Fakulta riadenia a informatiky
Informatika

Domáce zadanie 2

Analýza výkonu ADT zoznam

doc. Ing. **Miroslav Kvaššay** PhD.
STREDA 12, 13
2021/2022

Maroš Gorný, 5ZYI21

Obsah

Úloha 2

Zadanie – Overenie výkonu v scenári

Realizácie ADT zoznam, 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 sprístupnenie prvkov). Parametre do operácií sú taktiež náhodné; ako index je možné zvoliť akýkoľvek aktuálne platný index.

Tab. 1 Testovacie scenáre pre ADT zoznam

Scenár	Podiel operácií			
	`insert`	`reomoveAt`	`at`	`getIndexOf`
A	20	20	50	10
B	35	35	20	10
C	45	45	5	5

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

List<T>* - Pointer na list s ktorým budem pracovať.
SimpleTest - Inštancia s ktorou môžem merať čas
FileLogConsumer - Pointer na Logger s ktorým budem schopný ukladať dáta.
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 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, at, removeAt, getIndexOf.**

Výber metódy – at, removeAt (komplikácie)

V prípade metód at a removeAt môže nastať komplikácia v prípade,

- Že list je **prázdny** a teda nebudeme môcť zvoliť platný index. V takomto prípade skontrolujem všetky podmienky uvedené vyššie a vyberám ďalšiu prvú vhodnú metódu, teda insert alebo getIndexOf.
- Že list je prázdny a metódy insert a getIndexOf dosiahli svoj **maximálny počet volaní.** V takomto prípade si pri metóde
    - at, pridám jeden prvok do listu, zavolám metódu at a následne ten prvok vymažem. (čas meriam len pre metódu at)
    - removeAt, pridám jeden prvok do listu a zavolám metódu removeAt. (čas meriam len pre metódu removeAt)

Tieto komplikácie mi ale zapríčinia **tvorenie zhlukov** daných metód, avšak len pri konci, keď už nemôžem úplne voľne volať dané metódy.

Volanie metódy a meranie času

Pri volaní metódy musím metódu zavolať s platným indexom. V sekcii Výber metódy – at, removeAt (komplikácie) som vyriešil problém pri metóde at a removeAt s veľkosťou listu 0.

Najprv si vytvorím **náhodné dáta**, ktoré budem vkladať do metód insert a getIndexOf. Tieto dáta budú obsahovať čísla od 0 po RAND_MAX.

Platný index nastavím tak, že si vygenerujem číslo od 0 po veľkosť listu – 1, teda pri veľkosti 3 by sa mi vygenerovali čísla 0,1,2. A pri prázdnom liste nastavím index na 0.

- insert
    - Zavolám metódu s náhodnými dátami a indexom v intervale od 0 po veľkosť listu.
    - Odmeriam čas
- getIndexOf
    - Zavolám metódu s náhodnými dátami
    - Odmeriam čas

- At
    - Zavolám metódu s platným indexom
    - Odmeriam čas
- removeAt
    - Zavolám metódu s platným indexom
    - Odmeriam čas

Ukladanie údajov

Dáta som ukladal do CSV súboru, kde

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

Size	Time[ns]	Method
0	589500	getIndexOf
0	1500	getIndexOf
0	1513100	insert
1	4200	getIndexOf
1	217800	at
1	1800	at
1	600	getIndexOf
1	158500	removeAt

Testovanie – Výsledky

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

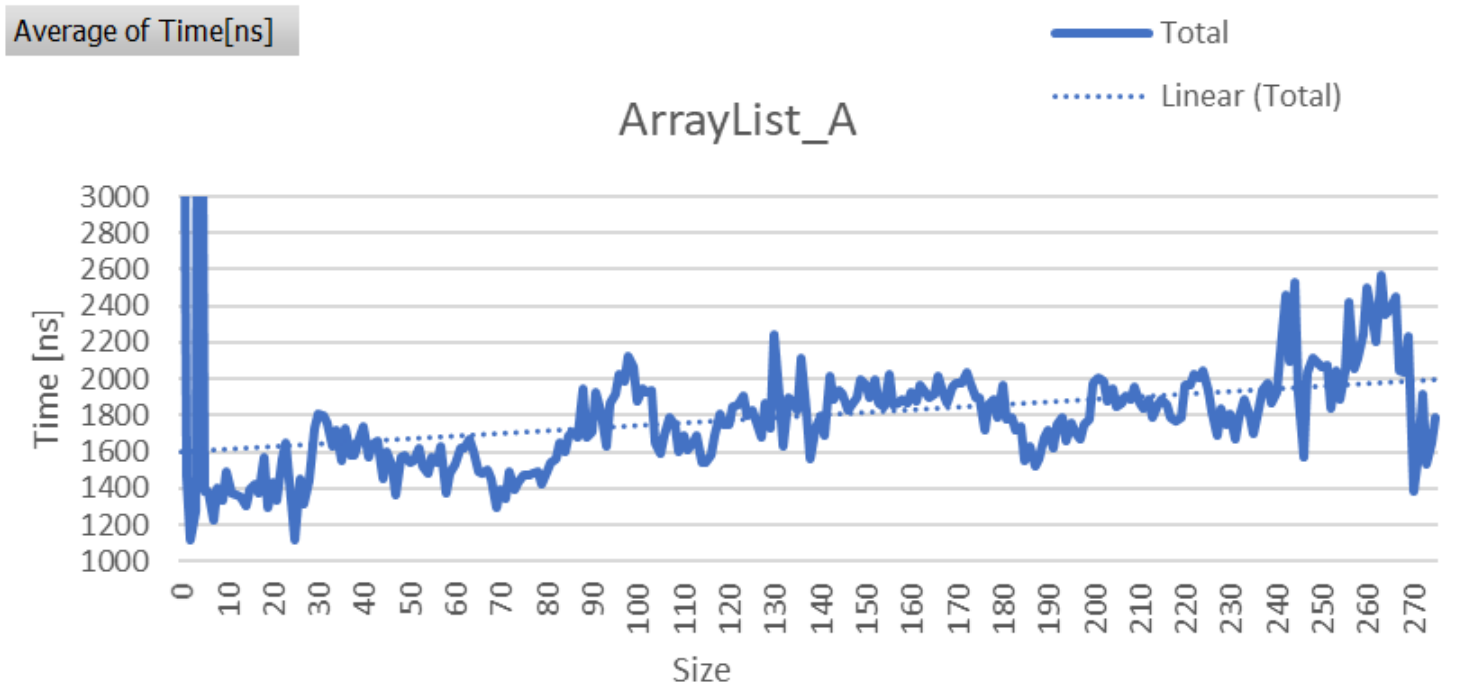
Tým pádom môžem porovnať scenáre ako celok a nemusím sa pozerať na jednotlivé metódy. Pozerám sa na ich spoločný priemer, takže porovnávam celkovú rýchlosť zoznamu a nie jeho najlepšiu alebo najhoršiu zložitosť.

