Fakulta riadenia a informatiky
Informatika

Domáce zadanie 4
Analýza výkonu ADT prioritný tabuľka

doc. Ing. **Miroslav Kvaššay** PhD.
STREDA 12, 13 Maroš Gorný, 5ZYI21
2021/2022

Obsah

Úloha 2

Zadanie – Overenie výkonu v scenári

Realizácie ADT tabuľka, ktoré ste implementovali v úlohe 1, otestujte v scenároch definovaných v Tab. 1. V každom scenári vykonajte spolu 100 000 operácií. Jednotlivé operácie sú v jednotlivých scenároch volané náhodne tak, aby na konci súhlasil podiel jednotlivých operácií (neplatí, že najskôr sa zavolajú operácie na vkladanie prvkov, potom operácie na mazanie prvkov a nakoniec operácie na vyhľadávanie prvkov). Parametre do operácií sú taktiež náhodné; kľúč je náhodné číslo z intervalu <0; 100 000>.

Tab. 1 Testovacie scenáre pre ADT tabuľka

Scenár	Podiel operácií		
	`insert`	`remove`	`tryFind`
A	20	20	60
B	40	40	20

V rámci analýzy výkonu v scenári je nutné merať len dĺžku trvania vybranej operácie. To znamená, že do merania **sa nesmie započítavať čas potrebný pre generovanie pomocných údajov**.

Testovanie – Implementácia

Použité inštancie a funkcie

Table<T>* - Pointer na prioritný front s ktorým budem pracovať.
SimpleTest - Inštancia s ktorou môžem merať čas
FileLogConsumer - Pointer na Logger s ktorým budem schopný ukladať dáta.

Random numbers

```
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_real_distribution<> distribution(0,100000);
distribution(gen)
```
- Random number between 0 and 100000

srand(time(NULL)); - Funkcia s ktorou môžem nastaviť náhodný seed pre funkciu rand().
rand() - Funkcia ktorá vráti náhodné číslo medzi 0 a RAND_MAX.

Logika testovania

Výber vhodnej implementácie

Konečný podiel operácií má byť rovný podielu zvoleného v tabuľke. Implementácia bude teda veľmi podobná ako v predošlom zadaní. To znamená, že si náhodne vygenerujem **100 čísel** a určím si hranice, v ktorých sa má daná metóda volať.

Implementácia hraníc pre výber metódy

Hranice sú určené typom scenáru, ale pre príklad môžeme povedať, že máme **4** metódy a každá má pravdepodobnosť 25 %, preto prvá hranica bude v bode **25**, druhá v bode **50**, tretia v bode **75** a štvrtá teda posledná v bode **100**.

Výber metódy

Keď sú **určené hranice** a vygenerované náhodné číslo **od 0 po 99**, môžeme určiť ktorá metóda sa zavolá.

V prípade že číslo bude pod prvou hranicou, teda bude **menšie ako 25**, tak sa zavolá **prvá metóda**. Ak číslo bude väčšie, ale bude **pod hranicou 50**, zavolá sa **druhá metóda** atď..

Taktiež musím splniť **podiel operácií**, takže budem počítať koľko krát sa mi operácia zavolala a ak dosiahla svoje maximum, potom budem vyberať prvú metódu ktorá toto maximum ešte nedosiahla **v poradí: insert, tryFind, remove.**

Výber metódy – remove (komplikácie)

V prípade metód remove môže nastať komplikácia v prípade,

- Že tabuľka je **prázdna** a teda nebudeme môcť zvoliť platný index. V takomto prípade skontrolujem či môžem zavolať metódu insert, ktorá bude fungovať aj pri prázdnej tabuľke a ak spĺňam všetky podmienky na jej zavolanie, tak ju zavolám.
- Že tabuľka je prázdna a metóda insert dosiahla svoj **maximálny počet volaní.** V takomto prípade si pri metóde remove pridám jeden nulový prvok do tabuľky, ktorý bude mať nulový kľúč. Pri vložení jedného prvku pri prázdnej tabuľke tieto parametre nehrajú žiadnu rolu a následne zavolám metódu remove pri ktorej odmeriam čas metódy remove a inkrementujem počítadlo volaní metódy remove.

Táto komplikácia mi ale zapríčiní **tvorenie zhlukov** danej metódy, avšak zväčša len pri konci pretože metóda insert bude mať malý percentuálny podiel a už bude plne využitá.

Volanie metódy a meranie času

Pri volaní metódy musím metódu zavolať s platným indexom. V sekcii Výber metódy – remove (komplikácie) som vyriešil problém pri metóde remove s veľkosťou prioritného frontu 0.

Pre trošku rýchlejšie vkladanie a vymazávanie prvkov s daným kľúčom, som si spravil jeden **Array** booleanov „**isInside_**“ a jeden **ArrayList** intov „**keyArray_**“. Keďže viem

konečnú veľkosť, môžem si do môjho **Array**-u vložiť boolean hodnoty na danom indexe, ktoré budú predstavovať, či je daný **kľúč vložený alebo nie**. **ArrayList** zase bude predstavovať všetky **kľúče ktoré sú prítomn**e v danej tabuľke.

Príklad:
Ak vložím prvok s kľúčom, ktorý bude mať hodnotu 3, tak sa v „isInside_“ na 3 indexe zmení boolean na TRUE a do ArrayListu sa pridá prvok 3.

Pri ďalšom vkladaní:
Ak budem chcem vložiť ďalší prvok, ktorý bude mať napríklad hodnotu kľúču 3, najprv sa pozriem na index 3 v Array „isInside“, či je daný prvok už dnu, alebo nie. Ak tam nie je, tak opakujem predošlí krok a teda ho tam vložím a ak tam je, tak vygenerujem nové náhodné číslo.

