Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ КОМП'ЮТЕРНИХ НАУК ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Катедра «Комп'ютерна інженерія та програмування»

ЗВІТ

про виконання лабораторної роботи №14

з навчальної дисципліни «Алгоритми та структури даних»

Варіант 5

Виконав студент:

Омельніцький Андрій Миколайович
Група: КН-1023б

Перевірив:

Старший викладач

Бульба Сергій Сергійович

Харків - 2024

Зміст

1 Мета роботи **2**

2 Завдання **2**

3 Хід виконання **2**

3.1 Червоно-чорне дерево 3

3.1.1 Node 3

3.1.2 RBTree 4

3.1.3 Форматуання дерева 5

3.2 Приклад роботи програми 7

4 Висновки **8**

1 Мета роботи

Набути досвіду практичної роботи розв’язання задач з використанням бінарних дерев.

Теми для попередньої роботи:

- балансування дерев;
- АВЛ-дерева;
- червоно-чорні дерева;
- дерева Хаффмана;
- дерева для арифметичних виразів.

2 Завдання

Розробити програму, що створює бінарне дерево та розв’язує індивідуальне завдання. Видати вміст дерева та результати індивідуального завдання на екран.

Побудувати червоно-чорне дерево. Визначити висоту дерева. Додати вузол із ключем, значення якого визначено висотою дерева.

3 Хід виконання

Для виконання завдання було обрано мову Rust. Увесь код також додатково був розміщений в GitHub репозитарії: https://github.com/blackgolyb/algos-labs.

3.1 Червоно-чорне дерево

3.1.1 Node

Для початку почнемо з розробки вузла нашого дерева також будемо будувати наше дередо за принципом ключ-значення. Тобто за впорядкування вузлів будуть відповідати саме ключі, а значення будуть просто зберігатися там.

```rust
use core::mem;

pub enum TreeNodePtr<K: Ord, V> {
    Some(Box<TreeNode<K, V>>),
    None,
}

impl<K: Ord, V> TreeNodePtr<K, V> {
    #[inline]
    pub fn take(&mut self) -> Self {
        mem::take(self)
    }

    pub fn height(&self) -> usize {
        match self {
            TreeNodePtr::Some(node) => node.height(),
            _ => 0,
        }
    }

    pub fn is_red(&self) -> bool {
        match self {
            TreeNodePtr::Some(node) => node.is_red,
            _ => false,
        }
    }

    pub fn left(&self) -> &Self {
        match self {
            TreeNodePtr::Some(node) => &node.left,
            _ => self,
        }
    }

    pub fn right(&self) -> &Self {
        match self {
            TreeNodePtr::Some(node) => &node.right,
            _ => self,
        }
    }
}

impl<K: Ord, V> Default for TreeNodePtr<K, V> {
    fn default() -> Self {
        TreeNodePtr::None
    }
}

pub struct TreeNode<K: Ord, V> {
    pub key: K,
    pub value: V,
    pub left: TreeNodePtr<K, V>,
    pub right: TreeNodePtr<K, V>,
    pub is_red: bool,
}

impl<K: Ord, V> TreeNode<K, V> {
    pub fn create(key: K, value: V) -> Self {
        TreeNode {
            key,
            value,
            left: TreeNodePtr::None,
            right: TreeNodePtr::None,
            is_red: true,
```

```
        }
    }

    pub fn height(&self) -> usize {
        let left = self.left.height();
        let right = self.right.height();
        left.max(right) + 1
    }
}
```

