ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

Εργασία Δεύτερη

Μάθημα : Δομές Δεδομένων

Υπεύθυνος Καθηγητής : Χαράλαμπος Κωνσταντόπουλος

Πανεπιστήμιο Πειραιά

Τμήμα Πληροφορικής

Κωνσταντίν Ταχίρη Π10127

Κωνσταντίνος Γκρέκας Π09028

Περιεχόμενα

Εισαγωγή

Στη παρούσα εργασία, απαντώνται τα ερωτήματα της εκφώνησης, δηλαδή τα Α, Β και Γ. Παραδίδεται, δηλαδή, η υλοποίηση του δυαδικού δέντρου αναζήτησης, το οποίο λειτουργεί ταυτόχρονα και ως δέντρο μεγίστων. Στον πηγαίο κώδικα, περιλαμβάνονται οι λειτουργίες εισαγωγής και διαγραφής, καθώς και οι συναρτήσεις Find_second_next και Print_between. Επιπλέον, παραδίδονται και οι λειτουργίες της προδιάταξης, της ενδοδιάταξης, της μεταδιάταξης, καθώς και της διάσχισης του δέντρου ανά επίπεδα, για τον ευκολότερο έλεγχο της ορθότητας των λειτουργιών του δέντρου. Τα αρχεία, που δημιουργήθηκαν για την λειτουργία της υλοποίησης του συγκεκριμένου δέντρου, είναι τα εξής : BnPTree.h, BnPTree.cpp και main.cpp. Παρακάτω, θα παρατίθεται ο πηγαίος κώδικας, χωρίς σχόλια, ώστε να είναι εύκολα αναγνώσιμος, αλλά περιλαμβάνεται μια λεπτομερής περιγραφή των τεχνικών που χρησιμοποιήθηκαν. Για να είναι δυνατή η μελέτη του εκτελέσιμου κώδικα του προγράμματος, μαζί με τα σχόλια, καθώς και η εκτέλεση του ίδιου του προγράμματος, τα αρχεία περιλαμβάνονται σε CD, το οποίο παραδίδεται μαζί με την εκτυπωμένη εργασία.

Πηγαίος Κώδικας

Παρακάτω, παρατίθεται ο κώδικας του αρχείου BnPTree.h :

```
#ifndef BSMTREE_H
#define BSMTREE_H

class BSMTree {
    private:
        typedef struct TreeNode {
            int priority;
            int element;
            TreeNode* leftChild;
            TreeNode* rightChild;
        }* TreeNodePtr;

        TreeNodePtr rootNode;
        TreeNodePtr currNode;
        TreeNodePtr parentNode;
        TreeNodePtr tempNode;

        void PreOrder(TreeNodePtr t);
        void InOrder(TreeNodePtr t);
        void PostOrder(TreeNodePtr t);
        void LeverOrder(TreeNodePtr t);
        void InOrderEditedV1(TreeNodePtr t, int RefElement, int
```

```
&counter, bool &secondNextFound, int &element);
        void InOrderEditedV2(TreeNodePtr t, int startNum, int endNum,
bool &numsExist, bool &numsCease);

    public:
        BSMTree ();
        bool AddTreeNode(int addPriority, int addElement);
        bool DeleteTreeNode(int delElement);
        void PrintPreOrder();
        void PrintInOrder();
        void PrintPostOrder();
        void PrintLevelOrder();
        int Find_second_next(int refElement);
        void Print_between(int startNum, int endNum);
};

#endif // BSMTREE_H
```