Ukladanie údajov

Dáta som ukladal do CSV súboru, kde

1. Stĺpec – **počet** prvkov v tabuľke po zavolaní metódy
2. Stĺpec – **čas** metódy v nanosekundách
3. Stĺpec – volaná **metóda**

Size	Time[ns]	Method
0	8800	tryFind
0	3700	tryFind
1	6800	insert
1	11100	tryFind
1	5000	tryFind
2	10800	insert

Testovanie – Výsledky

Na vyhodnotenie daných výsledkov som použil kontingenčnú tabuľku, kde na **X osi je veľkosť tabuľky** a na **osi Y je priemerný čas metód** v nanosekundách [ns].

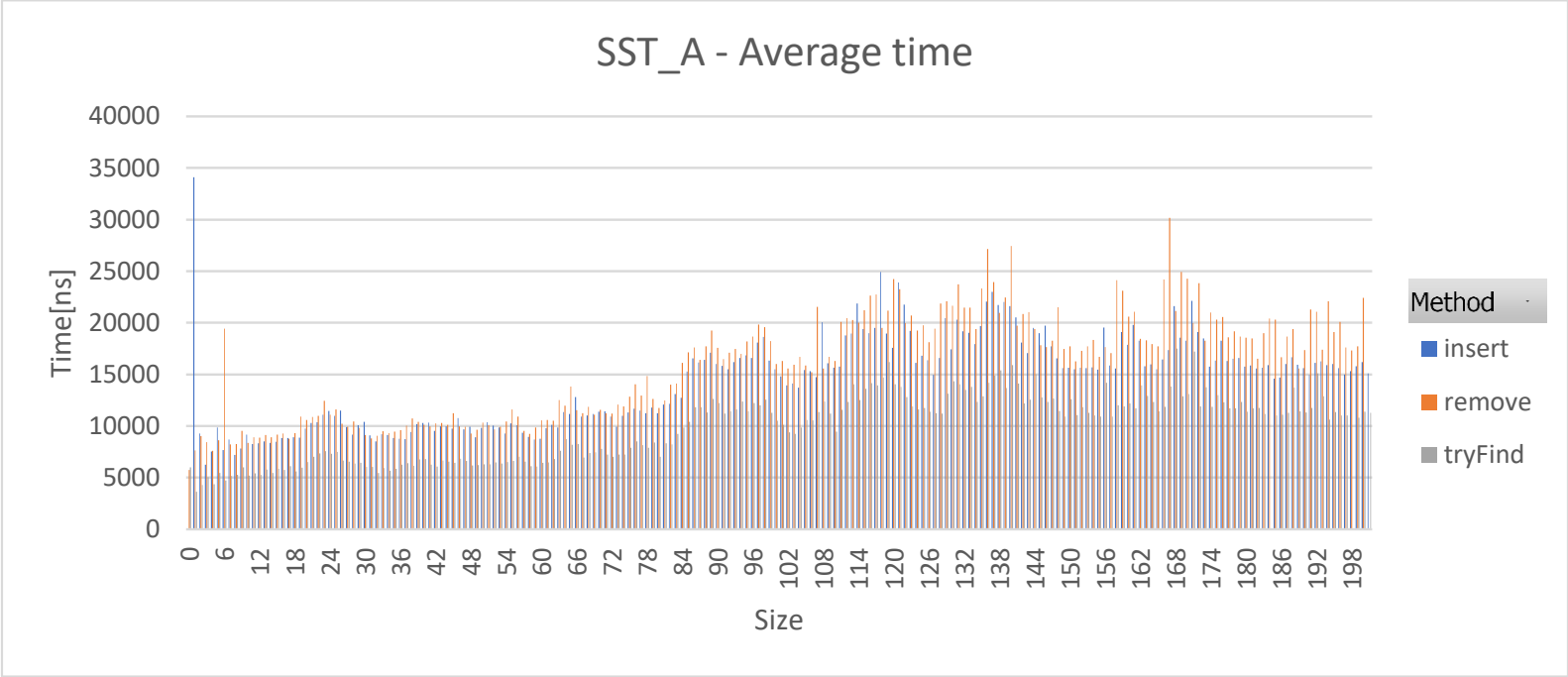
Tým pádom môžem porovnať scenáre ako celok a nemusím sa pozerať na jednotlivé metódy. Pozerám sa na ich spoločný priemer, takže porovnávam celkovú rýchlosť tabuľky a nie jej najlepšiu alebo najhoršiu zložitosť. Pre lepšie pochopenie rýchlosti daných metód, som pridal do grafu **rozdelenie podľa metód**.

