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 №12

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

Закріпити теоретичні знання та набути практичного досвіду впорядкування набору статичних та динамічних структур даних.
Теми для попередньої роботи:

- масиви та списки;
- алгоритми сортування вибором та включенням;
- алгоритми сортування вибором на деревах, розподілом та злиттям.

2 Завдання

Написати програму, що реалізує три алгоритми сортування набору даних згідно з табл. 12.1.

Визначити кількість порівнянь та обмінів для початкових наборів даних, що містять різну кількість елементів (50, 1000, 5000, 10000, 50000).

Оцінити час сортування. Дослідити вплив початкової впорядкованості набору даних (відсортований, відсортований у зворотному порядку, випадковий).

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 Пірамідальне сортування

```rust
use super::super::sort_preamble::*;

fn heapify<T: Ord>(sort: &mut HeapSort, arr: &mut Vec<T>, n: usize, i: usize) {
    let mut largest = i;
    let left = 2 * i + 1;
    let right = 2 * i + 2;

    if left < n && sort.gt(arr, left, largest) {
        largest = left;
    }

    if right < n && sort.gt(arr, right, largest) {
        largest = right;
    }

    if largest != i {
        sort.swap(arr, i, largest);
        heapify(sort, arr, n, largest);
    }
}

sort! {
    HeapSort | args: SortArgs<T> | {
        let arr = args.0;
        let sort = args.1;

        let n = arr.len();
        for i in (0..n / 2).rev() {
            heapify(sort, arr, n, i);
        }

        for i in (1..n).rev() {
            sort.swap(arr, 0, i);
            heapify(sort, arr, i, 0);
        }
    }
}
```

3.2 Порозрядне цифрове сортування

```
use super::super::sort_preamble::*;

fn counting_sort(sort: &mut RadixSort, arr: &mut Vec<i64>, exp: i64) {
    let n = arr.len();
    let mut output = vec![0; n];
    let mut count = vec![0; 10];

    for &num in arr.iter() {
        let index = (num / exp % 10) as usize;
        count[index] += 1;
    }

    for i in 1..10 {
        count[i] += count[i - 1];
    }

    for &num in arr.iter().rev() {
        let index = (num / exp % 10) as usize;
        output[count[index] as usize - 1] = num;
        sort.log_swap();
        count[index] -= 1;
    }

    for i in 0..n {
        arr[i] = output[i];
        sort.log_swap();
    }
}

sort! {
    RadixSort<i64> |args: SortArgs<i64>| {
        let arr = args.0;
        let sort = args.1;

        let mut max_i = 0;
        for i in 0..arr.len() {
            if sort.gt(arr, i, max_i) {
                max_i = i;
            }
        }
        let max = arr[max_i];

        let mut exp = 1;
        while max / exp > 0 {
            counting_sort(sort, arr, exp);
            exp *= 10;
        }
    }
}
```

3.3 «Кишенькове» сортування

Для цього сортування додамо додаткову умову для сортованого типа. Сортований тип повинен буде реалізовувати інтерфейс AsIndex який буде дозволяти отримати порядковий номер для кожного елемента.

```rust
use super::super::sort_preamble::*;
use super::super::variants::InsertionSort;

struct Bucket<T> {
    hash: usize,
    values: Vec<T>,
}

impl<T> Bucket<T> {
    fn new(hash: usize, value: T) -> Bucket<T> {
        Bucket {
            hash,
            values: vec![value],
        }
    }
}

sort!(
    BucketSort + AsIndex
    |args: SortArgs<T>| {
        let arr = args.0;
        let sort = args.1;

        let mut sub_sort = InsertionSort::new();

        let mut buckets: Vec<Bucket<T>> = vec![];
        let n = arr.len();

        for _ in 0..n {
            let val = arr.pop().unwrap();
            let hash = val.as_index();
            sort.log_swap();
            match buckets.binary_search_by(|bucket| bucket.hash.cmp(&hash)) {
                Ok(index) => buckets[index].values.push(val),
                Err(index) => buckets.insert(index, Bucket::new(hash, val)),
            }
        }

        unsafe {
            arr.set_len(n);
        }

        let mut i = 0;
        for mut bucket in buckets.into_iter() {
            sub_sort.sort(&mut bucket.values);
            sort.logger.marge_sub_logger(sub_sort.logger());
            for value in bucket.values {
                sort.log_swap();
                arr[i] = value;
                i += 1;
            }
        }
    }
);
```

3.4 Приклад роботи програми

Код програми для перевірки:

```rust
use crate::libs::sort::{
    testing::test_perf,
    variants::{BucketSort, HeapSort, RadixSort},
    AsIndex,
};

impl AsIndex for i64 {
    fn as_index(&self) -> usize {
        (self / 10) as usize
    }
}

pub fn main() {
    let test_cases: Vec<usize> = vec![20, 1000, 5000, 10000, 50000];

    let mut sort = HeapSort::new();
    let file_name = "heap.csv".to_string();
    test_perf(&mut sort, file_name, test_cases.clone());

    let mut sort = BucketSort::new();
    let file_name = "bucket.csv".to_string();
    test_perf(&mut sort, file_name, test_cases.clone());

    let mut sort = RadixSort::new();
    let file_name = "radix.csv".to_string();
    test_perf(&mut sort, file_name, test_cases.clone());
}
```