Εν συνεχεία, παρατίθεται ο κώδικας του αρχείου BnPTree.cpp :

```
#include "BSMTree.h"
#include <iostream>
#include <queue>
using namespace std;

BSMTree::BSMTree() {
    rootNode = NULL;
    currNode = NULL;
```

```
    parentNode = NULL;
    tempNode = NULL;
}

bool BSMTree::AddTreeNode(int addPriority, int addElement) {
    bool helpingVar = true;
    bool tempParExists = false;
    if (rootNode != NULL) {
        currNode = rootNode;
        while (currNode) {
            parentNode = currNode;
            if (currNode->element > addElement) {
                currNode = currNode->leftChild;
            } else if (currNode->element < addElement) {
                currNode = currNode->rightChild;
            } else {
                cout << "Element " << addElement << " already exists!"
<< endl;
                helpingVar = false;
                break;
            }
        }
        if (helpingVar) {
            TreeNodePtr NewNode = new TreeNode;
            NewNode->priority = addPriority;
            NewNode->element = addElement;
            NewNode->leftChild = NULL;
            NewNode->rightChild = NULL;
```

```
if (NewNode->element < parentNode->element) {
    parentNode->leftChild = NewNode;
} else {
    parentNode->rightChild = NewNode;
}
while (NewNode->priority > parentNode->priority) {
    currNode = rootNode;
    if (parentNode->element != rootNode->element) {
        tempParExists = true;
        while (currNode->element != parentNode->element) {
            tempNode = currNode;
            if (currNode->element < parentNode->element) {
                currNode = currNode->rightChild;
            } else {
                currNode = currNode->leftChild;
            }
        }
    }
    if (tempParExists) {
        if (parentNode->element < tempNode->element) {
            tempNode->leftChild = NewNode;
        } else {
            tempNode->rightChild = NewNode;
        }
    }

    if (NewNode->element < parentNode->element) {
        if (NewNode->rightChild) {
```

```
                    parentNode->leftChild = NewNode->rightChild;
                } else {
                    parentNode->leftChild = NULL;
                }
                NewNode->rightChild = parentNode;
            } else {
                if (NewNode->leftChild) {
                    parentNode->rightChild = NewNode->leftChild;
                } else {
                    parentNode->rightChild = NULL;
                }
                NewNode->leftChild = parentNode;
            }

            if (tempParExists) {
                parentNode = tempNode;
            } else {
                rootNode = NewNode;
                break;
            }
        }
        return true;
    } else {
        return false;
    }
} else {
    TreeNodePtr NewNode = new TreeNode;
    NewNode->priority = addPriority;
```

```
            NewNode->element = addElement;
            NewNode->leftChild = NULL;
            NewNode->rightChild = NULL;
            rootNode = NewNode;
            return true;
        }
    }

bool BSMTree::DeleteTreeNode(int delElement) {
    bool elementFound = false;
    bool hasParent = false;
    if (rootNode) {
        currNode = rootNode;
        tempNode = currNode;
        while (tempNode) {
            currNode = tempNode;
            if (tempNode->element > delElement) {
                tempNode = tempNode->leftChild;
            } else if (tempNode->element < delElement) {
                tempNode = tempNode->rightChild;
            } else {
                elementFound = true;
                break;
            }
        }
        if (elementFound) {
            while ((currNode->leftChild) || (currNode->rightChild)) {
                if (currNode->element != rootNode->element) {
```

```
                    hasParent = true;
                    tempNode = rootNode;
                    parentNode = tempNode;
                    while (tempNode->element != currNode->element) {
                        parentNode = tempNode;
                        if (tempNode->element > delElement) {
                            tempNode = tempNode->leftChild;
                        } else {
                            tempNode = tempNode->rightChild;
                        }
                    }
                }

                if (currNode->leftChild && currNode->rightChild) {
                    if (currNode->leftChild->priority >= currNode-
>rightChild->priority) {
                        tempNode = currNode->leftChild;
                        if (tempNode->rightChild) {
                            currNode->leftChild = tempNode->rightChild;
                        } else {
                            currNode->leftChild = NULL;
                        }
                        tempNode->rightChild = currNode;
                    } else {
                        tempNode = currNode->rightChild;
                        if (tempNode->leftChild) {
                            currNode->rightChild = tempNode->leftChild;
                        } else {
```

```
                    currNode->rightChild = NULL;
                }
                tempNode->leftChild = currNode;
            }
        } else {
            if (currNode->leftChild) {
                tempNode = currNode->leftChild;
                if (tempNode->rightChild) {
                    currNode->leftChild = tempNode->rightChild;
                } else {
                    currNode->leftChild = NULL;
                }
                tempNode->rightChild = currNode;
            } else if (currNode->rightChild) {
                tempNode = currNode->rightChild;
                if (tempNode->leftChild) {
                    currNode->rightChild = tempNode->leftChild;
                } else {
                    currNode->rightChild = NULL;
                }
                tempNode->leftChild = currNode;
            }
        }

        if (hasParent) {
            if (parentNode->leftChild && (parentNode-
>leftChild->element == currNode->element)) {
                parentNode->leftChild = tempNode;
```

```
                    } else {
                        parentNode->rightChild = tempNode;
                    }
                } else {
                    rootNode = tempNode;
                }
            }

            if (tempNode->leftChild && (tempNode->leftChild->element ==
currNode->element)) {
                tempNode->leftChild = NULL;
            } else {
                tempNode->rightChild = NULL;
            }
            delete currNode;
            return true;
        } else {
            cout << "Element " << delElement << " does not exist..." <<
endl;
            return false;
        }
    } else {
        cout << "Element " << delElement << " cannot be found! The tree
is already empty..." << endl;
        return false;
    }
}
```

```
void BSMTree::PreOrder(TreeNodePtr t) {
    if (t) {
        cout << "\t" << t->priority << "," << t->element;

        if (t->leftChild) {
            PreOrder(t->leftChild);
        }

        if (t->rightChild) {
            PreOrder(t->rightChild);
        }
    }
}

void BSMTree::InOrder(TreeNodePtr t) {
    if (t) {
        if (t->leftChild) {
            InOrder(t->leftChild);
        }

        cout << "\t" << t->priority << "," << t->element;

        if (t->rightChild) {
            InOrder(t->rightChild);
        }
    }
}
```

```
void BSMTree::PostOrder(TreeNodePtr t) {
    if (t) {
        if (t->leftChild) {
            PostOrder(t->leftChild);
        }

        if (t->rightChild) {
            PostOrder(t->rightChild);
        }

        cout << "\t" << t->priority << "," << t->element;
    }
}

void BSMTree::LeverOrder(TreeNodePtr t) {
    queue <TreeNodePtr> q;
    while (t) {
        cout << "\t" << t->priority << "," << t->element;

        if (t->leftChild) {
            q.push(t->leftChild);
        }

        if (t->rightChild) {
            q.push(t->rightChild);
        }

        if (!q.empty()) {
```

```
            t = q.front();
            q.pop();
        } else {
            t = NULL;
        }
    }
}

void BSMTree::InOrderEditedV1(TreeNodePtr t, int refElement, int
&counter, bool &secondNextFound, int &element) {
    if (t) {
        if (t->leftChild) {
            InOrderEditedV1(t->leftChild, refElement, counter,
secondNextFound, element);
        }

        if (t->element > refElement) {
            counter++;
            if (counter == 2) {
                cout << "The second next element, greater than the
element " << refElement << ", is " << t->element << ".\n";
                secondNextFound = true;
                element = t->element;
            }
        }

        if (secondNextFound == false) {
            if (t->rightChild) {
```

```
                InOrderEditedV1(t->rightChild, refElement, counter,
secondNextFound, element);
            }
        }
    }
}

void BSMTree::InOrderEditedV2(TreeNodePtr t, int startNum, int endNum,
bool &numsExist, bool &numsCease) {
    if (t) {
        if (t->leftChild) {
            InOrderEditedV2(t->leftChild, startNum, endNum, numsExist,
numsCease);
        }

        if (t->element > endNum) {
            numsCease = true;
        }

        if (!numsCease) {
            if (t->element >= startNum && t->element <= endNum) {
                numsExist = true;
                cout << t->element << "\t";
            }

            if (t->rightChild) {
                InOrderEditedV2(t->rightChild, startNum, endNum,
numsExist, numsCease);
```

```
            }
        }
    }
}

void BSMTree::PrintPreOrder() {
    if (rootNode) {
        cout << "The Pre-Order of the tree, is the following : " <<
endl;
        PreOrder(rootNode);
        cout << endl;
    } else {
        cout << "The tree is empty..." << endl;
    }
}

void BSMTree::PrintInOrder() {
    if (rootNode) {
        cout << "The In-Order of the tree, is the following : " <<
endl;
        InOrder(rootNode);
        cout << endl;
    } else {
        cout << "The tree is empty..." << endl;
    }
}

void BSMTree::PrintPostOrder() {
```

```
    if (rootNode) {
        cout << "The Post-Order of the tree, is the following : " <<
endl;
        PostOrder(rootNode);
        cout << endl;
    } else {
        cout << "The tree is empty..." << endl;
    }
}


void BSMTree::PrintLevelOrder() {
    if (rootNode) {
        cout << "The Level Order of the tree, is the following : " <<
endl;
        LeverOrder(rootNode);
        cout << endl;
    } else {
        cout << "The tree is empty..." << endl;
    }
}

int BSMTree::Find_second_next(int refElement) {
    if (rootNode) {
        bool nodeExists = false;
        currNode = rootNode;
        while (currNode) {
            if (currNode->element > refElement) {
```

