Rapport
Laboration 3

Datastrukturer och Algorimeter

Mathias Andersson
Henrik Bäck

Datastrukturer och Algoritmer

Laboration 3

Innehållsförteckning

AVL-träd

Det AVL-träd som implementerats har byggts från ett BST (Binary Search Tree) som konstruerats tidigare. Detta BST har kompletterats med funktioner för att kontrollera och hålla invarianten för ett AVL-träd. Dessutom har funktionalitet för att läsa in trädet från fil konfigurerats om för att kunna skapa en exakt kopia av det AVL-träd som sparats.

Roteringar

För att kunna bibehålla invarianten har två stycken roteringsfunktioner implmenterats. Dessutom har två funktioner för dubbelrotation implementerats. Alltså totalt fyra stycken roteringsfunktioner. Dessa funktioner används för att korrigera ett BST som bryter mot invarianten och funktionerna anropas både vid adderandet och borttagandet av element.

Vid insättning av nya noder

När den nya nodens position tas fram sparas samtidigt information om var i trädet den nya noden kommer att befinnas sig. För varje rekursivt anrop sparas information om hur funktionen arbetat sig fram till denna från föregående nod, det vill säga vilka höger och vänstersvängar som har gjorts. Detta används sedan för att kunna göra eventuella rotationer korrekt.

Vid raderandet av befintliga noder

När det gäller borttagandet av noder från trädet är fallet lite mer komplicerat. Genom att, för den noden som skall raderas kontrollera om vi får en övervikt på ett delträd på motstående sida i någon av de gemensamma föräldranoderna kan vi avgöra om vi behöver rotera. Samma rotationer körs som om en nod skulle adderats till det subträd som numer har övervikt. På detta sätt kan det avgöras vilken typ av rotation som behöver göras.

Höger-rotering

Vänsterbartnet till den nod som skall roteras sätts till nodens vänsterbarns högerbarn. Efter detta så kommer nodens vänsterbarns högerbarn att sättas till noden. Då har en rotation skett och en annan nod har satts som rot i subträdet.

Vänster-rotering

Högerbarnet till den nod som skall roteras sätts till nodens högerbarnets vänsterbarn. Efter detta så kommer nodens högerbarn vänsterbarn att sättas till noden. Då har rotation skett och en annan nod har satts rot i subträdet.

Höger-Vänster-rotering

Vid en rotering höger-vänster i en nod, n, kommer till en början att på nodens högerbarn en höger-rotering att utföras. Därefter kommer en vänster-rotering genomföras på själva noden, n.

Vänster-Höger-rotering

Vid en rotering vänster-höger i en nod, n, kommer till en början att på nodens vänsterbarn en vänste-rotering att utföras. Därefter kommer en höger-rotering genomföras på själva noden, n.

