# Data Structures and Algorithms

## Structures benchmarking

## Overview

Structures benchmarking is semester project for the school subject Data Structures and Algorithms.   
The goal was to program and experimentally compare the performance of different implementations of data structures.

The structures and the test application are written in C ++ with manual memory management.  
Qt Framework was used for better and more convenient testing and visualization.

## Krátka príručka k aplikácií

1

A screen shot of a computer

Description automatically generated

7

8

5

6

4

3

2

1. Komponent ComboBox:   
   Slúži pre výber ADT testovacej štruktúry. V aktuálnej implementácií sú na výber ADT: BitSet, List, Matrix, PriorityQueue.
2. Komponent List:   
   Obsahuje inštancie jednotlivých ADS vybranej ADT.
3. Komponent pre výber operácií pre testovanie ADT:   
   Užívateľ si dokáže sám zostaviť pravdepodobnosti operácií nad ADS. Pre lepší užívateľský komfort sa na pravej strane nachádza komponent ProgressBar, ktorý znázorňuje celkovú pravdepodobnosť vykonania zadaných operácií.
4. Komponent pre prepojenie zadávania testov s tvorbou a vizualizáciou grafov. Užívateľ dokáže zadať počet vykonaných operácií pomocou SlideBaru (0 - 1M).  
   V ponuke je CheckBox „CSV“. Pokiaľ je tlačidlo odškrtnuté, výsledky sa budú ukladať do CSV súboru.  
   Pomocou tlačidla Generate sa vygeneruje graf/zapíše do CSV.  
   Tlačídlo Clear zmaže vygenerovaný graf a uvoľní jednotlivé komponenty.
5. Legenda pre grafy. Znázorňuje farbu jednotlivých ADS v grafoch.   
   Užívateľ dokáže zobraziť/skryť „series“ pre jednotlivé ADS kliknutím na legendu.
6. Komponent pre vykreslenie a manipulovanie s výslednými časmi operácií nad danými ADT.  
   X-ová os znázorňuje počet vykonaných operácií

Y-ová os znázorňuje čas vykonania operácií v nanosekundách [ns]  
Pre vizualizáciu som sa rozhodol použiť ScatterSeries, nakoľko sa z nej dá vyčítať lepšie informácie, ako z LineSeries  
Užívateľ dokáže pohybovať grafom (mouse/touchpad).

Stlačením kláves ArrowUp/ArrowDown dokáže meniť range na Y-ovej osy v násobkoch 2. Táto funkcia sa zíde hlavne pri porovnávaní operácií s veľkými časovými rozdielmi.

1. Komponent pre vykreslenie časov vykonania všetkých operácií nad vybranými ADS.  
   Jednotlivé PieSeries znázorňujú celkový čas spracovania vybraných ADS.
2. A screen shot of a computer

   Description automatically generatedKomponent pre vykreslenie a manipulovanie s výslednými pomermi vykonania operácie nad danými ADS.  
   Jednotlivé BarSeries znázorňujú vybrané operácie pre testovanie nad vybranými ADS.  
     
   Po stlačení klávesy SPACE dokáže užívateľ zobraziť Komponent 8 na celú obrazovku a lepšie vyčítať údaje.   
     
   BarChart po stlačení klávesy Space

## Popis testovaných štruktúr

* **List<T>**

**ArrayList** : Dynamické pole v súvislej pamäti.   
**LinkedList** : Zreťazený zoznam.   
D**oublyCircularLinkedList** : Dvojitý zreťazený cyklický zoznam.

