ICCS315: Assignment 1

Vanessa Rujipatanakul

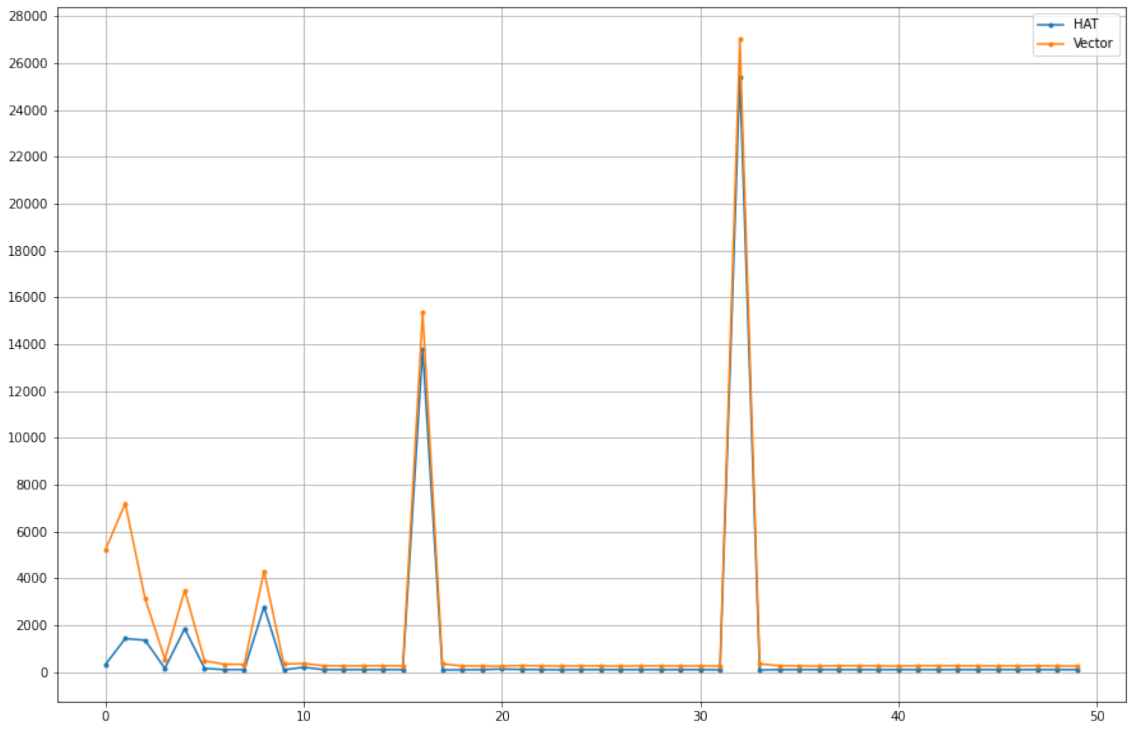
30/01/2023

Note: Implementations and resources are uploaded to this github

---

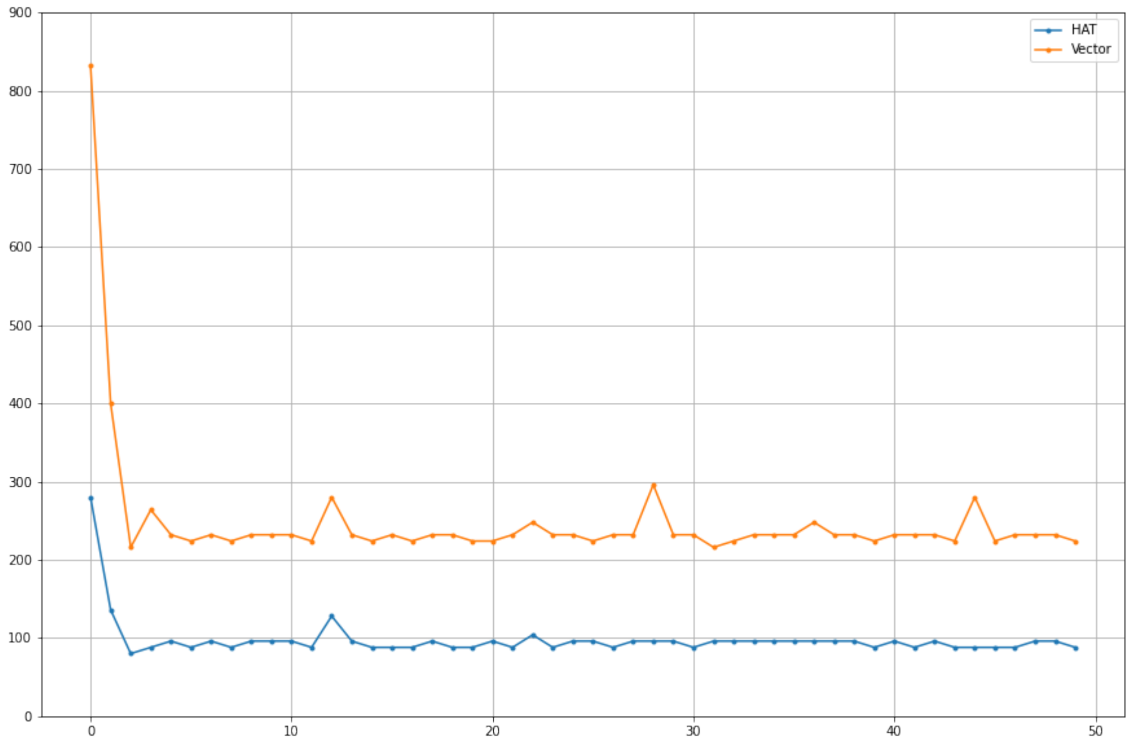
