Practical Tasks 6 and 7

Due dates:

- 09:00pm Friday, August 31, 2018 (Parts 1.1 and 1.2 of the main task only)
- 09:00pm Friday, September 7, 2018 (Part 1.3 of the main task)

(Very strict, late submission is not allowed)

General Instructions

1. You may need to read Chapter 11 of SIT232 Workbook available in Resources → Course materials → SIT232 Workbook to remind / study Generics and their application in object-oriented programming. You may also need to read Chapter 6 of SIT232 Workbook about Polymorphism and Interfaces as you need excellent understanding of these topics to progress well through the practical tasks of the course. Make sure you are proficient with them as they form a basis to design and develop programming modules in this and all the subsequent tasks. You may find other important topics required to complete the task, like exceptions handling, in other chapters of the workbook.
2. This time, your task is to realize a binary search tree that can be used to collect and store elements of generic type T. Read Chapter 9 of SIT221 Workbook available in CloudDeakin in Resources → Course materials → SIT221 Workbook to learn this data structure. Alternatively, you may explore chapters 8.2, 8.3, and relevant sections of 8.4 of the SIT221 course book "Data structures and algorithms in Java" (2014) by M. Goodrich, R. Tamassia, and M. Goldwasser. You may access the book on-line for free from the reading list application in CloudDeakin available in Resources → Course materials → Course Book: Data structures and algorithms in Java.
3. Download the zip file called "Practical Tasks 6 and 7(template)" attached to this task. It contains a template for the program that you need to work on. Open the solution file in Microsoft Visual Studio. There are two connected projects inside: DataStructures_Algorithms and Runner. There is one file only in Week06 subfolder that provides enumeration type for traversal mode for the binary search tree. The objective of the practical task is to add a new class implementing the data structure. There are no constraints on how to design and implement the new module internally, but it must meet all the requirements of the task in terms of functionality accessible by a user.

   The second project has Runner06_Task1.cs, which is new. It implements IRunner interface and acts as a main method in case of this practical task by means of the Run(string[] args) method. Make sure that you set the right runner class in Program.cs. You may modify this class as you need as its main purpose is to test the implementation of the binary search tree for its correctness. However, it has already a good structure to check the data structure. Therefore, you should first explore the content of the method and see whether it satisfies your goals in terms of testing. Some lines are commented out to make the current version compilable, so make sure you make them active as you progress with implementation of different methods of the required binary search tree class. In general, you should use this class and its method to thoroughly test your code. Your aim here is to cover all

potential logical issues and runtime errors. This (testing) part of the task is as important as writing the code. We will later on replace your runner by our special versions in order to test functionality of your code for common mistakes.

Objectives

1. Learn implementation of a Binary Search Tree data structure

Specification

Main task

In this practical session, your goal is to implement a new generic data structure known as Binary Search Tree (BST) capable to maintain arbitrary number of data elements. Similarly to the linked list, the BST relies on a node, a simple data structure which acts as a building block for the whole tree. In fact, the BST is a collection of nodes, whose order is not given by their physical positions in memory but based on links connecting parent nodes to their child nodes. Every node consists of a data record (called value) that holds valuable information to be stored, a special (unique) key, and two auxiliary pointers referring to the left and the right child of the node in the tree. The two pointers make the navigation from the parent to its child nodes possible. To simplify implementation of the BST, the node may also contain a reference to its parent node so that navigation is possible in both directions: from parents to child nodes and vice versa. Binary search trees keep their keys in sorted order, so that lookup and other operations can use the principle of binary search: when looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, on the basis of the comparison, to continue searching in the left or right subtrees. On average, this means that each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree. This is much better than the linear time required to find items by key in an (unsorted) array.

Task 1. Implementation of a tree node data structure

To succeed with this assignment, you must develop a new generic class called “BinarySearchTree”. However, you must first focus on the node data structure implementing it via a generic private class (so it must be built-in) inside the larger BinarySearchTree<T> class. Therefore, in the Week06 subfolder, create a new public class called “BinarySearchTree” and inside it create another ***private*** class called “BSTNode” capable to store a value of generic type TYPE. Ensure that it provides the following functionality to a user.