* **PriorityQueue<T>**  
  **PriorityQueueUnsortedArrayList** : Prioritný front implementovaný pomocou dynamického poľa neutriedene.  
  **PriorityQueueSortedArrayList** : Prioritný front implementovaný pomocou dynamického poľa utriedene.  
  **PriorityQueueLinkedList** : Prioritný front implementovaný pomocou zreťazeného zoznamu.  
  **PriorityQueueTwoList** : Prioritný front implementovaný pomocou 2 zoznamov. Kratší zoznam je limitované dynamické pole implementované ako prioritný front, dlhší zoznam je zreťazený zoznam.  
  **Heap** : Prioritný front implementovaný ľavostrannou haldou.
* **Matrix<T>  
  MatrixHeap** : Matica v nesúvislej pamäti.  
  **MatrixHeapContinous** : Matica implementovaná v súvislej pamäti za pomoci mapovacej funkcie.  
  **MatrixStack** : Matica implementovaná na Stacku(zásobníku).
* **BitSet<T>  
  BaseSet:** bázová množina implementovaná poľom.  **BitSet :** Bitová množina namapovaná na bázovú s podporou bitových operácií(prienik, zjednotenie, rozdiel...). **BitSetLarge** Bitová množina s 10x väčším počtom prvkov oproti pôvodnej.

## Popis testovania

Jadrom testovania vybraných štruktúr sú triedy **TestTime**(Timer) a **TestStructure.** Obe triedy sú navrhnuté genericky, aby dokázali testovať štruktúry nezávislé na type.   
Ďalej aplikácia obsahuje triedy **ListOperations**, **PriorityQueueOperations**, **MatrixOperations**, **BitSetOperations**, ktoré obaľujú jednotlivé operácie nad ADT, a ďalšie operácie potrebné pre testovanie vybraného scenára(generovanie náhodných čísel...).  
  
**TestTime**Triedu **TestTime** som sa snažil navrhnúť čo najvšeobecnejšie, aby dokázala časovať akúkoľvek funkciu s rozličným počtom parametrov.   
Pre tieto potreby som použil pomerne novú feature C++, šablóny s variabilným počtom parametrov.   
Pre časovanie používam std::chrono s vysokou precíznosťou na nanosekundy.   
Trieda **TestTime** je navrhnutá ako statická, nakoľko s ňou aplikácia pracuje často a počas celej doby trvania aplikácie.  
  
Tu je ukážka implementácie časovača.   
A screenshot of a cell phone

Description automatically generated  
  
**TestStructure**Trieda **TestStructure** ukladá a testuje jednotlivé operácie predané ako pointre na funkcie.   
Na základe zadanej pravdepodobnosti ďalej vyberá z jednotlivých pointrov na funkcie, a posiela ich do **TestTime**(Timeru).   
A screenshot of a cell phone

Description automatically generated

Spracovanie výsledkov testov

Úroveň 1:

A screen shot of a computer

Description automatically generated Scenár A:  
  
V **scenári A** je vhodnejšie použiť **ArrayList**, nakoľko prístup k prvkom na náhodnej pozícií predstavuje prevažnú väčšinu testovaných operácií.  
Dôvodom je, že prístup k prvkom v súvislej pamäti je rýchli, nakoľko pozícia prvku sa dá efektívne vypočítať podľa bázovej adresy začiatku ArrayListu.

Zložitosť O(1).  
Naopak pri LinkedListe je táto operácia pomerne neefektívna, nakoľko nemá žiadné dodatočné informácie a jediný spôsob je preiterovať zoznam pokiaľ sa nedostane na daný index.   
Zložitosť O(n).  
  
V hore uvedenom grafe je bližší pohľad na porovnanie výkonu jednotlivých operácií.   
A screen shot of a computer

Description automatically generated

A screen shot of a television

Description automatically generated**Scenár B:**

V **scenári B** je taktiež efektívnejší ArrayList, nakoľko stále je pomerne veľká časť operácia závislá na rýchlosti prístupu na náhodný prvok, ale ako je vidieť podľa grafu, tento scenár je o niečo vhodnejší pre **LinkedList** ako predošlí, nakoľko je častejšie vkladanie na oba konce zoznamu, vyberanie zo začiatku a o niečo menej prístupov.   
Aj napriek tomu je vhodnejšie použiť pre takýto scenár **ArrayList**, nakoľko sa k prvkom častejšie pristupuje, ako modifikuje.  
  
  
Podrobnejší pomer operácií sa nachádza v nižšie uvedenom grafe.

