17
Trees, Heaps, and Priority Queues

Objectives

• To know how to represent and access the elements in a binary tree (§§17.2.1-17.2.2).
• To insert an element to a binary tree (§17.2.3).
• To traverse a binary tree in inorder, postorder, and preorder (§17.2.4).
• To define a binary tree class (§17.2.5).
• To know the properties of a heap (§17.3).
• To know how to add and remove elements from a heap (§§17.3.2-17.3.3).
• To implement a heap using a vector (§17.3.4).
• To implement a priority queue using a heap (§17.5).
17.1 Introduction
The preceding chapter introduced three basic data structures: linked lists, stacks, and queues. This chapter introduces three advanced data structures: binary trees, heap, and priority queues.

17.2 Binary Trees
A list, stack, or queue is a linear structure that consists of a sequence of elements. A binary tree is a hierarchical structure. It is either empty or consists of an element, called the root, and two distinct binary trees, called the left subtree and right subtree. Examples of binary trees are shown in Figure 17.1.

![Binary Trees Diagram](image)

**Figure 17.1**
Each node in a binary tree has zero, one, or two branches.

The root of a left (right) subtree of a node is called a left (right) child of the node. A node without children is called a leaf. A special type of binary tree called a binary search tree is often useful. A binary search tree (with no duplicate elements) has the property that for every node in the tree, the value of any node in its left subtree is less than the value of the node, and the value of any node in its right subtree is greater than the value of the node. The binary trees in Figure 17.1 are all binary search trees. This section is concerned with binary search trees.

17.2.1 Representing Binary Trees
A binary tree can be represented using a set of linked nodes. Each node contains a value and two links named left and right that reference the left child and right child, respectively, as shown in Figure 17.2.
A binary tree can be represented using a set of linked nodes. A node can be defined as a class, as follows:

```cpp
template<typename T>
class TreeNode
{
public:
    T element; // Element contained in the node
    TreeNode<T> * left; // Pointer to the left child
    TreeNode<T> * right; // Pointer to the right child

    TreeNode() // No-arg constructor
    {
        previous = NULL;
        next = NULL;
    }

    TreeNode(T element) // Constructor
    {
        this->element = element;
        left = NULL;
        right = NULL;
    }
};
```

The variable `root` refers to the root node of the tree. If the tree is empty, `root` is null. The following code creates the first three nodes of the tree in Figure 17.1.

```cpp
// Create the root node
TreeNode<int> *root = new TreeNode<int>(60);

// Create the left child node
root->left = new TreeNode<int>(55);

// Create the right child node
root->right = new TreeNode<int>(100);
```

### 17.2.2 Accessing Nodes in Binary Trees

Suppose a tree with three nodes is created as in the preceding section. You can access the nodes in the tree through the `root` pointer. Here are the statements to display the elements in the tree.
17.2.3 Inserting an Element into a Binary Search Tree

If a binary tree is empty, create a root node for the new element. Otherwise, locate the parent node for the new element node, as follows. If the new element is less than the parent element, the node for the new element becomes the left child of the parent. If the new element is greater than the parent element, the node for the new element becomes the right child of the parent. Here is the algorithm:

```cpp
if (root == null)
    root = new TreeNode<T>(element);
else
    // Locate the parent node
    current = root;
    while (current != null)
        if (element value < the value in current.element)
            parent = current;
            current = current->left;
        else if (element value > the value in current.element)
            parent = current;
            current = current->right;
        else
            return false; // Duplicate node not inserted
    // Create the new node and attach it to the parent node
    if (element < parent->element)
        parent->left = new TreeNode<T>(element);
    else
        parent->right = new TreeNode<T>(element);
    return true; // Element inserted
```

For example, to insert 101 into the tree in Figure 17.2, the parent is the node for 107. The new node for 101 becomes the left child of the parent. To insert 59 into the tree, the parent is the node for 57. The new node for 59 becomes the right child of the parent. Both of these insertions are shown in Figure 17.3.
17.2.4 Tree Traversal
Tree traversal is the process of visiting each node in the tree exactly once in certain order. There are several ways to traverse a tree. This section presents inorder, preorder, postorder, depth-first, and breadth-first traversals.

<Side Remark: inorder>
With inorder traversal, the left subtree of the current node is visited first, then the current node, and finally the right subtree of the current node. The inorder traversal displays all the nodes in a binary search tree in increasing order.

