VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

IMPLEMENTACE KALENDÁŘE UDÁLOSTÍ

PENDING EVENT SET IMPLEMENTATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. DANIEL KOZOVSKÝ**
AUTHOR

VEDOUCÍ PRÁCE — **Dr. Ing. PETR PERINGER**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2020/2021

Zadání diplomové práce

Student: **Kozovský Daniel, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Inteligentní systémy
Název: **Implementace kalendáře událostí**
Pending Event Set Implementation
Kategorie: Modelování a simulace
Zadání:

1. Analyzujte různé implementace kalendáře událostí pro použití v simulačních systémech. Zaměřte se především na různé varianty *Calendar Queue*.
2. Navrhněte knihovnu implementující několik různých kalendářů s možností změny implementace kalendáře při simulaci. Knihovna musí obsahovat minimálně 8 různých implementací.
3. Knihovnu implementujte v C++. Napište sadu testů pro ověření správnosti a výkonnostních charakteristik jednotlivých implementací. Výsledky testů přehledně zpracujte a porovnejte je s výsledky publikovanými v dostupné literatuře.
4. Zhodnoťte dosažené výsledky a navrhněte možná vylepšení.

Literatura:

- Brown R.: *Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem*. Communications of the ACM, 1988.
- Tan K., Thng L.: *SNOOPy Calendar Queue*. 2000 Winter Simulation Conference Proceedings. IEEE, 2000.

Při obhajobě semestrální části projektu je požadováno:

- Splnění prvních 2 bodů zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/
Vedoucí práce: **Peringer Petr, Dr. Ing.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 19. května 2021
Datum schválení: 11. listopadu 2020

Abstrakt

Táto práca sa zaoberá vytvorením knižnice v C++, ktorá implementuje rôzne varianty kalendára udalostí, ktorý sa používa na riadenia diskrétnych simulácií. Knižnica zahŕňa deväť rôznych implementácií kalendára udalostí, ktoré sú prístupné cez jednotné rozhranie. Toto rozhranie je navrhnuté tak, aby bolo jednoduché knižnicu rozšíriť o ďalšie implementácie. Okrem samotnej knižnice sa práca zaoberá aj návrhom a popisom testovacej aplikácie a zhodnotením časových zložitostí jednotlivých implementácií.

Abstract

This work aims to create a library in C++, which implements various variants of the pending event set, which is used in discrete simulations. The library includes nine different implementations of the pending event set, accessible through a single interface. This interface is designed to make it easy to extend the library with additional implementations. In addition to the library itself, the work also describes the design of the test application and evaluates the time complexity of individual implementations.

Kľúčové slová

kalendár udalostí, modelovanie, simulácie, prioritná fronta, calendar queue

Keywords

pending event set, modelling, simulation, priority queue, calendar queue

Citácia

KOZOVSKÝ, Daniel. *Implementace kalendáře událostí.* Brno, 2021. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Dr. Ing. Petr Peringer

Implementace kalendáře událostí

Prehlásenie

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne pod vedením Dr. Ing. Petra Peringera. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Daniel Kozovský
17. mája 2021

Poďakovanie

Chcel by som sa poďakovať vedúcemu práce Dr. Ing. Petrovi Peringerovi za jeho pomoc pri vypracovaní tejto práce.

Obsah

Kapitola 1

Úvod

V dnešných dňoch informačných technológií sa často snažíme odhaliť vlastnosti systémov pomocou počítačových simulácií. Tieto simulácie sa vykonávajú na určitých modeloch, ktoré reprezentujú iné systémy. Simulácie môžu byť buď spojité alebo diskrétne. Na riadenie diskrétnych simulácií sa často používa kalendár udalostí. Kalendár udalostí je štruktúra, ktorá je schopná uchovať udalosti a zaručuje, že pri každom výbere bude vybraná udalosť s najmenšou časovou značkou.

Cieľom tejto práce je návrh a implementácia knižnice, ktorá takýto kalendár udalostí implementuje. Knižnica má byť navrhnutá tak, aby obsahovala viac implementácií, keďže každá z implementácií má svoje výhody a nevýhody. Knižnica sa snaží tieto implementácie zabaliť do jednoducho používateľného celku, ktorý umožňuje použitie akejkoľvek implementácie cez jednotné rozhranie. Taktiež má byť možné tieto implementácie meniť za behu simulácie.

Kapitola 2 sa zaoberá kalendárom udalostí a možnosťami jeho implementácie. Obsahuje základnú teóriu a niekoľko rôznych možností ako implementovať kalendár udalostí, vrátane rozboru zložitostí operácií. Kapitola 3 sa zaoberá návrhom knižnice pre kalendár udalostí. Predstavuje diagram tried, popisuje jednotlivé triedy, ich metódy a nakoniec predstavuje príklad použitia navrhovanej knižnice. Kapitola 4 obsahuje popis implementačných detailov a výsledky experimentov nad knižnicou.

Kapitola 2

Kalendár udalostí a jeho implementácie

Kalendár udalostí [7] je využívaný v diskrétnych simuláciách [9]. Ide o simulácie diskrétnych systémov, kde udalosti prichádzajú diskrétne v čase. Každá udalosť má pridelený svoj čas a každá udalosť značí zmenu stavu v systéme. Medzi dvoma po sebe idúcimi udalosťami k zmene stavu nedochádza. Z tohoto plynie aj simulačná metóda next-event, ktorá vždy "skočí"na čas nasledujúcej udalosti. Jej pseudokód je možné vidieť na kóde 2.1.

```
cas = T_START
inicializacia kalendara a modelu
while( Kalendar je neprazdny ) {
    vyber prvy aktivacny zaznam (AZ) z kalendara
    if ( cas atime v AZ > T_END )
        break; Ukoncenie cyklu
    Nastav cas na aktivacny cas atime v AZ
    Vykonaj popis chovania udalosti event v AZ
}
cas = T_END; Koniec simulacie.
```

Kód 2.1: Pseudokód algoritmu next-event

Vykonávanie udalostí obvykle zahŕňa aj vytváranie, plánovanie a rušenie ďalších udalostí. V prípade popisu formou procesov je vykonávanie udalostí implementované ako pokračovanie v popise procesu od miesta, kde bol prerušený. Ak nieje požadovaná žiadna ďalšia budúca udalosť, je kalendár prázdny a simulácia môže skončiť.

V tejto kapitole je vysvetlené čo je kalendár udalostí, na čo slúži a možnosti jeho implementácie. U jednotlivých implementácií sú priblížené ich výhody, nevýhody a ich fungovanie.

2.1 Výpočtová zložitosť algoritmu

Pre hodnotenie kvality algoritmov a ich následné porovnávanie je potrebné zvoliť vhodné kritérium. Takýmto kritériom je čas potrebný na vykonanie algoritmu a pamäťový priestor ktorý je potrebný na vykonanie daného algoritmu. Čas potrebný na výpočet algoritmu je uvedený ako časová zložitosť algoritmu[3], a potrebný pamäťový priestor sa uvádza ako priestorová zložitosť algoritmu. Čas aj priestor potrebný pre algoritmus závisí na počte

spracovávaných dát, a preto majú tieto zložitosti podobu funkcie, ktorá závisí na počte položiek N.

Asymptotická časová zložitosť je najčastejším kritériom pre hodnotenie algoritmov. Vyjadruje sa porovnaním algoritmu s určitou funkciou pre N blížiacemu sa nekonečnu. Porovnanie má podobu troch rôznych zložitostí, a to:

- O Omikron - vyjadruje hornú hranicu chovania
- Ω Omega - vyjadruje dolnú hranicu chovania
- Θ Theta - vyjadruje triedu chovania

2.2 Kalendár udalostí

Kalendár udalostí je dátová štruktúra typu prioritná fronta. Presnejšie ide o frontu záznamov o udalostiach, ktoré sú naplánované na určitý čas. Každá naplánovaná budúca udalosť má v kalendári záznam v tvare (čas,priorita,udalosť). Kalendár udalostí sa používa v rôznych simulačných nástrojoch pri simuláciách, ktoré potrebujú plánovať udalosti do budúcnosti.

Táto fronta obsahuje metódy:

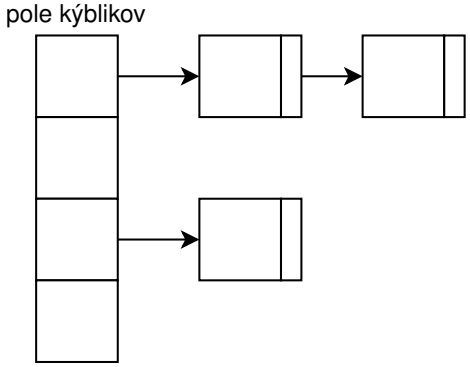
- nájdenie prvku na čele fronty (anglicky find min)
- výber prvku na čele fronty (anglicky dequeue)
- vloženie novej udalosti (anglicky enqueue)
- odstránenie ľubovolnej naplánovanej udalosti (anglicky delete event)
- inicializácia fronty (anglicky initialize)
- zrušenie fronty (anglicky destroy)

Pre príklad je možné kalendár udalostí prirovnať k bežnému kalendáru dní, ktorý všetci dobre poznáme. Bežný kalendár obsahuje dni, a my sme schopný nejakú budúcu udalosť naplánovať na ľubovolný deň nachádzajúci sa v budúcnosti. V tomto prípade ide o frontu udalostí predstavenú jednotlivými dňami, ale zároveň je možné nejakú udalosť vložiť už medzi dve naplánované udalosti.

2.3 Calendar Queue

Calendar queue [1] je prioritná fronta inšpirovaná reálnym kalendárom dní. Jej grafické znázornenie je možné vidieť na obrázku 2.1. Pozostáva z poľa, v ktorom jeden prvok sa nazýva kýblik (anglicky bucket). Každý kýblik predstavuje časový interval a môže obsahovať viac udalostí. Čiže obsahuje udalosti spadajúce do intervalu, ktorý tento kýblik reprezentuje. Udalosti sú v kýbliku usporiadané podľa priority tak, že udalosti, ktoré sa majú vykonať skôr majú vyššiu prioritu ako tie, ktoré sa majú vykonať neskôr.

Veľkosť a počet kýblikov ovplyvňuje rozpoznávaciu silu tejto fronty. Snažíme sa mať toľko kýblikov, aby každý kýblik obsahoval v ideálnom prípade práve jednu položku. Toto sa snažíme docieliť pre to, aby sme dokázali vkladať aj vyberať udalosti v skoro konštantnom



Obr. 2.1: Grafické znázornenie implementácie calendar queue

čase. Príliš malé množstvo kýblikov spôsobí, že sa spomalí vkladanie nových udalostí do fronty. Naopak príliš veľké množstvo kýblikov spomalí výber a odstraňovanie z fronty.

Práve pre toto je veľmi dôležitou súčasťou tejto fronty spôsob dynamickej úpravy veľkosti počtu kýblikov. Dynamický spôsob zaručí, že aj napríklad fronta, v ktorej očakávame veľké množstvo udalostí, napríklad 10000, bude efektívna aj pri počiatočnej nižšej záťaži. Frontu počiatočne inicializujeme na menšiu veľkosť, a vždy, keď fronta dosiahne počet udalostí viac ako dva-krát počet kýblikov, tak inicializujeme novú frontu, ktorá bude väčšia a presunieme do nej naplánované udalosti. Naopak, keď pri odoberaní udalostí dosiahneme počet udalostí menší ako polovica kýblikov, tak vytvoríme menšiu frontu. Pri zväčšovaní používame typicky zdvojnásobenie počtu kýblikov a pri zmenšovaní zase polovicu počtu kýblikov. Keď sa pozrieme na podmienku zväčšovania, respektíve zmenšovania, tak zistíme že po úprave dosiahneme rovnaký počet kýblikov ako je počet udalostí. Pre túto vlastnosť sa nám rýchlosť tejto fronty drží približne konštantná, až na body, kde je nutné robiť zväčšenie, respektíve zmenšenie počtu kýblikov vo fronte. Táto fronta je efektívna, keď má obsahovať veľké množstvo udalostí.

Keby chceme implementovať bežný kalendár pomocou calendar queue, použijeme 365 kýblikov, kde každý kýblik predstavuje jeden deň. Následne, ak deň obsahuje viac ako jednu udalosť, tak tieto udalosti usporiadame podľa ich času. Pri dosiahnutí veľkého množstva udalostí môžme ďalej pre zefektívnenie prejsť na rozoznávaciu silu pol dňa o veľkosti 730 kýblikov.

Časová zložitosť

Výber prvku - $O(1)$.
Vloženie prvku - $O(1)$.

Tieto časové zložitosti sú dosiahnuté pri ideálnom fungovaní kalendára. Calendar queue predpokladá rovnomerne rozložené udalosti v čase, a preto nemusí fungovať správne pri udalostiach, ktoré sú nerovnomerne rozložené v čase. Časová zložitosť je konštantná, ale len ak nedochádza k zmene veľkosti kalendára a následnému presunu udalostí. Táto operácia zmeny veľkosti je závislá na aktuálnom počte udalostí v kalendári a dosahuje zložitosť $O(n)$.

