPostOrder(nodePtr->left);
128           displayPostOrder(nodePtr->right);
129           cout << nodePtr->value << endl;
130      }
131 }
132
133 template <class T>
134 bool BinaryTree<T>::searchNode(T num)
135 {
136      TreeNode *nodePtr = root;
137
138      while (nodePtr)
139      {
140           if (nodePtr->value == num)
141                return true;
142           else if (num < nodePtr->value)
143                nodePtr = nodePtr->left;
144           else
145                nodePtr = nodePtr->right;
```

```
146       }
147
148       return false;
149  }
150
151  template <class T>
152  void BinaryTree<T>::removeNode(T num)
153  {
154       deleteNode(num, root);
155  }
156
157  template <class T>
158  void BinaryTree<T>::deleteNode(T num, TreeNode *&nodePtr)
159  {
160       if (num < nodePtr->value)
161            deleteNode(num, nodePtr->left);
162       else if (num > nodePtr->value)
163            deleteNode(num, nodePtr->right);
164       else
165            makeDeletion(nodePtr);
166  }
167
168  // makeDeletion member function deletes node from tree and
169  // reattach the deleted node's subtrees
170  template <class T>
171  void BinaryTree<T>::makeDeletion(TreeNode *&nodePtr)
172  {
173       // Define a temp pointer to use in reattaching the left subtree
174       TreeNode *tempNodePtr = nullptr;
175
176       if (nodePtr == nullptr)
177            cout << "Cannot delete empty node. \n";
178       else if (nodePtr->right == nullptr)
179       {
180            tempNodePtr = nodePtr;
181            nodePtr = nodePtr->left; // Reattach left child
182            delete tempNodePtr;
183       }
184       else if (nodePtr->left == nullptr)
185       {
186            tempNodePtr = nodePtr;
187            nodePtr = nodePtr->right; // Reattach right child
188            delete tempNodePtr;
189       }
190       // If the node has two children
191       else
192       {
193            // Move one node to the right
194            tempNodePtr = nodePtr->right;
```

```
195
196             // Go to the end of left node
197             while (tempNodePtr->left)
198                  tempNodePtr = tempNodePtr->left;
199
200             // Reattach the left subtree
201             tempNodePtr->left = nodePtr->left;
202             tempNodePtr = nodePtr;
203
204             //Reattach the right subtree
205             nodePtr = nodePtr->right;
206             delete tempNodePtr;
207        }
208   }
209
210
```