Bilaga - Programkod

avl.h

```
#ifndef AVL_CLASS_12344321
#define AVL_CLASS_12344321

#include "TreeNode.h"
#include <cstring>
#include <fstream>
using namespace std;

template <class T>
class AVL
{
 public:

  //Description
  //AVL = AVL-Tree
  //e = element

  //ø => AVL
  //Pre: True
  //Post: A new tree is created
  AVL();

  //AVL => ø
  //Pre:
  //Post: The tree and all of it's nodes has been destryed
  ~AVL();

  //AVL x e => AVL
  //Pre:  !exists(e)
  //Post: Element has been added
  void insert(T element);

  //AVL x e => AVL
  //Pre:  exists(e)
  //Post: Element has been removed
  void removeValue(T element);

  //AVL x e => Boolean
  //Pre:  True
  //Post: Returned true if element exists, else false
  bool exists(T element);

  //AVL => AVL
  //Pre:  True
  //Post: Printed the tree in pre-order
  void printPreOrder(void (*ut)(T data));

  //AVL => AVL
  //Pre:  True
  //Post: Printed the tree in post-order
  void printPostOrder(void (*ut)(T data));

  //AVL => AVL
```

```
  //Pre:  True
  //Post: Printed the tree in in-order
  void printInOrder(void (*ut)(T data));

  //AVL => AVL
  //Pre:  True
  //Post: Tree has been wrote to filename
  void saveToFile(char filename[]);

  //AVL => AVL
  //Pre:  True
  //Post: Tree has been read from file
  bool openFromFile(char filename[]);

  //AVL => AVL
  //Pre:  True
  //Post: Tree has been emptied
  void emptyTree();

  //AVL => Integer
  //Pre:  True
  //Post: Returned the number of elements in the tree
  int numEl();


 private:
  //AVL x e => AVL
  //Pre: !exists(e)
  //Post: Element has been added
  bool insertPr(TreeNode<T> * & n, T element, bool doBalance);

  //AVL x e => AVL
  //Pre: !exists(e)
  //Post: Element has been removed
  bool removeValuePr(TreeNode<T> *& n, T value, TreeNode<T> *& parent, bool
goLeft);

  //AVL x e => Boolean
  //Pre: True
  //Post: Returned true if the element exists, else false
  bool existsPr(TreeNode<T> * n, T element);

  //AVL => AVL
  //Pre: True
  //Post: Elements has been printed in pre-order
  void printPreOrderPr(TreeNode<T> * n, void (*ut)(T data));

  //AVL => AVL
  //Pre: True
  //Post: Elements has been printed in post-order
  void printPostOrderPr(TreeNode<T> * n, void (*ut)(T data));

  //AVL => AVL
  //Pre: True
  //Post: Elements has been printed in in-order
  void printInOrderPr(TreeNode<T> * n, void (*ut)(T data));
```

```
  //AVL => AVL
  //Pre: True
  //Post: Tree has been been written to file
  void printToFile(TreeNode<T> * n, T arr[], int & i);

  //AVL => AVL
  //Pre: True
  //Post: The tree has been empty
  void emptyTreePr(TreeNode<T> *& n);

  //AVL x e => e
  //Pre: True
  //Post: The smallest value in the subtree is found in T
  void findSmallPr(TreeNode<T> * n, T & aktuell);

  //AVL x e => Boolean
  //Pre: True
  //Post: Returned true if the node and its subtrees are balanced, else
false
  bool isBalanced(TreeNode<T> *&n);

  //AVL x e => AVL
  //Pre: True
  //Post: If needed, the tree is balanced at the node n
  void balance(TreeNode<T> * &n, bool goLeftChild, bool goLeftSubTree);

  //AVL x e => AVL
  //Pre: True
  //Post: Rotated the node and the tree at the right
  TreeNode<T>* rotateRight(TreeNode<T> * &n2);

  //AVL x e => AVL
  //Pre: True
  //Post: Rotated the node and the tree at the left
  TreeNode<T>* rotateLeft(TreeNode<T> * &n2);

  //AVL x e => AVL
  //Pre: True
  //Post: Rotated the node and the tree at the right then left
  TreeNode<T>* rotateRightLeft(TreeNode<T> *& n2);

  //AVL x e => AVL
  //Pre: True
  //Post: Rotated the node and the tree at the left then right
  TreeNode<T>* rotateLeftRight(TreeNode<T> *& n2);

  //AVL x e => Integer
  //Pre: True
  //Post: Returned the height of the nods subtree
  int subTreeHeight(TreeNode<T> * &n);

  //AVL x e => AVL
  //Pre: True
  //Post: If needed, the tree is balanced at the node n (when element is
removed)
  void balanceRemove(TreeNode<T> * n, TreeNode<T> * parent, bool goLeft);
```

```
  //AVL x e => AVL
  //Pre: True
  //Post: Inserted an element to the AVL-tree without running any balancy
check
  void insertUnBalance(T element);

  //Data
  TreeNode<T>* root;
  int size;

};

#include "avl.cpp"

#endif
```

