Balancing Search Trees

1. Tree Balance and Rotation
   - binary search trees
   - right rotation of a tree around a node
   - code for right rotation

2. AVL Trees
   - self-balancing search trees
   - four kinds of critically unbalanced trees

3. code for rotation
   - from left-right to left-left tree

MCS 360 Lecture 33
Introduction to Data Structures
Jan Verschelde, 13 April 2020
Balancing Search Trees

1. Tree Balance and Rotation
   - binary search trees
   - right rotation of a tree around a node
   - code for right rotation

2. AVL Trees
   - self-balancing search trees
   - four kinds of critically unbalanced trees

3. code for rotation
   - from left-right to left-left tree
Binary Search Trees

Consider 4, 5, 2, 3, 8, 1, 7 (recall lecture 24).
Insert the numbers in a tree:

```
        4
       /|
      / 2
     /|
    1  3  5
     |
      8
       |
        7
```

Rules to insert $x$ at node $N$:

- if $N$ is empty, then put $x$ in $N$
- if $x < N$, insert $x$ to the left of $N$
- if $x \geq N$, insert $x$ to the right of $N$

Recursive printing: left, node, right sorts the sequence.
an unbalanced tree

Inserting 0, 1, 2, ..., 9.

depth of tree : 9

0
  1
    2
      3
        4
          5
            6
              7
                8
                  9
shaping binary search trees

To make a binary search tree with given shape:

```
       20
      /   \
     10    40
    /     /  \
   5     15   20
  /   \
 1    7 10
```

Insert numbers in a particular order: 20, 40, 10, 5, 15, 1, 7.

\[
\text{depth}(T) = \begin{cases} 
0, & \text{if } T \text{ is empty,} \\
1 + \max(\text{depth(left}(T)), \text{depth(right}(T))), & \text{otherwise.}
\end{cases}
\]

The tree is unbalanced because the depth of the left tree is two, while the depth of the right tree is zero.
Balancing Search Trees

1. Tree Balance and Rotation
   - binary search trees
   - right rotation of a tree around a node
   - code for right rotation

2. AVL Trees
   - self-balancing search trees
   - four kinds of critically unbalanced trees

3. code for rotation
   - from left-right to left-left tree
Right Rotation

To balance the binary search tree, we do a right rotate around the root:

Observe the effects of a right rotation:
- left tree has become the new root;
- old root is now at the right of new root;
- left tree of old root is now the right tree of the left tree of old root.
Right Rotation in 3 Steps

Tree with root node $T$:

1. Label left of $T$ with $L$.
2. New tree $N$ has right of $T$ as right and as left the right of $L$.
3. Result $R$ has $L$ as root, the tree $N$ as right, and the left of $L$ as left.
Balancing Search Trees

1. Tree Balance and Rotation
   - binary search trees
   - right rotation of a tree around a node
   - code for right rotation

2. AVL Trees
   - self-balancing search trees
   - four kinds of critically unbalanced trees

3. code for rotation
   - from left-right to left-left tree
a node struct

struct Node
{
    int data;       // numbers stored at node in tree
    Node *left;     // pointer to left branch of tree
    Node *right;    // pointer to right branch of tree

    Node(const int& item, Node* left_ptr = NULL, 
    Node* right_ptr = NULL) :
        data(item),
        left(left_ptr), right(right_ptr) {}
}
```cpp
#include "mcs360_integer_tree_node.h"

namespace mcs360_integer_tree
{
    class Tree
    {
    private:
        Node *root; // data member

    public:
        Tree(const int& item,
             const Tree& left = Tree(),
             const Tree& right = Tree() ) :
            root(new Node(item,left.root,right.root)) {}
        Tree get_left() const;
        Tree get_right() const;
        void insert(int item);
    }
}
```
Prototype of function in client of class Tree:

Tree rotate_right ( Tree t );

// Returns the tree rotated to the right
// around its root.

// Precondition: left of t is not null.
**Definition of `rotate_right`**

```c
Tree rotate_right ( Tree t )
{
    Tree left = t.get_left();

    Tree new_t = Tree(t.get_data(),
                       left.get_right(),t.get_right());

    Tree R = Tree(left.get_data(),
                   left.get_left(),new_t);

    return R;
}
```
Balancing Search Trees

1. Tree Balance and Rotation
   - binary search trees
   - right rotation of a tree around a node
   - code for right rotation

2. AVL Trees
   - self-balancing search trees
   - four kinds of critically unbalanced trees

3. code for rotation
   - from left-right to left-left tree
Define the balance of a tree as

\[ \text{balance} = \text{depth} (\text{right tree}) - \text{depth} (\text{left tree}). \]

G.M. Adel’son-Vel’skiî and E.M Landis published in 1962 an algorithm to maintain the balance of a binary search tree.

If balance gets out of range $-1 \ldots +1$, the subtree is rotated to bring into balance.