2.4 Dynamic Calendar Queue

Dynamic calendar queue [6] vychádza z bežného calendar queue (obrázok 2.1), kde rieši jeho veľký problém. Tento problém nastáva, keď máme nepravidelné rozloženie udalostí.

Keď rozložíme udalosti príliš blízko k sebe, tak môže dôjsť k tomu, že všetky udalosti padnú do jedného kýblika. V takomto prípade sa calendar queue bude chovať ako lineárny zoznam, čo nebude vôbec efektívne.

Dynamic calendar queue na riešenie tohto problému pridáva dva nové mechanizmy. Prvý mechanizmus sleduje vhodný čas na zmenu veľkosti tak, že sleduje priemerný čas potrebný na pridanie, respektíve odobranie udalosti. V bežnom calendar queue to závisí len na počte udalostí a nie na rozložení udalostí. Druhý mechanizmus je aproximácia priemerného časového rozdielu medzi udalosťami presne výberom vhodných udalostí. Tento časový rozdiel dokáže ukázať ako budú udalosti rozložené naprieč frontou.

Na detekciu nerovnomerného rozloženia používa dynamic calendar queue cenu každého pridania a odobrania prvku jednotlivo a spriemeruje tieto dve akumulované ceny po tom čo sú prvky pridané alebo odobrané toľkokrát ako je počet kýblikov. Takže pre frontu s počtom kýblikov osem urobí priemer po ôsmych operáciach pridania, respektíve odobrania udalostí. Cena operácie pridania udalosti je definovaná ako počet udalostí, ktoré sú pri konkrétnom pridávaní navštívené, čiže rastie s počtom udalostí s väčšou prioritou nachádzajúcich sa v príslušnom kýbliku. Cena operácie odobranie udalosti je definovaná ako počet kýblikov, ktoré je nutné navštíviť pri výbere prvého prvku, čiže rastie s počtom prázdnych kýblikov pred prvým naplneným kýblikom. Keď aspoň jeden z týchto dvoch priemerov presiahne určitú definovanú hranicu, tak to znamená, že udalosti nie sú rovnomerne rozložené medzi kýbliky. V takom prípade dynamic calendar queue prepočíta šírku kýblikov a pre distribuuje udalosti do konfigurovaných kýblikov.

Keď dynamic calendar queue zmení počet kýblikov, tak druhý mechanizmus určí priemernú medzeru medzi udalosťami, presnejšie šírku kýblika v závislosti na rozložení udalostí. Základná calendar queue zisťuje priemernú medzeru medzi udalosťami tak, že pozrie na niekoľko prvých udalostí a na ich základe spočíta priemernú hodnotu. Toto sa ale môže líšiť od reálnej medzery, ak nemáme rovnomerné rozloženie udalostí. Napríklad ak budeme mať rozdelených prvých 10 udalostí v časovom intervale 1 - 10 a ďalších 100 udalostí v intervale 10 - 12. V tomto prípade ak spočítame priemernú medzeru z prvých 10 udalostí, tak dosiahneme nepravdivú hodnotu. Pre dosiahnutie presnej hodnoty je nutné zvoliť vzorky z najhustejšie zaplneného intervalu.

Na hľadanie najhustejšieho regiónu používa dynamic calendar queue dve premenné. Prvá pre každý kýblik počíta počet udalostí v kýbliku, druhá obsahuje identifikáciu kýblika s najväčším množstvom udalostí. Po zmene veľkosti počtu kýblikov dôjde k bežnému prerozdeleniu udalostí do nových kýblikov, ale okrem toho dynamic calendar queue zistí či sú udalosti rovnomerne rozložené medzi tieto kýbliky. Toto zisťuje tak, že získa pomer z kýblikov v okolí kýblika s najväčším množstvom udalostí ku celkovému počtu udalostí. Keď tento pomer presiahne určitú hranicu, tak môžeme povedať že udalosti nie sú rovnomerne rozložené. Vtedy je potrebné prepočítať novú priemernú medzeru medzi udalosťami a následne upraviť šírku kýblikov tak, aby boli udalosti rovnomerne rozložené.

Časová zložitosť

Výber prvku - $O(1)$.
Vloženie prvku - $O(1)$.

Tieto časové zložitosti sú podobne ako aj calendar queue dosiahnuté za ideálneho fungovania kalendára. Dynamic calendar queue sa o niečo lepšie dokáže vysporiadať s nerovnomerne rozloženými udalosťami, ale taktiež nemusí dosiahnuť najlepšie nastavenia šírky kýblikov.

2.5 SNOOPy Calendar Queue

SNOOPy calendar queue [5] vychádza tiež zo základnej varianty calendar queue, viď obrázok 2.1. Taktiež ako dynamic calendar queue si uvedomuje problém tejto implementácie pri nerovnomernom rozložení udalostí v čase. Lenže obe, aj calendar queue aj dynamic calendar queue počítajú svoje parametre na základe vzorkovania udalostí v kalendári. Ale vzorkovanie nieje schopné vždy odhaliť optimálnu šírku kýblikov. Keď dôjde k nesprávnemu určeniu šírky kýblikov, tak sa zložitosť týchto implementácií zhoršuje od predstavenej zložitosti O(1).

SNOOPy calendar queue prináša metódy ako odhaliť správne nastavenie šírky kýblikov. K základnému calendar queue pridáva dva mechanizmy. Prvý je zodpovedný za iniciovanie zmeny veľkosti kýblikov (SNOOPy triggering process) a druhý zabezpečuje výpočet správnej šírky kýblikov vždy, keď je nutné zmeniť šírku (SNOOPy bucket width optimisation).

SNOOPy bucket width optimisation

Pri zmene šírky kýblika sa SNOOPy calendar queue snaží minimalizovať funkciu súčtu priemerného vybratia udalosti a priemerného vloženie udalosti. Premennú, ktorú optimalizuje, je šírka kýblika a počet kýblikov je konštantný. Matematicky zapísané :

$$\min_{B_W} C = C_E + C_D, \text{ za predpokladu že } N_B \text{ konštantné}$$

Kde C je cenová funkcia, C_E je cena priemernej operácie pridania prvku, C_E je cena priemernej operácie vybratia prvku, B_W je šírka kýblika a N_B je počet kýblikov. Zmenu veľkosti kýblika realizujeme tak, že aktuálnu šírku kýblika vynásobíme konštantou $k : B_W = k * B_W$. Zmena C_E a C_D závisí na konštante k, a preto môžme optimalizačný problém upraviť na :

$$\min_k C' = \min_k C'_D + C'_E = \min_k \frac{C_D}{g_1(k)} + g_2(k) * C_E$$

Kde $g_1(k)$ a $g_2(k)$ sú väčšie rovné jednej, a obe sú monotónne rastúce funkcie premennej k. Obe taktiež spĺňajú podmienku : $g_1(1) = g_2(1) = 1$. Nové priemerné ceny C'_D a C'_E zostanú optimalizované len krátku dobu po zmene šírky kýblika, kým sa nezmení rozloženie udalostí vo fronte.

Funkcie g_1 a g_2 závisia na rozložení udalostí vo fronte, ako aj od premennej k. Určenie presných funkcií je časovo náročné, a preto nemá zmysel tieto funkcie určovať presne. Namiesto toho určíme najlepší a najhorší scenár pri zmene ceny na základe zmeny šírky kýblikov.

Pre prípad priemernej ceny výberu prvku môžme usúdiť, že v najhoršom prípade nedôjde k žiadnemu zlepšeniu. To nastane, keď aj po zmene šírky zostane relatívne uloženie udalostí rovnaké. V najlepšom prípade môže dôjsť až k tomu, že dosiahneme k-krát menšiu cenu. Do úvahy musí brať aj to, že ideálna cena výberu prvku nastane keď rozšírime kýbliky tak veľmi, že všetky udalosti padnú do jedného kýblika. Táto varianta je ale nežiadúca, lebo by sme dostali jeden lineárny zoznam udalostí. Preto musíme obmedziť k tak, aby k tomu nedošlo. Podľa týchto hraníc môžme povedať, že optimalizované C'_D je :

$$\frac{C_D}{k} \leq C'_D \leq C_D$$

Podobne pre cenu vloženia prvku vieme povedať najlepší a najhorší scenár pri zmene šírky. Najlepší scenár je, že cena operácie vloženia sa nezmení. Toto nastane práve v situácií,

keď sa žiadne dva kýbliky nespoja v jeden, maximálne dôjde k natlačeniu kýblikov bližšie k sebe. V najhoršom prípade dôjde k tomu, že sa cena zvýši k-krát. Podľa tohto môžeme matematicky povedať, že optimalizované C'_E je :

$$C_E \leq C'_E \leq k * C_E$$

Po definovaní hraníc pre cenové funkcie C'_E a C'_D môžme spraviť permutáciu u cenovej funkcie nasledovne :

$$C'_D + C'_E = \frac{1}{2}(\frac{C_D}{k} + C_D) + \frac{1}{2}(k * C_E + C_E)$$

Táto rovnica spĺňa aj podmienku $g_1(1) = g_2(1) = 1$. Po vyriešení dostaneme parameter k a cenu optimalizovaných operácií C'_D a C'_E :

$$k = \sqrt{\frac{C_D}{C_E}}, C'_D = \frac{\sqrt{C_D * C_E}}{2} + \frac{C_D}{2}, C'_E = \frac{\sqrt{C_D * C_E}}{2} + \frac{C_E}{2}$$

Z tohto nám vyplýva že optimálna šírka kýblika pri zmene šírky je :

$$B^*_W = \sqrt{\frac{C_D}{C_E}} * B_W$$

Je možné jednoducho overiť, že túto hodnotu premennej k je možné použiť aj pri znižovaní šírky kýblika. V tomto prípade sa nám mení predpoklad, že $g_1(k)$ a $g_2(k)$ sú väčšie rovné ako jedna na predpoklad, že sú menšie rovné ako jedna. Keďže najhoršie a najlepšie prípady zostávajú rovnaké, tak dochádza k tomu že medze sú rovnaké, a preto aj hodnota parametra k zostáva rovnaká.

SNOOPy triggering process

Keďže zmena šírky kýblikov závisí na C_D a C_E, tak je nutné ich vedieť určiť. Najprv si definujeme pojmy Slot, $C_{D,1}$, $C_{E,1}$, Éra, $C_{D,n}$ a $C_{E,n}$.

Slot je časový interval predstavujúci N_B operáciám vloženia alebo odobrania prvku, nie však ich kombinácia. Čiže ide buď o operácie odobrania udalosti, alebo čisto o operácie vloženia udalosti, ktorých je toľko, aký je počet kýblikov.

$C_{D,1}$ je priemerná cena operácie vybrania prvku získaná spriemerovaním operácií cez jeden Slot. Zapamätaná hodnota $C_{D,1}$ sa neprenáša medzi Slotmi. Každý Slot určuje novú hodnotu $C_{D,1}$ na základe operácií vybrania prvku, ktoré sa uskutočnili v príslušnom Slote.

$C_{E,1}$ je priemerná cena operácie vloženia prvku získaná spriemerovaním operácií cez jeden Slot. Má podobné vlastnosti ako $C_{D,1}$.

Éra je časový interval medzi dvoma po sebe idúcimi operáciami zmeny šírky kýblikov.

$C_{D,n}$ je kĺzavý priemer n po sebe idúcich hodnôt $C_{D,1}$. Keď éra začne, tak vypočíta po sebe n idúcich $C_{D,1}$ do prvého $C_{D,n}$. Každý ďalší $C_{D,1}$ v danej ére nahrádza najstarší a tvorí kĺzavý priemer. Keď éra trvá kratšie ako n $C_{D,1}$, tak $C_{D,n}$ bude 0 po celý čas danej éry.

$C_{E,n}$ je kĺzavý priemer n po sebe idúcich hodnôt $C_{E,1}$ získaných v danej ére. Má rovnaké vlastnosti ako $C_{D,n}$.

V prípade dynamic calendar queue sledujeme iba $C_{D,1}$ a $C_{E,1}$. SNOOPy calendar queue sleduje okrem $C_{D,1}$ a $C_{E,1}$ aj $C_{D,10}$ a $C_{E,10}$. SNOOPy calendar queue používa základné

mechanizmy od calendar queue ako aj dynamic calendar queue. Okrem toho pridáva ďalšie dva mechanizmy a to :

$$C_{E,10} \geq 2 \times C_{D,10} \text{ alebo } C_{D,10} \geq 2 \times C_{E,10}$$

Toto znamená že keď sa cenové funkcie pre pridanie a odobranie udalosti líšia aspoň dva-krát, tak dôjde k zmene šírky kýblika. V tomto prípade ide o vyváženie cenových funkcií C_E a C_D. Inými slovami ide o dosiahnutie stavu $C_E = C_D$ s určitou toleranciou (tolerancia dva-krát).