1: ***Resizable Arrays***

(a) Append Latency (compared to built-in Vector)
HAT ≈ 96 cycles
Vector ≈ 260 cycles



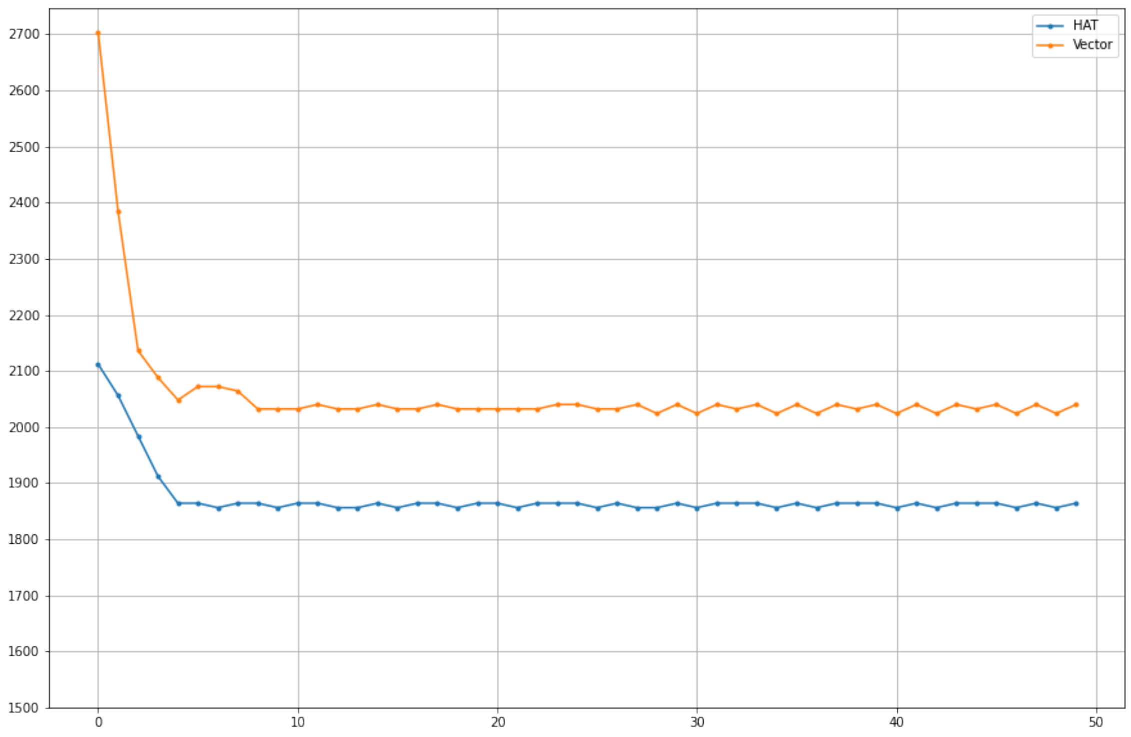
x-axis = N-th Iterations
y-axis = No. of Cycles

