16.01.2019

LABORATORIES 3

[EADS] Algorithms and data structure

THE AVL TREE

WIKTOR BARA

TASK

Design the template class, AVL tree, and apply to class internal supporting class iterator.

THE CLASS TEMPLATE

```
template<typename Key, typename Info>
class Dictionary {
private:
  struct Node{
  Key key;
  Info info;
  Node* left;
  Node* right;
  Node* parent;
  int height;

  Node(Key ke, Info in):
    key(ke),info(in), left(nullptr), right(nullptr),
    parent(nullptr), height(0) {};
  Node(Key ke, Info in, int h):
    key(ke), info(in), left(nullptr), right(nullptr),
    parent(nullptr), height(h) {};

  };

  Node* root;
```

int getHeight(Node* x){ // returns the number of nodes below, if nullptr returns -1

Node* search(const Key key, Node* x)const{ //returns node with given key among given node and his children, if node with given key doesn't exists returns nullptr

Node* copy(Node* x){ // returns copy of given node with his children

void print(Node* x)const{ // displays node and children of given node without view about the balance

void clear(Node* x){ //deletes given node and his children

Node* findMax(Node* x)const{ // returns node with the smallest key among given node and his children

Node* findMin(Node *x)const{ // returns node with the greatest key among given node and his children

Node* insert(const Key &key, const Info &info, Node* x){ //returns node in which structure creates new node and add it to given node, the balances the whole structure of tree

Node* remove(const Key &key, Node* x){ //returns node in which structure deletes the node with given key among children of given nodes, then balances the whole structure of tree, if there is no node with given key it does nothing

Node* rrotate(Node* &x){ //algorithm of single right rotation to balance the tree

Node* lrotate(Node* &x){ //algorithm of single left rotation to balance the tree

Node* llrotate(Node* &x){ //algorithm of double left rotation to balance the tree

Node* rrrotate(Node* &x){ //algorithm of double right rotation to balance the tree

//method taken from

// https://stackoverflow.com/questions/801740/c-how-to-draw-a-binary-tree-to-the-console

int _print_t(Node *tree, int is_left, int offset, int depth, char s[20][255]){
// prints balanced tree

public:

Dictionary(){

Dictionary(const Dictionary<Key,Info> &x){

~Dictionary(){ //constructors

Dictionary<Key, Info> &operator=(const Dictionary<Key,Info>& x){ //assignment opertator

bool operator==(const Dictionary &x)const{

bool operator!=(const Dictionary &x)const{ //comparators

```
friend ostream& operator<<(ostream &os, const Dictionary &x){           //use private method print to display tree
void print_t()   {                 //use private method _print_t

bool isEmpty(){            //checks if tree is empty
void clear(){              //deletes all nodes in the tree
bool keyExists(const Key &key){          //checks if node with given key exists
unsigned int getHeight(){                //returns height of tree

bool insert(const Key &key, const Info &info){          //use private method insert to add new node
bool remove(const Key &key){           //use private method remove to delete node with given key

Iterator begin(){                 //returns Iterator pointing to the smallest node
Iterator end(){                   //returns Iterator pointing to the root of tree
Iterator middle(){                //returns Iterator pointing to the greatest node
Const_Iterator const_begin()const{      //returns Const_Iterator pointing to the smallest node
Const_Iterator const_end()const{        //returns Const_Iterator pointing to the root of tree
Const_Iterator const_middle()const{     //returns Const_Iterator pointing to the greatest node
```

SUPPORTING CLASS

```
class Iterator {
    friend class Dictionary;
    mutable Node* curr;
    Iterator(Node *x)
  public:
Iterator(){
Iterator(const Iterator &x){
 ~Iterator(){}              //constructors

Iterator operator=(const Iterator &x){          //assignment operator
    Iterator &operator++()const{        //prefix
    Iterator operator++(int)const{      //postfix
    Iterator &operator--()const{        //prefix
    Iterator operator--(int)const {     //postfix

    Key operator*()const{               //returns key if iterator points to nullptr returns 0
    Node* operator->() const{           //returns key or info of given iterator
    bool operator!()const{              //returns true if iterator points to nullptr else false
```

TESTING CONDUCTED

Creating Tree by:

- Constructor
- Copy Constructor
- Constructor and assignment operator

Inserting nodes:

- In the order
- In the random way

To check if the tree is balanced

Removing nodes:

- remove of the main root
- remove of any node with children
- removing the nodes without children

Iterator:

- Constructors:
  - Assignment by operator =
  - Checking every possible start point (begin, middle, end)
  - Copy constructor
- Operator ++ or -- (post and pre):
  - Going through the whole tree
  - Going beyond nodes of tree
  - Using when Iterator points to nullptr
- Operator *:
  - Checking randomly selected nodes to check
  - Checking if the Iterator points to nullptr