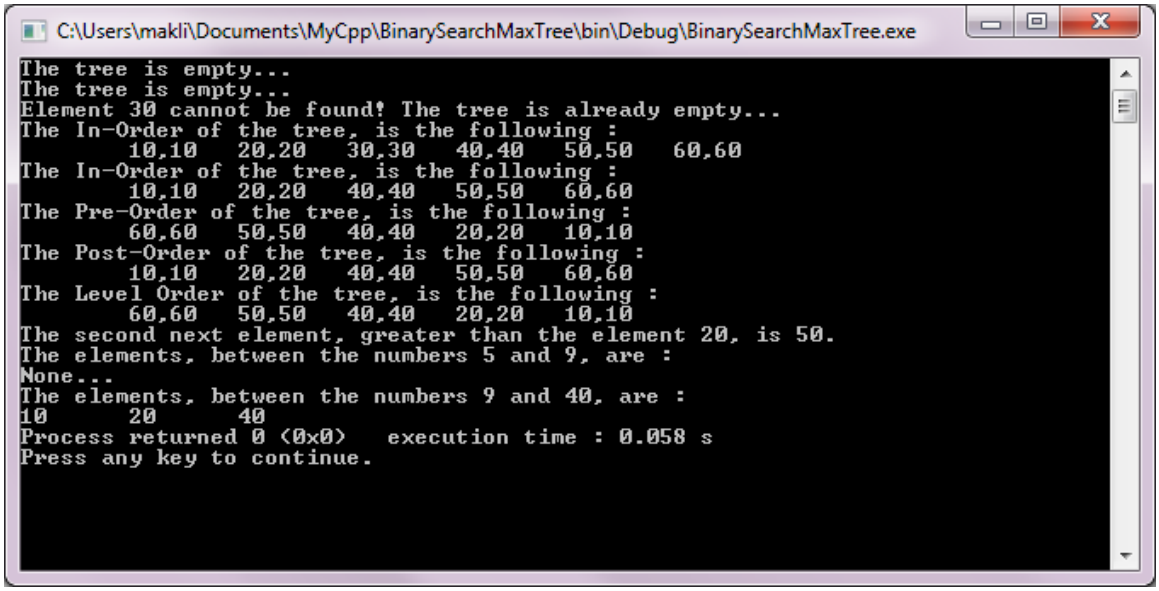
```
                currNode = currNode->leftChild;
            } else if (currNode->element < refElement) {
                currNode = currNode->rightChild;
            } else {
                nodeExists = true;
                break;
            }
        }

        bool secondNextFound = false;
        int counter = 0;
        int element = 0;

        if (nodeExists) {
            InOrderEditedV1(rootNode, refElement, counter,
secondNextFound, element);
            if (secondNextFound) {
                return element;
            } else {
                cout << "There is not second next element, of the
element " << refElement << "..." << endl;
                return -1;
            }
        } else {
            cout << "Element " << refElement << " does not exist..." <<
endl;
            return -1;
        }
    } else {
```

```
        cout << "The tree is empty..." << endl;
        return -1;
    }
}

void BSMTree::Print_between(int startNum, int endNum) {
    if (rootNode) {
        bool numsExist = false;
        bool numsCease = false;
        if (startNum < endNum) {
            cout << "The elements, between the numbers " << startNum <<
" and " << endNum << ", are :\n";
            InOrderEditedV2(rootNode, startNum, endNum, numsExist,
numsCease);
            if (!numsExist) {
                cout << "None..." << endl;
            }
        } else {
            cout << "Invalid values inserted..." << endl;
        }
    } else {
        cout << "The tree is empty..." << endl;
    }
}
```

Τέλος, παρατίθεται ενδεικτικά ο κώδικας του αρχείου main.cpp, όπου ενδείκνυται ο τρόπος χρήσης των συναρτήσεων της κλάσης BnPTree :

```
#include "BSMTree.h"
#include <iostream>
#include <queue>
using namespace std;

int main() {
    BSMTree makli;
    makli.Find_second_next(20);
    makli.Print_between(5, 9);
    makli.DeleteTreeNode(30);
    makli.AddTreeNode(10, 10);
    makli.AddTreeNode(20, 20);
    makli.AddTreeNode(30, 30);
    makli.AddTreeNode(40, 40);
    makli.AddTreeNode(50, 50);
    makli.AddTreeNode(60, 60);
    makli.PrintInOrder();
    makli.DeleteTreeNode(30);
    makli.PrintInOrder();
    makli.PrintPreOrder();
    makli.PrintPostOrder();
    makli.PrintLevelOrder();
    makli.Find_second_next(20);
    makli.Print_between(5, 9);
    makli.Print_between(9, 40);
    return 0;
}
```