A screen shot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generatedA screen shot of a computer

Description automatically generatedScenár C:

Výsledky **Scenára C** sú opäť v prospech ArrayListu, ale ako je vidieť na grafoch, tento scenár je z predošlých dvoch najvhodnejší pre LinkedList z dôvodu častejšieho vkladania ako pristupovaniu k prvkom.

Pre tento scenár je vhodnejšie použiť LinkedList s dvojitým zreťazením vzhľadom na efektívnejšiu implementáciu operácií zrusPosledny, a čiastočne aj prístupu k prvkom, nakoľko sme ho algoritmicky rozdelili na nutnosť preiterovať polovicu zoznamu. Obrázok nižšie porovnáva **Scenár C ArrayList vs DoublyLinkedList**

A screen shot of a computer

Description automatically generated

A picture containing monitor, screen

Description automatically generatedBližší pohľad na porovnanie operácií.

Úroveň 2:

Scenár A:

A screenshot of a computer screen

Description automatically generatedA screen shot of a computer

Description automatically generated

Pre lepšie porovnanie som pridal okrem **Heap** a **PQLinkedList** aj tretiu štruktúru, **PriorityQueueSortedArrayList**, nakoľko je pekne vidieť, rozdiel zložitosti vkladania/vyberania v utriedenom a neutriedenom liste.  
**Heap** je zo všetkých najefektívnejšia, keďže zložitosť operácií je O(log2n).   
**PriorityQueueList** je efektívny pri vkladaní, pri výbere je však veľmi neefektívny, nakoľko sa prvky musí iterovať nad celým listom a hľadať, či má daný prvok minimálnu prioritu.  
Vkladanie: 0(1), Výber O(n).  
**PriorityQueueSortedArrayList** ako je vidieť na grafoch má opačný problém. Vkladanie je neefektívne O(n), a vyberanie efektívne O(1).

A screen shot of a computer

Description automatically generated  
Scenár B:   
**Scenár B** je veľmi neefektívny pre implementáciu prior. Frontu zreťazeným zoznamom, keďže výber prvkov je o polovicu menší, ako vkladanie, tým pádom pri operáciach pop a peek je veľmi náročné prechádzať a hľadať prvok s najlepšou prioritou.  
V prípade ľavostrannej haldy je to vhodná kombinácia operácií.  
  
Pri tomto scenári som sa rozhodol znížiť počet operácií na cca 100 000, aby boli grafy ako tak čitateľné.

A screenshot of a computer screen

Description automatically generated

## Úroveň 3:

Scenár Prístup:

Pri maticiach som pridal tretiu implementáciu, **MatrixStack**(matica implementovaná viacrozmerným poľom v zásobníku.).

A screen shot of a computer

Description automatically generated

Prístup do pamäte, implementácie sú takmer zhodné (Mercedes ☺)

Nakoľko pri prístup do pamäte je veľmi rýchla operácia pri každej z implementácií, rozhodol som sa pridať tretí scenár (násobenie matíc). Taktiež sa jedná prevažne o rýchlosť prístupu do pamäti, ale prístupov je výrazne viac. Testujem násobenie matíc 100 x 100, takže pri jednej operácií násobenia ( O(n^3)) je tých prístupov výrazne viac a je vidieť rozdiel prístupu do pamäti v rôznych implementáciach.

A screen shot of a computer

Description automatically generatedAko je vidieť z grafu, častý prístup v prípade násobenia je vhodnejší pre súvislú pamäť.  
Najrýchlejší je samozrejme do Stacku.

Úroveň 4:  
  
Obojstranne zreťazený cyklický zoznam:   
  
Pri tejto implementácií som sa rozhodol najskôr implementovať **DoublyLinkedList** a pomocou dedičnosti som spravil drobné úpravy v implementácií a implementoval taktiež **DoublyCircularLinkedList**.   
  
Rozhodol som sa ho porovnať s implementáciou klasického **LinkedListu**.