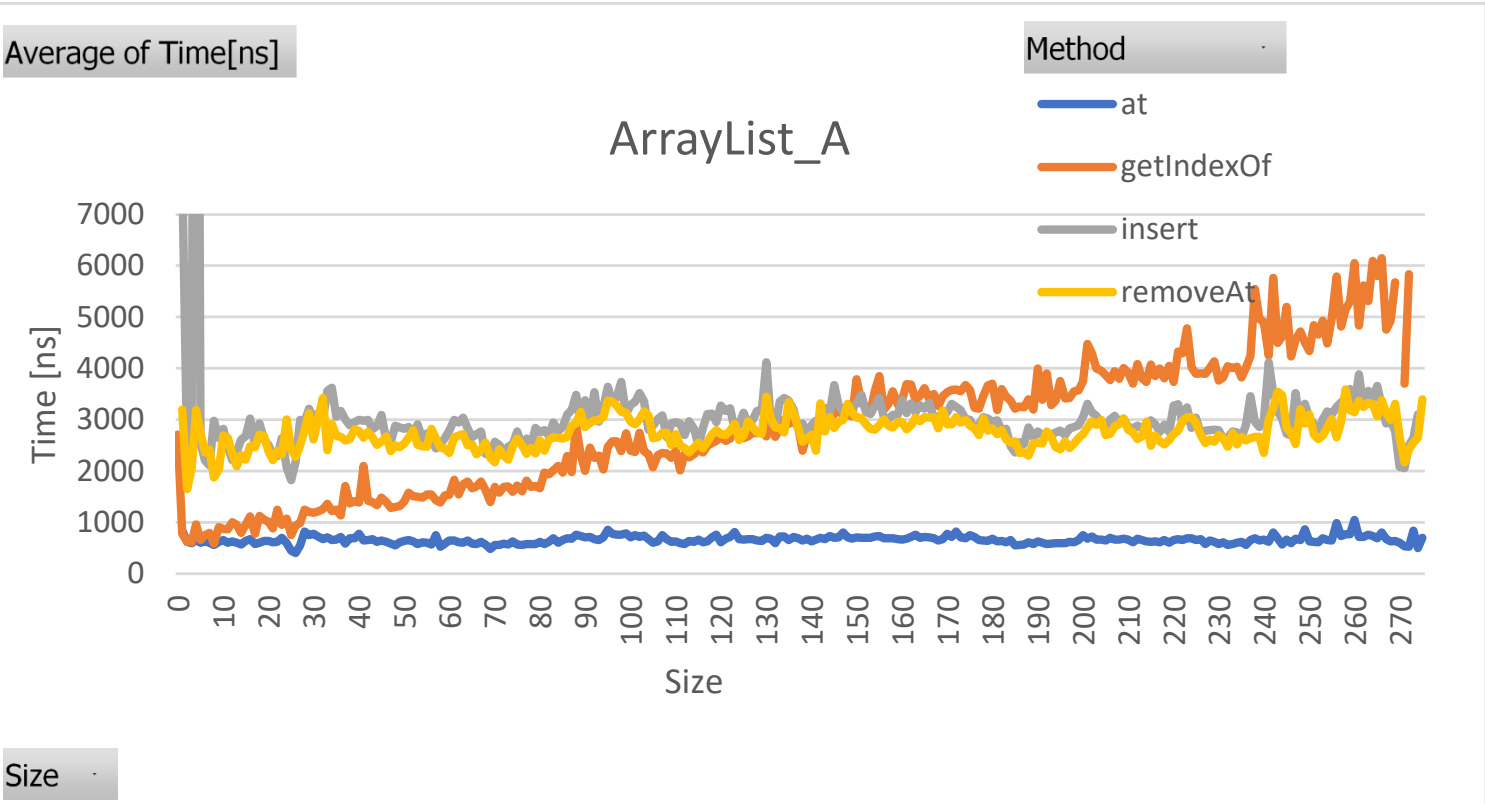
Scenár A

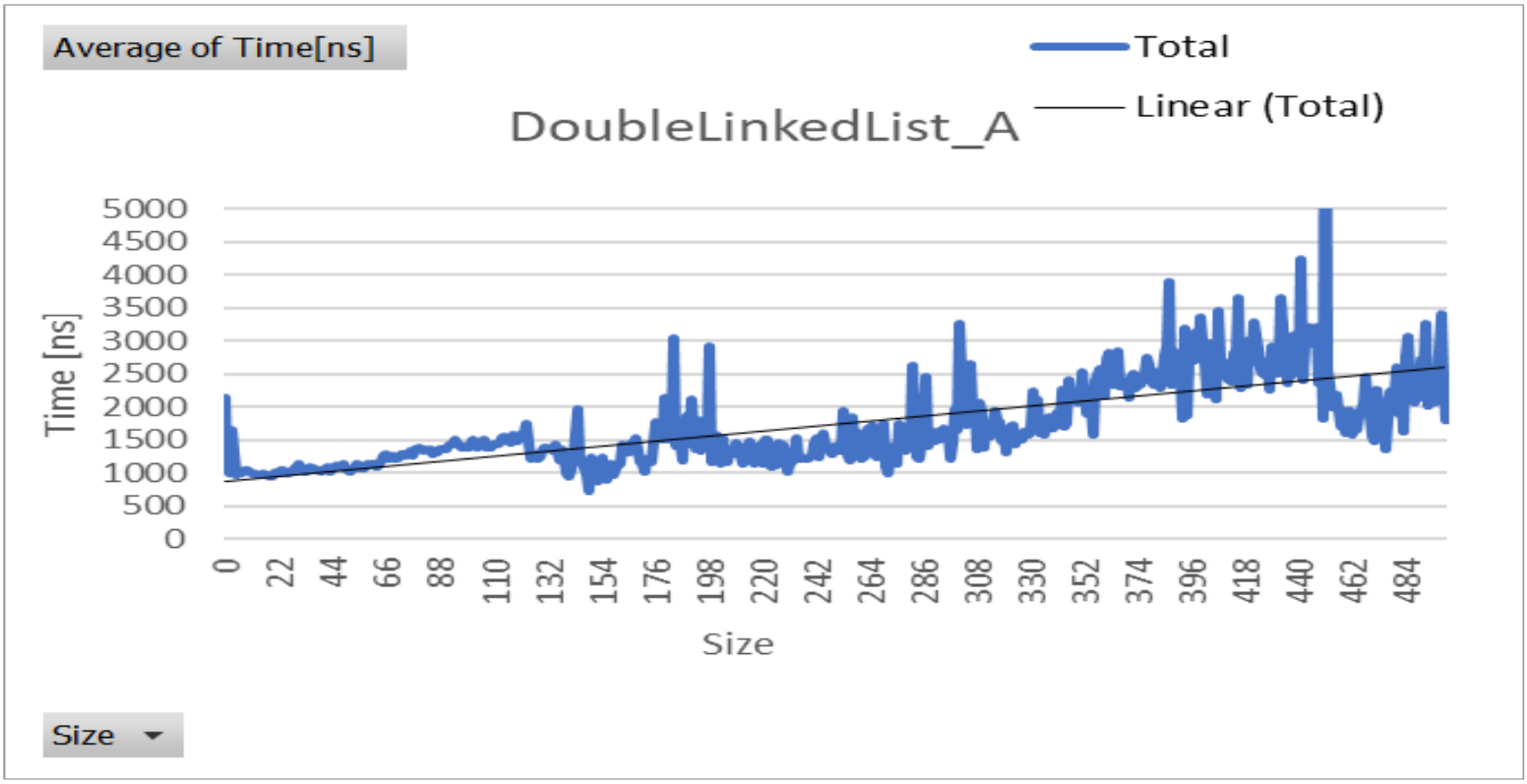
Zadanie

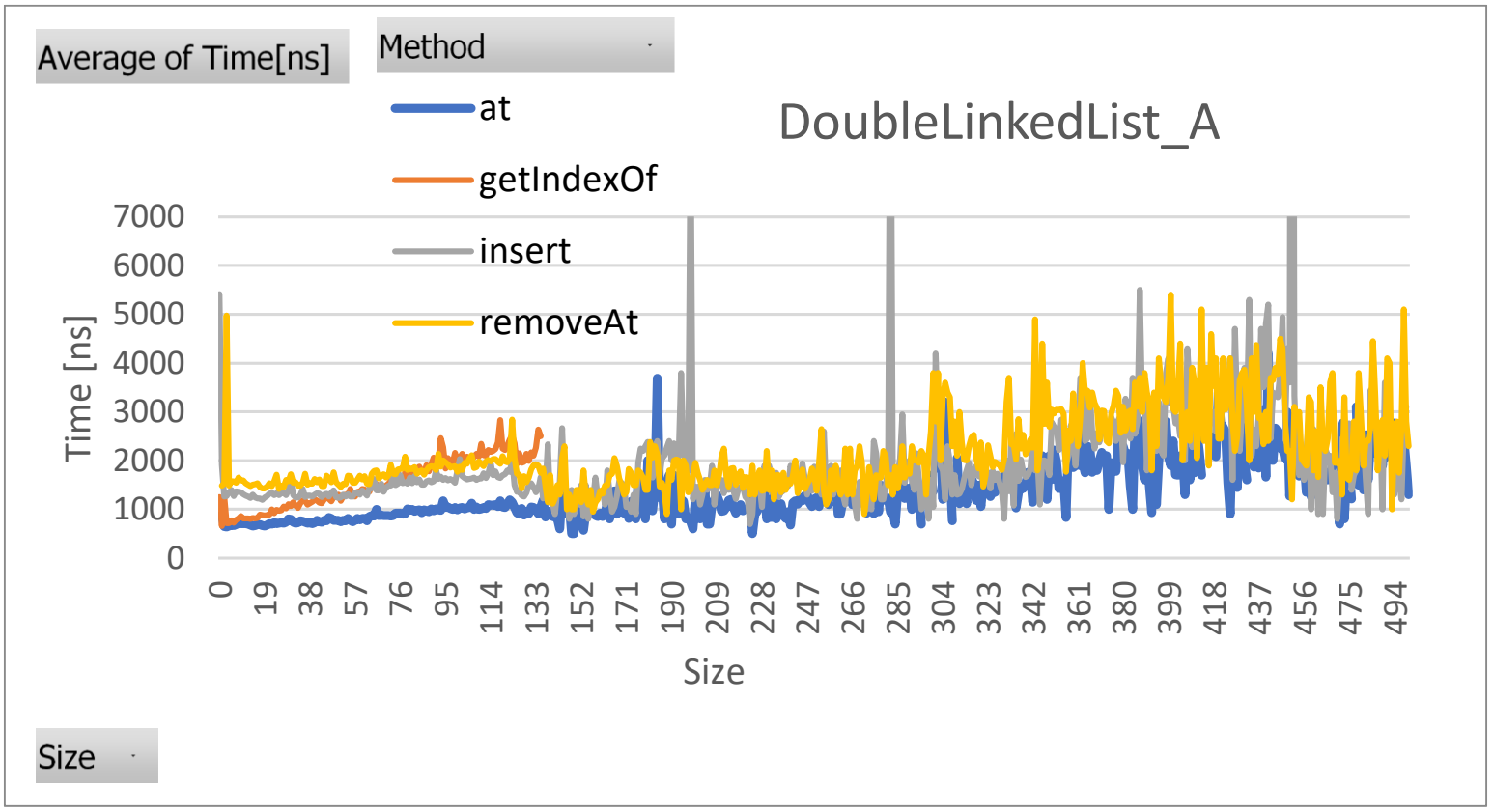
Scenár	Podiel operácií			
	insert	reomoveAt	at	getIndexOf
A	20	20	50	10

Prezentácia výsledkov






Average of Time[ns]
Total
Linear (Total)
DoubleLinkedList_A
Time [ns]
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
0 22 44 66 88 110 132 154 176 198 220 242 264 286 308 330 352 374 396 418 440 462 484
Size
Size


Average of Time[ns]
Method
at
getIndexOf
insert
removeAt
DoubleLinkedList_A
Time [ns]
7000
6000
5000
4000
3000
2000
1000
0
0 19 38 57 76 95 114 133 152 171 190 209 228 247 266 285 304 323 342 361 380 399 418 437 456 475 494
Size
Size

Vyhodnotenie výsledkov

Na základe grafov môžeme vidieť, že

ArrayList

- Ako celok mu **priemerná rýchlosť klesá**, teda čím väčší tento zoznam je, tým je rýchlosť pomalšia.
- Môžeme však vidieť, že najväčší vplyv pri veľkosti má metóda getIndexOf, čo je však logické, pretože pri mojej implementácií sa stane veľa krát, že táto metóda bude musieť prejsť celý zoznam. Metódy insert a removeAt sú rýchlejšie, ale pri danej veľkosti ich neovplyvňuje veľkosť zoznamu a drží sa medzi 2000 až 4000 nanosekúnd. Metóda at je konštantná teda veľkosť zoznamu ju neovplyvňuje.