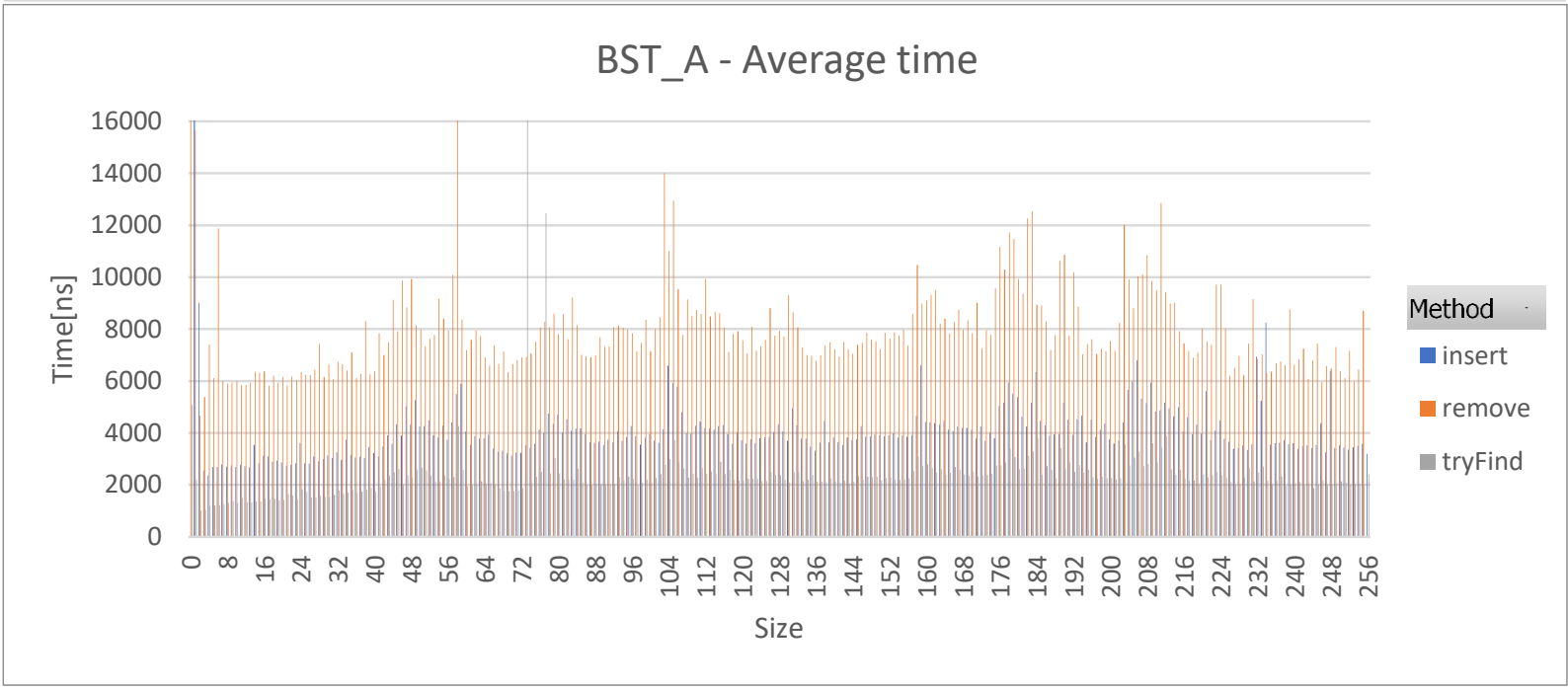
Scenár A

Zadanie

Scenár	Podiel operácií		
	`insert`	`remove`	`tryFind`
A	20	20	60

Prezentácia výsledkov





Vyhodnotenie výsledkov

Na základe grafov môžeme vidieť, že

SortedSequenceTable

- Ako celok jej **priemerná rýchlosť klesá**, teda čím väčšia táto tabuľka je, tým je rýchlosť pomalšia.
- Môžeme však vidieť, že zo začiatku rýchlosť klesá trošku rýchlejšie a následne sa klesanie rýchlosti spomaľuje.

BinarySearchTree

- Na **začiatku** aj na **konci** má približne **rovnakú rýchlosť**. V strede trošku spomalí, ale veľký rozdiel tam nie je.
- Pri danom grafe si však môžeme všimnúť, že **metóda remove** ma **najväčší vplyv** na rýchlosť.

SortedSequenceTable vs BinarySearchTree

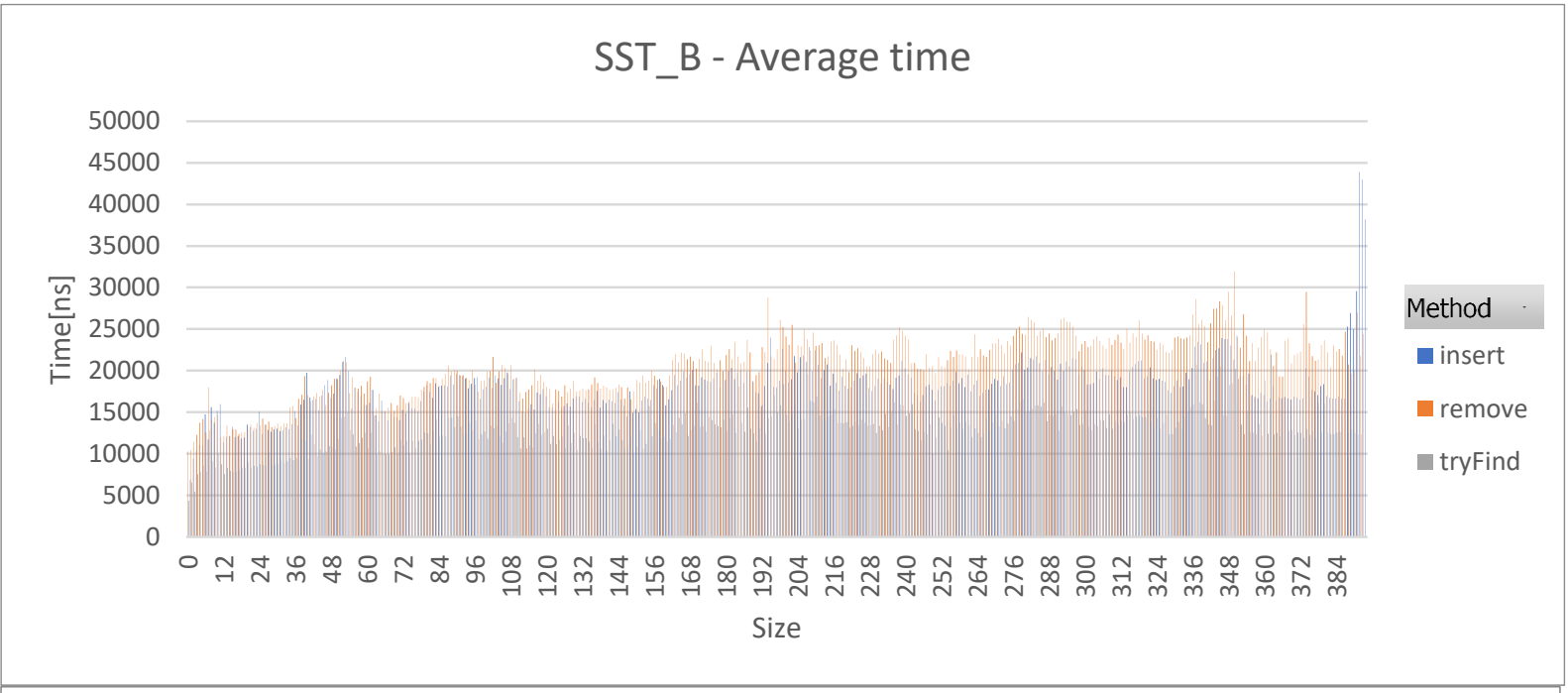
- V danom scenári je značne **rýchlejší BinarySearchTree**.
- Taktiež môžeme vidieť, že pri SortedSequenceTable sa rýchlosť pri počte prvkov pomaly spomaľuje, avšak pri BinarySearchTree rýchlosť dokonca začala v určitej časti klesať.