Το αποτέλεσμα, που θα εμφανισθεί στην οθόνη είναι το ακόλουθο :



Όπως προαναφέρθηκε, έχουν υλοποιηθεί οι λειτουργίες της προδιάταξης, της ενδοδιάταξης, της μεταδιάταξης, καθώς και της διάσχισης του δέντρου ανά επίπεδο (δηλαδή οι συναρτήσεις InOrder, PreOrder, PostOrder και LevelOrder, οι οποίες παίρνουν ως όρισμα έναν κόμβο – δείκτη, ως σημείο αναφοράς από το οποίο πρέπει να ξεκινήσει η διάσχιση). Οι συναρτήσεις αυτές είναι ιδιωτικές (private), για αυτό το λόγο χρησιμοποιούνται δημόσιες (public) συναρτήσεις, οι οποίες καλούν τις προηγούμενες συναρτήσεις, και οι οποίες μπορούν να χρησιμοποιηθούν στη συνάρτηση main() (δηλαδή οι αντίστοιχες συναρτήσεις PrintInOrder, PrintPreOrder, PrintPostOrder και PrintLevelOrder). Οι συναρτήσεις αυτές δημιουργήθηκαν με σκοπό την εύκολη κατανόηση της ορθότητας του δυαδικού δέντρου αναζήτησης – δέντρου μεγίστων.

Επίσης, στο αρχείο BSMTree.h, δηλώνεται μία δομή, η οποία αποτελεί τη δομή του κόμβου του δυαδικού δέντρου, περιλαμβάνοντας προτεραιότητα, τιμή στοιχείου, καθώς και δύο δείκτες προς τα δύο παιδιά, αριστερό και δεξί. Για τη δημιουργία ενός κενού δυαδικού δέντρου, δηλώνεται μια συνάρτηση κατασκευής (constructor), της οποίας η λειτουργία ορίζεται στο αρχείο BSMTree.cpp, δηλαδή η αρχικοποίηση των βοηθητικών δεικτών με τιμή NULL (συμπεριλαμβανομένου και του δείκτη προς τη ρίζα του δέντρου, ο οποίος εφόσον το δέντρο είναι κενό, παίρνει

και αυτός τη τιμή NULL).

Στο πρόγραμμα, ορίζεται η συνάρτηση Find_second_next, η οποία είναι τύπου int. Σε περίπτωση, όπου βρεθεί το επιθυμητό στοιχείο, αυτό επιστρέφεται. Σε αντίθετη περίπτωση, δηλαδή η συνάρτηση αποτύχει να βρει το επιθυμητό στοιχείο, γίνεται η παραδοχή ότι θα επιστρέφεται η τιμή -1. Γίνεται η παραδοχή, ότι η τιμή -1 είναι ο κωδικός λάθους, που επιστρέφεται από την συνάρτηση σε περίπτωση σφάλματος ή αποτυχίας.

Αξίζει να σημειωθεί, ότι οι συναρτήσεις Find_second_next, Print_between, PrintPreOrder, PrintInOrder, PrintPostOrder και η PrintLevelOrder, είναι δημόσιες συναρτήσεις οι οποίες χρησιμοποιούν τις αντίστοιχες ιδιωτικές συναρτήσεις InOrderEditedV1, InOrderEditedV2, PreOrder, InOrder, PostOrder και LevelOrder, οι οποίες έχουν αναδρομικό χαρακτήρα, δηλαδή μέσα στο σώμα τους, καλούν τους εαυτούς τους με διαφορετικά ορίσματα κάθε φορά. Οι δημόσιες, σε αντίθεση με τις αντίστοιχες ιδιωτικές, μπορούν να χρησιμοποιηθούν από το χρήστη, στο main αρχείο. Σημειώνεται επίσης, ότι ειδικά για τη διάσχιση ανά επίπεδα, γίνεται χρήση ουράς. Αυτό εξηγεί και την εντολή « #include <queue> », που υπάρχει στο BSMTree.cpp καθώς και στο main αρχείο. Καθώς, ελέγχεται κάθε κόμβος, γίνεται ένας επιπλέον έλεγχος της ύπαρξης των παιδιών του, και αν αυτά υπάρχουν, μπαίνουν στην ουρά χρησιμοποιώντας την εντολή push(). Όταν χρειαστεί να διαβαστεί κάποιο από τα στοιχεία της ουράς (η κεφαλή της ουράς πάντα), αυτό το στοιχείο διαβάζεται και στη συνέχεια με την εντολή pop(), αφαιρείται από την ουρά.

Στο αρχείο main.cpp, καλούνται οι συναρτήσεις, καθώς έτσι ενδείκνυται ο τρόπος χρήσης τους. Σε περίπτωση, κλήσης μιας συνάρτησης, η οποία δεν θα φέρει κάποιο ουσιαστικό αποτέλεσμα στο δέντρο, εμφανίζεται σχετικό, επεξηγηματικό μήνυμα.

Ο σκοπός, για τον οποίο παρατίθεται όλος ο κώδικας, είναι για να υπάρχει μια ολοκληρωμένη εικόνα του πηγαίου κώδικα. Παρακάτω, θα παρατίθεται αποκλειστικά το τμήμα κώδικα, το οποίο αφορά το κάθε ερώτημα, και το οποίο θα περιγράφεται διεξοδικά.

