MECHTRON 2MD3
Data Structures and Algorithms for Mechatronics
Winter 2022

# Assignment 03

Published: April 4, 2022
Due at 11:59pm on April 12th, 2022.

*Please read the following carefully:*

- Late submissions will be marked with a penalty of 20% per day.
- The answer to Questions 1 must be submitted as a source code file (cpp or zip file containing the source code), possibly with the same name as the base implementation. Do not submit the source code as PDF file, otherwise you will receive a mark of **zero**.
- Include your name and student ID number in the source code files. You can add a comment on top of the source code indicating your name and student ID.
- **Your work must be your own. Plagiarism and copying will not be tolerated. This is very important!**
- The answer is to be submitted via Avenue to Learn.
- Please ensure you have read the question carefully and watched the video in which I described the code.

This assignment consists of 1 question, and is worth 80 marks.

## Q 1 Modifying and Extending the Binary Search Tree [80 marks]

I uploaded an implementation of the binary search tree in the file `A3_BST.cpp` along with this assignment. We already discussed many details of implementing a binary search tree during the lectures. The implementation is closely related to those topics. I also provided a video describing the given C++ implementation. This assignment aims to familiarize you with the details of implementing a search tree and enable you to make modifications to it and add extra functionalities to it.

In the first part of this assignment, you are asked to modify the current implementation in a way that instead of storing integer **keys** on the nodes, the nodes can store **both the key and a list of values associated with that key**. It can be implemented by defining a new class and encapsulating the key

and list of values. You can then define getters and setters to access and modify keys and values. For simplicity, suppose that the keys are only integers and the associated values are integers as well.

In the resulting tree, we will have **no duplicate keys**. Instead, when inserting a key/value pair $(k, v)$ for which $k$ is already present in the tree, we will add the associated value to the list of values. Thus, you have to modify the insert function to store the given value in the list of values (currently, it skips the value). Remember that, after this modification, the resulting binary search tree will still work based on keys, and all decisions and computations in the tree will only consider keys. You have to make **all** required changes to the implementation to realize this modification.

The second part of the task is to add extra functions to the implementation of the binary search tree that is given with this assignment. You should edit **ONLY** the file `A3_BST.cpp` and implement the following functions:

1. Add a function `min()` that returns an iterator to the node having the minimum key stored in the tree.

2. Add a function `max()` that returns an iterator to the node having the maximum key stored in the tree.

3. Add a function `rank(k)` that returns the rank of the key $k$ given as parameter, in the sequence of nodes visited by the in-order traversal. You will need a helper function to count the number of nodes in a subtree that is rooted at a given node. Refer to Tutorial 10 for more details. Your implementation should throw an appropriate exception if $k$ is not in the tree.

4. Add a function `floor(k)` that returns an iterator to the node having the largest key that is less than or equal to $k$.

5. Add a function `ceil(k)` that returns an iterator to the node having the smallest key that is greater than or equal to $k$.

6. Provide a public method `eraseMin()` that removes the node with the minimum key.

7. Provide a public method `eraseMax()` that removes the node with the maximum key.

8. Implement the exception `NonExistentElement` and use it whenever is needed.

9. Add a function `selectAtRank(i)` that returns an iterator to the node that is at rank $i$ of the in-order traversal.

10. Add a function `countKeysBetween(l,h)` that returns the number of keys in the tree that are in the range $[l, h]$ inclusive.

11. Add a function `entriesBetween(l,h)` that returns list of nodes (or an iterator to those nodes) that has keys in the range $[l, h]$ inclusive. Hint: you may use a Queue to implement this.

12. In the `SearchTree::Iterator`, add a new method `nextInPostorder()` that when called on an iterator, returns an iterator to the node that is the next node in the post-order traversal. This method should not alter the iterator and should simply return a new iterator, so that after calling this method the iterator remains intact. You may need to change other parts of iterator's implementation.

13. In the `SearchTree::Iterator`, add a new method `nextInLevelorder()` that when called on an iterator, returns an iterator to the node that is the next node in the level-order traversal. This method should not alter the iterator and should simply return a new iterator, so that after calling this method the iterator remains intact. You may need to change other parts of iterator's implementation.

14. Add a function `median()` that returns an iterator to the node having the median key that is stored in the tree. Hint: the median key is the entry with rank $\lfloor \frac{n}{2} \rfloor$.

15. Add a function `height()` to the binary search tree that returns it's height. You should only consider the nodes storing real key/value pairs.

16. Add a function `depth(const SearchTree::Iterator& p)` returns the depth of the node that iterator $p$ is on it. You should only consider the nodes storing real key/value pairs, and skip the super root.

Some **important** notes to remember:

- many of the functions are already discussed in the lectures or tutorials and for some of them, even an implementation is given (check tutorial folders). But, those implementations were based on a minimal binary search tree ADT. The C++ implementation in `A3_BST.cpp` is more accurate and you have to realize above functions extending only `A3_BST.cpp`.

- I advise you to think carefully about the functions above as some of them can be implemented just by proper calling of other functions.

- some of the functions above may be implemented as a combination of a public and a utility function.

- The functions above have to be defined as public member functions. You are responsible for making appropriate design choices (such as defining extra functions that are needed as utility functions).