SNOOPy calendar queue dosahuje lepšie výsledky ako dynamic calendar queue. Aj napriek tomu, že obsahuje viac mechanizmov na spustenie zmeny šírky kýblika, aj tak v priemere dochádza k zmene menej často. To je spôsobené tým, že dokáže lepšie určiť správne hodnoty šírky kýblika.

Časová zložitosť

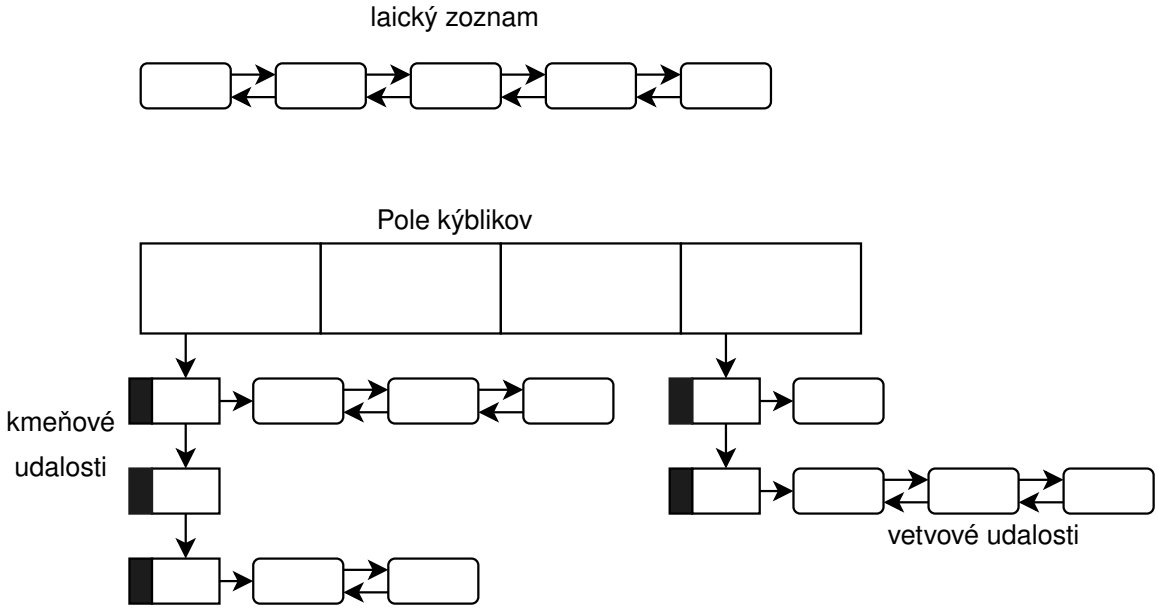
Výber prvku - $O(1)$.
Vloženie prvku - $O(1)$.

Tieto časové zložitosti fungujú rovnako ako v calendar queue. SNOOPy calendar queue rieši len voľbu správnej šírky kýblikov pri nerovnomernom rozložení udalostí. Dosahuje lepšie výsledky ako dynamic calendar queue, a tým pádom menej často dochádza k zmene veľkosti kýblikov a následnému presúvaniu udalostí so zložitosťou $O(n)$.

2.6 Sluggish Calendar Queue

Sluggish calendar queue [2] je prioritná fronta zameraná na simulácie komunikačných sietí. Pozostáva z dvoch častí a to z laického zoznamu udalostí (anglicky layman event list) a poľa kýblikov. Laický zoznam je obojsmerne viazaný zoznam, ktorý ukladá usporiadané udalosti. Udalosti v laickom zozname nazývame laické. Pole kýblikov je podobné ako u bežného calendar queue. Rozdiel od bežného kalendára je v tom, že udalosti v kýbliku sú uložené v obojsmernom zozname. Tento zoznam nazývame kmeňovým zoznamom (anglicky trunk list) a udalosti v ňom nazývame kmeňové udalosti. Na rozdiel od bežného calendar queue obsahujú kmeňové udalosti ukazovateľ na ďalší obojsmerný zoznam, ktorý sa nazýva vetvový zoznam (anglicky branch list), udalosť v ňom je zasa vetvová udalosť. Udalosti vo vetvovom zozname sú taktiež usporiadané, ale nemusia nutne spadať do kýblika, v ktorom je príslušná koreňová udalosť. Časová značka kmeňovej udalosti nesmie byť väčšia ako akejkoľvek udalosti z jeho vetvového zoznamu. Grafické znázornenie môžme vidieť na obrázku 2.2.

Pridanie udalosti najprv skontroluje či je možné pridať túto udalosť do laického zoznamu. Máme parameter α, ktorý udáva koľko udalostí z laického zoznamu prezeráme pri hľadaní správnej pozície. Skúšame pridať udalosť od zadnej strany zoznamu, ale kontrolujeme len posledných α udalostí. Ak nájdeme udalosť, ktorá má menšiu časovú značku alebo rovnakú, tak za ňu vložíme vkladanú udalosť. Ak v posledných α udalostiach nenájdeme zhodu, tak skúsime ešte kontrolovať odpredu. V tomto prípade skontrolujeme prvých α udalostí a ak nájdeme udalosť s väčšou časovou značkou, tak vložíme danú udalosť pred ňu. Ak je zoznam kratší ako 2α udalostí, tak nájdeme presné miesto prezeraním všetkých udalostí. Ak je kratší, tak sa použije vyššie zmienený postup. Na toto si zoznam ukladá počet udalostí v sebe uložených.



Obr. 2.2: Grafické znázornenie implementácie sluggish calendar queue

Je možné, že udalosť nebude možné vložiť na základe posledných a prvých α udalostí. V takom prípade treba presunúť laický zoznam do pola kýblikov. To sa vykonáva tak, že prvú udalosť v laickom zozname zmením na kmeňovú a vložím ju na príslušné miesto metódou ako základný calendar queue. Zvyšné udalosti sa stávajú vetvovým zoznamom príslušnej kmeňovej udalosti. Po prenesení udalostí z laického zoznamu do pola kýblikov vložíme novú udalosť do prázdneho laického zoznamu. Po vložení kmeňovej udalosti treba spraviť kontrolu, či netreba zmeniť veľkosť kýblikov.

Vybratie prvej udalosti môže zanedbať vetvové zoznamy, lebo žiadna udalosť nemôže byť skôr, ako príslušná kmeňová udalosť. V prvom momente zistíme udalosť s najmenšou časovou značkou z pola kýblikov. Následne musíme porovnať túto udalosť s prvou udalosťou v laickom zozname. Ak udalosť v laickom zozname má menšiu časovú značku, tak vrátime udalosť z laického zoznamu. Ak tomu tak nieje, tak prvou vrátenou udalosťou bude kmeňová udalosť z pola kýblikov. V tomto prípade je nutné spracovať jej vetvový zoznam. To vykonáme tak, že prvú udalosť z vetvového zoznamu vložíme do pola kýblikov ako kmeňovú udalosť a zvyšok vetvového zoznamu pridelíme ako vetvový zoznam novej kmeňovej udalosti. Pri tomto je nutné zachovať relatívne poradie udalostí s rovnakým časom. Pre tento prípad si zaznamenávame unikátny identifikátor, ktorého usporiadanie určuje v akom poradí boli príslušné laické zoznamy vkladané do pola kýblikov. Vďaka tomuto identifikátoru je možné zabezpečiť stabilitu, ktorá hovorí, že udalosti s rovnakým časom majú byť vykonané v takom poradí, v akom boli vytvorené.

Odstránenie udalosti pozostáva z ukazovateľa na udalosť, ktorú treba odstrániť. V tomto prípade je odstránenie veľmi jednoduché, ak je udalosť vo vetvovom alebo laickom zozname. V takom prípade dôjde jednoducho k jej odstráneniu z príslušného zoznamu. Ak je daná udalosť kmeňová, postup je identický ako pri vyberaní prvého prvku. To pozostáva spracovaním vetvového zoznamu príslušnej kmeňovej udalosti. To znamená, Že prvý prvok vetvového zoznamu sa stáva kmeňovým, a je vložený do pola kýblikov na príslušné miesto.

Zmena veľkosti je veľmi podobná tradičnému calendar queue. Táto implementácia viacej toleruje prázdne kýbliky pri vyberaní prvkov, lebo postupne pridáva nové udalosti z vet-

vových zoznamov pri odstraňovaní kmeňových udalostí. Preto operácia zmenšenia sa uskutočňuje až keď počet udalostí je menší ako 1/16 príslušného počtu kýblikov, na rozdiel od 1/2 z tradičného calendar queue. Zároveň je nutné meniť šírku kýblikov, keď počet udalostí v jednom kýbliku presiahne určitú hranicu Θ. V tejto implementácií je Θ konštanta rovná 100, čo predíde skorej zmene veľkosti pri začiatku, keď nieje ľahké presne odhadnúť rozloženie udalostí a tým pádom ani šírku kýblika. Určenie veľkosti kýblika sa vykoná tak, že skontrolujeme najviac naplnený kýblik, zistíme najväčšiu časovú značku T_{min} a najmenšiu časovú značku T_{min}. Keďže zoznamy sú obojsmerné, tak táto operácia je v konštantnom čase. Novú šírku kýblika určíme :

$$B_W = \frac{4(T_{max} - T_{min})}{min(N_B, \widetilde{n})}$$

Kde B_W je šírka kýblika, N_B je počet kýblikov a $\widetilde{n}$ je počet udalostí z rôznou časovou značkou v najviac naplnenom kýbliku.

Časová zložitosť

Výber prvku - $O(1)$.
Vloženie prvku - $O(1)$.

Keďže sluggish calendar queue používa calendar queue, tak je taktiež ovplyvnený vlastnosťami calendar queue. Na rozdiel od calendar queue sa lepšie vysporadúva s nerovnomerným rozložením udalostí, keďže obsahuje menší počet záznamov v calendar queue štruktúre. Menej často dochádza k pribúdaniu aj ubúdaniu udalostí v calendar queue, a preto menej často dochádza k zmene veľkosti.

2.7 FlexQueue

FlexQueue [11] je opäť inšpirovaná tradičným algoritmom calendar queue, viď obrázok 2.1. Okrem základného pola kýblikov pridáva ešte hierarchický bitový vektor. Tento vektor obsahuje toľko bitov, koľko je kýblikov. Pre každý kýblik uchováva informáciu o tom, či obsahuje nejaké prvky, alebo je prázdny. Bitová hodnota 1 hovorí, že nieje prázdny a hodnota 0 hovorí, že je prázdny. Toto umožňuje nájdenie prvého neprázdneho kýblika na základe bitových operácií, čo je rýchlejšie ako prechádzanie pola kýblikov prvok po prvku. Pre toto je nutné ukladať iba udalosti neprevyšujúce aktuálny rok. Všetky udalosti prevyšujúce aktuálny rok sú uložené v samostatnom úložisku. FlexQueue taktiež používa iný spôsob na určovanie zmeny šírky kýblikov.

Pole kýblikov je v tomto algoritme statické a nemení veľkosť. Jediný parameter ktorý táto implementácia mení je šírka kýblika B_W. Veľkosť tohoto pola je N kýblikov, tak, že N je mocninou čísla dva. Ďalej obsahuje vektor N bitov. Táto implementácia používa dva vektory a dve polia kýblikov. Toto sa uskutočňuje z dvoch dôvodov. Prvým je aby nebolo nutné robiť zmenu veľkosti na poli, ktoré obsahuje udalosti a tie následne rozmiestňovať do príslušných kýblikov. Druhý dôvod je zabezpečenie že prvý jednotkový bit v poli ukazuje vždy na prvý neprázdny kýblik, čomu by tak nebolo, keby pole bolo použité cyklicky. Následne je nutné implementovať úložisko udalostí, ktoré prevyšujú aktuálne pole kýblikov.

Aktuálny priemer je μ, smerodajná odchýlka je σ a nová udalosť je Δ časových jednotiek vzdialená, tak nové odhadnuté hodnoty sú:

$$\mu' = \alpha\mu + (1-\alpha)\Delta \quad , \quad \sigma' = \beta\sigma + (1-\beta)|\Delta - \mu|$$

kde α a β sú konštanty, ktoré by mali byť čísla čo majú v menovateli mocninu čísla dva. Pre túto implementáciu sú $\alpha = 0,875 = \frac{7}{8}$ a $\beta = 0,25 = \frac{1}{4}$.

Novú šírku určíme tak, že určíme hranice aktuálneho horizontu, minimálnu hranicu ako $max(\mu' - k\sigma', 0)$ a maximálnu hranicu ako $\mu' + k\sigma'$. Šírka jedného kýblika je rozdelenie celého horizontu na N rovnakých častí, keďže horizont je pole kýblikov. Táto hodnota je zaokrúhlená na celé číslo a maximum horizontu je posunutý ako N-krát minimum. Zmena veľkosti sa vykonáva vždy, keď sa aktuálne používané pole kýblikov vyprázdni. Vždy striedame prvé a druhé pole periodicky, a každé pole môže obsahovať inú šírku kýblikov.

