Comparing different data structures in merging efficiency.

Changhui (Eric) Zhou

April 25, 2025

word count: ???

Contents

1 Introduction **2**

2 Theory **2**

2.1 Data structure terminology 2

References **4**

1 Introduction

A *data structure* is a particular way of organising data in a computer so that it can be used effectively (GeeksforGeeks, n.d.). Designing and choosing more efficient data structures has always been a great persuit for computer scientists, for optimal data structures can save huge amount of computing resources, especially in face of large amount of data. Basic data structures include ordered data structures like arrays, linked lists and binary search trees and unordered data structures like hashtables.

For ordered data structures, merging two or more instances of them while maintaining its ordered property may be frequently used in practice. For example, to investigate the factors affacting the school grade, data from different schools may be grouped and merged according to various factors. The efficiency of combination varies significantly based on the data structure itself and the algorithm used in the process.

This essay will focus on invesitgating the theoratical time complexity (need definitions aa) and actual performance of merging algorithms of different data structures, namely arrays, and BSTs, which are the most commonly used data structure in real life.

Research question: to be done aa

2 Theory

2.1 Data structure terminology

When a homogeneous relation (a binary relation between two elements) $\leq$ on a set of element X satisfies

We say $P = (X, \leq)$ is a total order.

Simple C++ Code Example

Listing 1: Basic C++ Code

```cpp
#include <iostream>

// Simple factorial function
int factorial(int n) {
```

```cpp
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}

int main() {
    std::cout << "Factorial of 5: "
              << factorial(5) << std::endl;
    return 0;
}
```

Including External Code

Listing 2: External C++ File

```cpp
#include<vector>
#include<random>
#include<numeric> // iota
#include<time.h>
#include<tuple>
//#include<algorithm> // shuffle

using std::vector, std::tuple, std::mt19937_64, std::make_tuple;
typedef tuple<int, int, int> t3i;

class fenwick_tree{
    vector<int>*dat = nullptr;
    fenwick_tree(int len){
        dat = new std::vector<int>(len);
    }
};

vector<t3i> generate(int num_sets, int num_merge, int
    num_remove_if, int rand_seed){
    vector<t3i> ret = {};
    mt19937_64 rd = mt19937_64(rand_seed);
    for(int rcm = num_merge, rcr = num_remove_if; rcm + rcr > 0;
        ){
        if(rd() % (rcm + rcr) < rcm)
```

```
            ret.push_back(make_tuple(1, rd() % num_sets, rd() %
                num_sets));
        else
            ret.push_back(make_tuple(2, rd() % num_sets, 0));
    }

    //return 0;
    //std::vector<int> fa = std::vector<int>(num_sets);
    //iota(fa.begin(), fa.end(), 1);
}

vector<t3i> generate(int num_sets, int num_merge, int
    num_remove_if){
    return(generate(num_sets, num_merge, num_remove_if, time(NULL
        )));
}
```

References

GeeksforGeeks. (n.d.). *Introduction to data structures.* Retrieved from https://www.geeksforgeeks.org/introduction-to-data-structures/ ([Accessed April 13, 2025])