BSTNode(int Key, TYPE Value, BSTNode< TYPE > Parent)	Initializes a new instance of the BSTNode<TYPE >class, containing the specified value, key, and reference no the parent node. The Parent property is set to null if the node is the top most node (the root) of the binary search tree. LeftChild and RightChild properties must be initialized to null.

Value	Property. Gets or sets the value of type TYPE contained in the node.
Key	Property. Gets or sets the integer key assigned to the node.
LeftChild	Property. Gets or sets the reference to the left child node of the current BSTNode<TYPE> in a binary search tree. LeftChild property is set to null if the current BSTNode<TYPE> is a leaf; i.e., it is the bottom most one in its branch.
RightChild	Property. Gets or sets the reference to the right child node of the current BSTNode<TYPE> in a binary search tree. LeftChild property is set to null if the current BSTNode<TYPE> is a leaf; i.e., it is the bottom most one in its branch.
Parent	Property. Gets or sets the reference to the parent node of the current BSTNode<TYPE> node. Parent property is set to null if the BSTNode<TYPE> is the top most node (the root) of the binary search tree.

Note that since the BSTNode<TYPE> is a private class of the BinarySearchTree<T>, none of its properties are visible to the code outside of the BinarySearchTree<T>; that is, you are to be the only user of the class. Because of this, all the properties of the BSTNode<TYPE> can be declared as public, while the class itself gets the ***private*** access modifier.

Now, complete the implementation of BSTNode<T> with regard to its properties and move forward to realization of the BinarySearchTree<T> class.

Task 2. Implementation of the Binary Search Tree

In this part of the practical task, you must complete the BinarySearchTree<T> class, whose object must maintain arbitrary number of data elements as a collection of nodes and provide the following functionality to a user.

BinarySearchTree ()	Initializes a new instance of the BinarySearchTree<T> class that is empty. If the BinarySearchTree<T> is empty, the Count property is set to 0, the MinKey property is set to int.MaxValue, and the MaxKey equals to int.MinValue. This constructor is an $O(1)$ operation.
Count	Property. Gets the number of nodes contained in the BinarySearch-Tree<T>. Retrieving the value of this property is an $O(1)$ operation.
MinKey	Property. Gets the minimum integer key presented in the current instance of the BinarySearchTree<T>. If the BinarySearchTree<T> is empty, the MinKey property contains int.MaxValue. Retrieving the value of this property is an $O(1)$ operation.
MaxKey	Property. Gets the maximum integer key presented in the current instance of the BinarySearchTree<T>. If the BinarySearchTree<T> is empty, the MinKey property contains int.MinValue. Retrieving the value of this property is an $O(1)$ operation.
void Add(int Key, T Value)	Adds a new tree node containing the specified value to the BinarySearchTree<T>. The position of the new node is determined according to the specified key following the binary search tree policy. As the binary search tree may degenerate into a linked list, this method is an $O(\mathrm{n})$ operation, where n is Count.

void Clear()	Removes all nodes from the BinarySearchTree<T>. Count is set to zero, and the MinKey and MaxKey properties are set to int.MaxValue and int.MinValue, respectively. This method is an $O(1)$ operation.
T Find(int Key)	Finds the node associated with the specified key and returns its value. This method returns the default value for type T, i.e. default(T), if the BinarySearchTree<T> does not contain the integer key. As the binary search tree may degenerate into a linked list, this method is an $O(\mathrm{n})$ operation, where n is Count.
bool Contains(int Key)	Determines whether the specified key is in the BinarySearchTree<T>. It returns true if such key exists; otherwise, false. If the BinarySearch-Tree<T> is empty, it returns false, too. As the binary search tree may degenerate into a linked list, this method is an $O(\mathrm{n})$ operation, where n is Count.
T[] Traverse(TraversalMode mode)	Traverses BinarySearchTree<T> according the specified mode (Pre-order, In-order, or Post-order) and returns the array of data elements placed in the nodes following the order in which these nodes were examined. This method is an $O(\mathrm{n})$ operation, where n is Count. If BinarySearchTree<T> is empty, then the resulting array is to be empty (note that this is different to null), too.