Ερώτημα Α

Κώδικας Ερωτήματος Α

Παρακάτω, παρατίθεται το τμήμα του κώδικα, που αφορά το ερώτημα Α, δηλαδή τις λειτουργίες εισαγωγής και διαγραφής στο δυαδικό δέντρο αναζήτησης, το οποίο λειτουργεί και ως δέντρο μεγίστων :

```
bool BSMTree::AddTreeNode(int addPriority, int addElement) {
    bool helpingVar = true;
    bool tempParExists = false;
    if (rootNode != NULL) {
        currNode = rootNode;
        while (currNode) {
            parentNode = currNode;
            if (currNode->element > addElement) {
                currNode = currNode->leftChild;
            } else if (currNode->element < addElement) {
                currNode = currNode->rightChild;
            } else {
                cout << "Element " << addElement << " already exists!"
<< endl;
                helpingVar = false;
                break;
```

```
    }
}
if (helpingVar) {
    TreeNodePtr NewNode = new TreeNode;
    NewNode->priority = addPriority;
    NewNode->element = addElement;
    NewNode->leftChild = NULL;
    NewNode->rightChild = NULL;
    if (NewNode->element < parentNode->element) {
        parentNode->leftChild = NewNode;
    } else {
        parentNode->rightChild = NewNode;
    }
    while (NewNode->priority > parentNode->priority) {
        currNode = rootNode;
        if (parentNode->element != rootNode->element) {
            tempParExists = true;
            while (currNode->element != parentNode->element) {
                tempNode = currNode;
                if (currNode->element < parentNode->element) {
                    currNode = currNode->rightChild;
                } else {
                    currNode = currNode->leftChild;
                }
            }
        }
        if (tempParExists) {
            if (parentNode->element < tempNode->element) {
```

```
            tempNode->leftChild = NewNode;
        } else {
            tempNode->rightChild = NewNode;
        }
    }

    if (NewNode->element < parentNode->element) {
        if (NewNode->rightChild) {
            parentNode->leftChild = NewNode->rightChild;
        } else {
            parentNode->leftChild = NULL;
        }
        NewNode->rightChild = parentNode;
    } else {
        if (NewNode->leftChild) {
            parentNode->rightChild = NewNode->leftChild;
        } else {
            parentNode->rightChild = NULL;
        }
        NewNode->leftChild = parentNode;
    }

    if (tempParExists) {
        parentNode = tempNode;
    } else {
        rootNode = NewNode;
        break;
    }
```

```
            }
            return true;
        } else {
            return false;
        }
    } else {
        TreeNodePtr NewNode = new TreeNode;
        NewNode->priority = addPriority;
        NewNode->element = addElement;
        NewNode->leftChild = NULL;
        NewNode->rightChild = NULL;
        rootNode = NewNode;
        return true;
    }
}

bool BSMTree::DeleteTreeNode(int delElement) {
    bool elementFound = false;
    bool hasParent = false;
    if (rootNode) {
        currNode = rootNode;
        tempNode = currNode;
        while (tempNode) {
            currNode = tempNode;
            if (tempNode->element > delElement) {
                tempNode = tempNode->leftChild;
            } else if (tempNode->element < delElement) {
                tempNode = tempNode->rightChild;
```

```
            } else {
                elementFound = true;
                break;
            }
        }
        if (elementFound) {
            while ((currNode->leftChild) || (currNode->rightChild)) {
                if (currNode->element != rootNode->element) {
                    hasParent = true;
                    tempNode = rootNode;
                    parentNode = tempNode;
                    while (tempNode->element != currNode->element) {
                        parentNode = tempNode;
                        if (tempNode->element > delElement) {
                            tempNode = tempNode->leftChild;
                        } else {
                            tempNode = tempNode->rightChild;
                        }
                    }
                }

                if (currNode->leftChild && currNode->rightChild) {
                    if (currNode->leftChild->priority >= currNode-
>rightChild->priority) {
                        tempNode = currNode->leftChild;
                        if (tempNode->rightChild) {
                            currNode->leftChild = tempNode->rightChild;
                        } else {
```

```
                currNode->leftChild = NULL;
            }
            tempNode->rightChild = currNode;
        } else {
            tempNode = currNode->rightChild;
            if (tempNode->leftChild) {
                currNode->rightChild = tempNode->leftChild;
            } else {
                currNode->rightChild = NULL;
            }
            tempNode->leftChild = currNode;
        }
    } else {
        if (currNode->leftChild) {
            tempNode = currNode->leftChild;
            if (tempNode->rightChild) {
                currNode->leftChild = tempNode->rightChild;
            } else {
                currNode->leftChild = NULL;
            }
            tempNode->rightChild = currNode;
        } else if (currNode->rightChild) {
            tempNode = currNode->rightChild;
            if (tempNode->leftChild) {
                currNode->rightChild = tempNode->leftChild;
            } else {
                currNode->rightChild = NULL;
            }
```

```
                    tempNode->leftChild = currNode;
                }
            }

            if (hasParent) {
                if (parentNode->leftChild && (parentNode-
>leftChild->element == currNode->element)) {
                    parentNode->leftChild = tempNode;
                } else {
                    parentNode->rightChild = tempNode;
                }
            } else {
                rootNode = tempNode;
            }
        }

        if (tempNode->leftChild && (tempNode->leftChild->element ==
currNode->element)) {
            tempNode->leftChild = NULL;
        } else {
            tempNode->rightChild = NULL;
        }
        delete currNode;
        return true;
    } else {
        cout << "Element " << delElement << " does not exist..." <<
endl;
        return false;
```

```
        }
    } else {
        cout << "Element " << delElement << " cannot be found! The tree
is already empty..." << endl;
        return false;
    }
}
```