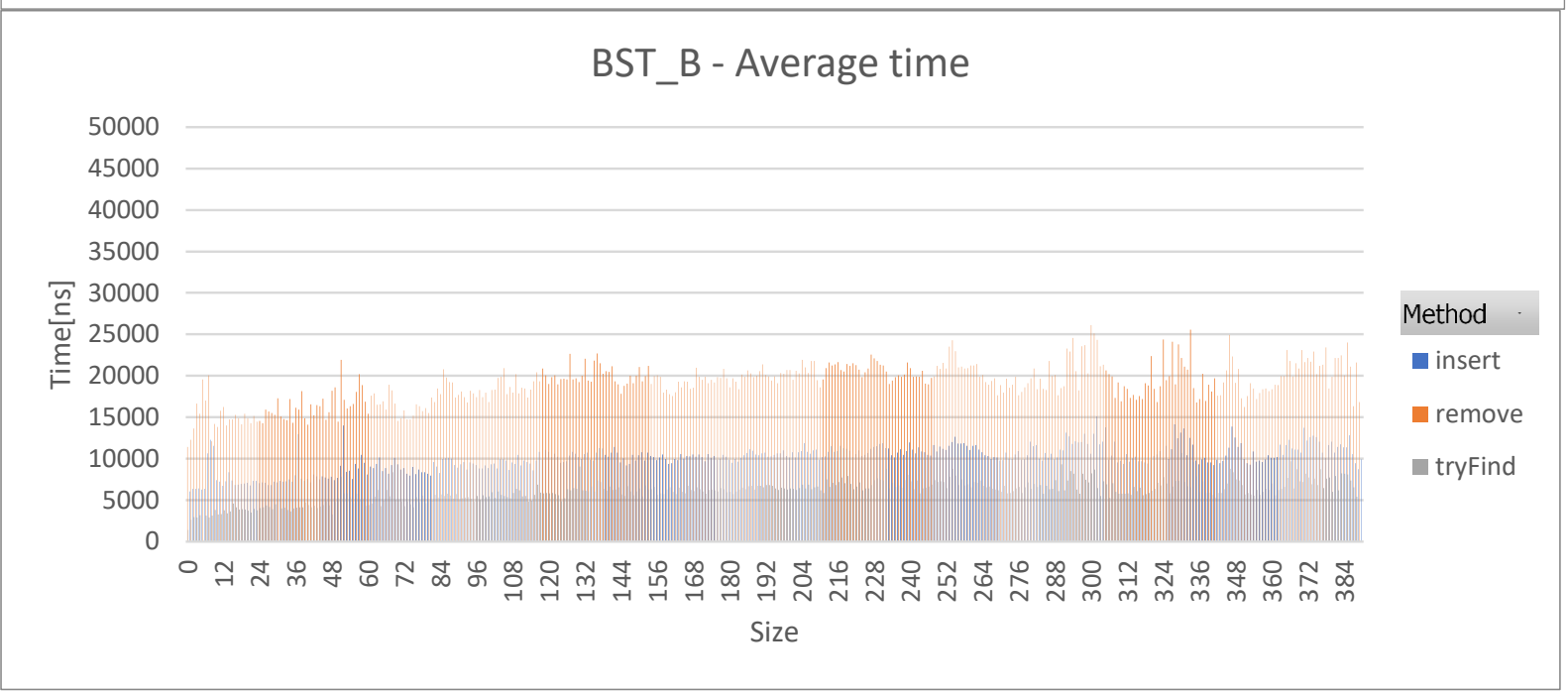
Scenár B

Zadanie

Scenár	Podiel operácií		
	`insert`	`remove`	`tryFind`
B	40	40	20

Prezentácia výsledkov





Vyhodnotenie výsledkov

Na základe grafov môžeme vidieť, že

SortedSequenceTable

- Ako celok jej **priemerná rýchlosť klesá**, teda čím väčšia táto tabuľka je, tým je rýchlosť pomalšia.
- Môžeme však vidieť, že zo začiatku rýchlosť klesá trošku rýchlejšie a následne sa klesanie rýchlosti spomaľuje.

BinarySearchTree

- Ako celok jej priemerná **rýchlosť klesá**, avšak len veľmi pomaly.
- Pri danom grafe si však môžeme všimnúť, že metóda **remove** ma **najväčší vplyv** na rýchlosť.

SortedSequenceTable vs BinarySearchTree

- V danom scenári sú **rýchlosti** takmer **rovnaké**.
- Avšak môžeme si všimnúť, že pri **BST** rýchlosť značne **ovplyvňuje** metóda **remove**, takže keby metódu remove nepoužívame, tabuľka BST by bola značne rýchlejšia.

Záver

Vďaka testom môžeme vidieť, že ak je tabuľka určená hlavne na **vyhľadávanie**, tak je určite lepšie použiť **BinarySearchTree**.