Vkladanie prvku je veľmi podobné klasickej calendar queue. Najprv kontroluje, či sa udalosť nachádza v aktuálnom horizonte alebo nie. Ak sa tam nenachádza, tak vkladá udalosť do úložiska na udalosti, ktoré nie sú v horizonte. Ak sa tam nachádza, tak vloží udalosť do správneho kýblika na základe intervalu. Následne prepočíta priemer a smerodajnú odchýlku.

Pri vyberaní položky kontroluje obe polia kýblikov aj úložisko ostatných udalostí. Ak sa udalosť nachádza v jednom z dvoch polí, tak je odstránená z kýblika a ak kýblik neobsahuje žiadne udalosti nastavíme príslušnú hodnotu v bitovom poli na 0. Ak sa nachádzame v jednom poli kýblikov, a prvá udalosť sa nachádza v ďalšom poli, to znamená, že prechádzame medzi poliami a prvé pole je prázdne. V takom prípade dochádza u neho k zmene šírky kýblikov.

Tento typ implementácie sa snaží predísť alokácie pamäte ako aj operáciám s pohyblivou rádovou čiarkou. Spolieha sa na rýchle určenie správnych udalostí na základe bitových operácií a predpokladá, že väčšina udalostí spadá do oblasti jedného horizontu.

Časová zložitosť

Výber prvku - $O(1)$.
Vloženie prvku - $O(n)$.

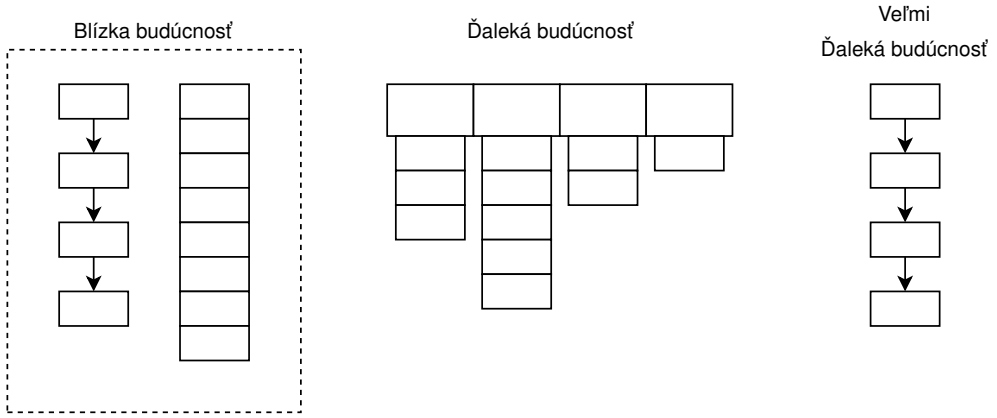
Keďže flex queue používa princíp calendar queue, ale bez zmeny veľkosti, tak dochádza k zhoršeniu zložitosti na vloženie prvku. Pri správnej voľbe počtu kýblikov je ale možné dosiahnuť lepšie výsledky ako calendar queue, keďže nedochádza k presúvaniu udalostí pri zmene veľkosti. Táto fronta môže byť vhodná pre simulácie, kde poznáme aké budú počty udalostí vo fronte.

2.8 Lazy Queue

Lazy queue [8] vychádza z myšlienky dvoj-zoznamovej štruktúry, ktorá delí udalosti na blízku budúcnosť a ďalekú budúcnosť. Blízka budúcnosť je v zoradenom zozname, kde vzdialená je nezoradený zoznam. Grafické znázornenie je možné vidieť na obrázku 2.3.

Tento spôsob rozvrhnutia udalostí je inšpirovaný tým, ako ľudia hodnotia nasledujúce udalosti. Môžme si udalosti rozvrhnúť do troch skupín. Prvá skupina udalostí je blízka budúcnosť. Tieto udalosti sú známe, a preto máme presný časový rozvrh a poradie. Druhá skupina je vzdialená budúcnosť, pri ktorej očakávame, že veľké množstvo udalostí je ešte neznáme. Preto si nerobíme ešte detailnejší rozvrh. Treťou skupinou je veľmi vzdialená budúcnosť. Udalosti spadajúce do tejto skupiny môžu byť uložené niekde inde, a nemusia byť veľmi brané do úvahy. S pohybom času sa pohybujú aj hranice medzi blízkou, vzdialenou a veľmi vzdialenou budúcnosťou.

V implementácií je blízka budúcnosť reprezentovaná usporiadaným polom elementov a jednosmerným zoznamom. Vzdialená budúcnosť je reprezentovaná ako niekoľko mesiacov,



Obr. 2.3: Grafické znázornenie implementácie lazy queue

kde každý mesiac obsahuje nezoradené pole udalostí. Veľmi vzdialená budúcnosť je implementovaná ako jednosmerný zoznam. na prechod mesiaca z ďalekej budúcnosti do blízkej sa používa algoritmus quicksort.

Pridanie prvku pozostáva z určenia budúcnosti, do ktorej prvok spadá. Predpoklad tejto implementácie je, že väčšina prvkov spadá do ďalekej budúcnosti. Ak je to tak, potom dôjde k identifikovaniu mesiaca a vloženiu tejto udalosti do pola v príslušnom mesiaci. Ak prvok spadá do blízkej budúcnosti, tak je pridaný do zoznamu blízkej budúcnosti. Tento zoznam slúži výhradne na vkladanie udalostí do blízkej budúcnosti.

Odoberanie prvku je realizované tak, že vráti prvý prvok v blízkej budúcnosti. Ak sa v blízkej budúcnosti nenachádza žiaden prvok, tak usporiadame najbližší mesiac a stane sa z neho blízka budúcnosť. Po tejto operácií je nutné zmeniť medze medzi blízkou, ďalekou a veľmi ďalekou budúcnosťou. Týmto spôsobom je usporiadanie odložené a vykonáva sa len na malých častiach kalendára udalostí. Z tejto vlastnosti vychádza meno "lenivá fronta"(anglicky lazy queue).

Na dosiahnutie dobrého výkonu je nutné správne určiť počet mesiacov a ich dĺžku. Dĺžka mesiaca sa počíta tak, aby do jedného mesiaca spadal konštantný počet udalostí a dĺžka každého mesiaca je rovnaká. Tento konštantný počet je reprezentovaný premennou EM, ktorá udáva predpokladaný počet udalostí v mesiaci. Aktuálny priemerný počet udalostí na mesiac je zasa reprezentovaný premennou AEM. Počet mesiacov je určený tak, že len malá časť udalostí spadá do veľmi vzdialenej budúcnosti. Pre efektívne fungovanie je preto nutné zaviesť mechanizmus na zmenu veľkosti a počtu mesiacov.

Na zmenu veľkosti mesiacov je nutných päť parametrov:

1. Počet udalostí v zozname pre blízku budúcnosť NF, $NF \in [0, MAXNF]$
2. Počet udalostí nachádzajúcich sa vo veľmi vzdialenej budúcnosti VFF, $VFF \in [0, MAXVFF]$
3. Aktuálny priemerný počet udalostí na mesiac AEM, $AEM \in [LOWER, UPPER]$
4. Počet prázdnych alebo skoro prázdnych mesiacov v hornej polovici ďalekej budúcnosti FF, $FF + VFF > MAXVFF/2$
5. Šírka a výška špičiek nachádzajúcich sa v rozložení udalostí, $KFF < PeakTreshold$ kde KFF je počet udalostí v prvých K mesiacoch a $PeakTreshold$ je konštantné číslo.

Podmienky plynúce z parametrov 1-4 sú tvrdé obmedzenia, ktoré nesmú byť nikdy porušené. Podmienka 5 spôsobí zmenu veľkosti len keď ostatné podmienky budú dodržané po zmene veľkosti. Vďaka podmienke 5 dokážeme odhaliť špičky v blízkosti blízkej budúcnosti a rozprestrieť tieto udalosti do okolitých mesiacov znížením dĺžky mesiaca na polovicu. Toto znižuje zbytočnú zmenu veľkosti spustenú pravidlom číslo 1. Hranice parametrov 3-5 môžu byť nastavené pomerne naširoko, Čo pomôže znížiť počet operácií zmeny veľkosti. Porušenie pravidla 4 by nemalo viesť na zníženie počtu mesiacov na polovicu, a preto sa v tomto prípade vykonáva menšia zmena počtu.

Aplikácia zmeny veľkosti

Zmeny veľkosti sú kontrolované vždy len pri operáciach vybratia a vloženia prvku. Keď pri vyberaní prvku zistíme že blízka budúcnosť je prázdna, tak skontrolujeme ďalekú budúcnosť na prvý neprázdny mesiac. Pred usporiadaním tohto mesiaca a jeho presunutím do blízkej budúcnosti dôjde ku kontrole kritérií 3-5. Pri zistení potreby dôjde k samotnej zmene veľkosti, ale tak, aby sa neporušili podmienky 1 alebo 2.

Pri vkladaní prvku, ktorý spadá do blízkej alebo veľmi ďalekej budúcnosti robíme kontrolu pravidiel 1 alebo 2. Príslušné operácie sú vykonané tak, aby boli splnené podmienky 3 a 4. Pri fáze zostrojovania, keď ešte nebol vybraný žiaden prvok kontrolujeme taktiež podmienky 3 a 4.

Operácie zmeny veľkosti

Na zmenu veľkosti používame štyri operácie :

1. Zmenšenie počtu mesiacov na polovicu
2. Zväčšenie počtu mesiacov na dvojnásobok
3. Zmenšenie dĺžky mesiacov na polovicu
4. Zväčšenie dĺžky mesiacov na dvojnásobok

Tieto operácie sa často musia kombinovať, napríklad pri zmenšení dĺžky na polovicu je niekedy nutné zdvojnásobiť počet mesiacov.

Zmenšovanie dĺžky mesiacov na polovicu sa vykonáva tak, že usporiadame každý mesiac. Následne skontrolujeme, či je nutné zdvojnásobiť počet mesiacov (či udalosti v hornej polovici ďalekej budúcnosti je možné vložiť do veľmi ďalekej budúcnosti) a následne zmenu veľkosti mesiaca a preusporiadanie udalostí do nových mesiacov.

Zväčšovanie dĺžky mesiacov sa vykonáva tak, že po pároch sú spojené mesiace dokopy a následne sú presunuté do spodnej polovice pola mesiacov. Ak počet udalostí v hornej polovici ďalekej budúcnosti je dosť malý, tak sú tieto udalosti presunuté do veľmi ďalekej budúcnosti a potom je zmenšený počet mesiacov na polovicu.

Lazy queue na rozdiel od calendar queue lepšie zvláda zmeny v rozložení udalostí. Taktiež zmeny veľkosti štruktúry nastávajú menej často. Táto štruktúra je ale nevhodná, ak veľké množstvo nových udalostí spadá do blízkej alebo veľmi ďalekej udalosti.

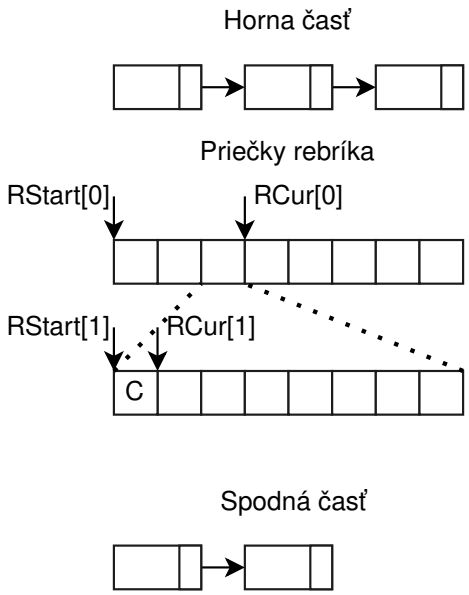
Časová zložitosť

Výber prvku - $O(1)$.
Vloženie prvku - $O(1)$.

Konštantná zložitosť je len za predpokladu, že sú udalosti vkladané do ďalekej budúcnosti, a že je správne zvolená šírka mesiacov a ich počet. Taktiež, ak dochádza k zmene veľkosti, tak dochádza k presúvaniu udalostí. Keďže veľkosť sa mení vždy v násobkoch čísla dva, dochádza k nie vždy presnému nastaveniu veľkosti pri aktuálnych podmienkach. Niekedy by bolo vhodnejšie nastavovať veľkosti v menších krokoch.

