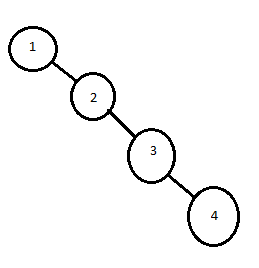
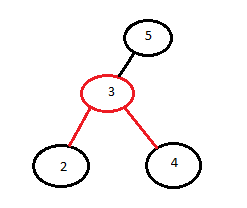
Röd-svarta träd

Röd svarta träd är en datastruktur ganska likt vanliga binära sök träd fast med flera villkor. Vanliga binära sökträd har vanligtvis komplexitets nivån O(log(n)), detta finns det dock ingen garanti för detta då trädet inte balanserar sig självt. Med en inmatning i ett binärt sökträd där en sekvens av nummer är stigande kan komplexitetsnivån nå O(n). Se figur 1.

Röd svarta träd är en självbalanserande datastruktur och kommer därför att garantera en O(log(n) komplexitet.



Figur 1. Denna bild demosterar ett vanligt binärt sökträd som har lagt in 1,2,3,4 i ordning. Att komma åt 4 kommer att ha komplexitets nivån O(n).

Till skillnad från vanliga binära sökträd så har rödsvarta träd två typer av noder. Noderna är antingen svarta eller röda. Detta betraktas implementärt som en boolean (sant eller falskt). Röd-svarta träd till skillnad från vanliga träd är och dubbelriktade. Vanliga träd håller bara koll på sina två barn, höger och vänster barn. Rödsvarta träd håller också koll på sin förälder. Se figur 2.

Figur 2. Den röda noden har föräldern 5. Högerbarn 4 och vänsterbarn 2.

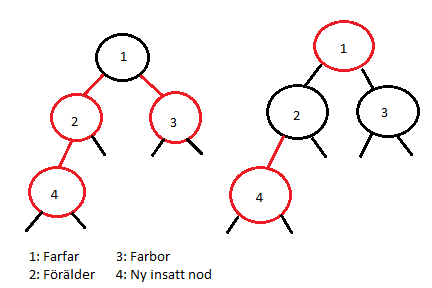
# Regler

Implementeringen kan vara lite krånlig för vissa och kan därför ses som en nackdel. Därför kan det vara bra att vända sig till standardbiblotekets STL set.

* Roten(noden högst upp i trädet) betraktas alltid som svart.
* noder som är nullptr kommer alltid betraktas som svarta.
* Nya noder som ska sättas in betraktas alltid som röda.
* Varje gren ska ha lika många svarta noder.
* Två röda noder får inte ligga efter varandra.

En rot deklareras ofta i början som nullptr som sedan används för att rekursivt ta sig igenom trädet. En insättning av rödsvart träd sker på ungefär samma sätt som ett vanligt träd. Efter varje insättning kommer en kontrollering att ske för att upptäcka om trädet är obalanserat, dvs om reglerna är uppfyllda.

Om reglerna inte är uppfyllda kommer man att använda två olika metoder, rotering eller färgbyte.



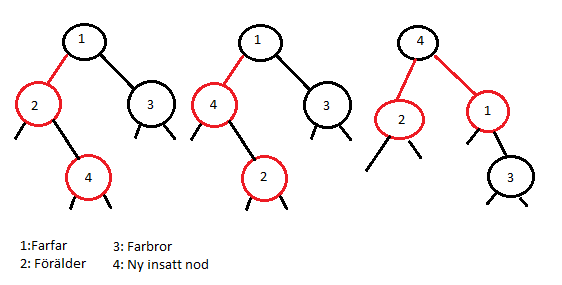
Figur 3. Illustering av färgbyte. Vänstra trädet är innan och höger är efter.

Om den nyinsatta nodens förälder inte är svart så måste ett färgbyte utföras vilket illusteras i figur 3. Farbror och förälder till den ny insatta noden blir svarta medans farförälder noden blir svart. Den nya noden blir flir den nya nodens farförälder och sedan görs stegen på nytt.

Om istället den nyinsatta nodens förälder är svart finns det fyra olika alternativ beroende på vart den nya instatta nodens förälder befinner sig jämtemot sin förälder.

* Föräldern är vänster barn till farföräldern och den nyinsatta noden är vänster barn til lföräldern
* Föräldern är vänster barn till farföräldern och den nyinsatta noden är höger barn till föräldern
* Föräldern är höger barn till farföräldern och den nyinsatta noden är höger barn till föräldern
* Föräldern är höger barn till farföräldern och den nyinsatta noden är vänster barn till föräldern

Dessa steg ser lite olika ut. Figur 4 illusterar hur deb andra regeln ser ut som även använder sig av den första regeln. Regel 3 och 4 är bara spegelvända gemtemot 1 och 2.



Figur 4. Bilden illusterar vänster höger steget ( Föräldern är vänster barn till farföräldern och den nyinsatta noden är höger barn till föräldern). Det första trädet kommer att göra en vänster rotering på föräldern som ger resultatet i mitten trädet. Efter detta kommer höger rotering farföräldern att ske och efter de ett färgbyte av farföräldern och föräldern.

Om det ska bli färgbyte eller de fyra andra alternativen itereras sålänge som alla dessa villkor är uppfyllda

* tillfälliga noden är skillt från roten
* tillfälliga noden är röd
* tillfälliga nodens pappa är röd

I bilagorna kan läsaren se hur författaren implementerat ett röd-svart träd med inmatning med inte borttagning.