<Side Remark: postorder>
With postorder traversal, the left subtree of the current node is visited first, then the right subtree of the current node, and finally the current node itself.

<Side Remark: preorder>
<Side Remark: depth-first>
With preorder traversal, the current node is visited first, then the left subtree of the current node, and finally the right subtree of the current node. Depth-first traversal is the same as preorder traversal.

<Side Remark: breadth-first>
With breadth-first traversal, the nodes are visited level by level. First the root is visited, then all the children of the root from left to right, then the grandchildren of the root from left to right, and so on.

For example, in the tree in Figure 17.3, the inorder is
45 55 57 59 60 67 100 101 107

The postorder is
45 59 57 55 67 101 107 100 60

The preorder is
60 55 45 57 59 100 67 107 101

The breadth-first traversal is
17.2.5 The BinaryTree Class
Let us define a binary tree class named BinaryTree with insert, inorder traversal, postorder traversal, and preorder traversal, as shown in Figure 17.4. Its implementation is given in Listing 17.1.

**Figure 17.4**
BinaryTree implements a binary tree with operations insert, inorder, preorder, and postorder.

**Listing 17.1 BinaryTree.h (Class Interface and Implementation)**

```cpp
***PD: Please add line numbers in the following code***
<Side Remark line 5: TreeNode class>
<Side Remark line 9: left pointer>
<Side Remark line 10: right pointer>
<Side Remark line 12: TreeNode constructor>
<Side Remark line 18: TreeNode constructor>
<Side Remark line 27: BinaryTree class>
<Side Remark line 30: constructor>
<Side Remark line 32: function>
<Side Remark line 39: root>
<Side Remark line 41: private function>
<Side Remark line 47: non-arg constructor>
<Side Remark line 54: constructor>
<Side Remark line 69: insert function>
<Side Remark line 72: new root>
<Side Remark line 76: locate parent>
<Side Remark line 90: duplicate element>
<Side Remark line 94: new left child>
<Side Remark line 96: new right child>
<Side Remark line 99: increase size>
<Side Remark line 105: inorder>
<Side Remark line 112: recursive helper function>
<Side Remark line 122: postorder>
<Side Remark line 129: recursive helper function>
<Side Remark line 139: preorder>
<Side Remark line 146: recursive helper function>
```
#ifndef BINARYTREE_H
#define BINARYTREE_H

template<typename T>
class TreeNode
{
public:
    T element; // Element contained in the node
    TreeNode<T> * left; // Pointer to the left child
    TreeNode<T> * right; // Pointer to the right child

    TreeNode(); // No-arg constructor
    TreeNode(T element); // Constructor
};

template<typename T>
class BinaryTree
{
public:
    BinaryTree();
    BinaryTree(T elements[]);
    bool insert(T element);
    void inorder();
    void preorder();
    void postorder();
    int getSize();

private:
    TreeNode<T> * root;
    int size;
    void inorder(TreeNode<T> *root);
    void postorder(TreeNode<T> *root);
    void preorder(TreeNode<T> *root);
};

template<typename T>
BinaryTree<T>::BinaryTree()
{
    root = NULL;
    size = 0;
}

template<typename T>
BinaryTree<T>::BinaryTree(T elements[], int arraySize)
{
    root = NULL;
    size = 0;
    for (int i = 0; i < arraySize; i++)
    {
        insert(elements[i]);
    }
}
/* Insert element into the binary tree
* Return true if the element is inserted successfully
* Return false if the element is already in the list
*/
template<typename T>
bool BinaryTree<T>::insert(T element)
{
    if (root == NULL)
root = new TreeNode<T>(element); // Create a new root
else
{
// Locate the parent node
TreeNode<T> *parent = NULL;
TreeNode<T> *current = root;
while (current != NULL)
{
    if (element < current->element)
    {
        parent = current;
        current = current->left;
    }
    else if (element > current->element)
    {
        parent = current;
        current = current->right;
    }
    else
        return false; // Duplicate node not inserted

// Create the new node and attach it to the parent node
if (element < parent->element)
    parent->left = new TreeNode<T>(element);
else
    parent->right = new TreeNode<T>(element);
}
size++;
return true; // Element inserted

/* Inorder traversal */
template< typename T>
void BinaryTree<T>::inorder()
{
inorder(root);
}