2.9 Ladder Queue

Ladder queue [10] je inšpirovaná z calendar queue a obsahuje miernu podobnosť štruktúre Lazy queue. Štruktúra sa skladá z troch častí, a to z hornej časti, priečok rebríka a spodnej časti. Spodná časť je usporiadaný zoznam udalostí. Obsahuje udalosti, ktoré sa majú vykonať najskôr. Horná časť obsahuje neusporiadané udalosti, ktoré sa vykonávajú neskôr ako je určitá hranica. Priečky rebríka sú štruktúra obsahujúca prvky podobné kýblikom v calendar queue. Narozdiel od calendar queue môžu rôzne priečky obsahovať rôzne šírky kýblikov. Grafické znázornenie je možné vidieť na obrázku 2.4.



Obr. 2.4: Grafické znázornenie implementácie ladder queue

Premenné, ktoré Ladder queue používa sú :

- $MaxTS$, najvyššia časová značka zo všetkých udalostí v hornej časti. Hodnota sa aktualizuje vždy, keď je nová udalosť vložená do hornej časti.
- $MinTS$, najnižšia časová značka zo všetkých udalostí v hornej časti. Hodnota sa aktualizuje vždy, keď je nová udalosť vložená do hornej časti.
- N_{TOP}, počet udalostí v hornej časti. Hodnota sa inkrementuje s príchodom novej udalosti do hornej časti.
- T_{Start}, hraničná hodnota na vloženie udalostí do hornej časti. Udalosť je vložená do hornej časti, len keď jej časová značka je vyššia alebo rovná ako je táto hodnota, inak je vložená do priečok alebo spodnej časti.

- $B_W[i]$, šírka kýblikov pre i-tu priečku v rebríkovej časti.
- N_{B_c}, počet udalostí v kýbliku, ktorý reprezentuje najbližší časový interval v rebríkovej časti.
- $N_{Bucket}[j,k]$, počet udalostí v k-tom kýbliku j-tej priečky.
- N_{Rung}, počet aktívne používaných priečok v rebríkovej časti.
- $R_{Cur}[x]$, hraničná hodnota na vloženie udalostí do príslušnej priečky. Táto hodnota je podobná ako T_{Start}, ale pre príslušnú priečku.
- $R_{Start}[x]$, hodnota začiatku intervalu pre danú priečku. Koncovú hodnotu intervalu reprezentuje šírka kýblika vynásobená počtom kýblikov, alebo začiatok vyššej priečky rebríka, prípadne začiatok hornej časti pre najvyššiu priečku rebríka.
- $THRES$, hodnota, ktorá reprezentuje hranicu pre vytváranie nových priečok.
- N_{Bot}, počet udalostí v spodnej časti.

Výber prvého prvku pozostáva z viacerých variant na základe toho, ktoré z troch častí sú neprázdne. Ak jediná neprázdna časť je horná časť, tak pri výbere prvku vytvoríme novú priečku rebríka a presunieme všetky udalosti z hornej časti do tejto priečky. Šírku jednotlivých kýblikov v tejto priečke sa určí ako :

$$B_W[i] = \frac{MaxTS - MinTS}{N_{TOP}}$$

Následne nastavíme parametre $R_{Start}[1] = R_{Cur}[1] = MinTS$ a $T_{Start} = MaxTS + B_W[1]$.

Po určení šírky sú udalosti presunuté z hornej časti do príslušných kýblikov v priečke. Index správneho kýblika sa urči podla vzťahu :

$$B_{index} = \left\lfloor \frac{TS - R_{Start[i]}}{B_W[i]} \right\rfloor$$

Kde TS predstavuje časovú značku aktuálne presúvanej udalosti. Udalosti sú do kýblika pridávané neusporiadané. Táto šírka kýblikov sa snaží zabezpečiť, že priemerne je v každom kýbliku jedna udalosť.

Keď presunieme udalosti z hornej časti do priečky, nasleduje prechádzanie priečky od najmenšieho kýblika, kde sa hľadá prvý neprázdny kýblik. Tento kýblik je skontrolovaný, či neobsahuje viac udalostí ako je hodnota $THRES$. Keď neobsahuje, tak je tento kýblik označený ako B_c a jeho udalosti sú usporiadané a presunuté do spodnej časti. Ak tento kýblik obsahuje viac udalostí ako je $THRES$, tak je z tohto kýblika vytvorená nová priečka rebríka. Toto sa uskutoční podobne ako pri transformácií hornej časti do priečky. Šírka kýblika sa určí na základe :

$$B_W[i+1] = \frac{B_W[i]}{THRES} \tag{2.1}$$

Toto predpokladá rovnomerné rozloženie udalostí v danom kýbliku. Ďalej sa nastavia parametre $R_{Start}[i+1] = R_{Cur}[i+1] = R_{Cur}[i]$ a následne sa nastaví $R_{Cur}[i] = R_{Cur}[i] + B_W[i]$. Následne začne prechádzanie novo vzniknutej priečky opäť od začiatku po prvý neprázdny kýblik. Tu sa následne opäť kontroluje na počet udalostí menší ako $THRES$. Táto generácia sa môže opakovať viackrát,až po kým nie je splnená podmienka počtu udalostí v kýbliku predstavujúcom najskorší interval a jeho udalosti presunuté do spodnej časti.

Po presunutí udalostí do spodnej časti máme usporiadaný zoznam udalostí. V tomto zozname môžme jednoducho vrátiť udalosť na čele zoznamu ako výsledok operácie vybratia prvku. Nasledujúce udalosti vybratia opäť kontrolujú neprázdnosť spodnej časti, a v prípade neprázdnosti vracajú prvky zo spodnej časti. V prípade prázdnej spodnej časti nastáva prechod rebríka od spodnej časti, keďže spodné priečky obsahujú vždy skoršie časy ako vyššie priečky. Po vyprázdnení priečky je daná priečka zmazaná a algoritmus sa presúva na vyššiu priečku.

Vloženie prvku prechádza najprv hornú časť a kontroluje, či udalosť obsahuje časovú značku, ktorá je vyššia alebo rovná ako T_{Start}. Ak je vyššia alebo rovná, tak je táto udalosť vložená do hornej časti. Ak je menšia, tak prechádza po rebríku od horných priečok a kontroluje, či je časová značka vyššia alebo rovná ako $R_{Cur}[i]$, kde i je index aktuálnej priečky. Ak nevloží udalosť do hornej časti ani do žiadnej z priečok, tak je následne udalosť vložená do spodnej časti. Ak je počet udalostí v spodnej časti väčší ako je hodnota $THRES$, tak nastáva vytvorenie novej priečky z udalostí v spodnej časti. Šírka sa určuje na základe rovnice 2.1. V novej priečke je následne nájdený prvý neprázdny kýblik, je označený ako B_c a sú z neho premiestnené udalosti do spodnej časti. Týmto mechanizmom sa zabezpečí, že netreba prechádzať veľa udalostí v spodnej časti pri vkladaní nových udalostí do krátkej budúcnosti.

Táto implementácia nemusí na rozdiel od calendar queue často presúvať veľké množstvo udalostí, keďže každé generovanie priečok vždy presúva len časť udalostí a nie celý kalendár udalostí ako je to pri zmene veľkosti u calendar queue.

Časová zložitosť

Výber prvku - $O(1)$.
Vloženie prvku - $O(1)$.

Ladder queue dosahuje dobré výsledky počas vkladania udalostí. Toto vloženie nemusí byť plne konštantné, ale je závislé na počte priečok a umiestnenia udalostí. Najrýchlejšie vkladanie udalostí je v prípade, že udalosti spadajú do vrchnej časti. V spodnej časti je vloženie udalosti $O(n)$, ale keďže počet udalostí v spodnej časti je obmedzený, tak to nie je prekážkou. Náročné operácie sú tvorba nových priečok a hlavne výber prvku pri prázdnej spodnej časti aj priečkach rebríka. V tomto prípade dochádza k presunu všetkých udalostí z hornej časti do priečky rebríka a následné prípadné zjemňovanie priečok až po tvorbu novej spodnej časti. Táto operácia má zložitosť $O(n)$, keďže dochádza presunu všetkých udalostí v štruktúre.

2.10 Far Future Event Leaf Tree

Far Future Event Leaf Tree (ďalej označované ako FELT) [4] je implementácia, ktorá k primárnej calendar queue pridáva sekundárnu FELT štruktúru. Týmto spôsobom sa snaží obmedziť udalosti uložené v calendar queue. Preto udalosti, ktoré sa nachádzajú veľmi ďaleko v budúcnosti ukladá do samostatnej sekundárnej štruktúry.

Táto implementácia obsahuje mierne prispôsobenú implementáciu calendar queue. Túto calendar queue využíva ako primárnu štruktúru na ukladanie udalostí. Ako sekundárnu štruktúru využíva FELT, ktorý používa podľa potreby. Túto potrebu sleduje na základe trendu medzi príchodom a odoberaním udalostí. Tento trend môže byť buď rastúci, klesajúci alebo stabilný. Štruktúra preto musí byť schopná sledovať tento trend.

Trend sleduje na základe pohyblivého okna počet udalostí vo fronte. Na sledovanie sa použije kruhový buffer o veľkosti N_C prvkov. Do neho ukladá vždy aktuálny stav počtu udalostí vo fronte. Tento presun vždy vykonáva po N_O počte operácií výberu prvku, respektíve operácií vloženia prvku. Na základe sledovania bufferu dokáže odhaliť stav trendu. Ak sú počty udalostí plne rastúce, tak je rastúci trend fronty. Ak sú naopak plne klesajúce, tak je klesajúci trend fronty. Ak nie je trend ani rastúci ani klesajúci, musí skontrolovať, či počty udalostí nepresahujú statické hranice na zmenu veľkosti odvodené od aktuálneho počtu udalostí. Ak hranice nepresahujú, tak trend je stabilný. Ak trend nie je ani rastúci, ani klesajúci a dokonca ani stabilný, tak to znamená, že fronta nie je schopná určiť aktuálny trend.

Na začiatku použitia funguje len primárna štruktúra a sekundárna FELT fronta sa nepodieľa na ukladaní udalostí. Na zistenie nutnosti práce zo sekundárnou FELT štruktúrou sa používajú dve hranice a to $LOWBOUND$ a $UPPBOUND$.

Pri rastúcom trende a počte udalostí v primárnej calendar queue presahujúcom hodnotu $UPPBOUND$ dôjde k presunutiu udalostí do sekundárnej FELT štruktúry. Ak nie je FELT štruktúra inicializovaná, dôjde v tomto bode k jej inicializácií. Presun pozostáva v zachovaní prvých $2 \times LOWBOUND$ udalostí v calendar queue a presunutí ostatných udalostí do FELT.

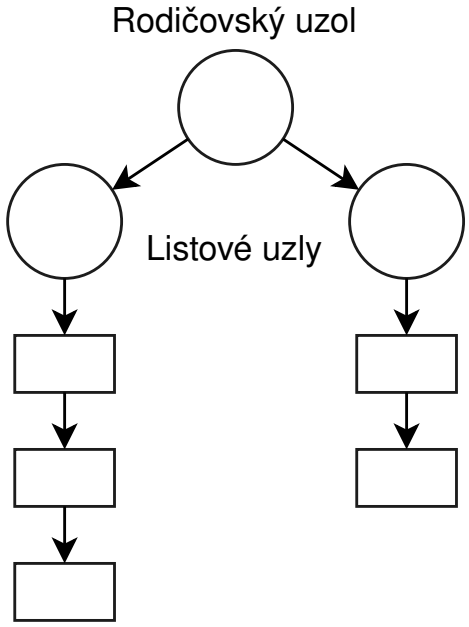
Keď počet udalostí v calendar queue dosiahne hodnotu menšiu ako je $LOWBOUND$, dôjde k presunu udalostí z FELT do calendar queue. Situácia nastane len ak FELT už existuje, čo znamená, že nie pokiaľ nedošlo k skoršiemu presunu do FELT. Presun do FELT môže byť dvojakého typu. Toto rozhodnutie sa vykonáva na základe trendu.

Ak je trend stabilný, dôjde k presunu všetkých udalostí z FELT do calendar queue, a následnému zničeniu FELT štruktúry. V tomto bode môže byť počet udalostí v calendar queue väčší ako $UPPBOUND$. Tento presun pozostáva v prenesení najľavejšieho listu a zväčšenia rozmeru kalendára tak, aby sa doň zmestili všetky udalosti a následného prenesenia všetkých udalostí postupne zľava. Ak trend nie je stabilný, tak sa vykoná prenesenie len najľavejšieho listu.

Štruktúra FELT pozostáva z binárneho stromu. Každý listový uzol obsahuje neusporiadaný zoznam udalostí, viď obrázok 2.5. Žiaden uzol nesmie obsahovať viac udalostí ako je určitý maximálny limit. Tento limit môže byť maximálne hodnota $UPPBOUND - LOWBOUND$. Pri presiahnutí tohto limitu vytvorí listový uzol nové listové uzly a nastaví ich ako svojich následníkov. Následne vypočíta priemernú časovú značku z udalostí, ktoré sa v ňom nachádzali. Túto hodnotu použije na rozdelenie udalostí. Udalosti, ktoré majú časovú značku väčšiu ako je táto hodnota pridelí pravému následníkovi, ostatné pridelí ľavému následníkovi. Každý uzol obsahuje hodnotu *minNode*, ktorá reprezentuje najmenšiu časovú značku zo všetkých udalostí, ktoré sa nachádzajú v listových uzloch pod ním uložených.