DoubleLinkedList

- Ako celok mu **priemerná rýchlosť klesá**, teda čím väčší tento zoznam je, tým je rýchlosť pomalšia.
- Pri danom grafe je ťažšie spozorovať ako sa správajú metódy samostatne, pretože všetky sa správajú približne rovnako. Vidíme, že cca po veľkosti 130 čas začne kolísať ale to je pravdepodobne spôsobené pomerom metód a tým pádom by mala byť najpresnejšia metóda at.

ArrayList vs DoubleLinkedList

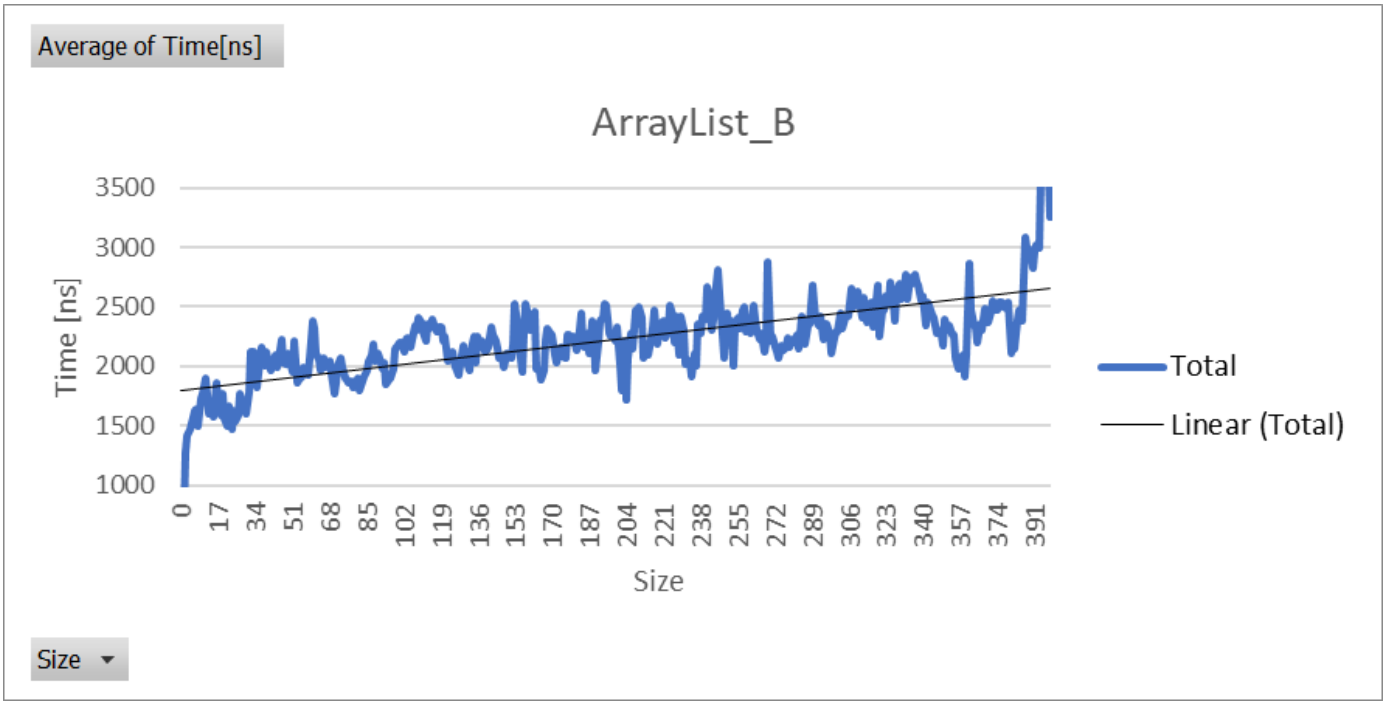
- V danom scenári je **rýchlejší ArrayList**. Začiatok majú obidva zoznamy podobné avšak DoubleLinkedList podľa grafu stúpa trošku rýchlejšie ako ArrayList. Tým pádom pri častom používaní metódy at je vhodnejšie použiť ArrayList ako DoubleLinkedList.