/* Inorder traversal from a subtree */
template< typename T>
void BinaryTree<T>::inorder(TreeNode<T> *root)
{
    if (root == NULL) return;
inorder(root->left);
    cout << root->element << " ";
inorder(root->right);
}

/* Postorder traversal */
template< typename T>
void BinaryTree<T>::postorder()
{
    postorder(root);
}

/** Inorder traversal from a subtree */
template< typename T>
void BinaryTree<T>::postorder(TreeNode<T> *root)
{
    if (root == NULL) return;
    postorder(root->left);
    postorder(root->right);
    cout << root->element << " ";
}

/* Preorder traversal */
template< typename T>
void BinaryTree<T>::preorder()
{
    preorder(root);
}

/* Preorder traversal from a subtree */
template<typename T>
void BinaryTree<T>::preorder(TreeNode<T> *root)
{
    if (root == NULL) return;
    cout << root->element << " ";
    preorder(root->left);
    preorder(root->right);
}

/* Get the number of nodes in the tree */
template<typename T>
int BinaryTree<T>::getSize()
{
    return size;
}
#endif

The no-arg constructor (lines 46-51) constructs an empty binary tree with root NULL and size 0. The constructor (lines 53-62) constructs a binary tree initialized with the elements in the array.

The insert(T element) function (lines 68-101) creates a node for element and inserts it into the tree. If the tree is empty, the node becomes the root (line 72). Otherwise, the function finds an appropriate parent for the node to maintain the order of the tree. If the element is already in the tree, the function returns false (line 90); otherwise it returns true (line 100).

The inorder() function (lines 104-108) invokes inorder(root) to traverse the entire tree. The function inorder(TreeNode root) traverses the tree with the specified root. This is a recursive function. It recursively traverses the left subtree, then the root, and finally the right subtree. The traversal ends when the tree is empty.

The postorder() function (lines 121-125) and the preorder() function (lines 138-142) are implemented similarly using recursion.

Listing 17.2 gives an example that creates a binary tree for strings using BinaryTree<string> (line 8). Add strings into the binary tree (lines 9-15) and traverse the tree in inorder (line 18), postorder (line 21), and preorder (line 24).

**Listing 17.2 TestBinaryTree.cpp (Using BinaryTree)**
***PD: Please add line numbers in the following code***
<Side Remark line 3: include class>
<Side Remark line 8: create tree1>
<Side Remark line 9: insert string>
<Side Remark line 18: inorder>
<Side Remark line 21: postorder>
<Side Remark line 24: preorder>
<Side Remark line 26: getSize>
<Side Remark line 29: create tree2>
<Side Remark line 31: inorder>

#include <iostream>
#include <string>
#include "BinaryTree.h"
using namespace std;

int main()
After all the string elements are inserted, tree1 should appear as shown in Figure 17.5(a). Tree tree2 is created as shown in Figure 17.5(b).
Figure 17.5
The binary search trees are pictured here after they are created.

If the elements are inserted in a different order, the tree will look different. However, the inorder traversal prints elements in the same order as long as the set of elements is the same. The inorder traversal displays a sorted list.

17.3 Heaps

Heap is a useful data structure for designing efficient sorting algorithms and priority queues. A heap is a binary tree with the following properties:

- It is a complete binary tree.
- Each node is greater than or equal to any of its children.

A binary tree is complete if every level of the tree is full except that the last level may not be full and all the leaves on the last level are placed left-most. For example, in Figure 17.6, the binary trees in (a) and (b) are complete, but the binary trees in (c) and (d) are not complete. Further, the binary tree in (a) is a heap, but the binary tree in (b) is not a heap, because the root (39) is less than its right child (42).

Figure 17.6
A heap is a special complete binary tree.

17.3.1 Representing a Heap

A heap is a binary tree. So it can be represented using a binary tree data structure. However, a more efficient representation for a heap is using an array or a vector if the heap size is known in advance. The heap in Figure 17.7(a) can be represented using an array in Figure 17.7(b). The root is at position 0, and its two children are at positions 1 and 2. For a node at position \( i \), its left child is at position \( 2i+1 \) and its right child is at position \( 2i+2 \), and its parent is \( (i-1)/2 \). For example, the node for element 39 is at position 4, so its left child (element 14) is at 9 \( (2\times4+1) \), its right child (element 33) is at 10 \( (2\times4+2) \), and its parent (element 42) is at 1 \( ((4-1)/2) \).
A binary heap can be implemented using an array.