(b) Access Latency (compared to built-in Vector)
HAT ≈ 96 cycles
Vector ≈ 232 cycles



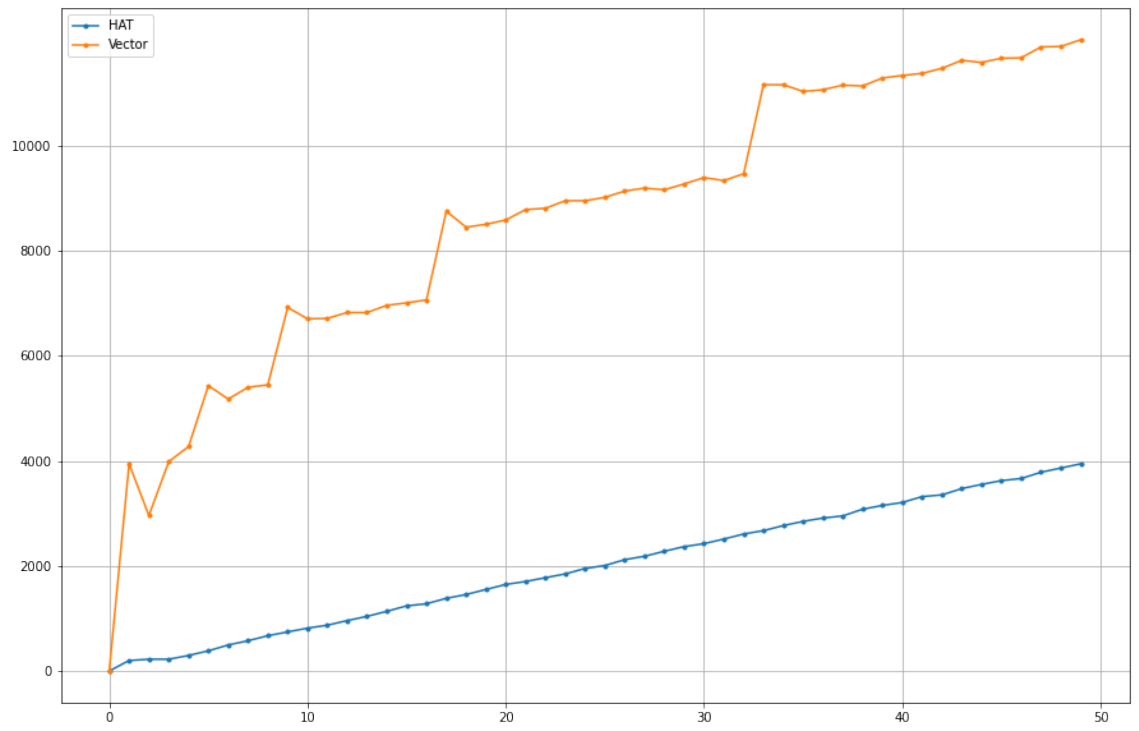
x-axis = N-th Iterations
y-axis = No. of Cycles

(c) Scan Latency (compared to built-in Vector)
HAT ≈ 1864 cycles
Vector ≈ 2036 cycles



x-axis = N-th Iterations
y-axis = No. of Cycles

(d) Overall Latency (compared to built-in Vector)
HAT ≈ 1980 cycles
Vector ≈ 8984 cycles