Taktiež si môžeme všimnúť, že pri väčšom **používaní** metód **insert** a **remove**, sú **rýchlosti** veľmi **podobné**. Avšak ak by sme metódu remove prestali používať, tak rýchlosť by bola rýchlejšia pri BinarySearchTree

V konečnom dôsledku by sme teda mohli vyvodiť záver taký, že ak by som bez ohľadu na situáciu potreboval využiť jednu z týchto dvoch tabuliek, zvolil by som **BinarySearchTree**, pretože dosahovala **podobné** alebo **lepšie výsledky** ako SortedSequenceTable.

Úloha 3

Zadanie – Analýza časových zložitostí

V rámci analýzy časových zložitostí je nutné otestovať rýchlosť operácií `insert`, `remove` a `tryFind` v závislosti od počtu prvkov a implementácie tabuľky a na základe nameranýcha spracovaných údajov **odhadnúť hornú asymptotickú zložitosť jednotlivých operácií**.

V rámci analýzy časových zložitostí vybraných operácií je nutné merať len dĺžku trvania vybranej operácie. To znamená, že do merania **sa nesmie započítavať čas potrebný pre generovanie pomocných údajov**.

Testovanie – Implementácia

Použité inštancie a funkcie

Table<T>* - Pointer na prioritný front s ktorým budem pracovať.
SimpleTest - Inštancia s ktorou môžem merať čas
FileLogConsumer - Pointer na Logger s ktorým budem schopný ukladať dáta.

Random numbers

```
std::random_device rd;
std::mt19937 gen(rd());
std::uniform_real_distribution<> distribution(0,100000);
```

distribution(gen) - Random number between 0 and 100000

srand(time(NULL)); - Funkcia s ktorou môžem nastaviť náhodný seed pre funkciu rand().
rand() - Funkcia ktorá vráti náhodné číslo medzi 0 a RAND_MAX.

Logika testovania

Testovanie budem robiť tak, že zavolám metódu **100 000 krát**.

- Pri metóde **insert** pôjde veľkosť od 0 po 99 999.
- Pri metóde **remove** pôjde veľkosť od 100 000 po 1.
  - Teda najprv sa tabuľka naplní a potom sa bude vyprázdňovať.
- Pri metóde **tryFind** pôjde veľkosť od 100 000 po 1.
  - Teda najprv sa tabuľka naplní potom sa zavolá metóda tryFind a následne sa tabuľka zmenší o jednu veľkosť
  -

Vzhľadom na zadanie, tu už nemusí byť kľúč určený z nejakého intervalu, preto ho budem vyberať z intervalu **0 až 1 000 000** a tým pádom je menšia šanca na to, že trafím ten istý kľúč. Ak by som ho náhodou trafil, tak vygenerujem nový náhodný kľúč.

Ukladanie údajov

Také isté ukladanie ako v úlohe 2, teda dáta som ukladal do CSV súboru, kde

1. Stĺpec – **počet** prvkov v tabuľke
2. Stĺpec – **čas** metódy v nanosekundách
3. Stĺpec – **volaná metóda**

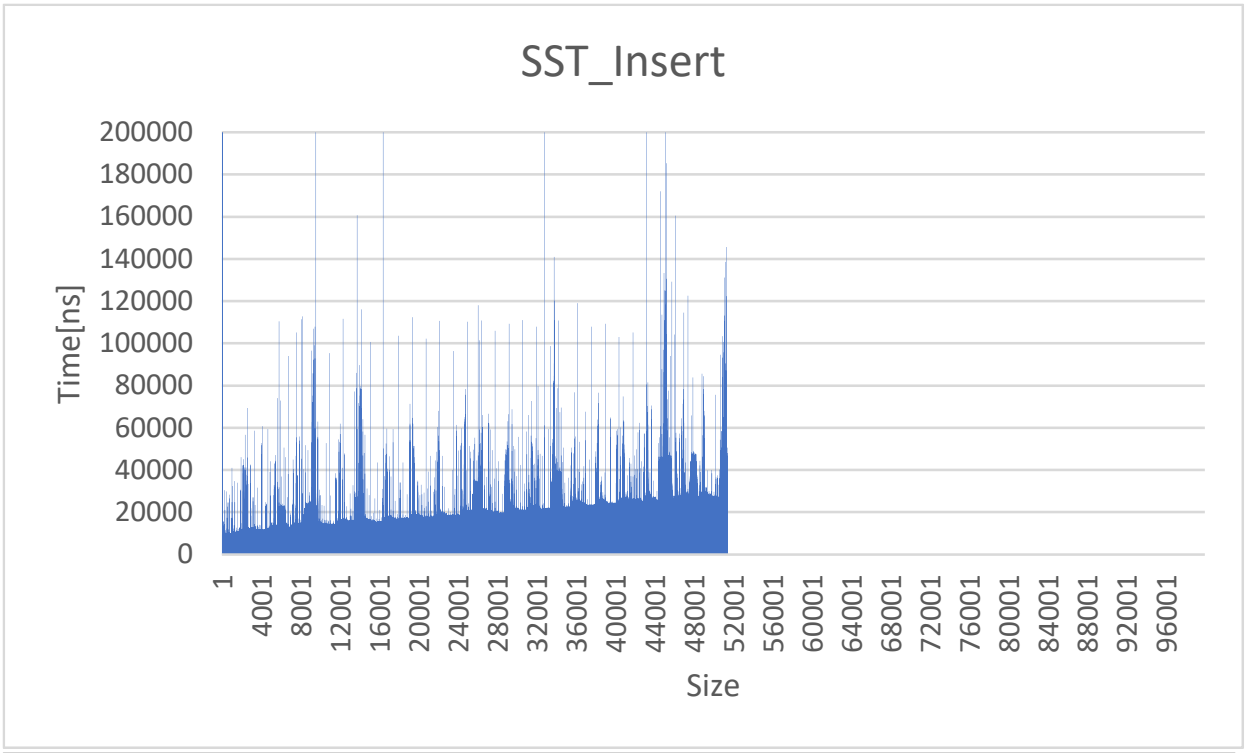
Size	Time[ns]	Method
1	1884100	insert
2	9400	insert
3	628900	insert
4	8400	insert
5	408000	insert
6	7300	insert
7	4100	insert