If the heap size is not known in advance, it is better to use a vector to store a heap.

17.3.2 Removing the Root

Often you need to remove the max element which is the root in a heap. After the root is removed, the tree must be rebuilt to maintain the heap property. The algorithm for building the tree can be described as follows:

```
Move the last node to replace the root;
Let the root be the current node;
while (the current node has children and the current node is smaller than one of its children)
    Swap the current node with the larger of its children;
    Now the current node is one level down;
```

Figure 17.8 shows the process of rebuilding a heap after the root 62 is removed from Figure 17.7(a). Move the last node 9 to the root, as shown in Figure 17.8(a). Swap 9 with 59 as shown in Figure 17.8(b). Swap 9 with 44 as shown in Figure 17.8(c). Swap 9 with 30 as shown in Figure 17.8(d).
(a) After moving 9 to the root

(b) After swapping 9 with 59

(c) After swapping 9 with 44

(d) After swapping 9 with 30

Figure 17.8

Rebuild the heap after the root is removed.

17.3.3 Adding a New Node

To add a new node to the heap, first add it to the end of the heap and then rebuild the tree as follows:

```
Let the last node be the current node;
while (the current node is greater than its parent)
{   Swap the current node with its parent;
    Now the current node is one level up;
}
```

Figure 17.9 shows the process of rebuilding a heap after adding a new node 88 to the heap in Figure 17.8(d). Place the new node 88 at the end of the tree as shown in Figure 17.9(a). Swap 88 with 13 as shown in Figure 17.9(b). Swap 88 with 44 as shown in Figure 17.9(c). Swap 88 with 59 as shown in Figure 17.9(d).
17.3.4 The Heap Class

Now you are ready to design and implement the Heap class. The class diagram is shown in Figure 17.10. Its implementation is given in Listing 17.10.

**Figure 17.10**

Heap provides operations for manipulating a heap.

<table>
<thead>
<tr>
<th>Heap&lt;T&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>-vector: vector&lt;T&gt;</td>
</tr>
<tr>
<td>+Heap()</td>
</tr>
<tr>
<td>+Heap(elements: T[], arraySize: int)</td>
</tr>
<tr>
<td>-remove(): T</td>
</tr>
<tr>
<td>+add(element: T): void</td>
</tr>
<tr>
<td>+getSize(): int</td>
</tr>
</tbody>
</table>

*Figure 17.9*

Rebuild the heap after adding a new node.
#ifndef HEAP_H
#define HEAP_H
#include <vector>
#include <stdexcept>
using namespace std;

template<typename T>
class Heap
{
public:
    Heap();
    Heap(T elements[], int arraySize);
    T remove() throw (runtime_error);
    void add(T element);
    int getSize();

private:
    vector<T> v;
};

template<typename T>
Heap<T>::Heap()
{
}

template<typename T>
Heap<T>::Heap(T elements[], int arraySize)
{
    for (int i = 0; i < arraySize; i++)
    {
        add(elements[i]);
    }
}

/* Remove the root from the heap */
template<typename T>
T Heap<T>::remove() throw (runtime_error)
{
    if (v.size() == 0)
        throw runtime_error("Heap is empty");
    T removedElement = v[0];
    v[0] = v[v.size() - 1]; // Move the last to root
    v.pop_back(); // Remove root
    // Maintain the heap property
    int currentIndex = 0;
    while (currentIndex < v.size())
    {
        int leftChildIndex = 2 * currentIndex + 1;
        int rightChildIndex = 2 * currentIndex + 2;
        if (leftChildIndex < v.size())
            v[currentIndex] = v[leftChildIndex];
        if (rightChildIndex < v.size())
            v[currentIndex] = v[rightChildIndex];
        if (v[currentIndex] < removedElement)
            v[currentIndex] = removedElement;
        currentIndex = leftChildIndex;
    }
    return removedElement;
}
// Find the maximum between two children
if (leftChildIndex >= v.size()) break; // The tree is a heap
int maxIndex = leftChildIndex;
if (rightChildIndex < v.size())
{
if (v[maxIndex] < v[rightChildIndex])
maxIndex = rightChildIndex;
}

// Swap if the current node is less than the maximum
if (v[currentIndex] < v[maxIndex])
{
T temp = v[maxIndex];
v[maxIndex] = v[currentIndex];
v[currentIndex] = temp;
currentIndex = maxIndex;
}
else
break; // The tree is a heap

return removedElement;

