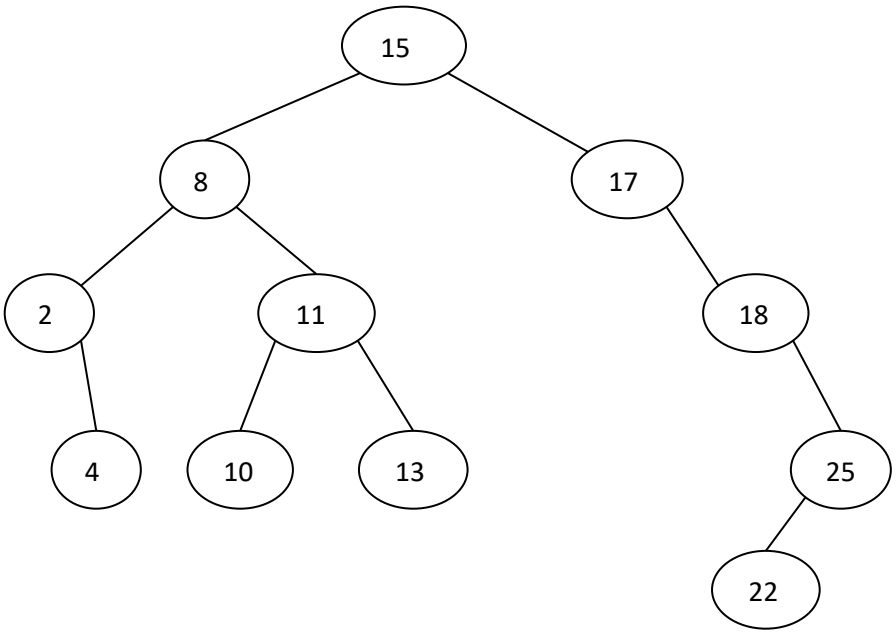
European Funds Knowledge Education Development
Wrocław University of Science and Technology
European Union European Social Fund

ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej

Laboratory – **List 8**

Introduction:

Binary Search tree is a data structure to store data in an ordered way. The structure is built from top (from a root). The is a rule about relation between elements (here called node) in left and right subtree of any node. Nodes in left subtree have to be smaller of the element in a node and nodes in right subtree have to be bigger than the element. The place of equal elements depend of the assumption. They can be in any subtree, it can be prohibited to add the same element or equal elements are stored in one node which is a complex structure (e.a. a list).



During insertion we are looking for a place for a new element. The new element have to be inserted as a new leaf. In the above example if we want to insert a new element with a key 12 we start from a root 15, then we go to 8, than to 11, than to 13 and finally we will find a place to insert 12 as a left child to 13.
During deletion if the node has up to one child the node is simple removed from the structure. If a node, which have to be removed has two children, we have to exchange it with it successor and remove the node which posses before the successor. In the above example if we want to delete a node with the key 8, we have to exchange it with 10 and then delete the node which posses 10 before exchange.
There are two ways to store a node in the memory. One way is to remember (beside element and probably the key) references to left child, right child and a parent. The second one is to not remember a reference to a parent. In the below list of task will be used the first idea.

To check the state it is good to implement traversal of the tree. It refers to the process of visiting (checking and/or updating) each node in a tree data structure, exactly once. There are 3 standard traversal in a tree: in-order, pre-order, post-order. In each of them in the procedure we visit the current node and its left and right substree. The difference is only when we visit the current node: as a first (pre-order), between subtrees (in-order) or as a last (post-order).
In the above example the sequence of visited nodes will be:
Pre-order: 15,8,2,4,11,10,13,17,18,25,22
In-order:2,4,8,10,11,13,15,17,18,22,25
Post-order: 4,2,10,13,11,8,22,25,18,17,15

List of tasks.

1. Take the implementation of class `HashTable`, `Link` and interface `IWithName` from previous list of tasks. You have to modify the way you collect the information about links in the `Document` class. In this list links are stored in a binary search tree.
2. Implement generic class BST which has the possibility to add and remove an element. The elements have to be stored in a node with additional fields `left`, `right` and `parent`. Base on the first implementation from the lecture. There is a need to have also method which will return strings with nodes presentation in possible orders: in-order, pre-order and post-order.
3. It is assumed that objects stored in BST implements `Comparable` interface, so you have to cast to `Comparable` during `add` and `remove` operation to use `compareTo` method.
4. In the class BST implement function `size()` which will return the number of stored element and method `successor` which will return the successor of an argument.