Ανάλυση Κώδικα Ερωτήματος Α

Καταρχάς, η συνάρτηση AddTreeNode, είναι τύπου bool, δηλαδή επιστρέφει τιμές false ή true, ανάλογα με την επιτυχία προσθήκης του νέου κόμβου. Δέχεται δύο ορίσματα, τη τιμή του στοιχείου και την προτεραιότητα, που έχει κάθε κόμβος του δέντρου. Στη συνέχεια, ορίζονται δύο μεταβλητές, τύπου bool, η helpingVar και η tempParExists, οι οποίες χρησιμεύουν, ώστε να γίνει έλεγχος ύπαρξης κόμβου στο δέντρο με ίδια τιμή στοιχείου, και για την ύπαρξη γονέα του κόμβου που πρόκειται να περιστραφεί, ώστε να πάρει τη θέση του ο νέος κόμβος που προστίθεται στο δέντρο, και να γίνει το νέο παιδί του γονικού κόμβου, αντίστοιχα. Έπειτα, γίνεται έλεγχος ύπαρξης της ρίζας του δέντρου. Σε περίπτωση, που δεν υπάρχει ρίζα, δηλαδή το δέντρο είναι κενό, ο νέος κόμβος γίνεται ρίζα. Σε αντίθετη περίπτωση, γίνεται μια δυαδική διάσχιση του δέντρου, ώστε να βρεθεί ο γονέας του νέου κόμβου, με τη βοήθεια ενός βρόγχου while. Σε περίπτωση, που βρεθεί ήδη υπάρχον στοιχείο, ίσο με το στοιχείο του κόμβου που πρόκειται να προστεθεί, η μεταβλητή helpingVar παίρνει την τιμή false, ώστε στην επόμενη εντολή διακλάδωσης if, να μην επιτραπεί η διαδικασία εισαγωγής του νέου κόμβου. Εφόσον, το πρόγραμμα εξέλθει από το βρόγχο while, ο δείκτης currNode θα είναι ίσος με NULL, ενώ ο

δείκτης parentNode, θα δείχνει προς τον γονέα του νέου κόμβου. Θα δημιουργηθεί ο νέος κόμβος, και θα γίνει αριστερό ή δεξί παιδί του γονικού κόμβου, ανάλογα με την τιμή του στοιχείου του. Στη συνέχεια, θα αρχίσει η διαδικασία των περιστροφών. Όσο υπάρχει ο γονικός κόμβος του νέου στοιχείου, του οποίου η προτεραιότητα είναι μικρότερη από την προτεραιότητα του νέου κόμβου, θα γίνεται περιστροφή. Η διαδικασία αυτή επιτυγχάνεται με τη χρήση του βρόγχου while. Ξεκινώντας, πρέπει να ελεγχθεί αν υπάρχει γονέας του γονικού κόμβου, ώστε μετά την περιστροφή, το νέο του παιδί να γίνει ο νέος κόμβος. Γίνεται έλεγχος με την εντολή if (parentNode->element != rootNode->element), και εφόσον αυτή η συνθήκη είναι αληθής, εξάγεται το συμπέρασμα ότι υπάρχει γονέας του parentNode. Τότε, η μεταβλητή tempParExists παίρνει την τιμή true. Έτσι, με τον επόμενο βρόγχο while, αρχίζει η αναζήτηση του γονέα του γονικού κόμβου. Ο γονέας αυτός, όταν βρεθεί, θα αναγνωρίζεται καθώς ο δείκτης tempNode θα δείχνει προς αυτόν. Στη συνέχεια, με έναν απλό έλεγχο, χρησιμοποιώντας την μεταβλητή tempParExists, γίνεται ο ορισμός του νέου παιδιού του κόμβου tempNode. Έπειτα, γίνεται η περιστροφή μεταξύ των δύο κόμβων, καθώς και ο προσδιορισμός της σωστής θέσης των υποδέντρων, σε περίπτωση που αυτά υπάρχουν, ώστε να συνεχίζουν να τηρούνται οι βασικοί κανόνες, που διέπουν τη δομή του δυαδικού δέντρου αναζήτησης. Τέλος, γίνεται επαναπροσδιορισμός του νέου γονικού κόμβου του νεοεισαχθέντος κόμβου. Σε περίπτωση, που ο δείκτης parentNode δεν έχει ένδειξη προς κάποιον κόμβο, σημαίνει ότι ο νέος κόμβος αποτελεί πλέον τη ρίζα του δέντρου, οπότε το πρόγραμμα εξέρχεται από τον βρόγχο while. Ο έλεγχος αυτός γίνεται με την εντολή if και την συνθήκη της, (!parentNode), ενώ η έξοδος από το βρόγχο while σε περίπτωση, που δεν υπάρχει γονέας, γίνεται με την εντολή break.Το πρόγραμμα, με εξαίρεση αυτή τη περίπτωση, εξέρχεται από το βρόχο while όταν πια ο γονέας του νέου κόμβου πάψει να έχει μεγαλύτερη προτεραιότητα από ότι ο νέος κόμβος, όπως και προαναφέρθηκε.