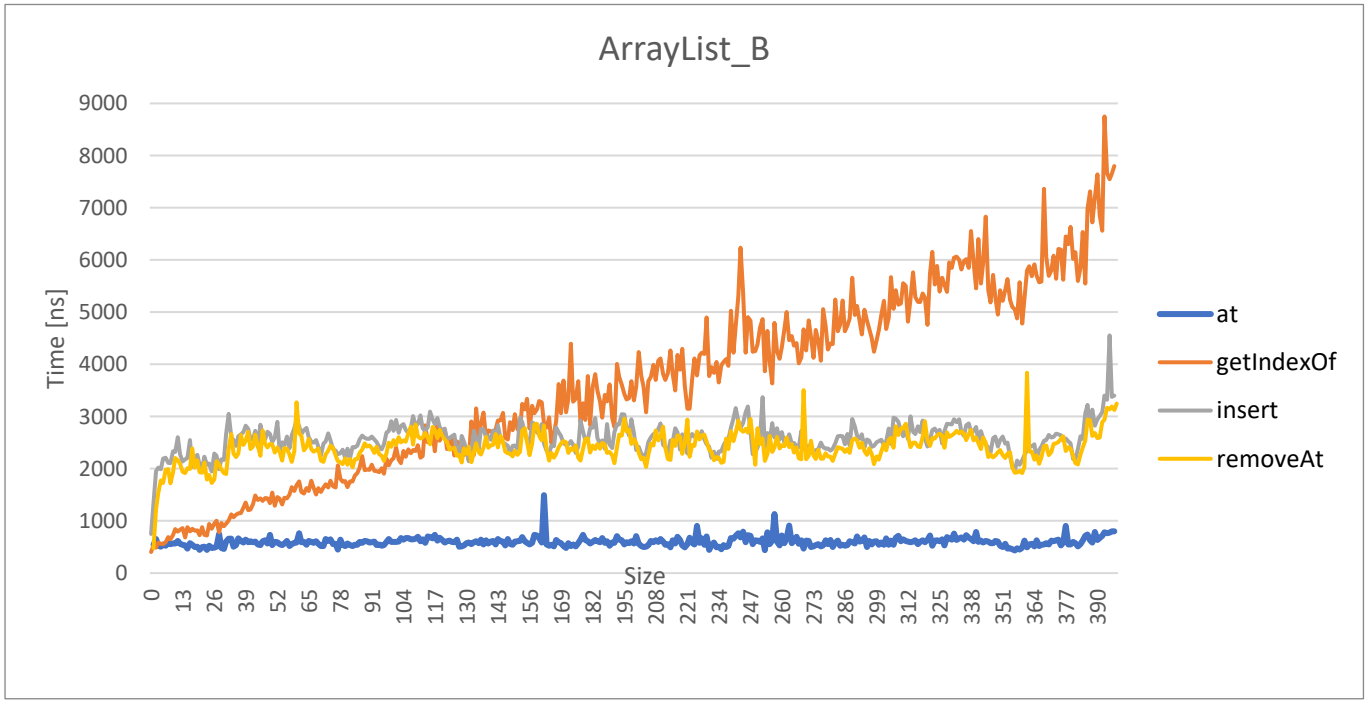
Scenár B

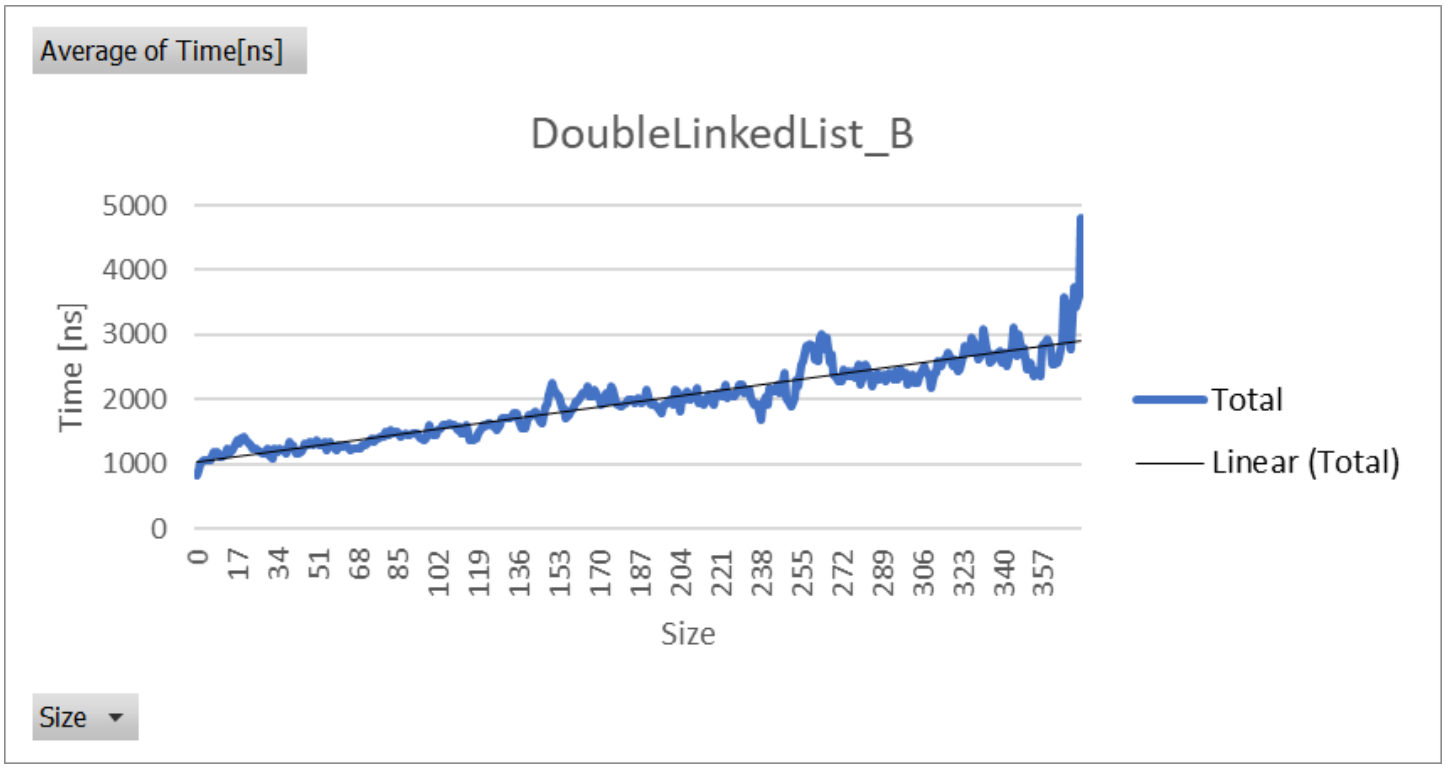
Zadanie

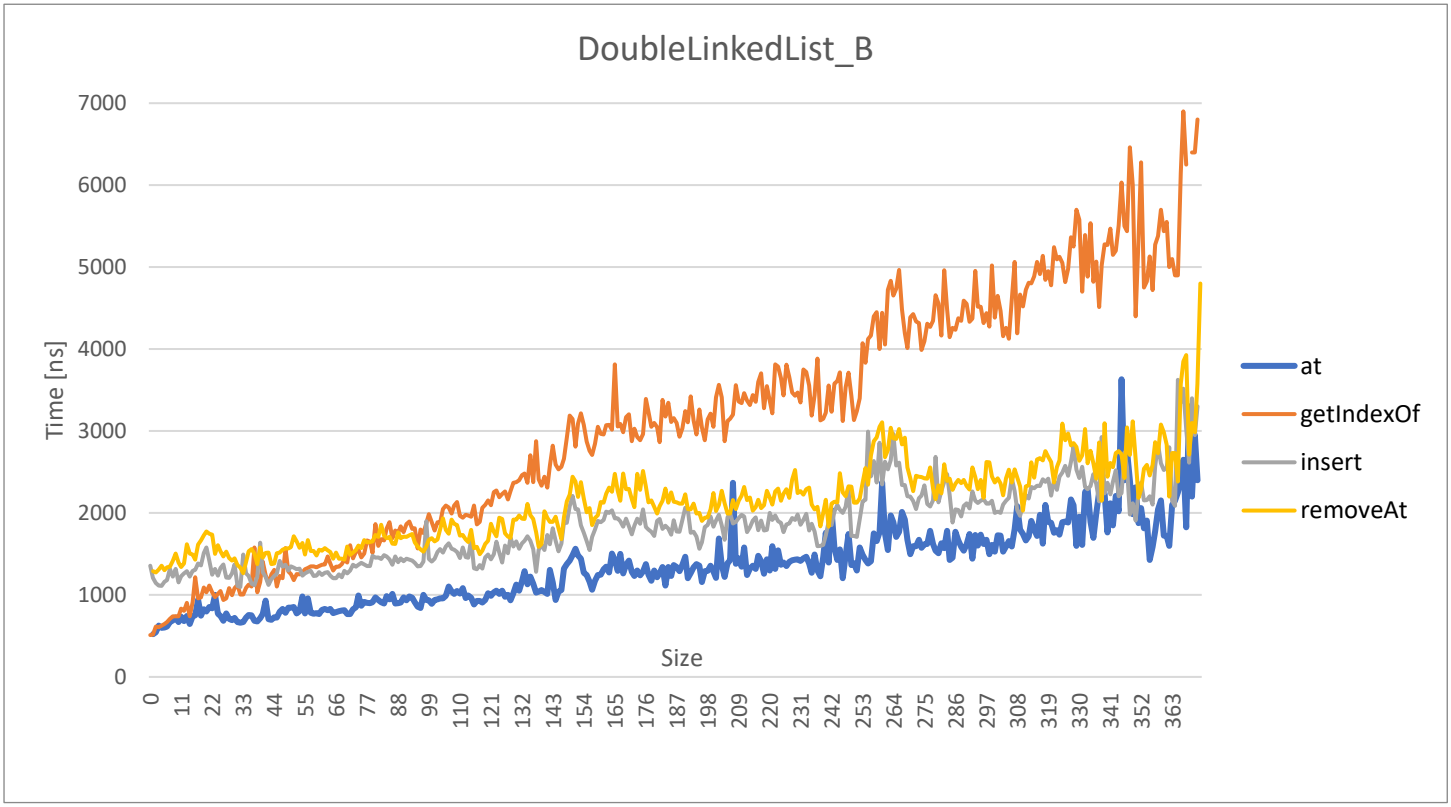
Scenár	Podiel operácií			
	`insert`	`reomoveAt`	`at`	`getIndexOf`
B	35	35	20	10

Prezentácia výsledkov






Average of Time[ns]
DoubleLinkedList_B
Time [ns]
5000
4000
3000
2000
1000
0
0
17
34
51
68
85
102
119
136
153
170
187
204
221
238
255
272
289
306
323
340
357
Size
Total
Linear (Total)
Size


DoubleLinkedList_B
Time [ns]
7000
6000
5000
4000
3000
2000
1000
0
0
11
22
33
44
55
66
77
88
99
110
121
132
143
154
165
176
187
198
209
220
231
242
253
264
275
286
297
308
319
330
341
352
363
Size
at
getIndexOf
insert
removeAt

Vyhodnotenie výsledkov

Na základe grafov môžeme vidieť, že

ArrayList

- Ako celok mu **priemerná rýchlosť klesá**, teda čím väčší tento zoznam je, tým je rýchlosť pomalšia.
- Môžeme však vidieť, že najväčší vplyv pri veľkosti má metóda getIndexOf, čo je však logické, pretože pri mojej implementácií sa stane veľa krát, že táto metóda bude musieť prejsť celý zoznam. Metódy insert a removeAt sú rýchlejšie, ale pri danej veľkosti ich neovplyvňuje veľkosť zoznamu a držia sa medzi 2000 až 4000 nanosekúnd. Metóda at je konštantná teda veľkosť zoznamu ju neovplyvňuje.

DoubleLinkedList

- Ako celok mu **priemerná rýchlosť klesá**, teda čím väčší tento zoznam je, tým je rýchlosť pomalšia.
- Metóda getIndexOf má na celkovú rýchlosť najväčší vplyv, ale je to znova spôsobené tým, že pri mojej implementácií sa veľa krát môže stať, že táto metóda prejde celý zoznam. Ostatné metódy sú pri väčšej veľkosti pomalšie, ale rýchlosť im klesá pomaly.