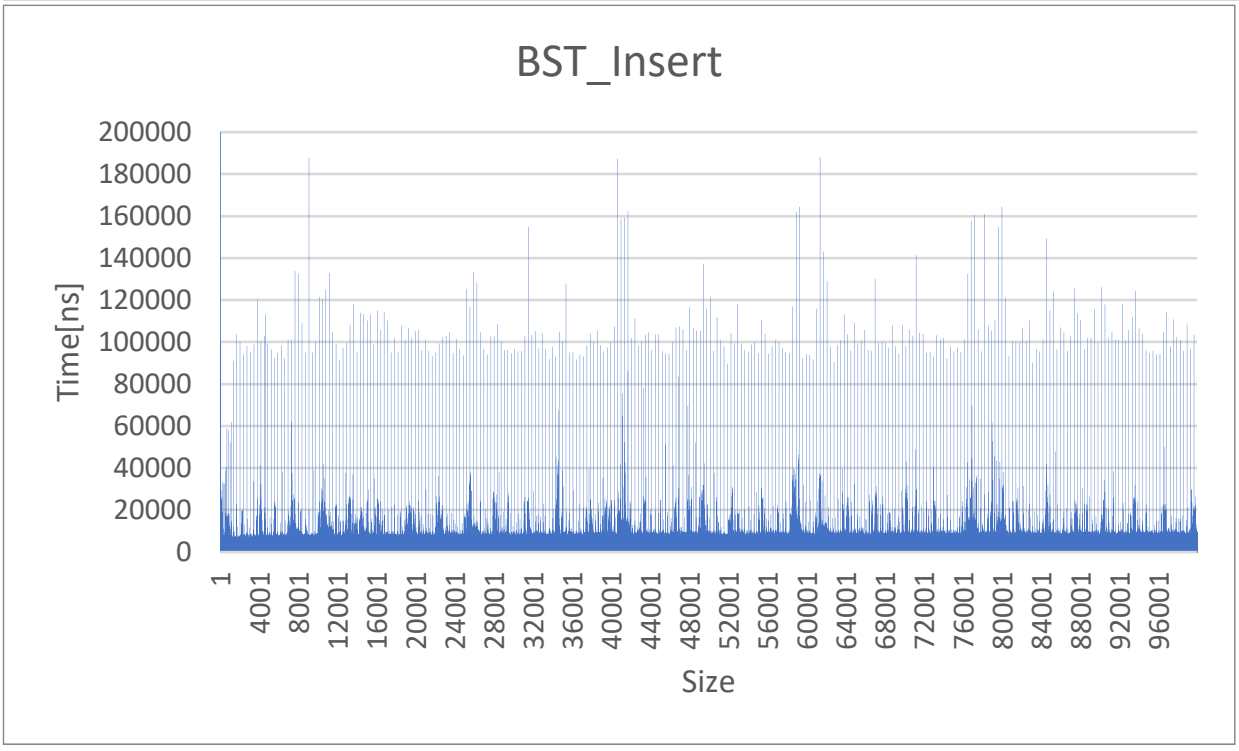
Testovanie – Výsledky

Na vyhodnotenie daných výsledkov som použil **column graph** , kde na **X** osi je **veľkosť tabuľky** a na **Y** osi je **čas** v nanosekundách.

Insert

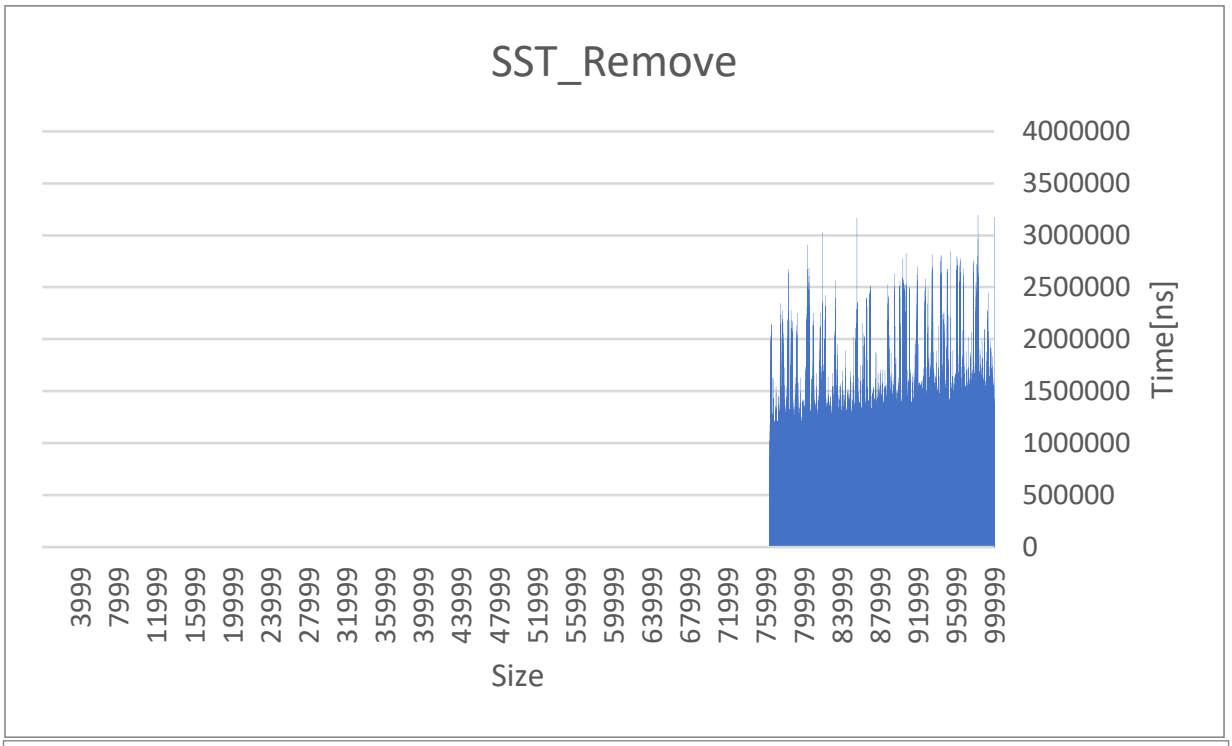
Môžeme vidieť, že **SortedSequenceTable spomaľuje** s **veľkosťou tabuľky**, avšak rýchlosť sa s pribúdajúcimi prvkami pomaly stabilizuje, preto bude zložitosť **logaritmická**. Pri **BinarySearchTree** je rýchlosť zo začiatku veľmi rýchla a následne sa **stabilizuje**, preto zložitosť odhadujem na **logaritmickú**.