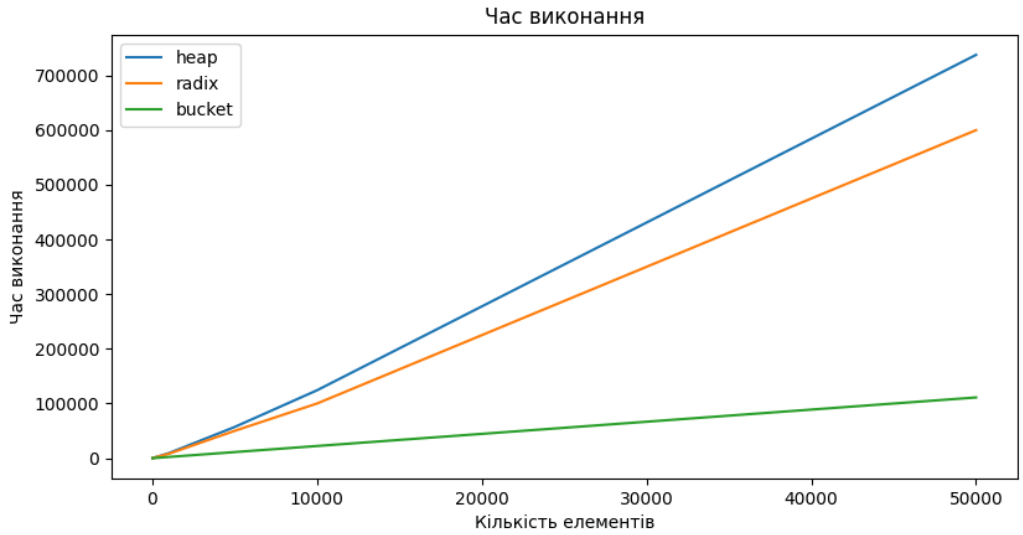

Рис. 1. Залежність часу пошуку від кількості елементів

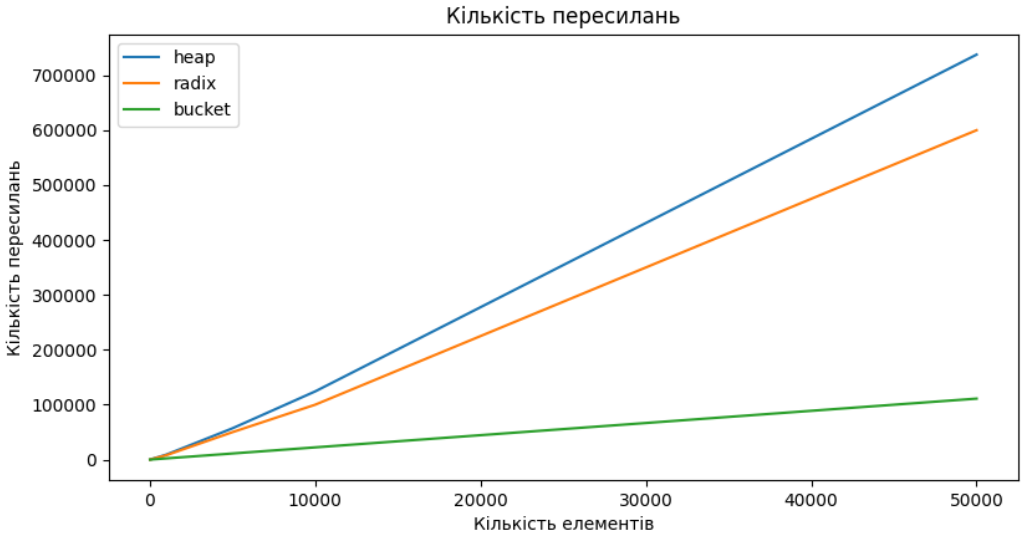

Рис. 2. Залежність кількості пересилань від кількості елементів

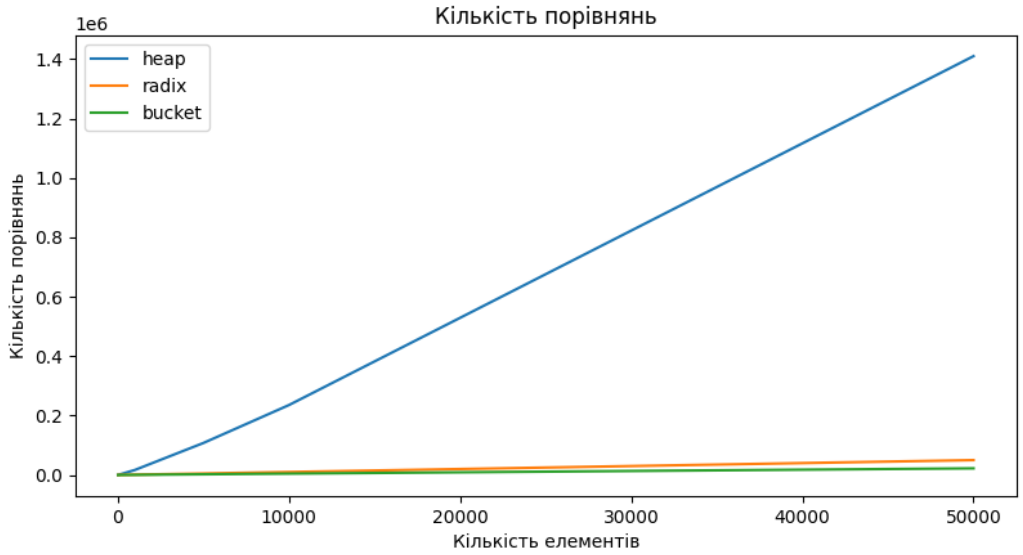

Рис. 3. Залежність кількості порівнянь від кількості елементів

4 Висновки

В ході виконання лабораторної робити було створено такі сортування: heap sort, radix sort, bucket sort. Також було протестовано їх на різних обємах даних та побудовано графіки для наглядної демонстрації їх характеристик.