3.1.2 RBTree

```
use super::node::{TreeNode, TreeNodePtr};
use std::cmp::Ordering;

pub struct RBTree<K: Ord, V> {
    pub(super) root: TreeNodePtr<K, V>,
}

impl<K: Ord, V> RBTree<K, V> {
    pub fn new() -> Self {
        RBTree {
            root: TreeNodePtr::None,
        }
    }

    pub fn height(&self) -> usize {
        self.root.height()
    }

    pub fn put(&mut self, key: K, value: V) {
        self.root = RBTree::put_node(&mut self.root, key, value);
        if let TreeNodePtr::Some(h) = &mut self.root {
            h.is_red = false; //Tree root is always black
        }
    }

    pub fn get(&self, search_key: K) -> Option<&V> {
        let mut current = &self.root;
        while let TreeNodePtr::Some(node) = current {
            match search_key.cmp(&node.key) {
                Ordering::Less => current = &node.left,
                Ordering::Greater => current = &node.right,
                Ordering::Equal => return Some(&node.value),
            }
        }
        None
    }

    fn put_node(node: &mut TreeNodePtr<K, V>, new_key: K, new_value: V) -> TreeNodePtr<K, V> {
        match node {
            TreeNodePtr::None => {
                return TreeNodePtr::Some(Box::new(TreeNode::create(new_key, new_value)))
            }
            TreeNodePtr::Some(node) => match new_key.cmp(&node.key) {
                Ordering::Less => {
                    node.left = RBTree::put_node(&mut node.left, new_key, new_value);
                }
                Ordering::Greater => {
                    node.right = RBTree::put_node(&mut node.right, new_key, new_value);
                }
                Ordering::Equal => node.value = new_value,
            },
        }
        // Red-Black Tree Balancing
        if node.right().is_red() && !node.left().is_red() {
            //Case 2
            *node = RBTree::rotate_left(node.take());
        }
        if node.left().is_red() && node.left().left().is_red() {
            //Case 4
            *node = RBTree::rotate_right(node.take());
```

```
        }
        if node.left().is_red()      && node.right().is_red()    {
            //Case 3, 4
            RBTree::flip_colors(node);
        }
        node.take()
    }

    fn rotate_left(mut node: TreeNodePtr<K, V>) -> TreeNodePtr<K, V> {
        debug_assert!(node.right().is_red());
        if let TreeNodePtr::Some(h) = &mut node {
            let mut right_node = h.right.take();
            if let TreeNodePtr::Some(x) = &mut right_node {
                x.is_red  = h.is_red;
                h.is_red  = true;

                h.right   = x.left.take();
                x.left    = node;
            }
            return right_node;
        }
        node
    }

    fn rotate_right(mut node: TreeNodePtr<K, V>) -> TreeNodePtr<K, V> {
        debug_assert!(node.left().is_red());
        if let TreeNodePtr::Some(h) = &mut node {
            let mut left_node = h.left.take();
            if let TreeNodePtr::Some(x) = &mut left_node {
                x.is_red  = h.is_red;
                h.is_red  = true;
                h.left    = x.right.take();
                x.right   = node;
            }
            return left_node;
        }
        node
    }

    fn flip_colors(node:    &mut TreeNodePtr<K, V>) {
        debug_assert!(!node.is_red());
        debug_assert!(node.left().is_red());
        debug_assert!(node.right().is_red());
        if let TreeNodePtr::Some(h) = node {
            h.is_red  = true;
            if let TreeNodePtr::Some(left)  = &mut h.left  {
                left.is_red    = false;
            }
            if let TreeNodePtr::Some(right)  = &mut h.right  {
                right.is_red    = false;
            }
        }
    }
}

impl<K: Ord, V> Default for RBTree<K, V> {
    fn default()  -> Self  {
        Self::new()
    }
}
```

3.1.3 Форматування дерева

Для нашого дерева також імплементуємо інтерфейс виводу на екран, щоб було зручно ливитися його вміст. Будемо виводити тільки ключ, щоб не накладати обмеження на зберігаймий тип.

```
use core::fmt;
use std::fmt::Write;