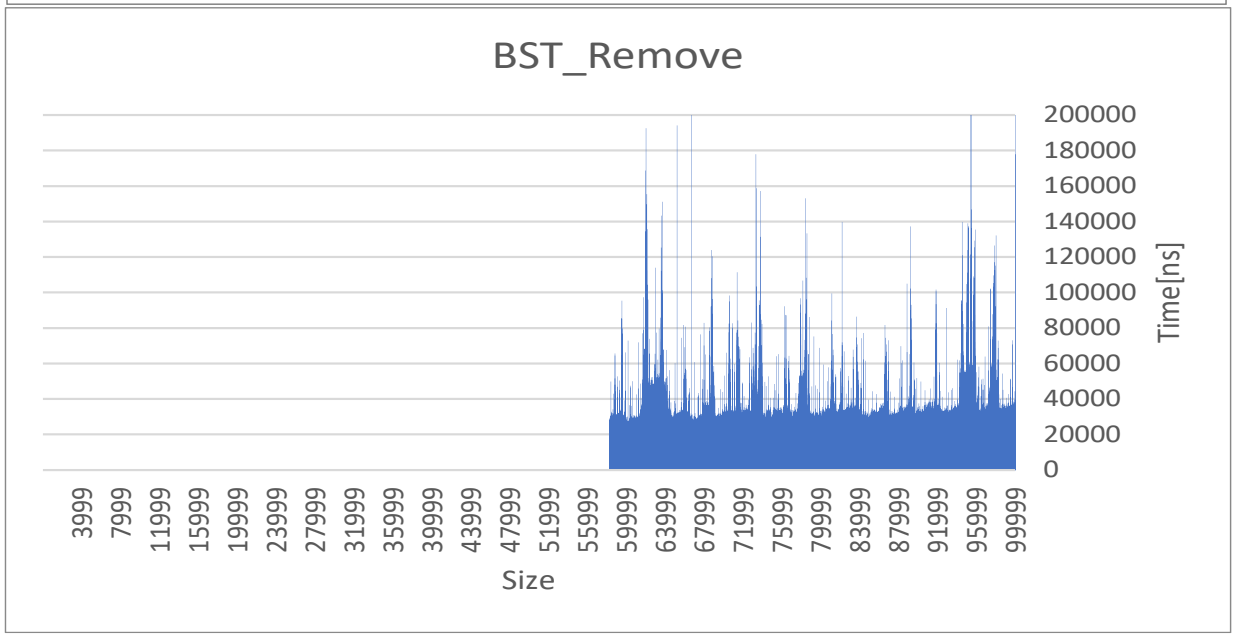




Remove

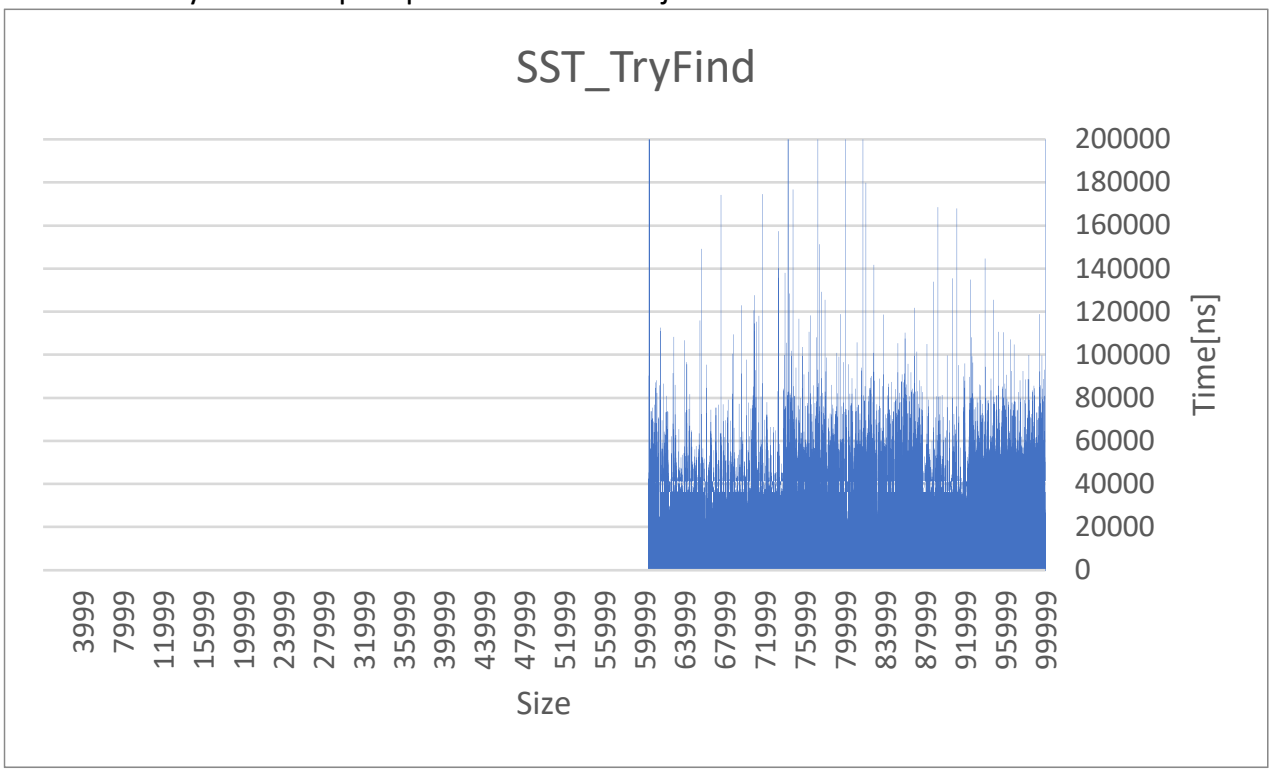
Môžeme vidieť, že pri **SortedSequenceTable** je zložitosť **lineárna**, avšak pri jednej tretine rýchlosť zrýchli. Pri BinarySerachTree je zložitosť **2x logaritmická**, najprv prvok musíme **nájsť** a potom ho **nahradiť** s inOrder nasledovníkom. **SST** má pri vymazávaní prvku zložitosť **lineárnu**, musíme ale pripočítať aj rýchlosť **vyhľadania** prvku ktorá je **logaritmická**.