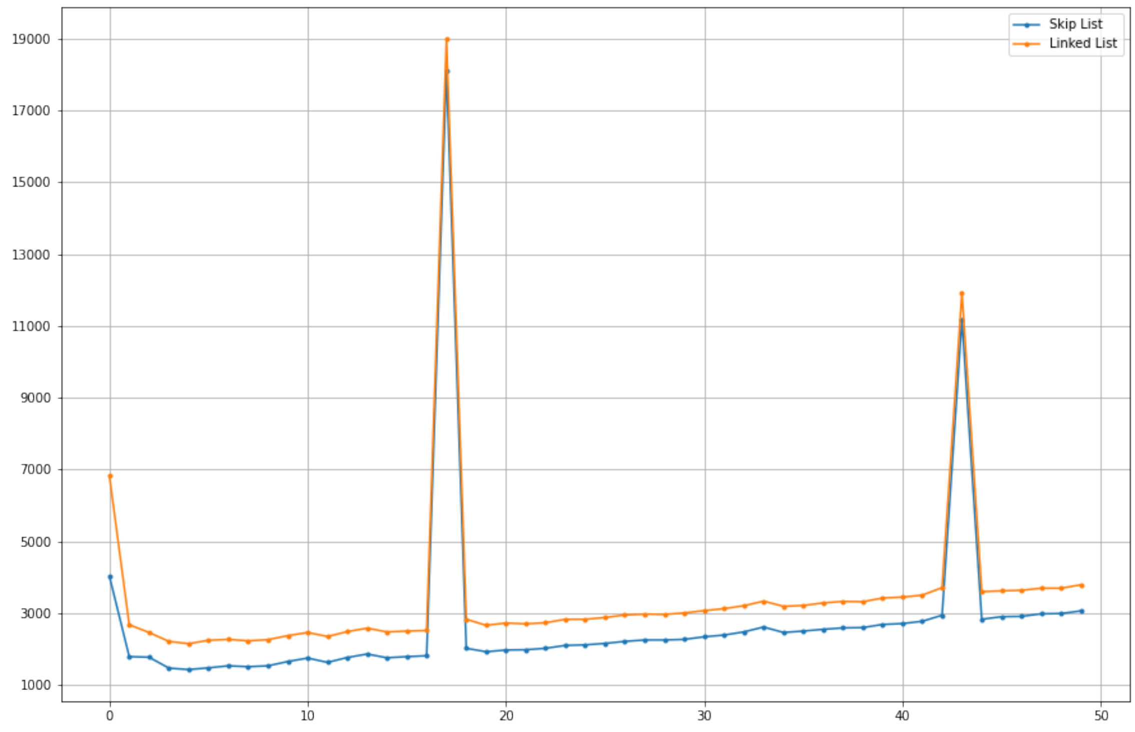


x-axis = No. of Elements
y-axis = No. of Cycles

3: ***Skip Lists***

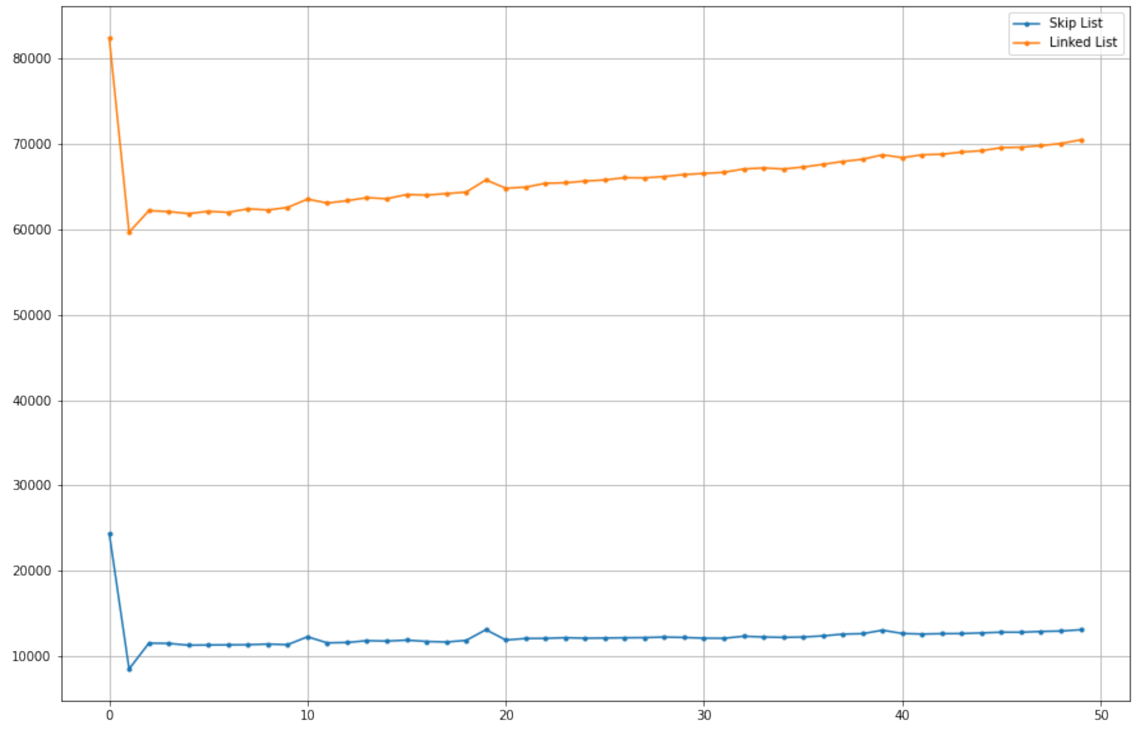
(a)
As Skip List is a data structure that is built upon the general idea of a linked list. The purpose of this experiment is to determine the performance difference between the 'insert' and 'search' functions of the two data structures by studying per-operation latency.