# 

# BILAGA A

#include <vector>

#include <iostream>

#include "red\_black\_tree.h"

int main() {

std::vector<int>vec = { 1,1,1,1,1,1,1,1};

red\_black\_tree rb\_tree;

for (auto e : vec) {

rb\_tree.insert(e);

}

rb\_tree.in\_order();

rb\_tree.level\_order();

return 0;

}

# BILAGA B

#ifndef RED\_BLACK\_TREE\_H

#define RED\_BLACK\_TREE\_H

#include <stdlib.h>

#include <queue>

#include <iostream>

enum Color { RED, BLACK };

struct node \*new\_node(int value);

struct node \*recoloring(struct node\*&a, struct node \*&b, struct node \*&c);

struct node\* bst\_insert(struct node\* node, struct node \*pt);

struct node

{

int value;

struct node \*left, \*right, \*parent;

bool color;

};

class red\_black\_tree {

struct node \*root;

protected:

void in\_order\_helper(struct node \*root)const;

void level\_order\_helper(struct node \*root)const;

void fix\_violation(struct node \*node);

bool get\_color(struct node \*&node);

public:

red\_black\_tree();

void rotate\_left(node \*&ptr);

void rotate\_right(node \*&ptr);

void insert(const int &value);

void in\_order()const;

void level\_order()const;

};

#endif

# BILAGA C

#include "red\_black\_tree.h"

red\_black\_tree::red\_black\_tree() :root(nullptr) {}

void red\_black\_tree::in\_order()const { in\_order\_helper(root); }

void red\_black\_tree::level\_order()const { level\_order\_helper(root); }

bool red\_black\_tree::get\_color(struct node \*&node) {

if (node == nullptr)

return BLACK;

return node->color;

}

void red\_black\_tree::insert(const int &value) {

struct node\*node = new\_node(value);

std::cout << node->color;

root = bst\_insert(root, node);

fix\_violation(node);

}

void red\_black\_tree::in\_order\_helper(struct node \*root)const

{

if (root == nullptr)

return;

in\_order\_helper(root->left);

std::cout << root->value << " ";

in\_order\_helper(root->right);

}

void red\_black\_tree::level\_order\_helper(struct node \*root)const

{

if (root == nullptr)

return;

std::queue<struct node\*> q;

q.push(root);

while (!q.empty())

{

node \*temp = q.front();

std::cout << temp->value << " ";

q.pop();

if (temp->left != nullptr)

q.push(temp->left);

if (temp->right != nullptr)

q.push(temp->right);

}

}

void red\_black\_tree::rotate\_left(node \*&ptr) {

node \*right\_child = ptr->right;

ptr->right = right\_child->left;

if (ptr->right != nullptr)

ptr->right->parent = ptr;

right\_child->parent = ptr->parent;

if (ptr->parent == nullptr)

root = right\_child;

else if (ptr == ptr->parent->left)

ptr->parent->left = right\_child;

else

ptr->parent->right = right\_child;

right\_child->left = ptr;

ptr->parent = right\_child;

}

void red\_black\_tree::rotate\_right(node \*&ptr)

{

node \*left\_child = ptr->left;

ptr->left = left\_child->right;

if (ptr->left != nullptr)

ptr->left->parent = ptr;

left\_child->parent = ptr->parent;

if (ptr->parent == nullptr)

root = left\_child;

else if (ptr == ptr->parent->left)

ptr->parent->left = left\_child;

else

ptr->parent->right = left\_child;

left\_child->right = ptr;

ptr->parent = left\_child;

}

void red\_black\_tree::fix\_violation(struct node \*ptr) {

struct node \*parent = nullptr;

struct node \*grand\_parent = nullptr;

// sålänge ptr inte är roten OCH ptr är röd OCH ptr pappa är röd ( två röda i rad)

while (ptr != root && get\_color(ptr) == RED && get\_color(ptr->parent) == RED) {

parent = ptr->parent;

grand\_parent = parent->parent;

// om de kom från vänstra sidan

if (get\_color(parent) == get\_color(grand\_parent->left)) {

struct node\*uncle = grand\_parent->right;

if (get\_color(uncle) == RED) {

ptr = recoloring(grand\_parent, uncle, parent);

}

else {

if (get\_color(ptr) == get\_color(parent->right)) {

rotate\_left(parent);

ptr = parent;

parent = ptr->parent;

}

rotate\_right(grand\_parent);

std::swap(parent->color, grand\_parent->color);

ptr->parent;

}

}

else {

struct node\*uncle = grand\_parent->left;

if (get\_color(uncle) == RED) {

ptr = recoloring(grand\_parent, uncle, parent);

}

else {

if (get\_color(ptr) == get\_color(parent->left)) {

rotate\_right(parent);

ptr = parent;

parent = ptr->parent;

}

rotate\_left(grand\_parent);

std::swap(parent->color, grand\_parent->color);

ptr = parent;

}

}

}

root->color = BLACK;

}

struct node \*recoloring(struct node \*&a, struct node \*&b, struct node \*&c) {

a->color = RED;

b->color = c->color = BLACK;

return a;

}

struct node \*new\_node(int value)

{

struct node \*temp = (struct node \*)malloc(sizeof(struct node));

temp->value = value;

temp->left = temp->right = temp->parent = nullptr;

temp->color = RED;

return temp;

}

struct node\* bst\_insert(struct node\* root, struct node\* ptr)

{

/\* If the tree is empty, return node \*/

if (root == nullptr) return ptr;

/\* Otherwise, recur down the tree \*/

if (ptr->value < root->value) {

root->left = bst\_insert(root->left, ptr);

root->left->parent = root;

}

else if (ptr->value >= root->value) {

root->right = bst\_insert(root->right, ptr);

root->right->parent = root;

}

/\* return the (unchanged) node pointer \*/

return root;

}