To traverse the tree as well as to support all the other operations, you need to keep the starting point; that is, a reference to the root node. Actually, it is better to realize this reference via a private property, say Root, in the BinarySearchTree<T>. It should keep track of the top most node of the tree and set it to null when the BinarySearchTree<T> is empty. Furthermore, note that the Traverse(TraversalMode mode) method accepts an argument which is represented via the enumeration type TraversalMode developed in TraversalMode.cs. Thus, there are exactly three ways to do traversal. Remember that Pre-order mode explores the tree studying a node first, then calls the Pre-order traversal for its left child followed by the Pre-order traversal call for the right child. In-order mode calls the In-order traversal of the left child, then considers the node, then completes with the In-order traversal of the right child. Finally, Post-order first proceeds with the Post-order traversal call for the left child, then for the right child, and completes with investigating the node itself.

Task 3. Implementation of the Remove operations for nodes of a binary search tree

In this last task, finalise the BinarySearchTree<T> class by adding the following function accessible by a user.

bool Remove(int Key)	Removes the data element related to the specified key from the BinarySearchTree<T>. It returns false if key was not found in the BinarySearchTree<T>; otherwise, true. This method is an $O(\mathrm{n})$ operation, where n is Count.

For simplicity, assume that only unique keys are to be stored in the BinarySearchTree<T>. Ensure that the Remove operation works correctly when a node is deleted at a leaf, at the root, or in the middle of the tree so that the binary search tree property is maintained, which imposes that the key of the left child is less than the key of the right child for every node presented in the tree.

Additional Task

These additional questions will not be marked, but we strongly recommend you to answer them as some of them will definitely appear in your final exam. If you seek feedback, your solutions must be submitted as a separate PDF/DOC document. Where you need to draw trees and heaps, your solutions can be hand-written and scanned.

1. If we would like to represent our data in ascending order, what traversal mode should we choose?

2. Assume you are given a list of first $n = 2k + 1$ positive integers, $\mathrm{k} \in \mathbb{N}$, as

$$L = (k + 1,1, \dots, k, k + 2, \dots, 2k + 1).$$

   For example, $L = (4,1,2,3,5,6,7)$ for $n = 2 \cdot 3 + 1$. Draw a binary search tree resulted from inserting these integers in the order of the list. What is the best, worst and average time for the searching operation?

3. Assume you have an array-based binary heap A with the contents:

$$1; 4; 7; 8; 9; 10; 14; 12; 15; 13; 17; 12$$

   Show the contents of A after each of the following two operations.
   - Insert 5 into A.
   - Delete the least element from A.

   Show your work for each operation including the content of the list at its intermediate stages. You can assume A is large enough to contain all the values inserted.

4. Is it right or wrong that in a heap of depth A, there must be at least 2^d elements. (Assume the depth of the first element (or root) is zero). If it is right, provide your prove. If it is wrong, explain why you think so.

5. Draw a sequence of diagrams showing the insertion of the values:

$$[11,4,2,7,19,9,20]$$

   into an empty AVL tree. You must:
   - Show the resulting tree immediately after each insertion step (that is before any balancing has taken place).
   - Indicate the node(s) at which each rotation is performed.
   - Where there is a double rotation, show the tree after each single rotation.
   - Show the resulting tree after balancing operation(s).

6. Draw a sequence of diagrams showing the deletion of the values:

$$[20,9,19,7,2,4,11]$$

   from the tree built in the question above. The deletions must occur in the given order. Again, show the tree after each deletion and rotation (if any).

7. What order should we insert the elements $\{1,2, \dots,7\}$ into an empty AVL tree so that we do not have to perform any rotations on it?

Submission instructions

Submit your program code as an answer to the main task via the CloudDeakin System by the deadline. You should zip your BinarySearchTree.cs file only and name the zip file as PracticalTask6.zip (should contain solutions to parts 1.1 and 1.2) or PracticalTask7.zip (should contain solution for the whole task, including part 1.3). Please, **make sure the file names are correct**. You **must not submit** the whole MS Visual Studio project / solution files.

Marking scheme

You will get 1 mark for outstanding work, 0.5 mark only for reasonable effort, and zero for unsatisfactory work, which means that most of your methods fail to work correctly and return the right result, or any compilation errors.

To achieve **at least partial marks in week 7**, your Remove method must work correctly when deleting a node at a leaf of the tree or when the node to be deleted has exactly one child.