Labworks 1: Binary Search Trees and Red-Black trees

October 2020

Deadline for these labworks: in 2 weeks (October 14, resp. October 21)
How?

- Make an archive with the source files of your implementation, and send it by email to `mircea.marin@e-uvt.ro`

Binary search trees

Overview:

- Data structures that support many dynamic-set operations.
- Basic operations take time proportional to the height of the tree.
  - For complete binary tree with n nodes: worst case $O(\log\ n)$.
  - For tree degenerated into a chain of n nodes: worst case $\Theta(n)$
- Different kinds of search trees include binary search trees, red-black trees, and B-trees.

Objectives of this labwork

1. remember the data structure for binary search trees
2. write C++ implementations for some of its operations

Objective of Lecture 1

- Learn a new data structure: red-black trees, which can be used to perform more efficiently (faster!) some dynamic-set operations.

What are binary search trees?

An important data structure for dynamic sets:

- Accomplish many dynamic-set operations in $O(h)$ time, where h = height of tree.
- We represent a binary search tree as an instance of the class

  ```
  struct BSTree {
      Node* root; // pointer to root node of the tree
      ...
  }
  ```

  and every node of the tree as an instance of the class

  ```
  struct Node {
      int key;       // key
      Node *p;      // pointer to parent
      Node *left;   // pointer to left child
      Node *right;  // pointer to right child
      ...
  }
  ```

  Note: the only node of a binary search tree without a parent is the root node. (`root->p == 0`)

- Stored keys must satisfy the **binary-search-tree property**:
  - if x is a node in the tree with left child y then $y.key \leq x.key$
  - if x is a node in the tree with right child y then $x.key \leq y.key$

Remark:

The binary-search-tree property allows us to print keys in a binary search tree in order, recursively, using an algorithm called an inorder tree walk. The elements of a binary search tree with root node x are printed in the monotonically increasing order of their key:

- Check to make sure that x is not NIL.
- Recursively, print the keys of the nodes in x's left subtree.
- Print x's key.
- Recursively, print the keys of the nodes in x's right subtree.

The method `void display(Node* x, int indent)` from the file `Node.h` in archive `BSTree.zip` implements this algorithm.

Labwork related to binary search trees

The archive BSTree.zip contains an incomplete implementation in C++ of an application which performs all the basic dynamic set operations on a binary search tree. Complete the missing implementations for the following methods of class BSTree:

1. `Node* predecessor(Node* x)` which returns a pointer to the node that precedes node `x` in an inorder walk of this tree, and NIL if `x` has no predecessor.

2. `int depth(Node* n)` which returns the depth of the tree whose root is pointed to by `n`. If `n` is NIL, the depth should be `-1`.

3. `Node* maximum(Node* n)` returns a pointer to the node with maximum key in the binary search tree with root `*n`. If `n` is NIL, return NIL.

Note: the missing implementations should be added to the file `Node.h`

Red-black trees

Overview:

- A red-black tree is a binary search tree with one extra field per node: an attribute `color`, which is either red or black.

  In C++, nodes can be represented as instances of

  ```
  struct RBNode {
  int key;       // key
  RBNode *p;     // pointer to parent
  RBNode *left;  // pointer to left child
  RBNode *right; // pointer to right child
  enum color { RED, BLACK };
  color col;
  ...
  };
  ```

- All leaves are empty (they do not contain elements with keys) and are colored black. In object-oriented implementations, there are two choices to represent leaves:

  1. with the null pointer `Nil`, which is assumed implicitly to be black.
  2. with a single sentinel node `Nil`, which is also an instance of class `RBNode`. This sentinel node is also the root's parent.

  In graphic representations, we usually do not draw the empty leaves.

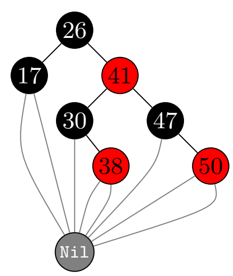
- A red-black tree must fulfil the following **red-black properties**:

1. Every node is either red or black.
2. The root is black
3. Every leaf is black
4. If a node is red, then both its children are black. (Hence no two reds in a row on a simple path from the root to a leaf.)
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.
6. Consider the following Ĉ++ classes to represent balanced red-black trees:

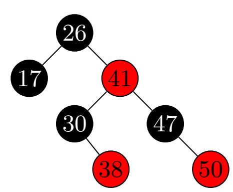
It can be shown that red-black trees are **balanced**: their height is $O(\log_2 n)$ where n is the number of nodes. Therefore, all operations will take $O(\log_2 n)$ time in the worst case.

Example

A red-black tree with number of nodes $n = 7$.



We won't bother with drawing `Nil` any more.



- Remarkable property: the dynamic set operations

  `minimum`, `maximum`, `successor`, `predecessor`, `search`, `insert`, `del`

  can be implemented with worst-case time complexity $O(\log_2 n)$ on red-black trees with n nodes.

Objectives of this labwork

1. Familiarization with the red-black properties of RB-trees.
2. Understand the functionality of the tree operations

   `minimum`, `maximum`, `successor`, `predecessor`, and `search`

   on red-black trees.
3. Efficient implementation of some operations on RB-trees, which make use of the red-black properties.

   - `RBInsert` and `RBDelete`

   as described in the lecture notes, to run with worst-time complexity $O(\log_2 n)$ on red-black trees with n nodes.

Note that these operations are based on the auxiliary operations `LeftRotate`, `RightRotate`, and `RBDeleteFixup`, which must be implemented too.

Labwork related to Red-Black trees

The archive `RBTree.zip` contains an incomplete implementation in `C++` of an application which performs all the basic dynamic set operations on a red-black tree, as described in the lecture notes.

1. Complete the missing implementations for the following methods of class RBTree:

   (a) `int bh()` which returns the black height the red-black tree.

   (b) `int maxBlackKey()` which returns the maximum key of black nodes in the red-black tree. If the red-black tree is empty, the method should return the value -1000.

   (c) `int maxRedKey()` which returns the maximum key of red nodes in the red-black tree. If the red-black tree has no red nodes, the method should return the value -1000.

   We assume that all nodes in the red-black tree have keys which are non-negative integers.

   Note: the missing implementations should be added to the file `RBNode.h`

For the next seminar/lab, prepare answer to the following questions:

1. What are the worst case runtime complexities of the methods

   `bh()`, `maxBlackKey()`, and `maxRedKey()`

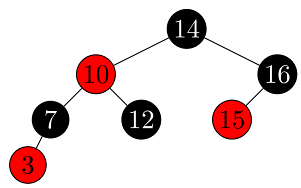
   implemented by you, if the red-black tree has n nodes?

2. What are the minimum and maximum number of red nodes in a red-black tree with black height 2?

3. Draw a red-black tree with minimum number of nodes and black height 2.

4. Draw a red-black tree with maximum number of nodes and black height 2.

5. Consider the red-black tree



   (a) What is the height of this red-black tree?

   (b) What is the black height of this red-black tree?

   (c) Draw the result or deleting the node with key 7.

   (d) Draw the result of inserting the node with key 11.