avl.cpp

```
#include <cstring>
#include <iostream>
#include <fstream>

template <class T>
AVL<T>::AVL()
{

  root = 0;
  size = 0;

}

template <class T>
AVL<T>::~AVL()
{

  emptyTreePr(root);
  root = 0;

}

template <class T>
bool AVL<T>::exists(T element)
{

  return existsPr(root, element);

}

template <class T>
bool AVL<T>::existsPr(TreeNode<T> * n, T element)
{

  if(n != 0)
    {
      if(n->data == element)
           return true;
```

```
      else if(n->data > element)
           return existsPr(n->leftChild, element);
      else
           return existsPr(n->rightChild, element);

    }
  else
    {
      return false;
    }
}

template <class T>
bool AVL<T>::insertPr(TreeNode<T> * & n, T element, bool doBalance)
{

  bool goLeftChild = false;
  bool goLeftSubTree = false;

  if(n==0)
    {
      TreeNode<T> * newNode = new TreeNode<T>;
      newNode -> data = element;
      newNode -> leftChild = 0;
      newNode -> rightChild = 0;
      n = newNode;

    }
  else if(element < n -> data)
    {
      goLeftChild = true;
      goLeftSubTree = insertPr(n->leftChild, element, doBalance);
    }
  else
    {
      goLeftChild = false;
      goLeftSubTree = insertPr(n->rightChild, element, doBalance);
    }


  if(doBalance)
    {
      balance(n ,goLeftChild, goLeftSubTree);
    }

  return goLeftChild;

}


template <class T>
void AVL<T>::insert(T element)
{
  insertPr(root, element, true);
  size++;
}
```

```
template <class T>
void AVL<T>::insertUnBalance(T element)
{
  insertPr(root, element, false);
  size++;
}

template <class T>
void AVL<T>::printPreOrder(void (*ut)(T data))
{

  printPreOrderPr(root,ut);

}

template <class T>
void AVL<T>::printPreOrderPr(TreeNode<T> * n, void (*ut)(T data))
{
  using namespace std;

  if(n!=0)
    {

      ut(n->data);
      printPreOrderPr(n->leftChild, ut);
      printPreOrderPr(n->rightChild, ut);

    }


}

template <class T>
void AVL<T>::printInOrderPr(TreeNode<T> * n, void (*ut)(T data))
{

  if(n!=0)
    {

      printInOrderPr(n->leftChild, ut);
      ut(n->data);
      printInOrderPr(n->rightChild, ut);


    }

}

template <class T>
void AVL<T>::printInOrder(void (*ut)(T data))
{

  printInOrderPr(root,ut);

}

template <class T>
```

```
void AVL<T>::printPostOrderPr(TreeNode<T> * n, void (*ut)(T data))
{

  if(n!=0)
    {

      printPostOrderPr(n->leftChild, ut);
      printPostOrderPr(n->rightChild, ut);
      ut(n->data);

    }

}

template <class T>
void AVL<T>::printPostOrder(void (*ut)(T data))
{

  printPostOrderPr(root,ut);

}

template <class T>
void AVL<T>::saveToFile(char filename[])
{

  T array[size];
  int talet = 0;
  T data;

  printToFile(root, array, talet);

  ofstream skrivFil(filename, ios::out);
  if(!skrivFil)
    cerr << endl << "Error while opening file" << endl;
  else
    {

      for(int i = 0; i<size; i++)
           {
             skrivFil.write(reinterpret_cast<const  char  *  >  (&array[i]),
sizeof(data));
           }

           skrivFil.close();
    }
}

template <class T>
void AVL<T>::printToFile(TreeNode<T> * n, T arr[], int &i)
{

  using namespace std;


  if(n!=0)
    {
```

```
      arr[i] = n->data;
      i++;
      printToFile(n->leftChild, arr, i);
      printToFile(n->rightChild, arr, i);

    }

}

template <class T>
bool AVL<T>::openFromFile(char filename[])
{
  T data;
  int i =0;
  bool bra;

  ifstream lasFil(filename, ios::in);
  if(!lasFil)
    bra = false;
  else
    {
      emptyTreePr(root);

      while(lasFil.peek()!= EOF)
           {
             lasFil.read(reinterpret_cast<char * > (&data), sizeof(data));
             insertUnBalance(data);
             i++;
             lasFil.seekg((i)*sizeof(data));

           }
      lasFil.close();
      bra = true;
    }

  return bra;

}

template <class T>
void AVL<T>::emptyTreePr(TreeNode<T> *& n)
{

  if(n!=0)
    {

      emptyTreePr(n->leftChild);
      emptyTreePr(n->rightChild);
      delete n;
      n = 0;
      size--;

    }

}

template <class T>
```

```
void AVL<T>::emptyTree()
{

  emptyTreePr(root);
}

template <class T>
int AVL<T>::numEl()
{

  return size;

}

template <class T>
void AVL<T>::removeValue(T value)
{

  removeValuePr(root, value, root, false);

}

template <class T>
bool  AVL<T>::removeValuePr(TreeNode<T>  *&  n,  T  value,  TreeNode<T>  *&
parent, bool goLeft)
{

  bool hasRemoved = false;

  if(n != 0)
    {
      if(value < n->data)
           {
             hasRemoved = removeValuePr(n->leftChild, value, n, true);
           }
      else if(value > n->data)
           {
             hasRemoved = removeValuePr(n->rightChild, value, n, false);
           }
      else if(value == n->data)
           {

             if((n->leftChild == 0) && (n->rightChild == 0))
               {
                 delete n;