Η συνάρτηση DeleteTreeNode, τύπου bool, παίρνει ένα μόνο

όρισμα, την τιμή του στοιχείου που πρόκειται να διαγραφεί. Ορίζονται δύο μεταβλητές, τύπου bool, οι elementFound και hasParent, οι οποίες αρχικοποιούνται με την τιμή false. Με την πρώτη εντολή if, ελέγχεται η περίπτωση το δέντρο να είναι κενό (δηλαδή ελέγχεται η ύπαρξη της ρίζας), στην οποία εμφανίζεται αντίστοιχο μήνυμα και επιστρέφεται η τιμή false. Αν υπάρχει ρίζα, γίνεται δυαδική αναζήτηση, για την εύρεση του στοιχείου, που επιλέχθηκε να διαγραφεί. Αν δεν βρεθεί, εμφανίζεται αντίστοιχο μήνυμα λάθους (η συνάρτηση και σε αυτήν την περίπτωση επιστρέφει τιμή false). Αν βρεθεί, η τιμή της μεταβλητής elementFound γίνεται true, και με τον αντίστοιχο έλεγχο ύπαρξης του κόμβου προς διαγραφή, ακολουθεί η διαδικασία περιστροφών, έως ότου ο κόμβος γίνει φύλλο, οπότε και διαγράφεται εύκολα. Η διαδικασία είναι αντίστροφη της εισαγωγής. Με τη χρήση ενός βρόγχου while, του οποίου η συνθήκη είναι η ((currNode->leftChild) || (currNode->rightChild)), γίνονται συνεχείς επαναλήψεις των περιστροφών, μέχρι να μην υπάρχει ούτε αριστερό, ούτε δεξί παιδί. Με λογική αντίστροφη από την προηγούμενη, ελέγχεται η περίπτωση να μην υπάρχει γονέας του κόμβου που πρόκειται να διαγραφεί. Σε περίπτωση, που υπάρχει, μετά την περιστροφή, ο κόμβος προς διαγραφή, παύει να είναι παιδί του, και τη θέση του παίρνει ο κόμβος, που προηγουμένως ήταν παιδί του κόμβου προς διαγραφή. Οι περιστροφές, συνεχίζονται όσο υπάρχει τουλάχιστον ένα παιδί του κόμβου προς διαγραφή. Μετά την εύρεση του γονέα του κόμβου προς διαγραφή, ακολουθούν διακλαδώσεις περιπτώσεων ύπαρξης παιδιών, και επιλογή του κατάλληλου παιδιού, για την περιστροφή, καθώς και ο σωστός ορισμός των υποδέντρων των δύο εμπλεκόμενων κόμβων, αν αυτά υπάρχουν. Όταν το πρόγραμμα, εξέλθει από το βρόγχο while, σημαίνει ότι ο κόμβος προς διαγραφή αποτελεί πλέον φύλλο, οπότε με την εντολή if διακλάδωσης, που ακολουθεί μετά την έξοδο του βρόγχου while, ο κόμβος αυτός, αρχικά αποκόπτεται από το υπόλοιπο δέντρο, και στη συνέχεια ελευθερώνεται ο χώρος, που καταλαμβάνει ο κόμβος αυτός στη μνήμη και επιστρέφεται η τιμή true από τη συνάρτηση, καθώς η διαγραφή έγινε με επιτυχία.

Ερώτημα Β

Κώδικας Ερωτήματος Β

Παρακάτω παρατίθεται το τμήμα του κώδικα, που αφορά το ερώτημα Β, δηλαδή τη λειτουργία της συνάρτησης Find_second_next, η οποία δέχεται σαν όρισμα ένα στοιχείο, και επιστρέφει το δεύτερο μικρότερο μεταξύ όλων των στοιχείων του δέντρου, τα οποία είναι μεγαλύτερα του στοιχείου, που δόθηκε σαν όρισμα στη συνάρτηση :

```
int BSMTree::Find_second_next(int refElement) {
    if (rootNode) {
        bool nodeExists = false;
        currNode = rootNode;
        while (currNode) {
            if (currNode->element > refElement) {
                currNode = currNode->leftChild;
            } else if (currNode->element < refElement) {
                currNode = currNode->rightChild;
            } else {
                nodeExists = true;
                break;
            }
        }

        bool secondNextFound = false;
```

```
        int counter = 0;
        int element = 0;

        if (nodeExists) {
            InOrderEditedV1(rootNode, refElement, counter,
secondNextFound, element);
            if (secondNextFound) {
                return element;
            } else {
                cout << "There is not second next element, of the
element " << refElement << "..." << endl;
                return -1;
            }
        } else {
            cout << "Element " << refElement << " does not exist..." <<
endl;
            return -1;
        }
    } else {
        cout << "The tree is empty..." << endl;
        return -1;
    }
}
```

Όπου, μέσα από την Find_second_next καλείται η συνάρτηση InOrderEditedV1, η οποία είναι μια ελαφρώς τροποποιημένη μορφή ενδοδιάταξης, με τη διαφορά ότι εμφανίζει στην οθόνη μόνο το δεύτερο μικρότερο στοιχείο, από όλα τα στοιχεία, που είναι μεγαλύτερα από το στοιχείο που επιλέχθηκε :

```
void BSMTree::InOrderEditedV1(TreeNodePtr t, int refElement, int
&counter, bool &secondNextFound, int &element) {
    if (t) {
        if (t->leftChild) {
            InOrderEditedV1(t->leftChild, refElement, counter,
secondNextFound, element);
        }

        if (t->element > refElement) {
            counter++;
            if (counter == 2) {
                cout << "The second next element, greater than the
element " << refElement << ", is " << t->element << ".\n";
                secondNextFound = true;
                element = t->element;
            }
        }

        if (secondNextFound == false) {
            if (t->rightChild) {
                InOrderEditedV1(t->rightChild, refElement, counter,
secondNextFound, element);
            }
        }
    }
}
```