Insert Latency (compared to built-in Linked List)
Skip List ≈ 2236 cycles
Linked List ≈ 2960 cycles



x-axis = N-th Iterations
y-axis = No. of Cycles

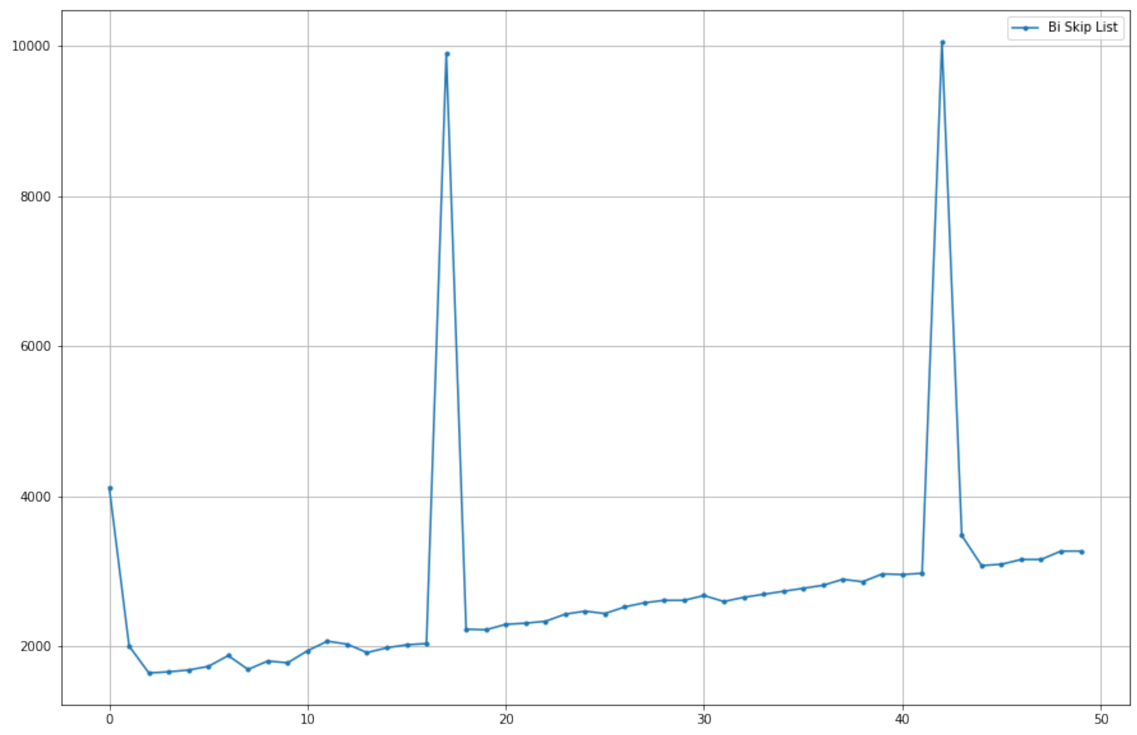
Search Latency (compared to built-in Linked List)
Skip List ≈ 12212 cycles
Linked List ≈ 65856 cycles