                 size--;
                 n = 0;
                 hasRemoved = true;
               }
             else if(n->rightChild == 0)
               {
                 TreeNode<T> * temp = n;
                 n = n->leftChild;
                 delete temp;
                 temp = 0;
               }
             else if(n->leftChild == 0)
```

```
                {
                  TreeNode<T> * temp = n;
                  n = n->rightChild;
                  delete temp;
                  temp = 0;
                }
              else
                {
                  T temp = n->rightChild->data;
                  findSmallPr(n->rightChild, temp);
                  n->data = temp;
                  hasRemoved = removeValuePr(n->rightChild, temp, n, false);
                }


            }

      if(hasRemoved && n != 0)
           balanceRemove(n, parent, goLeft);

    }

  return hasRemoved;

}

template <class T>
void AVL<T>::findSmallPr(TreeNode<T> * n, T & aktuell)
{

  if(n != 0)
    {
      if(aktuell > n->data)
           aktuell = n->data;

       findSmallPr(n->leftChild, aktuell);
    }

}


template <class T>
bool AVL<T>::isBalanced(TreeNode<T> * &n)
{

  bool isBal = true;

  if(abs(subTreeHeight(n->leftChild) - subTreeHeight(n->rightChild)) > 1)
    isBal = false;

  return isBal;

}


template <class T>
```

```
void AVL<T>::balance(TreeNode<T> * &n, bool goLeftChild, bool
goLeftSubTree)
{


  if(!isBalanced(n))
    {

      if(goLeftChild)
           {
             if(goLeftSubTree)
               n = rotateRight(n);
             else
               n = rotateLeftRight(n);
           }
      else
           {
             if(goLeftSubTree)
               n = rotateRightLeft(n);
             else
               n = rotateLeft(n);
           }

    }


}


template <class T>
void AVL<T>::balanceRemove(TreeNode<T> * n, TreeNode<T> * parent, bool
goLeft)
{


  if(!isBalanced(n))
    {
      if(subTreeHeight(n->leftChild) < subTreeHeight(n->rightChild))
           {
             if(subTreeHeight(n->rightChild->rightChild) > subTreeHeight(n-
>rightChild->leftChild))
               {

                 if(n == root)
                      root = rotateLeft(n);
                 else
                      {
                        if(goLeft)
                          parent->leftChild = rotateLeft(n);
                        else
                          parent->rightChild = rotateLeft(n);
                      }
               }
             else
               {

                 if(n == root)
```

```
                    root = rotateRightLeft(n);
                else
                    {

                      if(goLeft)
                        parent->leftChild = rotateRightLeft(n);
                      else
                        parent->rightChild = rotateRightLeft(n);

                    }
                }
            }
        else
            {
              if(subTreeHeight(n->leftChild->leftChild)   >   subTreeHeight(n-
>leftChild->rightChild))
                {

                  if(n == root)
                    root = rotateRight(n);
                  else
                    {
                      if(goLeft)
                        parent->leftChild = rotateRight(n);
                      else
                        parent->rightChild = rotateRight(n);
                    }

                }
              else
                {

                  if(n == root)
                    root = rotateLeftRight(n);
                  else
                    {
                      if(goLeft)
                        parent->leftChild = rotateLeftRight(n);
                      else
                        parent->rightChild = rotateLeftRight(n);
                    }
                }
            }

    }
}

template <class T>
TreeNode<T>* AVL<T>::rotateLeft(TreeNode<T> * &n2)
{

  TreeNode<T>* n1;
  n1 = n2->rightChild;
  n2->rightChild = n1->leftChild;
  n1->leftChild = n2;
  return n1;
```

```
}

template <class T>
TreeNode<T>* AVL<T>::rotateRight(TreeNode<T> *& n2)
{

  TreeNode<T>* n1;
  n1 = n2->leftChild;
  n2->leftChild = n1->rightChild;
  n1->rightChild = n2;
  return n1;

}

template <class T>
TreeNode<T>* AVL<T>::rotateLeftRight(TreeNode<T> *& n2)
{
  n2->leftChild = rotateLeft(n2->leftChild);
  return rotateRight(n2);
}

template <class T>
TreeNode<T>* AVL<T>::rotateRightLeft(TreeNode<T> *& n2)
{
  n2->rightChild = rotateRight(n2->rightChild);
  return rotateLeft(n2);
}


template <class T>
int AVL<T>::subTreeHeight(TreeNode<T> * &n)
{

  if(n != 0)
    {
      if(subTreeHeight(n->rightChild) > subTreeHeight(n->leftChild))
           return subTreeHeight(n->rightChild) + 1;
      else
           return subTreeHeight(n->leftChild) + 1;

    }
  else
    return 0;


}
```