ArrayList vs DoubleLinkedList

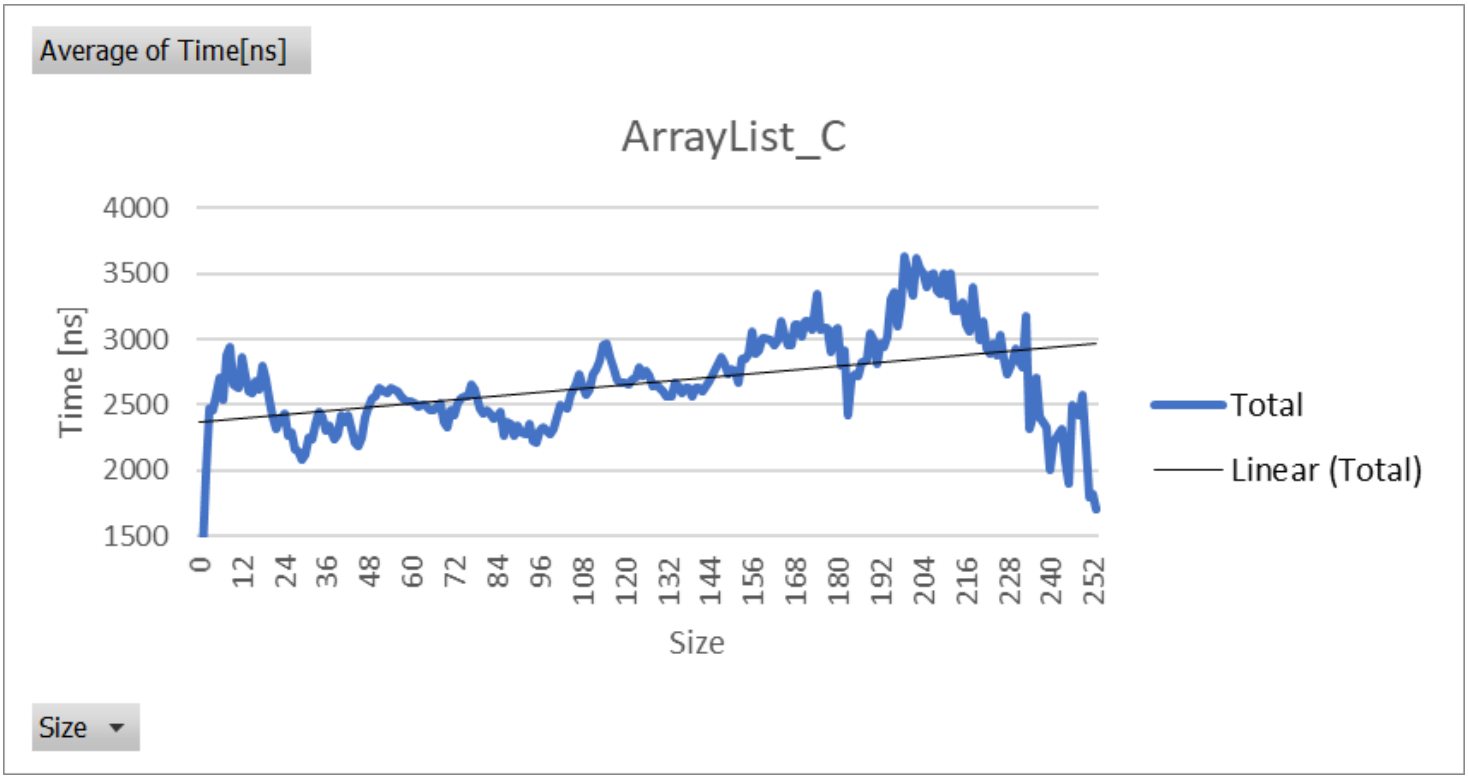
- V danom scenári sú **celkové rýchlosti podobné**, avšak keď sa pozrieme na metódy, pri ktorých máme najväčší podiel opakovaní tak prídeme na to, že metódy **insert** a **removeAt** sú pri menších veľkostiach **rýchlejšie pri DoubleLinkedListe**.