A screen shot of a computer

Description automatically generated

A picture containing monitor, clock

Description automatically generated

Ako je vidieť v grafe, výraznú zmenu predstavuje operácia ZrusPosledny, nakoľko DoublyLinkedList si uchováva aj pointer na predchádzajúci prvok, dokážeme túto operáciu pomerne efektívne implementovať. Taktiež je vidieť, že operácie s náhodným prístupom sú v DoublyLinkeListe implementované o trochu efektívnejšie, nakoľko sme našou implementáciou docielili, že stačí preiterovať polovicu Listu.

A screen shot of a computer

Description automatically generated**Dvojzoznam ako implementácia prioritného frontu**:

ScenarA:

Ako môžme vidieť z grafu, implementácia **prioritného frontu Dvojzoznamom** je o niečo menej efektívna, ako implementácia ľavostrannou haldou. Pomerne efektívne sú operácie vkladania, nakoľko sa kontroluje len či je vkladaný prvok menší ako prvky obmedzeného ArrayListu implementovaného prioritným frontom, a však pri operácií **Pop** nastáva pomerne zložitá operácia reštrukturizácie. Pokiaľ je však tento front implementovaný efektívne, dá sa tento nežiadúci jav výrazne obmedziť. Jedným z riešení je nastavovať kapacitu menšieho zoznamu na sqrt(dĺžka väčšieho). Týmto docielíme, že amortizovaná zložitosť operácií bude O(sqrt(n)). Čo je oproti **Heap** stále pomerne neefektívne, ale v porovnaní s ostatnými implementáciami je to pomerne efektívna implementácia.

A screenshot of a computer screen

Description automatically generated

Scenár B:

A screen shot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated**A screenshot of a computer

Description automatically generatedMnožina ako bitová mapa:**

Ako je z grafu vidieť, množinové operácie nie sú závislé na počte prvkov v množine, ale na veľkosti bázovej množiny. Operácie sú prevažne konštantné, ale ak máme bitovú množinu viazanú na bázovú, je prístup k prvku závislý od funkcionality getIndexOf(prvok) bázovej množiny. V tomto konkrétnom prípade je bázová množina implementovaná statickým poľom.

**Vplyv veľkosti menšieho zoznamu na PriorityQueueTwoList**

Rozhodol som sa testovať túto štruktúru s troma implementáciami.   
Prvá má počiatočnú kapacitu kratšieho zoznamu nastavenú na 4 prvky. (modrá)

Druhá má počiatočnú kapacitu kratšieho zoznamu nastavenú na 64 prvky. (oranžová)  
Tretia má počiatočnú kapacitu kratšieho zoznamu nastavenú na 1024 prvky. (zelená)  
  
Ako je z nižšie uvedených grafov vidieť, je dôležité trafiť počiatočnú kapacitu na správnu hodnotu pre budúce použitie štruktúry, nakoľko pri neskoršom používaní sa táto kapacita nebude zmenšovať, a môže sa stať, že sa z implementácie stane PriorityQueueSortedArrayList pokiaľ prestrelíme veľkosť kapacity.  
  
V prvom testovacom scenári je vkladanie a výber prvkov nastavený rovnomerne. Ako je vidieť,

**A screen shot of a computer

Description automatically generated**

A screenshot of a computer screen

Description automatically generated

V tomto scenári je vidieť, že počiatočná veľkosť 1024 bola príliš, nakoľko sú operácie Push a Pop rovnocenné, a tak veľký zoznam 1024^2 pravdepodobne nedocielime.   
Počiatočná kapacita 64 sa zdá byť najefektívnejšia, nakoľko aj keď je Push a Pop rovnocenné, pri milión operácií môžme očakávať, že bude zoznam obsahovať 64^2 prvkov.  
Ako môžme vidieť v stĺpcovom grafe, zložitosť operácie Pop sa prenáša na zložitosť operácie Push čím väčšiu počiatočnú kapacitu zvolíme.  
  
A screenshot of a computer screen

Description automatically generated

Utvrdiť sa v tom môzme aj pri tomto scenári. Zložitosť operácií sa prenáša z Pop na Push, čím väčšiu počiatočnú kapacitu PQ obsahuje.  
Je dôležité trafiť kapacitu tak, aby prípadne sedelo budúcemu použitiu štruktúry, ideálne na sqrt(možnej dĺžky).