For 100 points present solutions for this list till Week 10.
For 80 points present solutions for this list till Week 11.
For 50 points present solutions for this list till Week 12.
After Week 12 the list is closed.

There is a next page…

Appendix

The solution will be automated tested with tests from console of presented below format. The test assumes, that there are up to X different documents, which there are created as the first operation in the test. Each document can be constructed separately.

If a line is empty or starts from `'#'` sign, the line have to be ignored.
In any other case, your program should print an exclamation mark and write (copy) introduced a line and then, depending on the command follow the correct procedure / function.

If the current document is equal **`null`**, the program has to write "no current document".

If a line has a format:
`ch <str>`
your program has to choose a document with a name `str`, and all next functions will operate on this document.

If a line has a format:
`ld <name>`
your program has to call a constructor of a document with parameters `name` and `scan` for current position. The `name` have to be a correct identificator (in the constructor the `name` have to be stored in lower case letters). If it is not true, write in one line "`incorrect ID`". If the identificator is correct, the loaded document have to be now the current document. Try to add it to the hashtable. If it is impossible write "error" on the output.

If a line has a format:
`add <str>`
your program has to call `add(new Link(str))` for the current document's links. The result of the addition has to be written in one line. The `str` can be in the same format like a link in this task: only identificator or identificator with a weight in parenthesis.

If a line has a format:
`get str`
your program has to call `getElement(new Link(str))` for current document's links and write returned `Link` object. If the `str` is incorrect, write in one line "error".

If a line has a format:
`clear`
your program has to call `clear()` for current document links.

If a line has a format:
`show`
your program has to write on the screen the result of caling `toString()` for the current document's links. The first line has to present text "Document: <name>", the rest of lines – an in-order list of links separated by come and one space.

If a line has a format:
`preorder`

your program has to write on the screen the result of caling `toStringPreOrder()` for the current document's links. The first line has to present text "Document: <name>", the second line – a pre-order list of links separated by come and one space.

If a line has a format:
`postorder`
your program has to write on the screen the result of caling `toStringPostOrder()` for the current document's links. The first line has to present text "Document: <name>", the second line – a post-order list of links separated by come and one space.

If a line has a format:
`rem <str>`
your program has to call `remove(newLink(str))` for current document's links and write on the screen returned value in one line.

If a line has a format:
`successor <str>`
your program has to call `successor(newLink(str))` for current document's links and write on the screen returned value in one line.

If a line has a format:
`size`
your program has to write on screen the result of `size()` for current document's links and write on the screen returned value in one line.

If a line has a format:
`ht`
your program has to call `toString()` for hash table with documents.

If a line has a format:
`ha`
your program has to end the execution, writing as the last line "END OF EXECUTION". Every test ends with this line.

A simple test for this task:
INPUT:

```
#Test for Lab8
show
ld first
link=g(10)
eod
add e(11)
show
add k(12)
add a(10)
add f(10)
add i(10)
add n(10)
show
preorder
postorder
rem q
```

```
rem g
show
preorder
postorder
size
ha
```

OUTPUT:

```
START
!show
no current document
!ld first
!add e(11)
true
!show
Document: first
e(11), g(10)
!add k(12)
true
!add a(10)
true
!add f(10)
true
!add i(10)
true
!add n(10)
true
!show
Document: first
a(10), e(11), f(10), g(10), i(10), k(12), n(10)
!preorder
Document: first
g(10), e(11), a(10), f(10), k(12), i(10), n(10)
!postorder
Document: first
a(10), f(10), e(11), i(10), n(10), k(12), g(10)
!rem q
error
!rem g
g(10)
!show
Document: first
a(10), e(11), f(10), i(10), k(12), n(10)
!preorder
Document: first
i(10), e(11), a(10), f(10), k(12), n(10)
!postorder
Document: first
a(10), f(10), e(11), n(10), k(12), i(10)
!size
6
!ha
END OF EXECUTION
```