Vloženie prvku do tejto štruktúry pozostáva z rekurzívneho volania funkcie na vloženie. Vloženie do listového uzla pozostáva vo vložení udalosti do neusporiadaného zoznamu a ak prekročí limit, vytvoria sa nové listové uzly a presunú sa tam jeho udalosti. Vloženie do nelistového uzla pozostáva v porovnaní časovej značky s hodnotou *minNode* pravého následníka. Ak je táto časová značka väčšia alebo rovná tejto hodnote, tak rekurzívne vykoná vloženie na pravého následníka, inak rekurzívne vykoná vloženie na ľavého následníka.

Táto implementácia dokáže zredukovať záťaž pre calendar queue a obmedziť maximálny počet udalostí pri rastúcom trende fronty. Toto riešenie dokáže znížiť počet operácií na zmenu veľkosti a zlepšiť fungovanie prioritnej fronty. Implementácia spomínaná v článku obsahuje parametre $UPPBOUND = 4096$, $LOWBOUND = 256$, maximálny počet udalostí v listovom uzle je 3840, $N_C = 20$, $N_O = 10$ *% z aktuálnej veľkosti fronty.*



Obr. 2.5: Grafické znázornenie implementácie FELT sekundárneho úložiska

Časová zložitosť

Výber prvku - $O(1)$.
Vloženie prvku - $O(1)$ pri vložení do calendar queue, $O(\log n)$ pri vložení do FELT štruktúry.

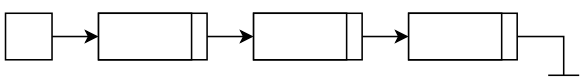
Tieto zložitosti sú dosiahnuté len pri väčšinovej práci štruktúry calendar queue. Práca zo štruktúrov FELT je logaritmická zložitosť $O(\log n)$. Štruktúra FELT mierne spomalí calendar queue a zabráni jeho expanzií pri rastúcom trende, ale zamedzí k presúvaniu veľkého množstva udalostí pri zmene veľkosti calendar queue. Naopak dochádza k presúvaniu menšieho množstva udalostí medzi štruktúrami calendar queue a FELT.

2.11 Ďalšie možné implementácie

Najjednoduchšia implementácia kalendára udalostí je pomocou lineárneho zoznamu, viď obrázok 2.6. Táto implementácia je pomerne intuitívna. Pri vkladaní udalostí prechádzame zoznam až kým nenájdeme miesto, kam je potrebné danú udalosť vložiť. Výber udalosti pozostáva vo výbere prvej položky v zozname. Implementácia je vhodná len pre malý počet udalostí [7].

Výber prvku - $O(1)$.
Vloženie prvku - $O(n)$.

Vloženie prvku v najhoršom prípade prechádza všetky udalosti uložené v zozname, až dôjde k porovnaniu s každou udalosťou a vloženie na koniec zoznamu. Výber prvku je veľmi rýchly, keďže dochádza vždy k výberu prvého prvku zo zoznamu.



Obr. 2.6: Grafické znázornenie implementácie pomocou zoznamu

Kalendár je možné implementovať aj pomocou niekoľkých stromových algoritmov, ktoré typicky dosahujú logaritmické zložitosti $O(\log n)$:

- heap
- pairing heap
- splay-tree
- a podobne

2.12 Analýza zložitosti operácií

Táto sekcia zahrňuje analýzu výpočtovej zložitosti pre všetky vyššie uvedené implementácie a porovnáva ich medzi sebou.

Priestorová zložitosť vyššie uvedených algoritmov je $O(n)$, keďže vždy je nutné uložiť samotnú udalosť, poprípade konštantný počet pomocných štruktúr na udalosť. V tomto prípade je dôležitejšia skôr časová zložitosť týchto implementácií. Časová zložitosť vloženia a výberu prvku je vždy uvedená na konci teórie k príslušnej implementácií. Výber prvku je pri každej implementácii $O(1)$, čiže konštantný. Vloženie prvku je taktiež u väčšiny implementácií $O(1)$, až na štandardný zoznam a flex queue, kde je zložitosť vloženia prvku $O(n)$ a stromových štruktúr, ktoré typicky dosahujú zložitosť $O(\log n)$.

Dôležité je ale podotknúť, že operácie na zmenu veľkosti, majú tiež svoje vlastné časové zložitosti. U calendar queue a štruktúrach na nej založených ide o presúvanie všetkých udalostí uložených v kalendári do nového pola. Preto táto operácia dosahuje lineárnu zložitosť $O(n)$. U flex queue dochádza len k zmene šírky kýblikov, a keďže sa táto zmena vykonáva nad prázdnym zoznamom, tak je táto operácia so zložitosťou $O(1)$.

Lazy queue vykonáva štyri rôzne operácie zmeny veľkosti. Zdvojnásobenie dĺžky alebo počtu mesiacov vyžaduje iba prenášanie udalostí medzi mesiacmi a veľmi ďalekou budúcnosťou, preto ide o lineárnu zložitosť $O(n)$. Aj zmena počtu mesiacov na polovicu vyžaduje iba prenesenie mesiacov vo vrchnej polovici ďalekej budúcnosti do veľmi ďalekej budúcnosti, čo je tiež zložitosť $O(n)$. Pri zmene dĺžky mesiaca na polovicu je okrem prenášania nutné vykonať aj usporiadanie mesiacov. Samotné usporiadanie využíva quicksort, ktorý má priemernú zložitosť $O(n \log n)$ a v najhoršom prípade $O(n^2)$. Preto je zložitosť tejto operácie $O(n + n \log n) = O(n(1 + \log n))$, čo je možné zjednodušiť na $O(n \log n)$. Takáto operácia závisí hlavne od zložitosti algoritmu usporiadania prvkov.

Ladder queue nevykonáva zmenu veľkosti, ale tiež presúva operácie pri začiatku etapy, a niektoré aj viackrát. Ale v priemere ide taktiež o lineárnu zložitosť $O(n)$. FELT Štruktúra obsahuje stromovú štruktúru, ktorej časová zložitosť vloženia prvku je $O(\log n)$. Prenášanie udalostí medzi calendar queue a FELT je zo zložitosťou $O(n \log n)$, keďže je nutné vykonať n vložení.

Pre ideálne fungovanie týchto implementácií je nutné, aby zmena veľkosti nevznikala často, respektíve len tak často, aby zlepšenie ceny operácií nad kalendárom prevyšovalo nad cenou operácie zmeny veľkosti.

Kapitola 3

Návrh knižnice

Táto kapitola popisuje návrh knižnice a obsahuje diagram tried. Knižnica je implementovaná v jazyku C++, ktorý je objektovo orientovaný. V sekcii 3.1 je popísaný diagram tried a význam jednotlivých tried pre knižnicu. Sekcia 3.1.1 obsahuje popis najdôležitejších metód pre knižnicu.

V návrhu som sa primárne zameral na varianty implementácie calendar queue. Všetky z implementácií popísaných v kapitole 2 sú buď inšpirované implementáciou calendar queue, alebo aspoň porovnateľne kvalitné v porovnaní s ňou, a taktiež sú založené na princípe multi-zoznamu. Návrh zabezpečuje jednoduchosť rozšírenia o ďalšie implementácie, a taktiež obsahuje možnosť výmeny implementácie za behu.

3.1 Diagram tried

Na obrázku 3.1 je možné vidieť diagram tried pre navrhnutú knižnicu. Najdôležitejšou triedou je `Pending_Event_Set`. Táto trieda predstavuje rozhranie medzi knižnicou a aplikáciou používajúcou túto knižnicu. Zároveň sú spolu s triedou `Event_Notice` jediné triedy dostupné z vonku knižnice. Všetky ostatné triedy sú interné pre knižnicu.

Trieda `Priority_Queue` je trieda, ktorá predstavuje rozhranie. Je to prioritná fronta, ktorá je implementovaná pomocou niektorej implementácie spomínanej v kapitole 2. Objekt triedy `Pending_Event_Set` môže obsahovať v jednom čase maximálne jednu implementáciu. Implementácia aktuálne sa nachádzajúca v príslušnom objekte je uložená v atribúte `queue_identifier`.

Príslušná implementácia obsahuje vždy objekty triedy `Event_Notice`. Tieto objekty by mali byť schopné zapuzdriť ľubovoľný objekt a sprístupniť prioritnej fronte jeho časovú značku. Prioritnú frontu ani nezaujíma obsah objektu, ak bude vždy schopný pristúpiť k časovej značke pre radenie. Časová značka je v tomto prípade dvojica (čas, priorita). Radenie prebieha primárne na základe času, a pri rovnakom čase rozhoduje priorita. Keď je rovnaký aj čas aj priorita, tak sa radí na základe mechanizmu FIFO.

Keďže implementácia SNOOPy calendar queue rozširuje implementáciu dynamic calendar queue a ten rozširuje obyčajný calendar queue, tak je tento vzťah vyjadrený dedičnosťou. Zároveň táto dedičnosť dovoľuje použiť rozšírenie sekundárneho FELT úložiska na ľubovolnú implementáciu calendar queue.

Každá z implementácií obsahuje príslušnú triedu. Triedy obsahujú potrebné parametre a tiež interné metódy. Sú to rôzne metódy na zmenu veľkosti a výpočet vnútorných para-

metrov a cien jednotlivých operácií. Tieto metódy nie sú uvedené v diagrame pre rozsiahlosť diagramu, ale je možné povedať, že sú súčasťou metód `enqueue`, `dequeue` a `delete`.

3.1.1 Popis metód

Jednotlivé implementácie obsahujú metódy, ktorých fungovanie je bližšie spomenuté v kapitole 2. Metóda `enqueue` predstavuje metódu na pridanie prvku do fronty. Táto metóda dostáva záznam o udalosti. Záznam je pridaný do fronty pomocou kópie. To znamená, že kalendár udalosti sa tvári ako typický C++ kontajner.

Naopak metóda `dequeue` predstavuje výber prvku s najmenšou časovou značkou. Metóda `find_min` funguje veľmi podobne ako metóda `dequeue`, s tým rozdielom, že udalosť zostáva vo fronte. Preto to, že udalosť vo fronte zostane, nie je vhodné túto udalosť nejako využívať k simulácií. Daná metóda funguje prevažne na zistenie prázdnosti kalendára, alebo zistenie, aká udalosť bude nasledovať. Nie však na spracovávanie danej udalosti.

Metóda `delete` dokáže odstrániť ľubovolnú udalosť za predpokladu, že užívateľ dokáže presne špecifikovať konkrétnu udalosť. Udalosť je následne lokalizovaná vo fronte na základe časovej značky a pri nájdení udalosti s totožnou časovou značkou sa následne kontroluje, či ide presne o špecifikovanú udalosť. Pre toto zistenie porovnáva prioritu udalosti a následne samotnú udalosť. Preto je nutné vedieť porovnať udalosti na zhodu. Pri využívaní vlastných dátových štruktúr musí užívateľ správne definovať operáciu `operator ==`.

Metóda `init` predstavuje konštruktor. Táto metóda inicializuje parametre, ak je ich nutné zadať. Bez zadania parametrov inicializuje objekt s prednastavenými parametrami. Prednastavené parametre budú určené na základe článkov spracúvajúcich konkrétne implementácie.

Metóda `destroy` predstavuje deštruktor. Táto metóda uvoľní udalosti, ktoré sa vo fronte nachádzajú. To znamená, že akékoľvek udalosti, ktoré sú v čase volania tejto metódy vo fronte zaniknú.