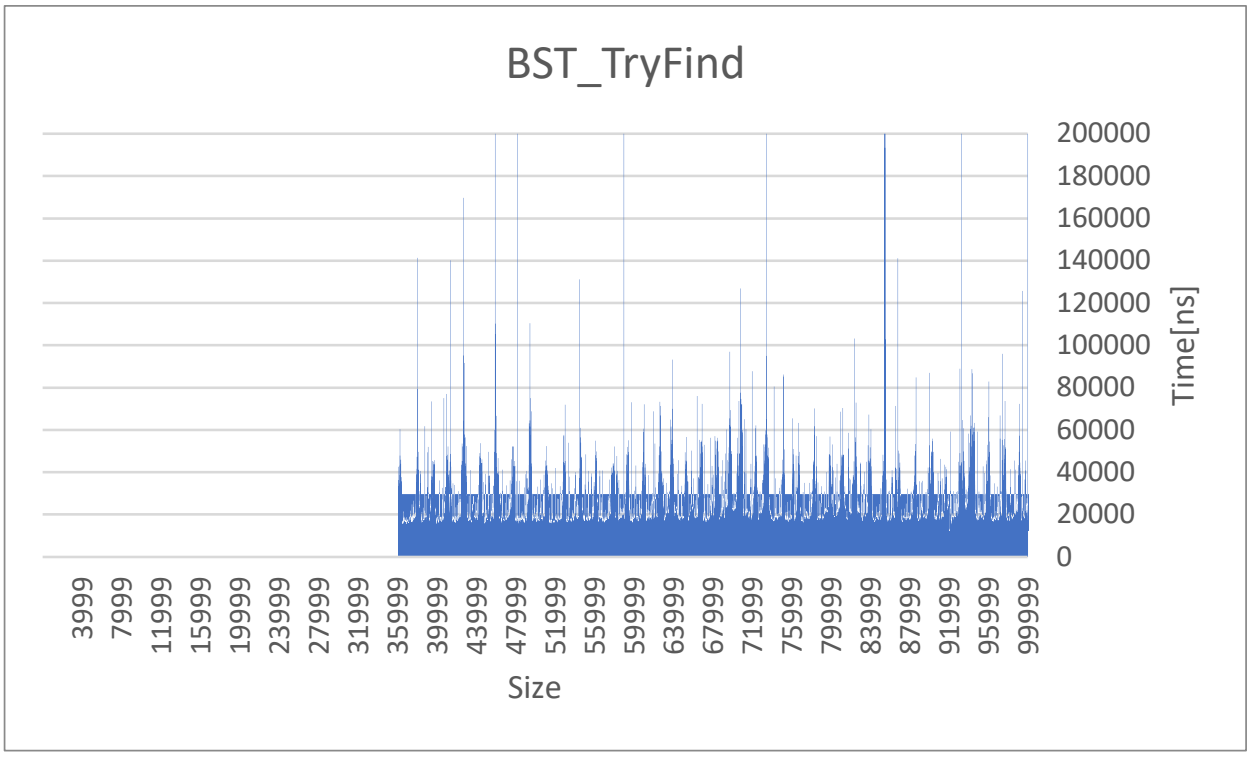




TryFind

Rýchlosť je značne **rýchlejšia** a stabilnejšia pri **BinarySearchTree**. Pri **SortedSequenceTable** je rýchlosť **pomalšia** ale vidíme, že sa postupom času stabilizuje preto pripomína **logaritmickú** zložitosť. Pri BST je zložitosť taktiež **logaritmická**, pretože zo začiatku rýchlosť stúpa a potom sa stabilizuje.





Záver – Odhad hornej asymptotickej zložitosti

Záver

Na základe grafov by som sa bez ohľadu na situáciu **rozhodol pre BinarySearchTree**, avšak musím byť **opatrný**, pretože BinarySearchTree sa môže **zdegenerovať** a tým pádom budú zložitosti **horšie ako** pri **SortedSequenceTable**. Ak by som teda vkladal prvky v utriedenom poradí, **BinarySearchTree** by malo všade **lineárnu zložitosť** a **nebola** by teda **vhodná** na použitie.

Insert

- SortedSequenceTable
  - **O(log (n))**
- BinarySearchTree
  - **O(log (n))**

Remove

- SortedSequenceTable
  - **O(n) + O(log (n))**
- BinarySearchTree
  - **O(log (n)) + O(log (n))**

TryFind

- SortedSequenceTable
  - **O(log (n))**
- BinarySearchTree
  - **O(log (n))**