```

```
use super::lib::RBTree;
use super::node::{TreeNode, TreeNodePtr};

impl<K: Ord + fmt::Display, V> fmt::Display for RBTree<K, V> {
    fn fmt(&self, formatter: &mut fmt::Formatter) -> fmt::Result {
        match &self.root {
            TreeNodePtr::Some(node) => write!(formatter, "{}", &node),
            _ => write!(formatter, "Empty"),
        }
    }
}

fn write_layers(layers: &Vec<bool>, formatter: &mut fmt::Formatter, pos: u8) -> fmt::Result {
    let n = layers.len();
    let mut s: String = String::new();
    if n > 1 {
        s = layers[..n - 1]
            .iter()
            .map(|l| match l {
                true => "│   ",
                false => "    ",
            })
            .collect();
    }
    if n != 0 {
        s += match pos {
            1 => "├── ",
            2 => "└── ",
            _ => "",
        }
    }
    write!(formatter, "{}", s)
}

fn write_nil(layers: &Vec<bool>, formatter: &mut fmt::Formatter, pos: u8) -> fmt::Result {
    write_layers(layers, formatter, pos)?;
    writeln!(formatter, "\x1b[40mNIL\x1b[0m")
}

impl<K: Ord + fmt::Display, V> TreeNode<K, V> {
    fn display_subtree(
        &self,
        formatter: &mut fmt::Formatter,
        levels: &mut Vec<bool>,
        position: u8,
    ) -> fmt::Result {
        let c = if self.is_red { 41 } else { 40 };
        write_layers(levels, formatter, position)?;
        writeln!(formatter, "\x1b[{}m{}\x1b[0m", c, self.key)?;

        levels.push(true);
        let n_levels = levels.len();
        if let TreeNodePtr::Some(leaf) = &self.left {
            leaf.display_subtree(formatter, levels, 1)?;
        } else {
            write_nil(levels, formatter, 1)?;
        }
        levels[n_levels - 1] = false;
        if let TreeNodePtr::Some(leaf) = &self.right {
            leaf.display_subtree(formatter, levels, 2)?;
        } else {
            write_nil(levels, formatter, 2)?;
        }

        levels.pop();
        Ok(())
    }
}

impl<K: Ord + fmt::Display, V> fmt::Display for TreeNode<K, V> {
    fn fmt(&self, formatter: &mut fmt::Formatter) -> fmt::Result {
        let mut levels = Vec::new();
        self.display_subtree(formatter, &mut levels, 0)
    }
}
```

3.2 Приклад роботи програми

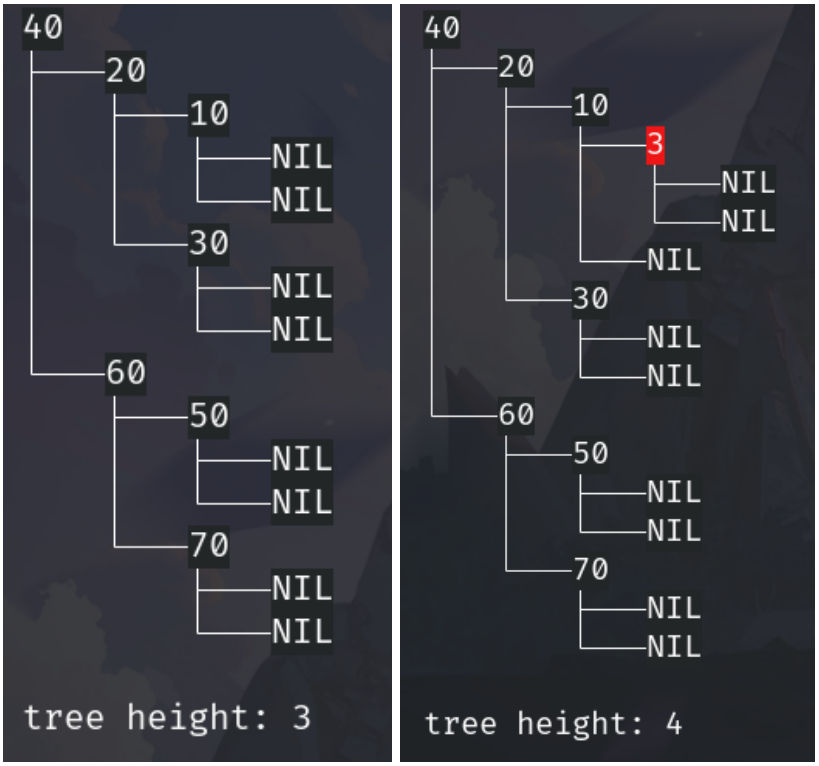
Код програми для перевірки:

```
use crate::libs::rb_tree::RBTree;

pub fn main() {
    let mut tree = RBTree::<i32, i32>::new();
    let data = [1, 2, 3, 4, 5, 6, 7];

    for i in data {
        tree.put(i * 10, i * 10);
        // println!("{tree}");
    }

    println!("{tree}");
    let h = tree.height() as i32;
    println!("tree height: {h}\n\n");
    tree.put(h, h);
    let h = tree.height();
    println!("{tree}");
    println!("tree height: {h}");
}
```



(а) (б)

Рис. 1. Приклад роботи програми (а) до додавання вузла (б) після додавання вузла

4 Висновки

В ході виконання лабораторної робити було створено червоно-чорне дерево та функції роботи з ним.