Trieda Pending_Event_Set

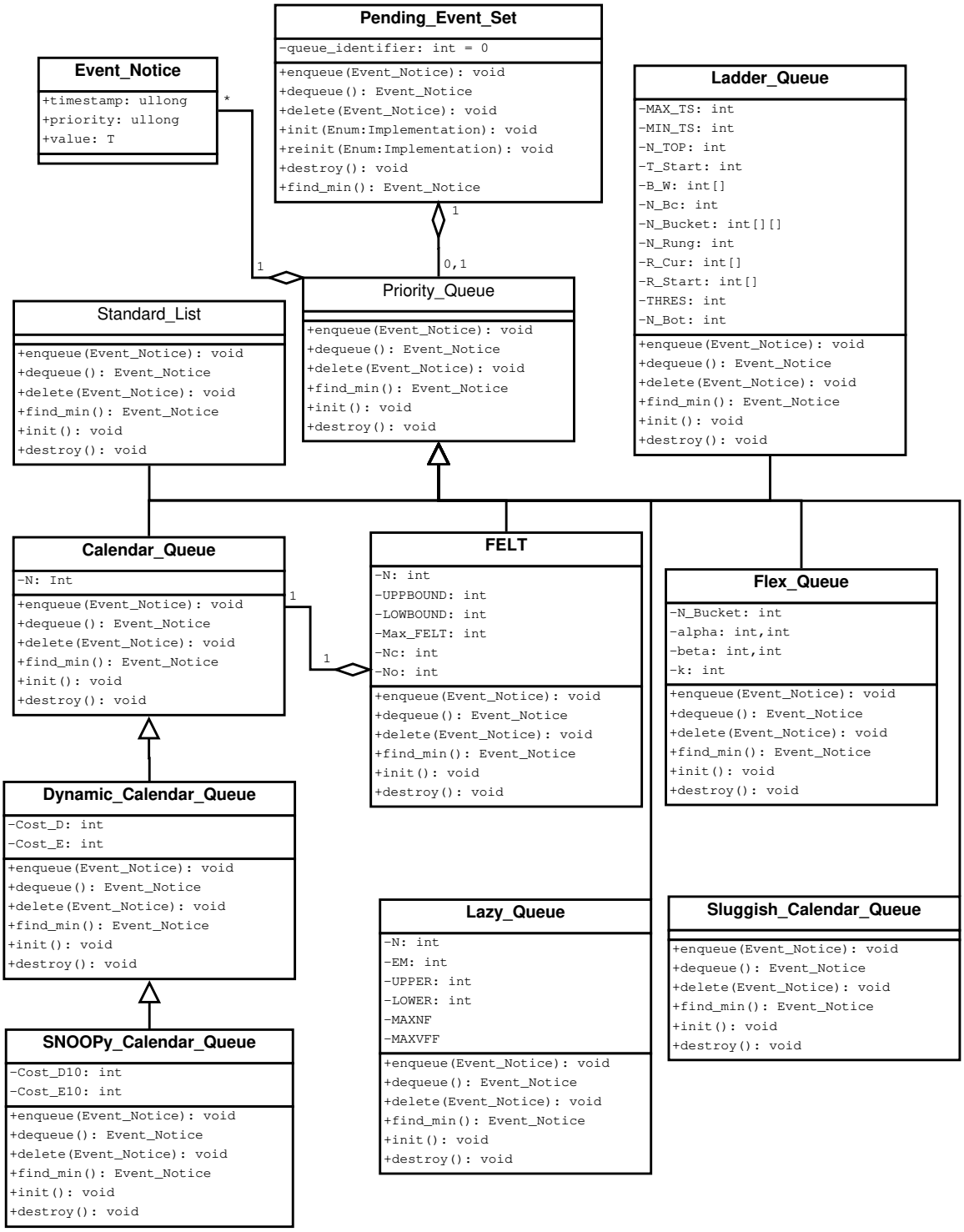
Najdôležitejšou metódou triedy `Pending_Event_Set` je metóda `init`. Táto metóda slúži na vytvorenie prioritnej fronty príslušného typu. Metóda obsahuje parameter, ktorý určí príslušnú implementáciu vnútornej prioritnej fronty. Po vytvorení objektu tejto triedy nie je zvolená žiadna implementácia a preto nie je možné tento objekt použiť na ukladanie udalostí. Objekt je použiteľný až po zavolaní metódy `init`. Určenie implementácie je reprezentované príslušnou enumeračnou hodnotou.

Po zavolaní metódy `init` s enumeračnou hodnotou reprezentujúcou platnú implementáciu (jedna enumeračná hodnota bude obsahovať prázdnu implementáciu) je možné používať takýto objekt na ukladanie a výber udalostí.

Vloženie udalosti sa vykoná pomocou metódy `enqueue`. Táto metóda deleguje udalosť na prioritnú frontu, ktorá sa postará o jeho uloženie. Výber sa zase vykoná pomocou metódy `dequeue`. Táto metóda taktiež zavolá príslušnú metódu prioritnej fronty. Veľmi podobne týmto metódam je možné použiť aj metódy `delete` a `find_min`.

Metóda `destroy` slúži na zničenie objektu kalendára udalostí. Tento objekt najprv zavolá `destroy` na prioritnú frontu, ak má nejakú špecifikovanú. Následne je objekt nastavený na počiatočný stav, to znamená, že obsahuje prázdnu implementáciu prioritnej fronty. V tomto bode je možné tento objekt bezpečne uvoľniť z pamäte.

Metóda `reinit` slúži na zmenu implementácie za chodu. Táto metóda vytvorí novú prioritnú frontu novo zvoleného typu a presunie všetky udalosti zo starej fronty do novej.



Obr. 3.1: Diagram tried knižnice

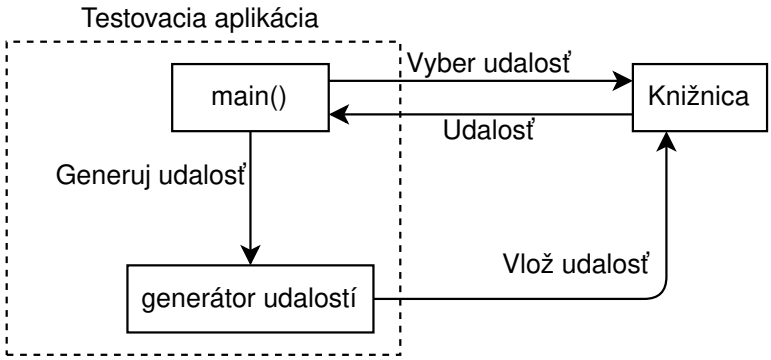
Po presunutí udalostí je stará fronta odstránená a kalendár používa už len novú frontu. V prípade podobnosti vnútornej štruktúry je možné namiesto presúvania udalostí postupne presunúť rovno celé bloky udalostí. Napríklad pri reinicializácii calendar queue na Dynamic calendar queue je možné použiť celé pole kýblikov, bez presúvania udalostí postupne do nového pola.

Touto metódou je možné reinicializovať prioritnú frontu na rovnakú, ktorú obsahuje, len zo zmenou parametrov. V takom prípade nedôjde k zničeniu pôvodnej fronty, len k potrebným príslušným modifikáciám na už existujúcej fronte.

Taktiež je možné za behu použiť aj metódu `init`. Táto metóda na rozdiel od metódy `reinit` odstráni všetky udalosti uložené v kalendári udalostí pred reinicializáciou kalendára. Je možné povedať, že volanie `init` na už inicializovaný objekt je totožné s volaním metódy `destroy` a následné použitie `init`. Táto realizácia zjednodušuje znovupoužitie toho istého objektu na ukladanie nových udalostí.

3.2 Návrh výkonnostných testov

Testy sú realizované ako aplikácia v C++ využívajúca navrhnutú knižnicu. Blokovú schému testovacej aplikácie je možné vidieť na obrázku 3.2. Aplikácia deleguje žiadosti na generátor udalostí, ktorý následne udalosť vloží do kalendára udalostí. Pri generovaní udalostí je možné definovať rozloženie udalostí v čase. Je možné zvoliť generovanie udalostí rovnomerné, trojuholníkové, alebo ťava. Testovacia aplikácia má informáciu o počte udalostí v čase a meria čas potrebný na vloženie udalosti, respektíve na výber udalosti. Tento čas je vždy vo vzťahu k aktuálnemu počtu udalostí v kalendári.



Obr. 3.2: Bloková schéma testovacej aplikácie

Jeden beh testovacej aplikácie sa skladá z cyklického generovania udalostí a vkladaní udalostí do kalendára až po požadované maximum udalostí v kalendári. Následne sa spustí cyklus, ktorý vyberie a následne vloží udalosť do kalendára. Aplikácia vždy vykoná niekoľko cyklov vkladania a výberu. Tento počet je možné špecifikovať. Po skončení cyklu je následne jedna udalosť z kalendára odstránená a nasleduje nový cyklus. Tento postup sa opakuje až do vyprázdnenia kalendára.

Aplikácia umožňuje špecifikovať implementácie kalendára, ktoré požadujeme otestovať. Na konci svojho behu vykoná priemer času vkladania a výberu zo všetkých vložení a výberov, ktoré vykonala. Následne je priemer vložený do csv súboru, ktorý je možné ďalej spracovať, alebo vykresliť z dát grafy. Názov výstupného súboru si užívateľ môže zvoliť pomocou parametra.

Kapitola 4

Implementácia a experimenty

Táto kapitola sa zaoberá implementačnými detailami a tvorbou knižnice. Približuje vývoj knižnice, ako aj postup a výsledky jej testovania. Ukazuje príklady použitia testovacej aplikácie, ako aj príklady výstupu vo forme grafov.

Zdrojové súbory na pamäťovom médiu sú uložené v priečinku *PES*. Viac o súborovej štruktúre média je možné vidieť v súbore `README.txt`, ktorý sa nachádza v koreňovom adresári pamäťového média.

4.1 Implementácia knižnice

Knižnica je naprogramovaná v jazyku C++. Skladá sa z 11 súborov. Deväť súborov reprezentuje jednotlivé implementácie, keďže každá implementácia je uložená pre prehľadnosť v samostatnom súbore. To sú súbory `list_queue.hh`, `calendar_queue.hh`, `dynamic_cq.hh`, `snoopy_cq.hh`, `sluggish.hh`, `ladder_queue.hh`, `lazy_queue.hh`, `flex_queue.hh` a súbor `felt.hh`. Súbor `structs.hh` obsahuje základnú triedu `Priority_Queue`, štruktúru `Event_Notice`, ako aj pomocné statické metódy pre porovnávanie udalostí pri radení a základné implementácie vloženia a odstránenia udalosti zo zoznamu. Posledným súborom je súbor `pes.hh`, ktorý obsahuje implementáciu triedy `Pending_Event_Set`. Všetky triedy a metódy nachádzajúce sa v knižnici sú zapúzdrené pomocou C++ namespace, ktorý má názov `pes`, čo je skratka od Pending Event Set. Všetky triedy sú definované ako C++ template, ktorý umožňuje vytvoriť kalendár udalostí pre ľubovoľný dátový typ, dokonca aj typy definované užívateľom. Jediná nutnosť je definovať nad daným dátovým typom `operátor==` na určenie zhody nájdenia udalosti pri metóde `delete`.

Na určenie implementácie pri metóde `init` u triedy `Pending_Event_Set` sa používa enum `Implementation`. Tento enum obsahuje hodnoty pre každú implementáciu ako aj pre prázdnu implementáciu. Jeho hodnoty je možné vidieť v tabuľke 4.1

Kalendár je možné inicializovať len zvolením implementácie bez ďalšieho nastavovania parametrov, alebo presnej voľby parametrov pre daný kalendár. Bez udania parametrov sú nastavené východzie parametre. Voľba parametrov sa vykonáva prostredníctvom štruktúry `S_PQ_settings`. Táto štruktúra obsahuje položky pre každú nastaviteľnú hodnotu pre každú implementáciu. Používajú sa vždy len položky pre práve zvolenú implementáciu, ostatné položky sú ignorované. Presnejší výpis položiek je možné vidieť v tabuľke 4.2.

Vývoj knižnice prebiehal na operačnom systéme EndeavourOS. EndeavourOS je linuxová distribúcia založená na Arch Linuxe. Knižnica využíva len štandardné knižnice, a preto nepotrebuje doinštalovávanie žiadnych ďalších knižníc. Testovanie prebiehalo pod

Enum hodnota	Význam
enum_empty	kalendár udalostí je neinicializovaný, prázdna implementácia
enum_linear	implementácia lineárneho zoznamu
enum_calendar_queue	implementácia calendar queue
enum_dynamic_CQ	implementácia dynamic calendar queue
enum_SNOOPy_CQ	implementácia SNOOPy calendar queue
enum_sluggish_CQ	implementácia sluggish calendar queue
enum_flex_queue	implementácia flex queue
enum_lazy_queue	implementácia lazy queue
enum_ladder_queue	implementácia ladder queue
enum_FELT	implementácia FELT

Tabuľka 4.1: Hodnoty enumerácie `Implementation` a ich význam.

Položka	Dátový typ	Východzia hodnota
Sluggish_Calendar_Queue_alpha	unsigned int	1000
Sluggish_Calendar_Queue_theta	unsigned int	100
Ladder_Queue_THRES	unsigned int	50
Lazy_Queue_UPPER	unsigned int	64
Lazy_Queue_LOWER	unsigned int	4
Lazy_Queue_MAXNF	unsigned int	1024
Lazy_Queue_MAXVFF	unsigned int	256
Lazy_Queue_peak_treshold	unsigned int	64
Lazy_Queue_peak_width	unsigned int	8
Flex_Queue_alpha	double	7/8
Flex_Queue_beta	double	1/4
Flex_Queue_k	double	2
Flex_Queue_N_bucket	unsigned int	32
FELT_Nc	unsigned int	20
FELT_No	unsigned int	50
FELT_LOWER	unsigned int	256
FELT_UPPER	unsigned int	4096
FELT_max_felt	unsigned int	3840
FELT_CQ_implementation	Implementation	enum_calendar_queue

Tabuľka 4.2: Položky štruktúry `S_PQ_settings`, ich dátový typ a východzia hodnota.