drivers.h

```
#include <iostream>
#include <stdlib.h>
#include <cstring>

using namespace std;

//Function for printing data to screen, used by the AVL
```

```
//Pre: True
//Post: Printed data to screen
void printInt(int data);

//Displays the menu
//Pre: True
//Post: Menu has been printed
int displayMenu();

//Get a menu choice from user
//Pre: True
//Post: Returned the choice of an user
int getMnuChoice();

//Displays the menu, takes an choice and runs the correct function
//Pre: true
//Post: User has chose to exit
void handleMenuChoice(AVL<int> & tree);

//Takes an integer from user
//Pre: True
//Post: Returned the integer
int inputInt();

//Add a user chosen element
//Pre: True
//Post: En element has been added if it not alredy exists
void dipAddElement(AVL<int> & tree);

//Remove a user chosen
//Pre: True
//Post: En element has been remove if it exists
void dipRemElement(AVL<int> & tree);

//Displays the tree in-order
//Pre: True
//Post: The tree has been displayed in-order
void dipIn(AVL<int> & tree);

//Displays the tree pre-order
//Pre: True
//Post: The tree has been displayed pre-order
void dipPre(AVL<int> & tree);

//Displays the tree post-order
//Pre: True
//Post: The tree has been displayed post-order
void dipPost(AVL<int> & tree);

//Saves the tree to a user chosen location and file
//Pre: True
//Post: The tree has been saved to the user chosen location and file
void dipSave(AVL<int> & tree);

//Opens the tree from a user chosen location and file
//Pre: True
//Post: The tree has been opended from the user chosen location and file
```

```
void dipOpen(AVL<int> & tree);

//Removes all elements
//Pre: True
//Post: all of the elements in the tree has been removed
void dipRemAll(AVL<int> & tree);

//Displays if an user choosen element is an element in the tree
//Pre: True
//Post: Displayed if an user choosen element is an element in the tree
void dipIsEl(AVL<int> & tree);

//Displays if the tree is empty
//Pre: True
//Post: Displayed if the tree is empty
void dipIsEmpt(AVL<int> & tree);
```