/* Insert element into the heap and maintain the heap property */
template<typename T>
void Heap<T>::add(T element)
{
v.push_back(element); // Append element to the heap
int currentIndex = v.size() - 1; // The index of the last node

// Maintain the heap property
while (currentIndex > 0)
{
int parentIndex = (currentIndex - 1) / 2;
// Swap if the current element is greater than its parent
if (v[currentIndex] > v[parentIndex])
{
T temp = v[currentIndex];
v[currentIndex] = v[parentIndex];
v[parentIndex] = temp;
}
else
break; // the tree is a heap now

currentIndex = parentIndex;
}

/* Get the number of element in the heap */
template<typename T>
tinline Heap<T>::getSize()
{
return v.size();
}
#endif

A heap is represented using a vector internally (line 18). You may change it to other data structures, but the Heap class contract will remain unchanged (see Exercise 17.5).

The remove() function (lines 36-77) removes and returns the root. To maintain the heap property, the function moves the last element to the root position and swaps it with its larger child if it is less than the larger child. This process continues until the last element becomes a leaf or it is not less than its children.
The `add(T element)` function (lines 80-102) appends the element to the tree and then swaps it with its parent if it is greater than its parent. This process continues until the new element becomes the root or it is not greater than its parent.

Listing 17.4 gives an example of using a heap to sort strings and numbers. The program creates a heap in line 8, adds strings to the heap (lines 9-14), and displays all strings in the heap (lines 16-17). The program creates a heap from an array of integers (line 21) and displays all elements in the heap in decreasing order (lines 22-23).

17.4 Priority Queues
A regular queue is a first-in and first-out data structure. Elements are appended to the end of the queue and are removed from the beginning of the queue. In a priority queue, elements are assigned with priorities. The element with the highest priority is accessed or removed first. A priority queue has a largest-in, first-out behavior. For example, the emergency room in a hospital assigns patients with priority numbers; the patient with the highest priority is treated first.

A priority queue can be implemented using a heap, where the root is the element with the highest priority in the queue. The class diagram for the priority queue is shown in Figure 17.11. Its implementation is given in Listing 17.5.

A regular queue is a first-in and first-out data structure. Elements are appended to the end of the queue and are removed from the beginning of the queue. In a priority queue, elements are assigned with priorities. The element with the highest priority is accessed or removed first. A priority queue has a largest-in, first-out behavior. For example, the emergency room in a hospital assigns patients with priority numbers; the patient with the highest priority is treated first.

A priority queue can be implemented using a heap, where the root is the element with the highest priority in the queue. The class diagram for the priority queue is shown in Figure 17.11. Its implementation is given in Listing 17.5.

A priority queue can be implemented using a heap, where the root is the element with the highest priority in the queue. The class diagram for the priority queue is shown in Figure 17.11. Its implementation is given in Listing 17.5.
template<typename T>
T PriorityQueue<T>::dequeue() throw (runtime_error)
{
    return heap.remove();
}

template<typename T>
int PriorityQueue<T>::getSize()
{
    return heap.getSize();
}

Listing 17.6 gives an example of using a priority queue for patients. The Patient class is declared in lines 18-34. Four patients are created with associated priority values in lines 3-6. Line 8 creates a priority queue. The patients are enqueued in lines 9-12. Line 15 dequeues a patient from the queue.

Listing 17.6 TestPriorityQueue.cpp (Testing Priority Queue)

***PD: Please add line numbers in the following code***
<Side Remark line 2: include PriorityQueue>
<Side Remark line 5: Patient class>
<Side Remark line 8: Patient constructor>
<Side Remark line 14: overload < operator>
<Side Remark line 19: overload > operator>
<Side Remark line 24: getName>
<Side Remark line 29: getPriority>
<Side Remark line 42: create a priority queue>
<Side Remark line 43: add to queue>
<Side Remark line 50: remove from queue>