Their approach is known as **AVL trees**.
a Class Hierarchy

- Binary Tree Node
- Binary Search Tree
- Binary Search Tree with Rotation
- AVL Tree
computing the balance

Recall the definition:

\[ \text{balance} = \text{depth(right tree)} - \text{depth(left tree)}. \]

At every node we compute the balance, displayed as subscript:

```
     20
   /   \
20     2
 /  \   /  \
10   10 40   0
 /  \           \
50   150         \
 /  \                 \
10   70
```
balancing a left-left tree

The tree below is *left heavy* as the balance is $-2$.
We also say that this is a *left-left tree*.

```
  20
 /   \
10   10
 /   / \
5   5   20
```

Executing a right rotation balances the tree.
Balancing Search Trees

1. Tree Balance and Rotation
   - binary search trees
   - right rotation of a tree around a node
   - code for right rotation

2. AVL Trees
   - self-balancing search trees
   - four kinds of critically unbalanced trees

3. code for rotation
   - from left-right to left-left tree
critically unbalanced trees

A tree is *critically unbalanced* if its balance is \(-2\) or \(+2\).

![Diagram of tree situations](image)

- A left-left tree
- A left-right tree
- A right-right tree
- A right-left tree
balancing trees of mixed kind

A right rotation balances a left-left tree and a left rotation balances a right-right tree.

Balancing a left-right tree happens in two stages:

1. rotate left-right tree to left-left tree:

```
        20 -2
       /   \
  5 +1   10 0
```

2. apply right rotation to left-left tree:

```
       20 -2
      /   \
  10 -1  5 0
      /   \
10 -1  5 0
```

Introduction to Data Structures (MCS 360)  Balancing Search Trees  L-33  13 April 2020  21 / 30
rotating a left-right tree

We rotate the left-right tree to a left-left tree:

Observe the effects of the rotation:

- the data at the left node of the new tree (10) is swapped with the data of the left of the old tree (5);
- the right of the left of the new tree (12) is the right of the right of the left of the old tree;
- the right of the left of the left of the new tree (7) is the left of the right of the left of the old tree.
rotating to left-left tree in 4 steps

Tree with root node $T$:

1. Label left of $T$ with $L$ and right of $L$ with $R$.
2. Tree $N$ has as its left the left of $L$, as its right the left of $R$.
3. Tree $M$ has as its left $N$, as its right the right of $R$.
4. Return the tree with its left $M$ and its right the right of $T$. 
a function to rotate a tree

Tree balance_by_rotation ( Tree t )
{
    if(is_left_left(t))
        return rotate_right(t);
    else if(is_right_right(t))
        return rotate_left(t);
    else if(is_left_right(t))
    {
        Tree R = rotate_to_left_left(t);
        return rotate_right(R);
    }
    else if(is_right_left(t))
    {
        Tree R = rotate_to_right_right(t);
        return rotate_left(R);
    }
}
else
{
    Tree L, R;
    if(!t.is_left_null())
        L = balance_by_rotation(t.get_left());
    if(!t.is_right_null())
        R = balance_by_rotation(t.get_right());
    return Tree(t.get_data(), L, R);
}

Balancing Search Trees

1. Tree Balance and Rotation
   - binary search trees
   - right rotation of a tree around a node
   - code for right rotation

2. AVL Trees
   - self-balancing search trees
   - four kinds of critically unbalanced trees

3. code for rotation
   - from left-right to left-left tree
prototype of the function

Tree rotate_to_left_left ( Tree t );

// Returns the tree rotated to a left-left tree.

// Preconditions:
//   (1) left of t is not null; and
//   (2) right of left of t is not null.

Test: insert 20, 5, 1, 10, 7, 12 to binary search tree.
definition of the function

Tree rotate_to_left_left ( Tree t )
{
    Tree left = t.get_left();
    Tree right = left.get_right();

    Tree new_left = Tree(left.get_data(),
                        left.get_left(),right.get_left());

    Tree new_right = Tree(right.get_data(),
                           new_left,right.get_right());

    Tree R = Tree(t.get_data(),
                  new_right,t.get_right());

    return R;
}
rebalancing search trees

After each insert (or removal):
- check the balance of the tree,
- and if critically unbalanced, rebalance.

Performance of the AVL tree:
- worst case: $1.44 \times \log_2(n)$,
- on average: $\log_2(n) + 0.25$ comparisons needed.

→ close to complete binary search tree.
Summary + Exercises

Started chapter 11 on balancing binary search trees.

Exercises:

1. Take a numerical example to left rotate a binary search tree with integer values. Formulate carefully each step in the left rotation. Justify the correctness of the algorithm.

2. Formulate the algorithm to rotate a right-left tree to a right-right tree and illustrate with an example.

3. The posted code provides functions to make an AVL tree. Design a class to represent an AVL tree.

4. Take your design of the previous exercise and define the methods of the class to represent an AVL tree.