Ανάλυση Κώδικα Ερωτήματος Β

Καταρχάς, η συνάρτηση InOrderEditedV1, όπως προαναφέρθηκε, αποτελεί τροποποίηση της ενδοδιάταξης, δηλαδή διατηρεί τον αναδρομικό χαρακτήρα της, με τη διαφορά, ότι χρησιμοποιούνται δύο βοηθητικές μεταβλητές, οι counter και secondNextFound, οι οποίες αποτελούν και ορίσματα της συνάρτησης, και τις οποίες μπορεί να επεξεργαστεί και να αλλάξει. Συγκεκριμένα, η μεταβλητή counter, η οποία αρχικοποιείται με τιμή 0, αυξάνεται κατά μία μονάδα κάθε φορά, που η τιμή του τρέχοντος στοιχείου που ελέγχεται, είναι μεγαλύτερη της τιμής του στοιχείου που επιλέχθηκε. Η μεταβλητή secondNextFound αρχικοποιείται με τιμή false, και αλλάζει μόνο όταν η τιμή του counter είναι ίση με 2, δηλαδή όταν βρεθεί το δεύτερο μικρότερο από όλα τα στοιχεία, που είναι μεγαλύτερα του επιλεχθέντος στοιχείου (ή πιο εύκολα, το δεύτερο σε σειρά στοιχείο, που είναι μεγαλύτερο του επιλεχθέντος στοιχείου). Η λογική της ενδοδιάταξης χρησιμοποιείται, διότι τα στοιχεία του δέντρου διασχίζονται ανάλογα με τη τιμή τους, δηλαδή από το μικρότερο στο μεγαλύτερο. Στο σώμα της συνάρτησης Find_second_next, αρχικά γίνεται μια δυαδική αναζήτηση για την εύρεση του στοιχείου, που δόθηκε ως όρισμα στη συνάρτηση. Ελέγχονται οι περιπτώσεις, να μην υπάρχει ρίζα ή το στοιχείο που δόθηκε ως όρισμα, και εμφανίζονται τα αντίστοιχα, επεξηγηματικά μηνύματα σφάλματος. Σε περίπτωση, που δεν υπάρχει σφάλμα, η μεταβλητή nodeExists, με αρχική τιμή false, αλλάζει σε true. Έπειτα, ορίζονται οι μεταβλητές counter, secondNextFound και element. Αν η τιμή της nodeExists είναι true, καλείται η συνάρτηση InOrderEditedV1, με όρισμα τον δείκτη προς την ρίζα, το στοιχείο του κόμβου αναφοράς, καθώς και τις βοηθητικές μεταβλητές, και εκτελείται. Σε περίπτωση, που βρεθεί το επιθυμητό στοιχείο, αυτό εμφανίζεται στην οθόνη και επιστρέφεται από τη συνάρτηση Find_second_next, αλλιώς επιστρέφεται η τιμή -1 (ο κωδικός λάθους, όπως προαναφέρθηκε) και εμφανίζεται ένα αντίστοιχο, επεξηγηματικό μήνυμα.

Ερώτημα Γ

Κώδικας Ερωτήματος Γ

Παρακάτω παρατίθεται το τμήμα του κώδικα, που υλοποιεί τη συνάρτηση Print_between, η οποία δέχεται ως ορίσματα δύο αριθμούς, οι οποίοι θα αποτελούν τα όρια ενός διαστήματος. Τα στοιχεία, των οποίων οι τιμές είναι εντός αυτού του διαστήματος, θα εμφανίζονται στην οθόνη :

```
void BSMTree::Print_between(int startNum, int endNum) {
    if (rootNode) {
        bool numsExist = false;
        bool numsCease = false;
        if (startNum < endNum) {
            cout << "The elements, between the numbers " << startNum <<
" and " << endNum << ", are :\n";
            InOrderEditedV2(rootNode, startNum, endNum, numsExist,
numsCease);
            if (!numsExist) {
                cout << "None..." << endl;
            }
        } else {
            cout << "Invalid values inserted..." << endl;
        }
    } else {
        cout << "The tree is empty..." << endl;
```

```
    }
}
```

Όπου, μέσα από την Print_between, καλείται η συνάρτηση InOrderEditedV2, η οποία έχει ως ορίσματα ένα δείκτη προς κόμβο του δέντρου, τους δύο αριθμούς, που επιλέχθηκαν ως όρια του διαστήματος, καθώς και δύο βοηθητικές μεταβλητές, τύπου bool :

```
void BSMTree::InOrderEditedV2(TreeNodePtr t, int startNum, int endNum,
bool &numsExist, bool &numsCease) {
    if (t) {
        if (t->leftChild) {
            InOrderEditedV2(t->leftChild, startNum, endNum, numsExist,
numsCease);
        }

        if (t->element > endNum) {
            numsCease = true;
        }

        if (!numsCease) {
            if (t->element >= startNum && t->element <= endNum) {
                numsExist = true;
                cout << t->element << "\t";
            }

            if (t->rightChild) {
                InOrderEditedV2(t->rightChild, startNum, endNum,
```

```
numsExist, numsCease);
                }
            }
        }
}
```

Ανάλυση Κώδικα Ερωτήματος Γ

Μέσα στο σώμα της συνάρτησης Print_between, γίνεται ένας έλεγχος ύπαρξης της ρίζας του δέντρου. Αν υπάρχει, ορίζονται δύο μεταβλητές τύπου bool, και αρχικοποιούνται με τιμή false, οι οποίες θα δοθούν αργότερα ως ορίσματα στη συνάρτηση InOrderEditedV2. Γίνεται ένας έλεγχος ορθότητας των τιμών των ορίων του διαστήματος, με την εντολή διακλάδωσης if. Σε περίπτωση σφάλματος, εμφανίζεται μήνυμα λάθους. Αλλιώς, καλείται η συνάρτηση InOrderEditedV2, με όρισμα τον δείκτη της ρίζας (από αυτό το σημείο πρέπει να ξεκινά η διαδικασία της ενδοδιάταξης). Στο σώμα της συνάρτησης αυτής, γίνεται ο έλεγχος της τιμής του στοιχείου του τρέχοντος κόμβου, ώστε να μην είναι μεγαλύτερη του ανώτατου ορίου του διαστήματος. Σε περίπτωση που είναι μεγαλύτερο το στοιχείο, η μεταβλητή numsCease παίρνει τιμή true, και η διαδικασία της ενδοδιάταξης σταματάει. Έτσι, ενδεχομένως υπάρχει μείωση του χρόνου εκτέλεσης της συνάρτησης, αν υπάρχουν πολλά στοιχεία μεγαλύτερα του ανώτατου ορίου του διαστήματος. Αν το στοιχείο είναι εντός των ορίων του διαστήματος, με τον αντίστοιχο έλεγχο, εμφανίζεται στην οθόνη. Επίσης, την πρώτη φορά που θα βρεθεί στοιχείο, το οποίο βρίσκεται εντός των ορίων του διαστήματος, η μεταβλητή numsExist θα λάβει την τιμή true. Στην περίπτωση, όπου μετά τον τερματισμό της συνάρτησης InOrderEditedV2, η τιμή της numsExist παραμένει false, εμφανίζεται στην οθόνη επεξηγηματικό μήνυμα, για τη μη ύπαρξη στοιχείων εντός του διαστήματος.