x-axis = N-th Iterations
y-axis = No. of Cycles

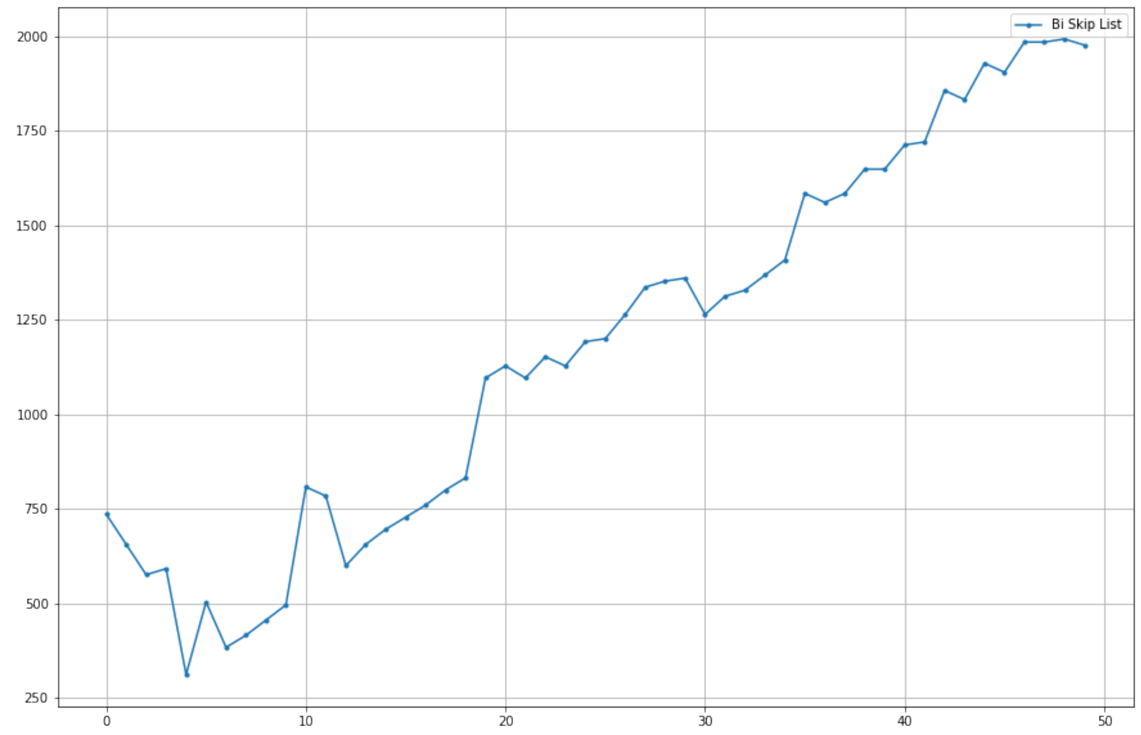
The built-in ‘list‘ library in c++ doesn’t have a function that can be used to retrieved an element from the list. The implementation of the 'get' function utilizes the 'begin()' function which returns an iterator of the list in order to scan through all the elements and determine if the element exist in the list or not.