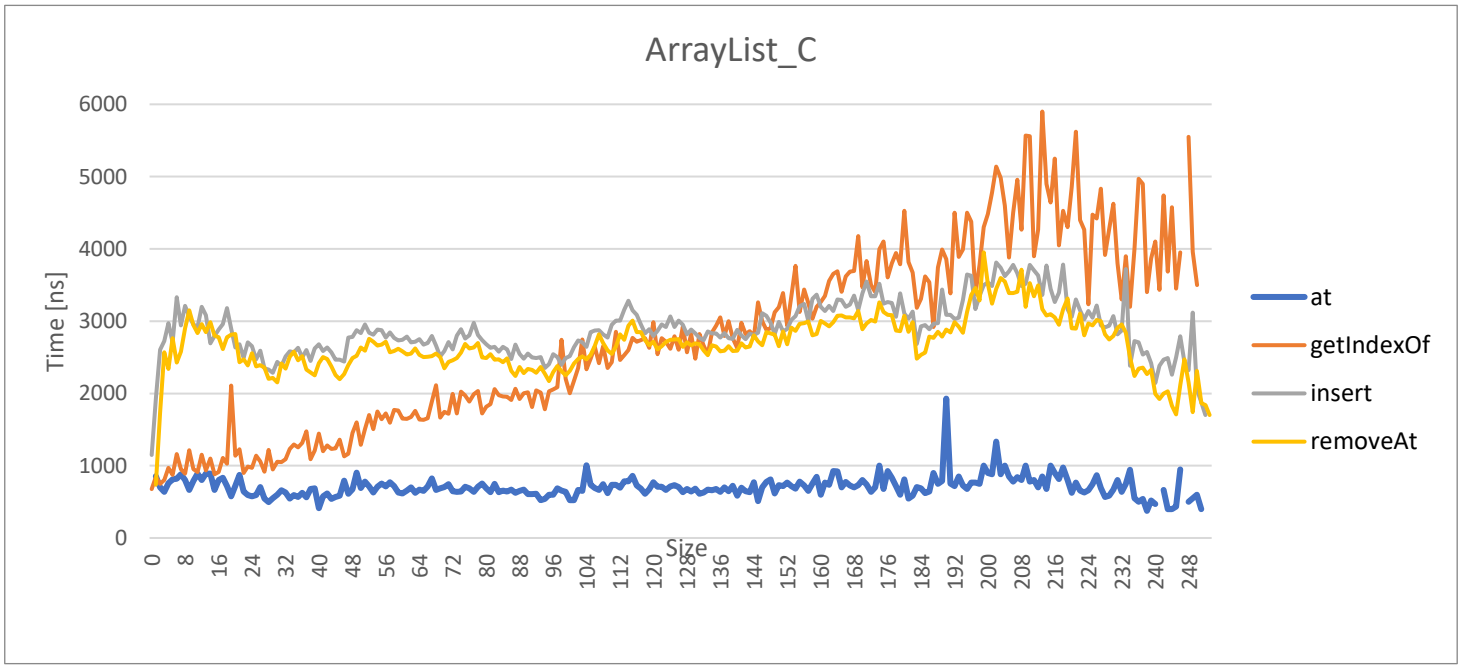
Scenár C

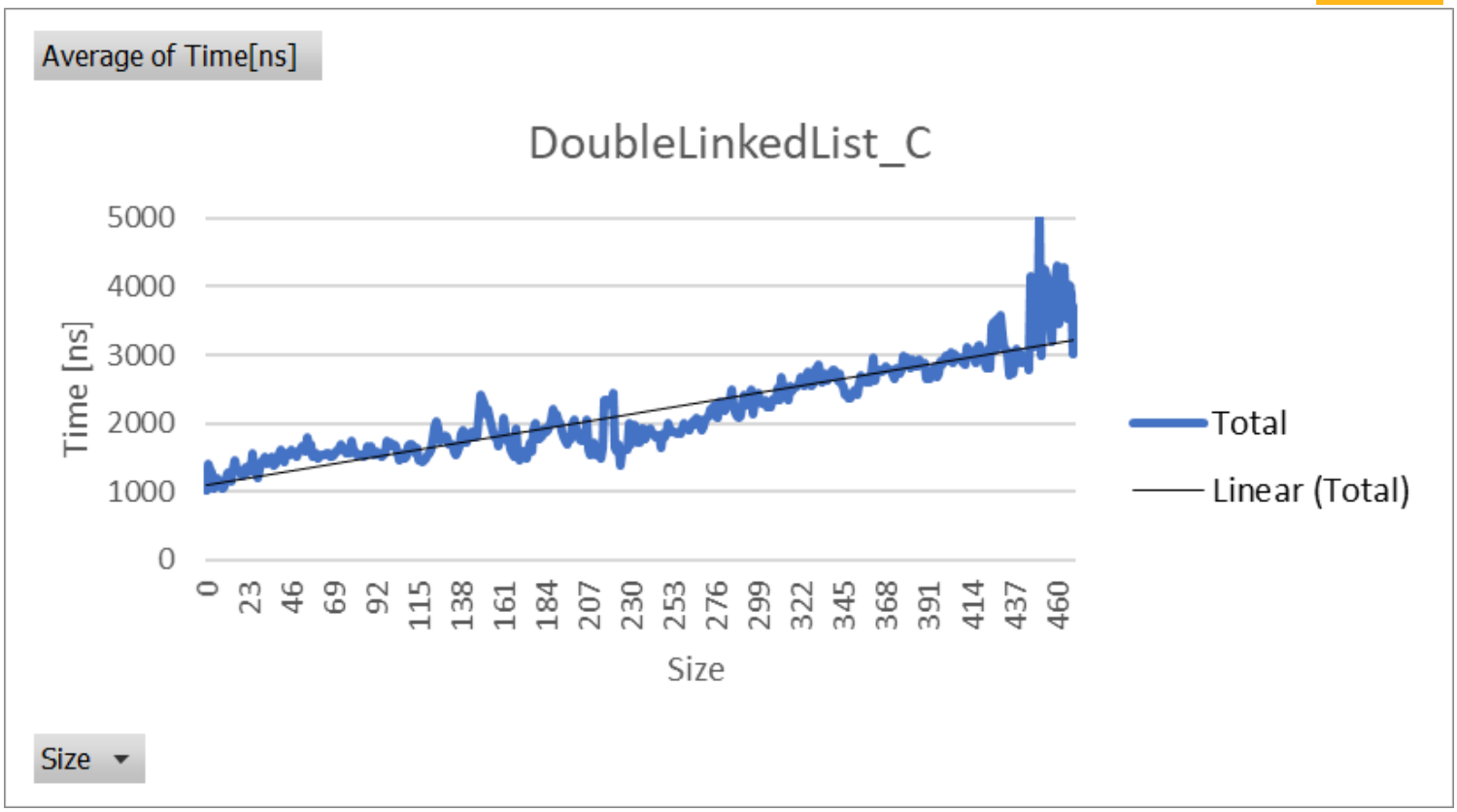
Zadanie

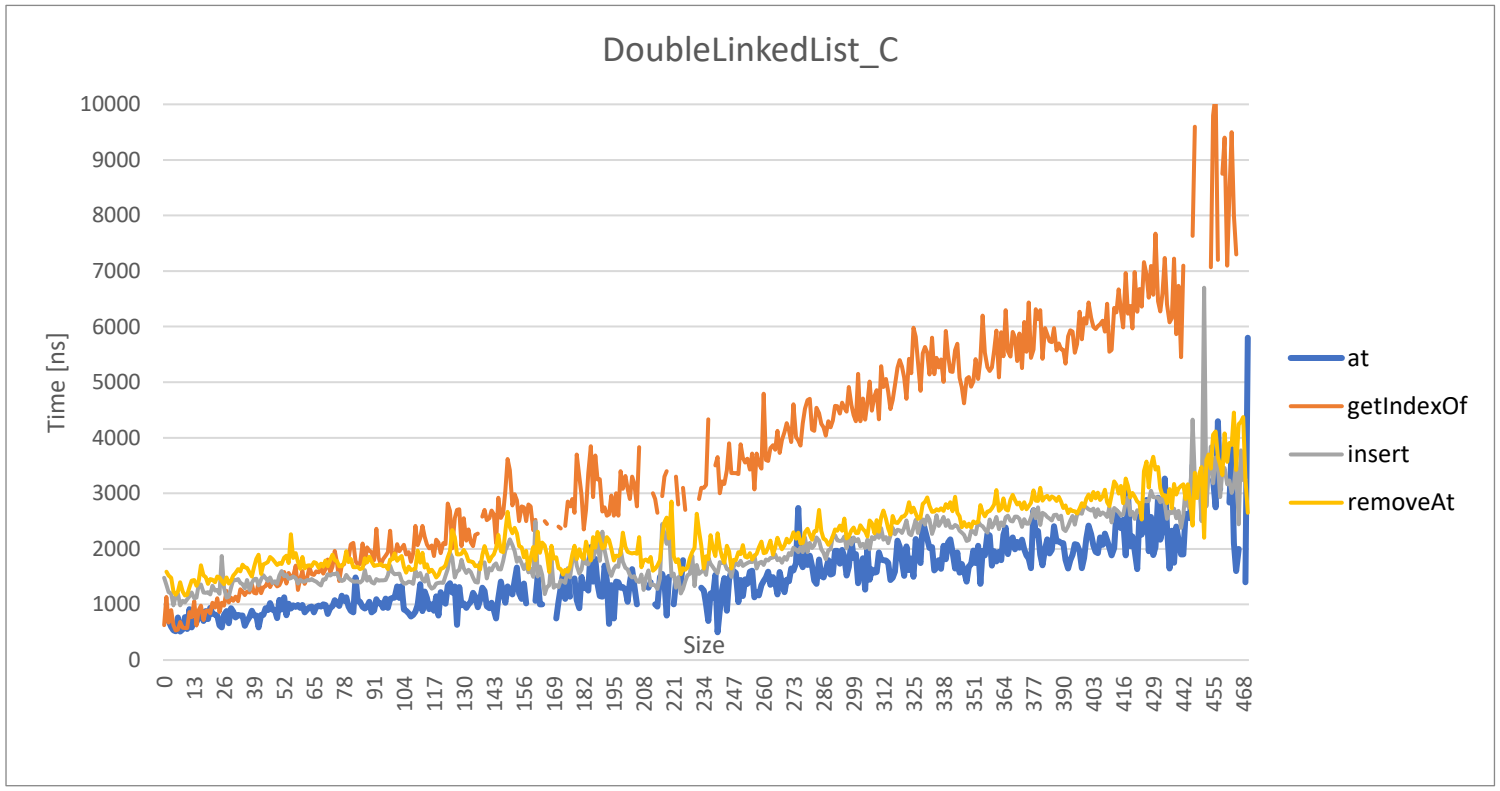
Scenár	Podiel operácií			
	insert	reomoveAt	at	getIndexOf
C	45	45	5	5

Prezentácia výsledkov






Average of Time[ns]
DoubleLinkedList_C
Time [ns]
5000
4000
3000
2000
1000
0
0
23
46
69
92
115
138
161
184
207
230
253
276
299
322
345
368
391
414
437
460
Size
Total
Linear (Total)
Size


DoubleLinkedList_C
Time [ns]
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
Size
0
13
26
39
52
65
78
91
104
117
130
143
156
169
182
195
208
221
234
247
260
273
286
299
312
325
338
351
364
377
390
403
416
429
442
455
468
at
getIndexOf
insert
removeAt

Vyhodnotenie výsledkov

Na základe grafov môžeme vidieť, že

ArrayList

- Ako celok mu **priemerná rýchlosť klesá**, teda čím väčší tento zoznam je, tým je rýchlosť pomalšia avšak približne po veľkosti 200 tam rýchlosť zase stúpa, ale stále sa to drží medzi 1500 až 3500 nanosekúnd.
- Môžeme však vidieť, že najväčší vplyv pri veľkosti má metóda getIndexOf, čo je však logické, pretože pri mojej implementácií sa stane veľa krát, že táto metóda bude musieť prejsť celý zoznam. Metódy insert a removeAt sú rýchlejšie, a na konci zoznamu rýchlosť stúpa, avšak to je pravdepodobne spôsobené len vygenerovaním vhodných indexov.

DoubleLinkedList

- Ako celok mu **priemerná rýchlosť klesá**, teda čím väčší tento zoznam je, tým je rýchlosť pomalšia.
- Metóda getIndexOf má na celkovú rýchlosť najväčší vplyv, ale to je stále spôsobené tým, že pri mojej implementácií sa veľa krát môže stať, že táto metóda prejde celý zoznam. Ostatné metódy sú pri väčšej veľkosti pomalšie, ale rýchlosť im klesá pomaly.

ArrayList vs DoubleLinkedList

- V danom scenári je **celková rýchlosť lepšia** v **ArrayList**. To súhlasí aj s vyhodnotením predošlého scenára, kde sme prišli na to, že pri menšej veľkosti je vhodnejší DoubleLinkedList, avšak tu veľkosť dosiahla dvojnásobok ArrayListu , takže **rýchlosť je lepšia v ArrayListe**.

Záver

Vďaka testom môžeme vidieť, že ak je v zozname viac prvkov, tak bez ohľadu na typ metód ktoré budeme používať, tak je vhodnejšie používať **ArrayList.**

Ak by sme sa chceli pozerať na metódy, tak s metódou **at** jednoznačne vyhráva **ArrayList**. Pri metódach **insert** a **RemoveAt** a pri menšej veľkosti je vhodnejšie používať **DoubleLinkedList** a pri metóde **getIndexOf** to vyzerá **podobne**.

Úloha 3

Zadanie – Analýza časových zložitostí

V rámci analýzy časových zložitostí je nutné otestovať rýchlosť operácií `insert`, `at` a `removeAt` v závislosti od počtu prvkov a implementácie zoznamu a na základe nameraných a 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

List<T>* - Pointer na list s ktorým budem pracovať.
SimpleTest - Inštancia s ktorou môžem merať čas
FileLogConsumer - Pointer na Logger s ktorým budem schopný ukladať dáta.
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 **at** pôjde veľkosť od 1 po 100 000.
    - Teda po každom volaní sa veľkosť zväčší o 1.
- Pri metóde **removeAt** pôjde veľkosť od 100 000 po 1.

Po spustení testov som zistil, že asi by bolo vhodnejšie čas premeniť na mikro alebo mili sekundy, avšak metóda at v ArrayListe by potom dosahovala hodnoty 0, preto som ešte ostal pri nanosekundách.

Ukladanie údajov

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

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

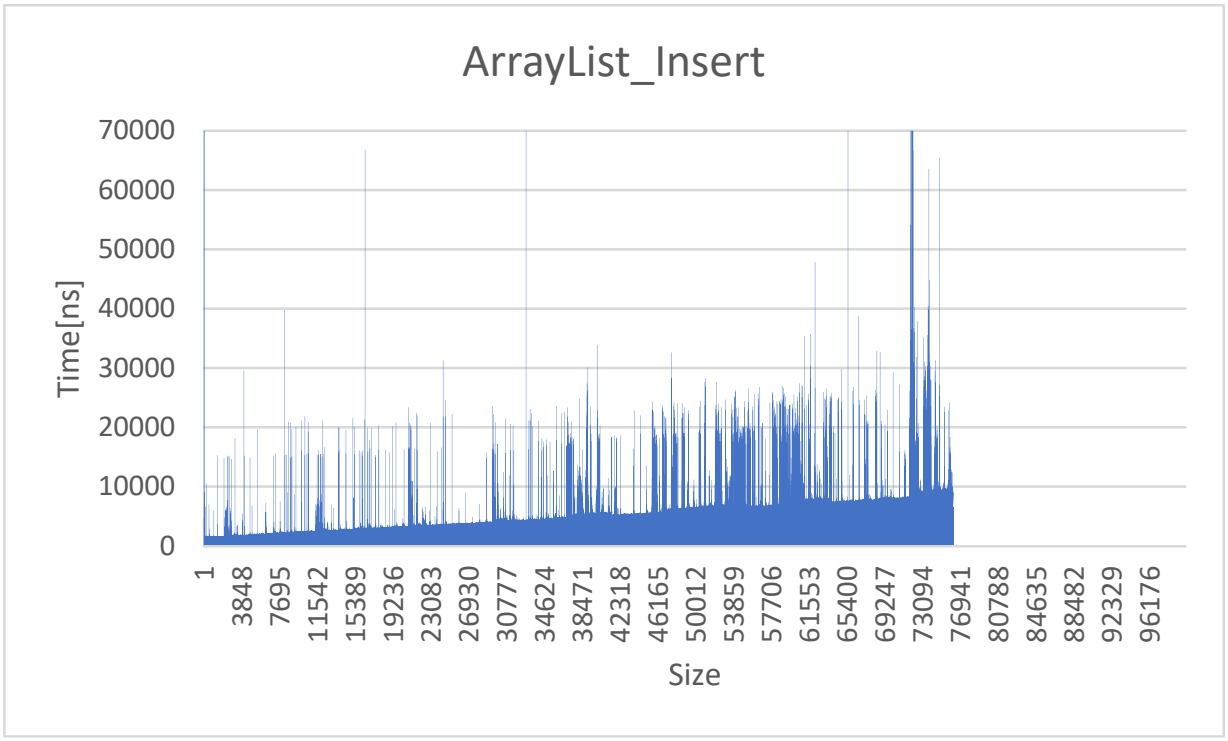
Size	Time[ns]	Method
1	244100	at
2	900	at
3	500	at
4	400	at
5	400	at
6	400	at
7	400	at