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

class Patient
{
public:
    Patient(string name, int priority)
    {
        this->name = name;
        this->priority = priority;
    }

    bool operator<(Patient &secondPatient)
    {
        return (this->priority < secondPatient.priority);
    }

    bool operator>(Patient &secondPatient)
    {
        return (this->priority > secondPatient.priority);
    }

    string getName()
    {
        return name;
    }

    int getPriority()
    {
        return priority;
    }
};
private:
    string name;
    int priority;
};

int main()
{
    // Queue of patients
    PriorityQueue<Patient> patientQueue;
    patientQueue.enqueue(Patient("John", 2));
    patientQueue.enqueue(Patient("Tim", 5));
    patientQueue.enqueue(Patient("Cindy", 7));

    while (patientQueue.getSize() > 0)
    {
        Patient element = patientQueue.dequeue();
        cout << element.getName() << " (priority: " << element.getPriority() << ") " << endl;
    }

    return 0;
}

<COutput>
Cindy(prioriy: 7) Tim(prioriy: 5) John(prioriy: 2) Jim(prioriy: 1)
</End Output>

The < and > operators are defined the Patient class so two patients can be compared. You can use any class type for the elements in the heap, provided that the elements can be compared using the < and > operators.

Key Terms
***PD: Please place terms in two columns same as in intro5e.

- binary search tree
- binary tree
- heap
- inorder traversal
- postorder traversal
- preorder traversal
- priority queue
- tree traversal

Chapter Summary

- A binary tree can be implemented using linked nodes. Each node contains the element value and two pointers that point to the left and right children.
- Tree traversal is the process of visiting each node in the tree exactly once in certain order. There are several ways to traverse a tree.
- With inorder traversal, the left subtree of the current node is visited first, then the current node, and finally the right subtree of the current node. The inorder traversal displays all the nodes in a binary search tree in increasing order.
• With postorder traversal, the left subtree of the current node is visited first, then the right subtree of the current node, and finally the current node itself.

• With preorder traversal, the current node is visited first, then the left subtree of the current node, and finally the right subtree of the current node. Depth-first traversal is the same as preorder traversal.

• With breadth-first traversal, the nodes are visited level by level. First the root is visited, then all the children of the root from left to right, then the grandchildren of the root from left to right, and so on.

• Heap is a useful data structure for designing efficient sorting algorithms and priority queues. A heap is a binary tree with two properties: (1) It is a complete binary tree; (2) Each node is greater than or equal to any of its children.

• You can implement a heap using a binary tree or an array or a vector.

• A regular queue is a first-in and first-out data structure. Elements are appended to the end of the queue and are removed from the beginning of the queue. In a priority queue, elements are assigned with priorities. When accessing elements, the element with the highest priority is removed first. A priority queue has a largest-in, first-out behavior.

• A priority queue can be implemented using a heap, where the root is the element with the highest priority in the queue.

Review Questions

Section 17.2 Binary Trees

17.1
If a set of the same elements is inserted into a binary tree in two different orders, will the two corresponding binary trees look the same? Will the inorder traversal be the same? Will the postorder traversal be the same? Will the preorder traversal be the same?

17.2
What is wrong if the following two highlighted lines are deleted in the following constructor for BinaryTree?

```
template <typename T>
BinaryTree<T>::BinaryTree(T elements[], int arraySize)
{
    root = NULL;
    size = 0;
    for (int i = 0; i < arraySize; i++)
    {
        insert(elements[i]);
    }
}
```
Add the elements 4, 5, 1, 2, 9, 3 into a binary tree in this order. Draw the diagrams to show the binary tree as each element is added.

17.4
Show the inorder, postorder, preorder, and breadth-first traversals for the binary tree in Figure 17.1.

Section 17.3 Heaps
17.5
What is a complete binary tree? What is a heap? Describe how to remove the root from a heap and how to add a new object to a heap.

17.6
What is the return value from invoking the remove function if the heap is empty?

17.7
Add the elements 4, 5, 1, 2, 9, 3 into a heap in this order. Draw the diagrams to show the heap as each element is added.

17.8
Show the heap after the root in the heap in Figure 17.9(d) is removed.

Section 17.6 Priority Queues
17.9
What is a priority queue?

Programming Exercises
Section 17.2 Binary Trees
17.1*
(Adding new functions in BinaryTree) Add the following new functions in BinaryTree.

```c
/* Search element in this binary tree */
bool search(T element)

/* Display the nodes in breadth-first traversal */
void breadthFirstTraversal()

/* Return the depth of this binary tree. Depth is the */
/* number of the nodes in the longest path of the tree */
int depth()
```

17.2**
(Implementing inorder traversal using a stack) Implement the inorder function in BinaryTree using a stack instead of recursion.

17.3**
(Sorting using a heap) Implement the following sort function using a heap.

```c
void sort(int list[], int arraySize)
```