drivers.cpp

```
#define BUFFERSIZE 100

void printInt(int data)
{

  cout << data << ",";

}

int displayMenu()
{

  cout << "    INTEGER AVL-TREE" << endl
       << "========================" << endl
       << endl
       << "1.  Add Element..." << endl
       << "2.  Remove an Element..." << endl
       << "3.  Display in Pre-Order" << endl
       << "4.  Display in In-Order" << endl
       << "5.  Display in Post-Order" << endl
       << "6.  Save to file..." << endl
       << "7.  Open from file..." << endl
       << "8.  Remove all elements" << endl
       << "9.  Is element in tree?..." << endl
       << "10. Is tree empty?" <<  endl
       << "0. Exit" << endl << endl << "Select your choice (0-10):";


}

int getMnuChoice()
{
  char buffer[BUFFERSIZE];
  int varde;
  bool felKoll = false;
  do
    {
```

```
      if(felKoll == true)
           cout << "Not a vaild selection, please try again!" << endl <<
"Select your choice (0-10):";
      cin.getline(buffer, BUFFERSIZE);
      if(buffer[0]=='0')
           return 0;
      felKoll = true;
    }while(!(varde=atoi(buffer)));
  return varde;

}

void handleMenuChoice(AVL<int> & tree)
{
  bool exit = false;
  int mnu;

    do
    {

      displayMenu();

      mnu = getMnuChoice();

      system("clear");
      switch(mnu)
           {

           case 1: dipAddElement(tree);break;
           case 2: dipRemElement(tree);break;
           case 3: dipPre(tree);break;
           case 4: dipIn(tree);break;
           case 5: dipPost(tree);break;
           case 6: dipSave(tree);break;
           case 7: dipOpen(tree);break;
           case 8: dipRemAll(tree);break;
           case 9: dipIsEl(tree);break;
           case 10: dipIsEmpt(tree);break;
           case 0: exit = true; break;
           default: cout << "Not a vaild selection, please try again!" <<
endl  << "Select your choice (0-15):";
           };
      system("clear");
    }while(!exit);

}

int inputInt()
{

  char buffer[BUFFERSIZE] = {'\0'};
  int varde;
  bool felKoll = false;
  do
    {
      if(felKoll == true)
           cout << "Not an integear! Please try again: ";
```

```
      cin.getline(buffer, BUFFERSIZE);
      if(buffer[0]=='0')
           return 0;
      felKoll = true;
    }while(!(varde=atoi(buffer)));
  return varde;

}

void dipAddElement(AVL<int> & tree)
{

  int tal;

  cout << "Add element" << endl;
  cout << "===========" << endl << endl;

  cout << "Input integer: ";

  tal = inputInt();

  if(!tree.exists(tal))
    {
      tree.insert(tal);
      cout << "Integer added, press RETURN to continue..." << endl;
    }
  else
    cout << "Integer already in tree, press RETURN to continue..." << endl;

  getc(stdin);

}

void dipRemElement(AVL<int> & tree)
{

  int tal;

  cout << "Remove element" << endl;
  cout << "==============" << endl << endl;

  cout << "Input integer to remove: ";

  tal = inputInt();

  if(tree.exists(tal))
    {
      tree.removeValue(tal);
      cout << "Integer removed, press RETURN to continue..." << endl;
    }
  else
    cout << "Integer is not in tree (can not remove), press RETURN to