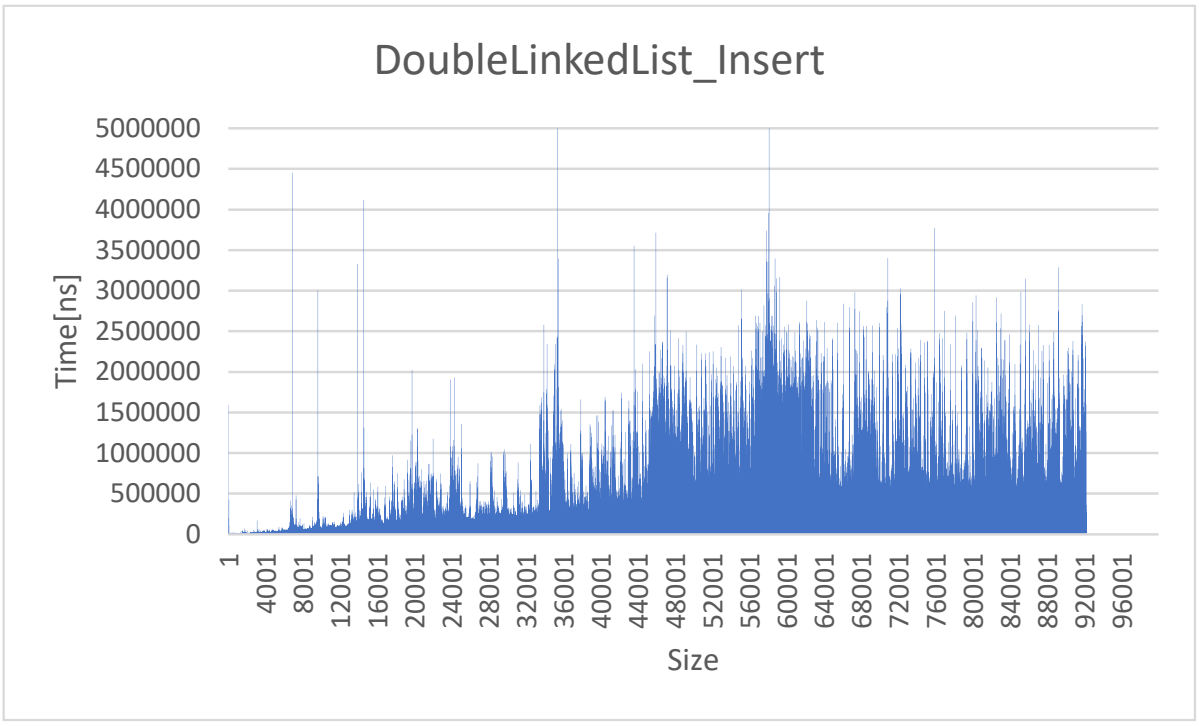
Testovanie – Výsledky

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

Insert

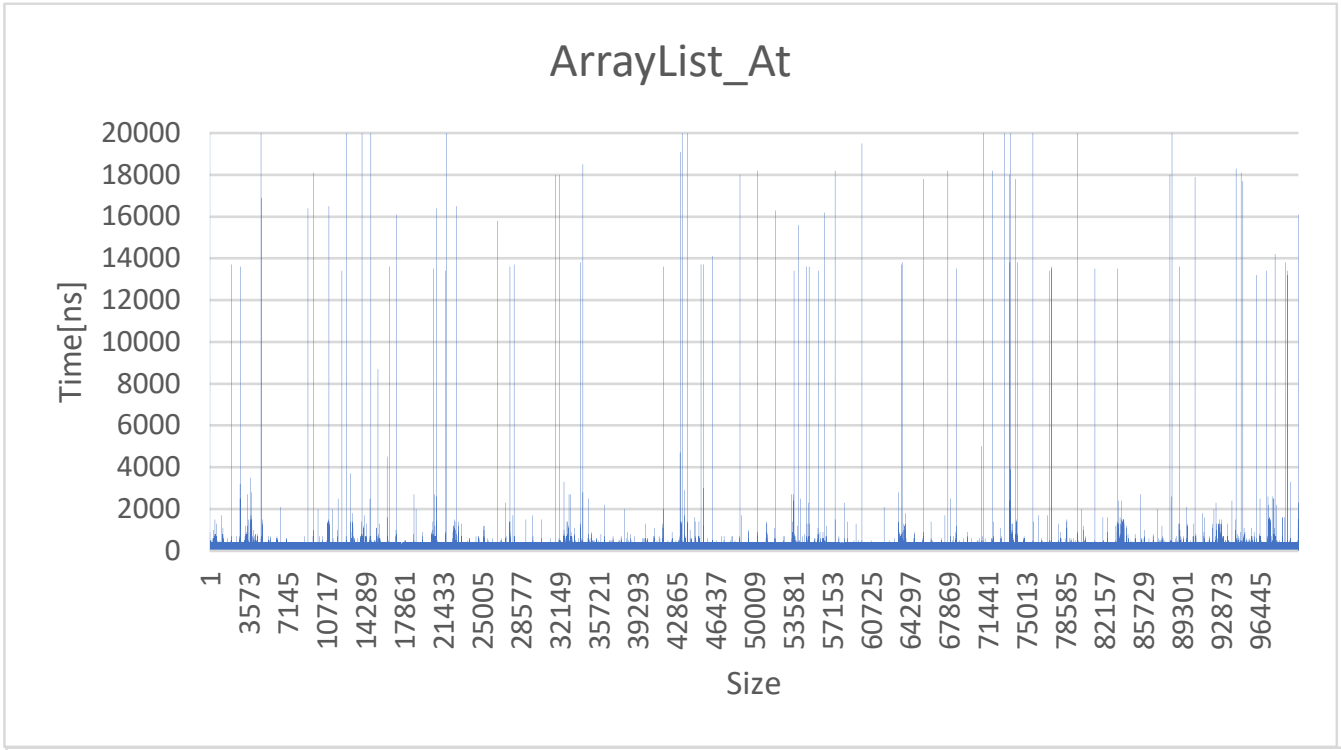
Môžeme vidieť, že ArrayList, je značne rýchlejší ako DoubleLinkedList.
Zložitosť pri ArrayListe pripomína **exponenciálnu zložitosť** pretože za polovicou začne čas vyskakovať veľmi vysoko. Pri DoubleLinkedListe by bolo vhodné použiť viac údajov, avšak vidíme, že zo začiatku čas stúpa a približne v polovici sa začne stabilizovať, preto by som zložitosť zvolil **logaritmickú**.