(b)
Insert Latency ≈ 2556 cycles



x-axis = N-th Iterations
y-axis = No. of Cycles

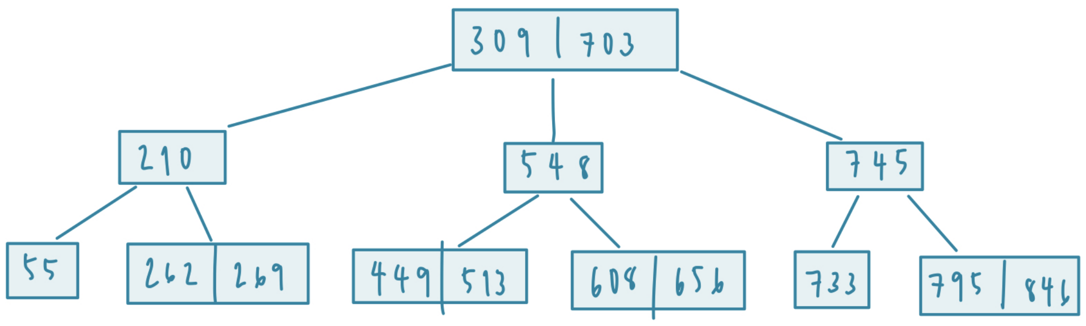
Search Latency ≈ 1196 cycles



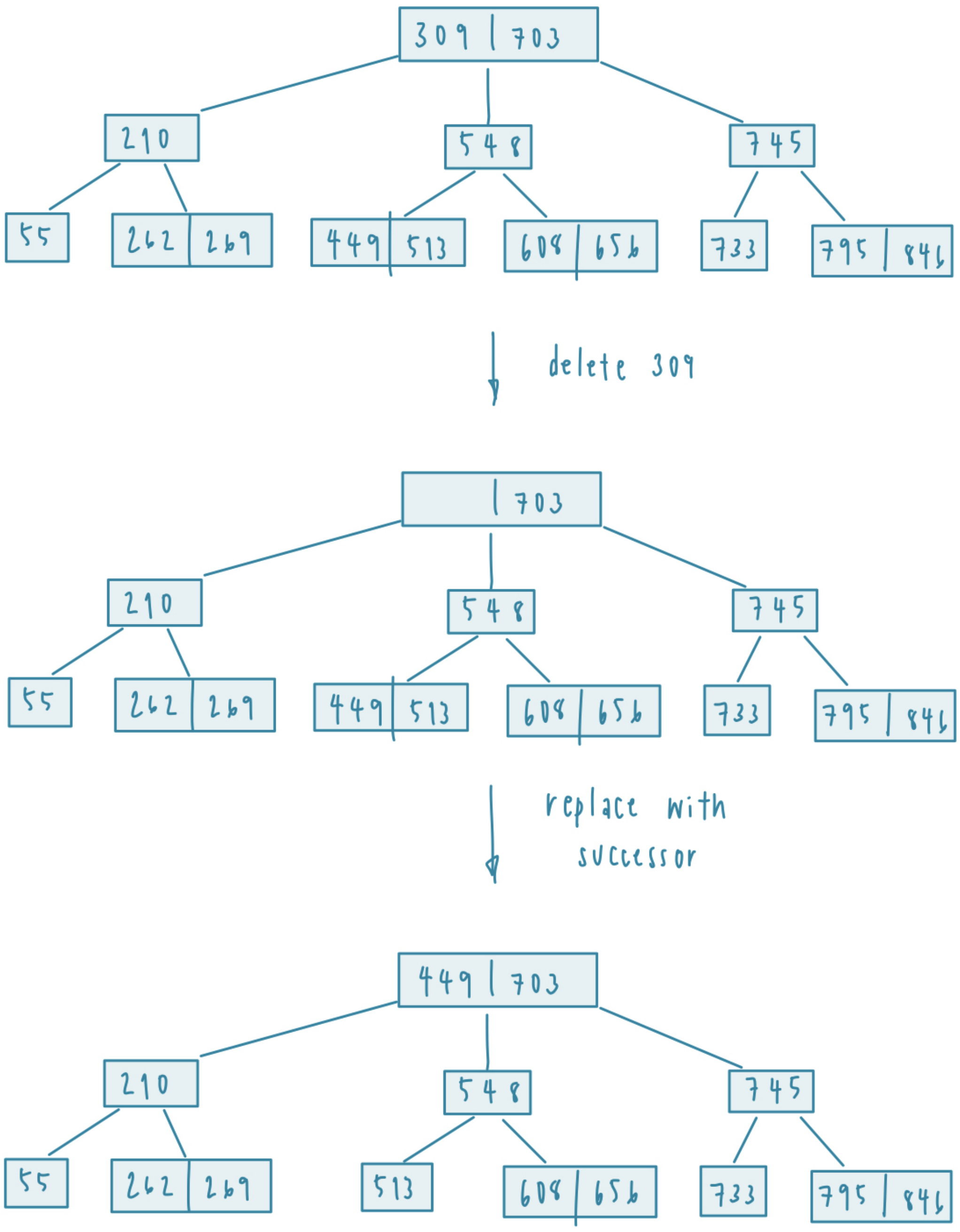
x-axis = N-th Iterations
y-axis = No. of Cycles

4: *(a, b) tree*

(a)



(b)



4: *B-Tree Speed*

An optimal value of b should be between 100 and 1000. This range seems to be the balance between minimizing the number of disk accesses and minimizing the space overhead per node. A smaller value of b means that each node contains fewer keys and values, reducing the space overhead per node. But this also increases the height of the tree, leading to more disk accesses and slower performance for operations that require disk access. A larger value of b means that each node contains more keys and values, reducing the height of the tree and the number of disk accesses required. However, this also increases the space overhead per node.