continue..." << endl;

  getc(stdin);

}
```

```
void dipIn(AVL<int> & tree)
{


  cout << "Display In-Order" << endl;
  cout << "================" << endl << endl;

  tree.printInOrder(&printInt);

  cout << endl << "Press RETURN to continue..." << endl;

  getc(stdin);

}

void dipPre(AVL<int> & tree)
{


  cout << "Display Pre-Order" << endl;
  cout << "=================" << endl << endl;

  tree.printPreOrder(&printInt);

  cout << endl << "Press RETURN to continue..." << endl;

  getc(stdin);

}

void dipPost(AVL<int> & tree)
{

  int tal;

  cout << "Display Post-Order" << endl;
  cout << "==================" << endl << endl;

  tree.printPostOrder(&printInt);

  cout << endl << "Press RETURN to continue..." << endl;

  getc(stdin);

}

void dipSave(AVL<int> & tree)
{

  char buffer[BUFFERSIZE] = {'\0'};

  cout << "Save to file" << endl;
  cout << "============" << endl << endl;

  cout << "Enter a relative path and filename: ";
```

```
  cin.getline(buffer, BUFFERSIZE);

  tree.saveToFile(buffer);

  cout << endl << "File saved, press RETURN to continue..." << endl;

  getc(stdin);

}

void dipOpen(AVL<int> & tree)
{

  char buffer[BUFFERSIZE] = {'\0'};

  cout << "Open from file" << endl;
  cout << "==============" << endl << endl;

  cout << "Enter a relative path and filename: ";

  cin.getline(buffer, BUFFERSIZE);

  if(tree.openFromFile(buffer))
    cout << endl << "File was opened, press RETURN to continue..." << endl;
  else
    cout << endl << "File not found, press RETURN to continue..." << endl;


  getc(stdin);

}

void dipRemAll(AVL<int> & tree)
{


  cout << "Remove all elements" << endl;
  cout << "===================" << endl << endl;

  tree.emptyTree();

  cout << "All elements has been removed, press RETURN to continue..." <<
endl;

  getc(stdin);

}

void dipIsEl(AVL<int> & tree)
{

  int tal;

  cout << "Is element in tree" << endl;
  cout << "==================" << endl << endl;

  cout << "Input integer: ";
```

```
  tal = inputInt();

  if(tree.exists(tal))
    {
      tree.exists(tal);
      cout << "Integer is in tree, press RETURN to continue..." << endl;
    }
  else
    cout << "Integer is not in tree, press RETURN to continue..." << endl;

  getc(stdin);

}


void dipIsEmpt(AVL<int> & tree)
{


  cout << "Is tree empty" << endl;
  cout << "=============" << endl << endl;


  if(tree.numEl() == 0)
    cout << "Tree is empty, press RETURN to continue..." << endl;
  else
    cout << "Tree is not empty, press RETURN to continue..." << endl;

  getc(stdin);

}
```

main.cpp

```
#include <iostream>
#include "avl.h"
#include "drivers.h"
#include "drivers.cpp"

using namespace std;

int main()
{

  AVL<int> taltraed;
  system("clear");

  handleMenuChoice(taltraed);

  return 0;
}
```

TreeNode.h

```
#if !defined(TREENODE_H___INCLUDED_)
#define TREENODE_H___INCLUDED_
```

```
template <class T> class AVL;

template <typename T>
class TreeNode
{
    friend class AVL<T>;

    private:
            T data;
            TreeNode* leftChild;
            TreeNode* rightChild;
};

#endif // !defined(TREENODE_H___INCLUDED_)
```