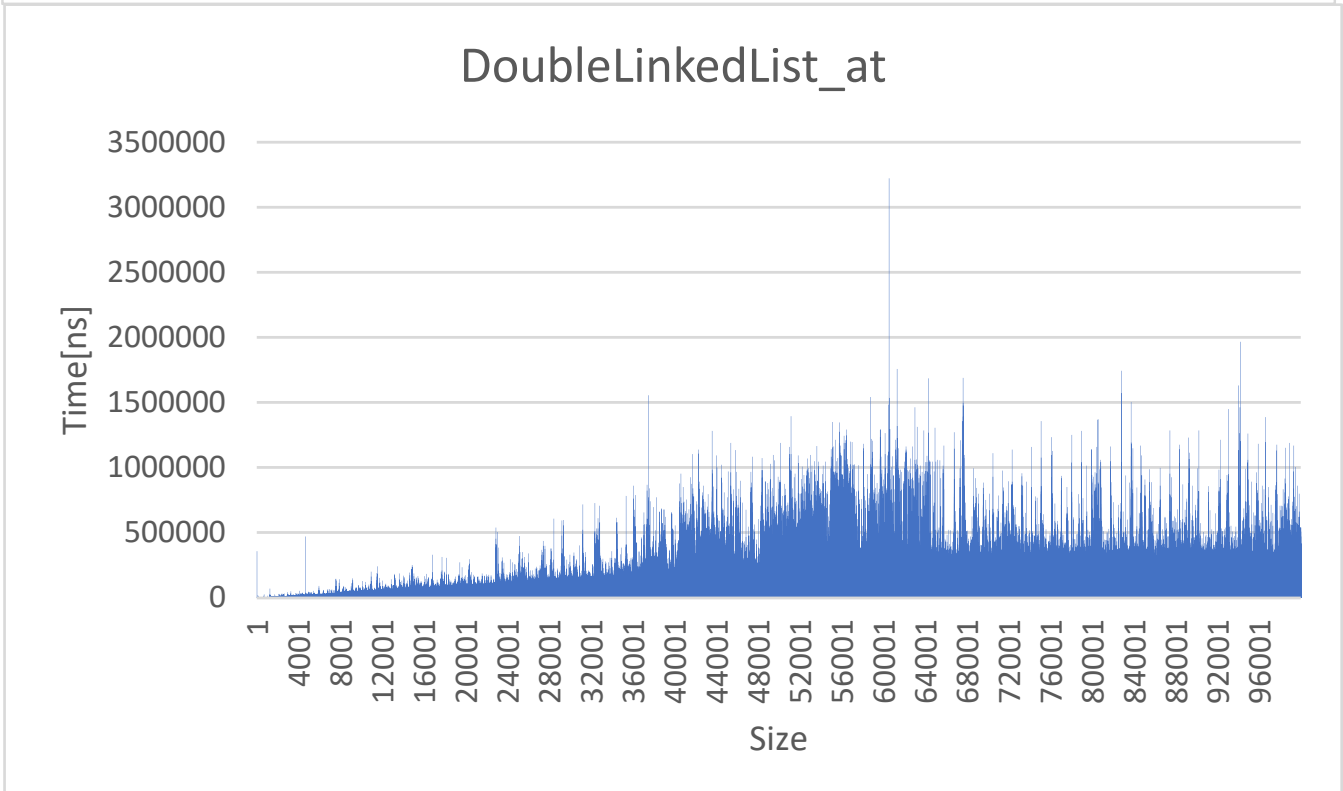




At

Znova môžeme vidieť, že ArrayList je oveľa rýchlejší ako DoubleLinkedList.
Zložitosť **ArrayListu** je **konštantná** a zložitosť **DoubleLinkedListu** znova vyzerá veľmi podobne ako **logaritmická zložitosť**, pretože v polovici sa stúpanie času začína stabilizovať.



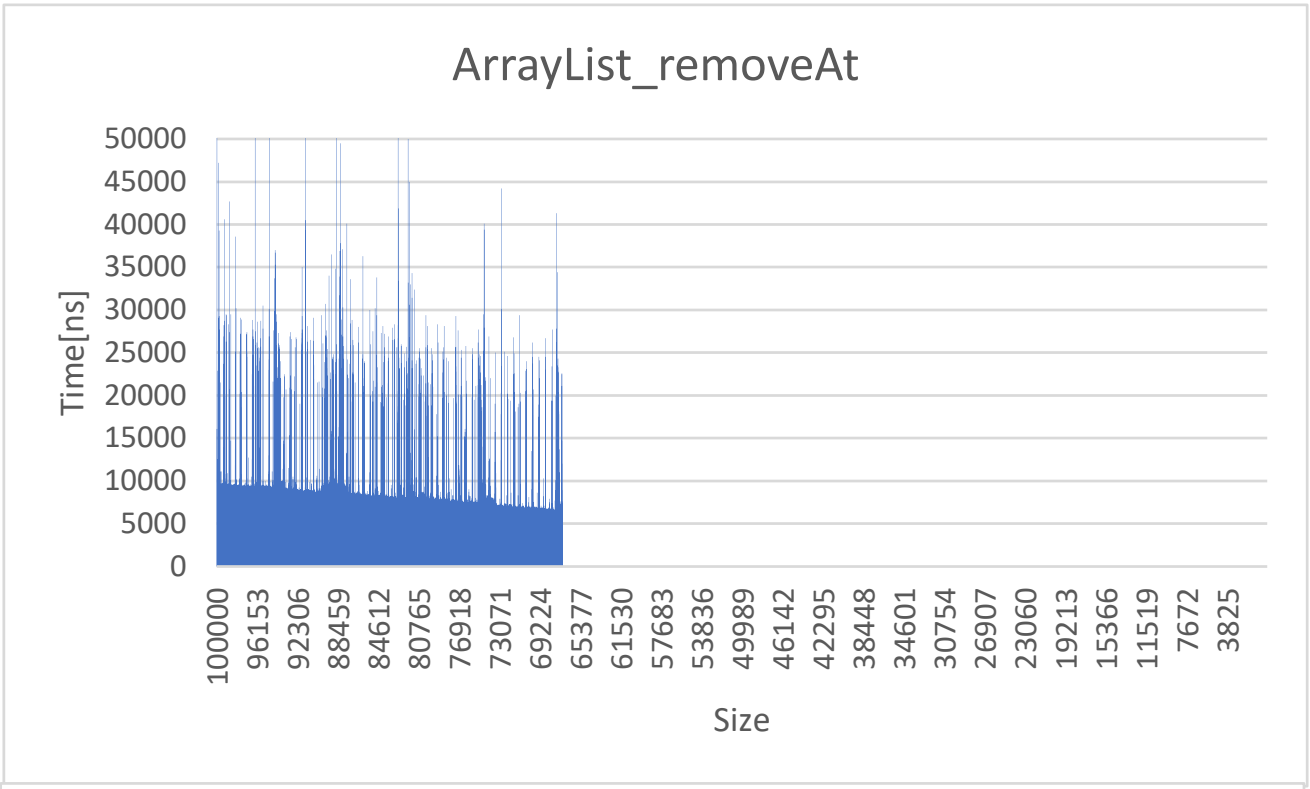


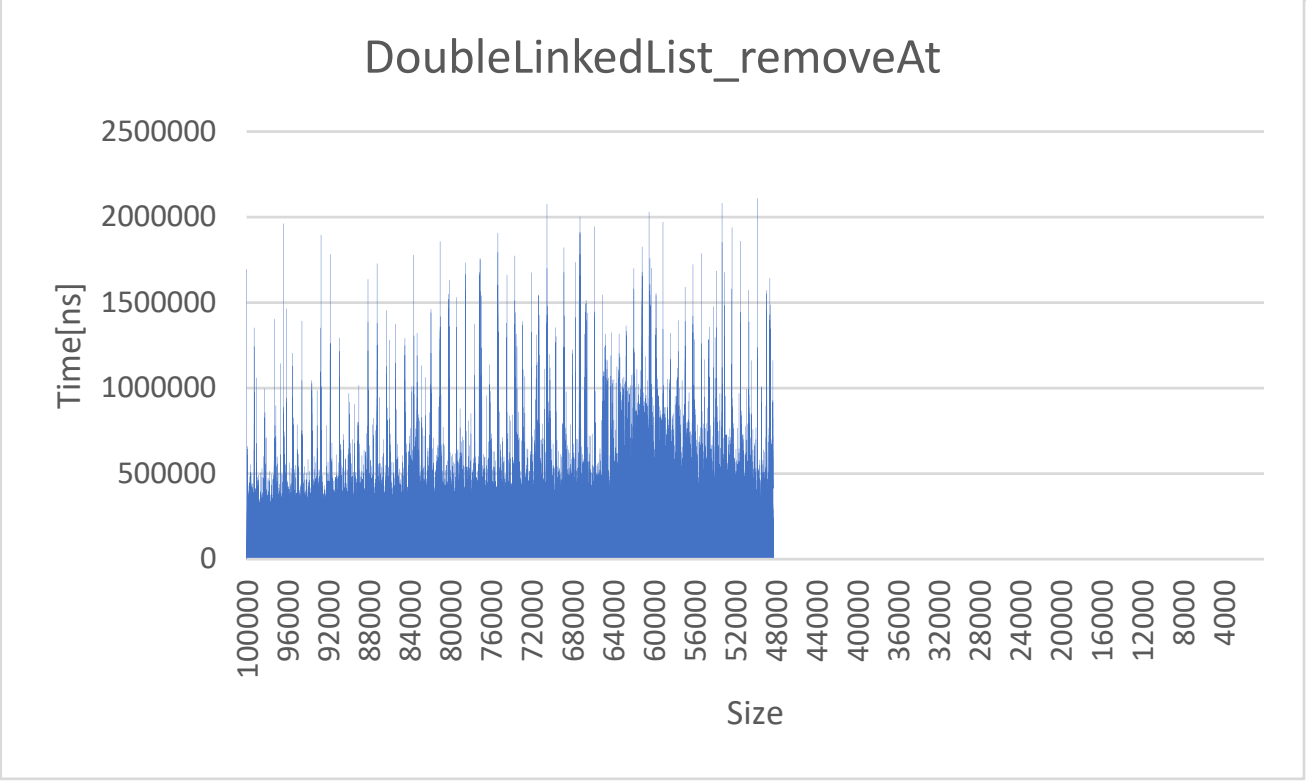
RemoveAt

Tak ako v predošlých príkladoch, tak aj tu je ArrayList rýchlejší.

Musíme sa na graf ale pozerať z opačnej strany, pretože veľkosť ide od najväčšej po najmenšiu.

Zložitosť pri **ArrayListe** vyzerá byť **lineárna** a zložitosť **DoubleLinkedListu** sa znova veľmi podobá na **logaritmickú** zložitosť pretože za polovicou sa čas stabilizuje.





Záver – Odhad hornej asymptotickej zložitosti

Záver

Na základe grafov by som sa pri **vkladaní** prvkov pri **väčšom množstve** rozhodol pre **DoubleLinkedList**, pri **sprístupnení** prvkov pri **väčšom množstve** pre **ArrayList** a pre **odstránenie prvkov** pri **väčšom množstve** pre **DoubleLinkedList**.

Insert

- ArrayList
  - **O(a^n)**
- DoubleLinkedList
  - **O(log(n))**

At

- ArrayList
  - **O(1)**
- DoubleLinkedList
  - **O(log(n))**

RemoveAt

- ArrayList
  - **O(n)**
- DoubleLinkedList
  - **O(log(n))**