štandardom C++11 aj C++17. Knižnica by nemala mať problém ani zo žiadnym novším štandardom. Použitie knižnice spočíva v zahrnutí hlavičkového súboru `pes.hh` pomocou príkazu `#include "pes.hh"`. Následne je možné používať všetky štruktúry a triedy implementované v knižnici. Užívateľ je síce schopný využívať priamo jednotlivé implementácie prioritnej fronty, ale odporúča sa používať tieto implementácie prostredníctvom triedy `Pending_Event_Set`. Pomocné štruktúry a metódy implementácie sú zabalená ako chránené pre jednotlivé triedy rozširujúce triedu prioritnej fronty.

4.2 Ukážka použitia knižnice

Príklad použitia navrhovanej knižnice je možné vidieť na kóde 4.1. Tento kód ukazuje jednoduchú ukážku použitia knižnice v simulácii typu next-event. Tento pseudokód je prevažne C++, keďže v tomto jazyku bude písaná aj knižnica, a v tomto jazyku budú aj aplikácie, ktoré ju budú používať. Funkcie napísané v typografickej notácii snake case pochádzajú z knižnice a funkcie písané notáciou camel case sú definované užívateľom.

Aplikácia inicializuje kalendár udalostí na implementáciu všeobecného calendar queue. Následne doň vloží počiatočnú udalosť a spustí cyklenie simulácie. Každá udalosť môže pri vykonaní generovať nové udalosti. Nové udalosti sú opäť vložené do kalendára udalostí, a proces pokračuje, kým sa v kalendári nachádzajú nejaké udalosti. Po dosiahnutí nejakej podmienky dôjde k zmene implementácie kalendára udalostí na SNOOPy calendar queue. K tomuto môže dôjsť napríklad na základe prechodu z rovnomerne generovaných udalostí v čase na udalosti, ktorých generovanie v čase je nerovnomerné. Túto podmienku musí nastaviť vykonanie nejakej udalosti.

```
    ...
Pending_Event_Set<int> set;
set.init(enum_calendar_queue);
Event_Notice e = generateStartEvent();
set.enqueu(e);
unsigned long long time = T_START;
while(set.find_min(e)) {
    e=set.dequeue();
    if (e.timestamp > T_END)
        break;
    time=e.timestamp;
    eventArray = doEventAndGenerateNewEvents(e);
    for(int i = 0; i < eventArray.size();++i)
        set.enqueue(eventArray[i]);
    if (something)
        set.reinit(enum_SNOOPy);
}
time=T_END
    ...
```

Kód 4.1: Kód použitia knižnice

4.3 Testovanie a merania výkonnosti

Na testovanie knižnice slúži krátky program používajúci túto knižnicu. Program sa skladá z jedného súboru `main.cc`. Preklad aplikácie sa uskutočňuje pomocou programu `make`, preto je k C++ súboru pribalený aj súbor `Makefile`. Preklad testovacej aplikácie využívajúcej implementovanú knižnicu sa vykonával pomocou prekladača gcc verzia 10.2.0. Po preklade sa vytvorí spustiteľný súbor s menom `main`. Testovacia aplikácia je plne parametrizovateľná prostredníctvom argumentov. Prehľad argumentov a ich význam je možné vidieť v tabuľke 4.3. Testovanie a meranie prebehlo na počítači z procesorom Intel Core i5-6200U, veľkosťou RAM 8GB a operačným systémom EndeavourOS.

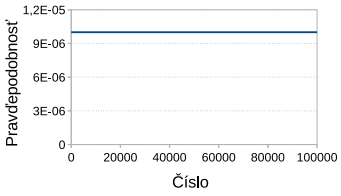
Argument	Význam
--help/-h	vypíše nápovedu pre obsluhu
--number/-n	počet udalostí po ktorý testuje
--repeat/-r	počet opakovaní vloženia a výberu pri každom počte udalostí
--generator/-g	typ generátoru udalostí (obdĺžnik,trojuholník,ťava)
--implementation/-i	voľba testovaných udalostí vo forme celého čísla z intervalu (0,512) kde každý bit predstavuje jednu implementáciu, číslo 511 testuje všetky implementácie
--output/-o	názov výstupného súboru

Tabuľka 4.3: Argumenty testovacej aplikácie a ich význam.

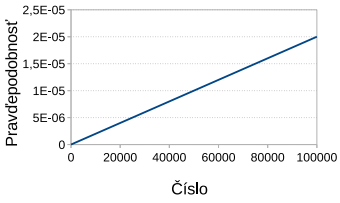
```
static default_random_engine re(time(NULL));
static uniform_int_distribution<int> obdlznik(0, 100000);
static vector<double> i_t {0,100000};
static vector<double> w_t {0,1};
piecewise_linear_distribution<double> trojuholnik
               (i_t.begin(),i_t.end(),w_t.begin());
static vector<double> i_c {0.0, 20000.0, 20000.0,
                           40000.0, 40000.0, 60000.0};
static vector<double> w_c {1.0, 1.0, 3.0, 3.0, 1.0, 1.0};
static piecewise_linear_distribution<double> tava
               (i_c.begin(),i_c.end(),w_c.begin());
switch (generator)
{
    case obdlznik:
        cas = obdlznik (re);
        while(cas < aktualny_cas)
            cas+=100000;
        return cas;
    case trojuholnik:
        cas = trojuholnik (re);
        while(cas < aktualny_cas)
            cas+=100000;
        return cas;
    case tava:
        cas = tava (re);
        while(cas < aktualny_cas)
            cas+=60000;
        return cas;
}
```

Kód 4.2: Pseudokód generátora udalostí

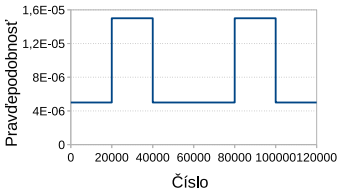
Výber generátora udalostí je možný z troch predpripravených generátorov udalostí, a to rovnomerné, trojuholníkové a ťava. Hustotu rozdelenia pravdepodobnosti je možné vidieť na obrázkoch 4.1, 4.2 a 4.3.



Obr. 4.1: Hustota pravdepodobnosti rovnomerné rozloženie



Obr. 4.2: Hustota pravdepodobnosti trojuholníkové rozloženie



Obr. 4.3: Hustota pravdepodobnosti rozloženie ťava

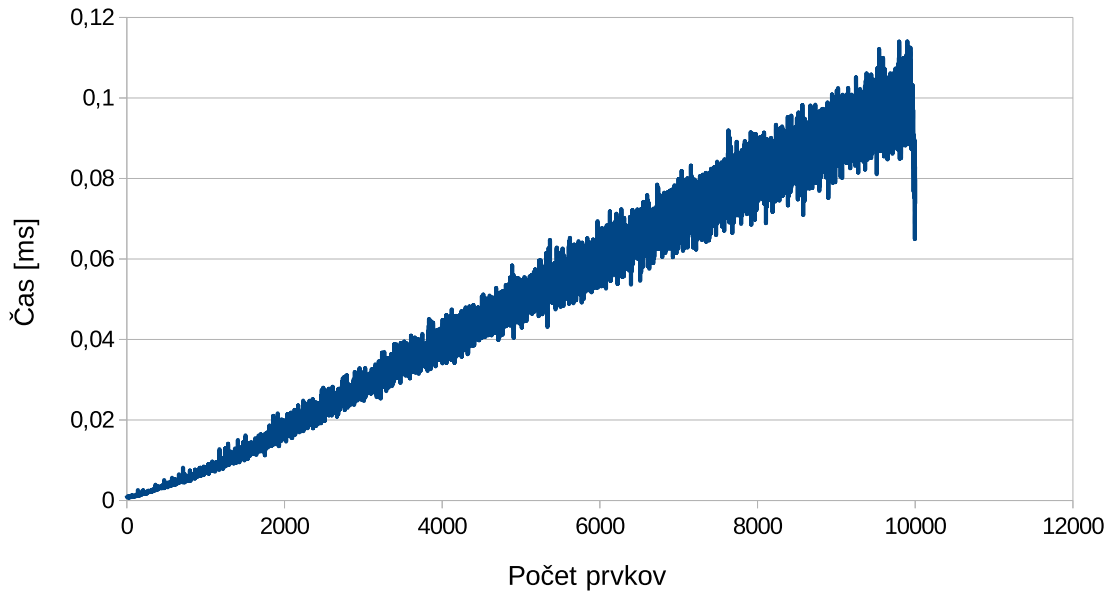
Generátor generuje udalosti len v určitom intervale a ak udalosť padne do času, ktorý sa nachádza pred aktuálnym časom, tak následne pripočíta hornú hranicu intervalu toľkokrát, aby čas udalosti bol vyšší, ako je aktuálny čas. Týmto je zabezpečené, že generovanie udalostí je schopné vygenerovať udalosti aj po prekročení aktuálneho času rozsahu generátora. Tento prístup taktiež zabezpečuje periodické opakovanie priebehu hustoty rozloženia pravdepodobnosti. Generovanie udalostí používa knižnicu `<random>`, presnejšie `uniform_int_distribution` a `piecewise_linear_distribution`. Okrem toho používa `default_random_engine`, ktorého semiačko je inicializované pomocou funkcie `time(NULL)`. Pseudokód generátora udalostí je možné vidieť na kóde 4.2.

V ďalších sekciách sú zobrazené grafické výsledky jednotlivých implementácií pri použití východzích nastavení testovacej aplikácie. To znamená, že argument `-n` je nastavený na 10000, `-r` je nastavené na 100, `-g` je nastavený na obdĺžnik a `-i` je nastavené na 511, čiže všetky implementácie. Výstupný súbor sa volá measure.csv.

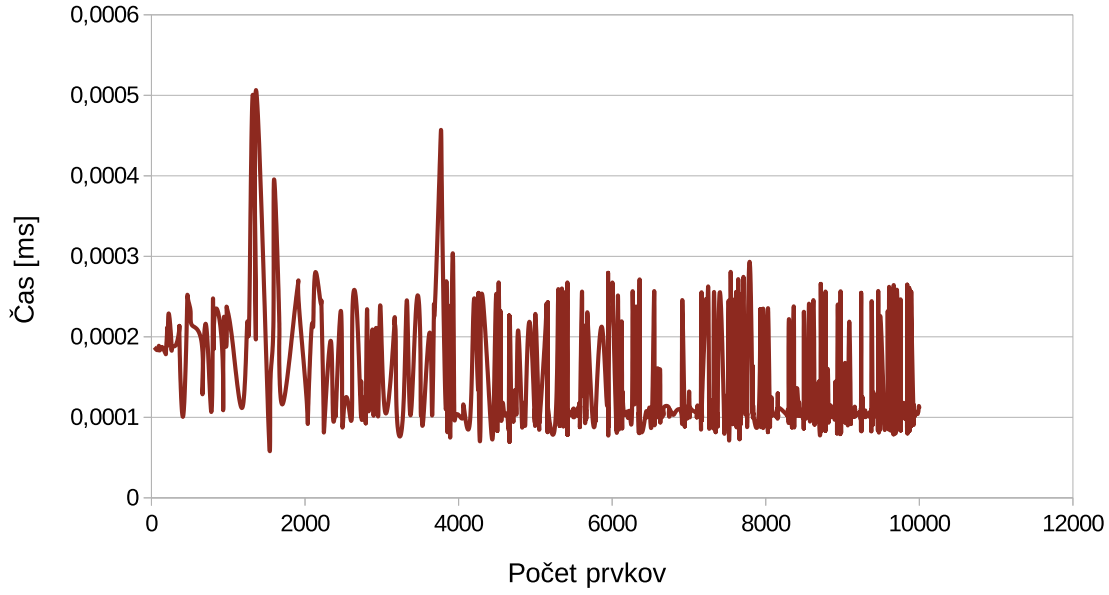
4.3.1 Lineárny zoznam

Lineárny zoznam dosahuje podľa teórie v kapitole 2.11 zložitosť $O(n)$ pre vloženie prvku. Z obrázka 4.4 je možné povedať že táto operácia skutočne dosahuje danú zložitosť. Implementácia je rýchla pri malom počte udalostí, ale s rastúcim počtom udalostí trvá vloženie udalosti čoraz dlhšie.

Výber prvku je naopak veľmi rýchly, keďže dochádza vždy k výberu prvej udalosti v zozname. Na obrázku 4.5 je možné vidieť, že skutočne je zložitosť výberu prvku konštantná, $O(1)$. Kmitanie hodnôt v grafe je spôsobené najmä spôsobom merania, ktorý sa mení so záťažou procesora.



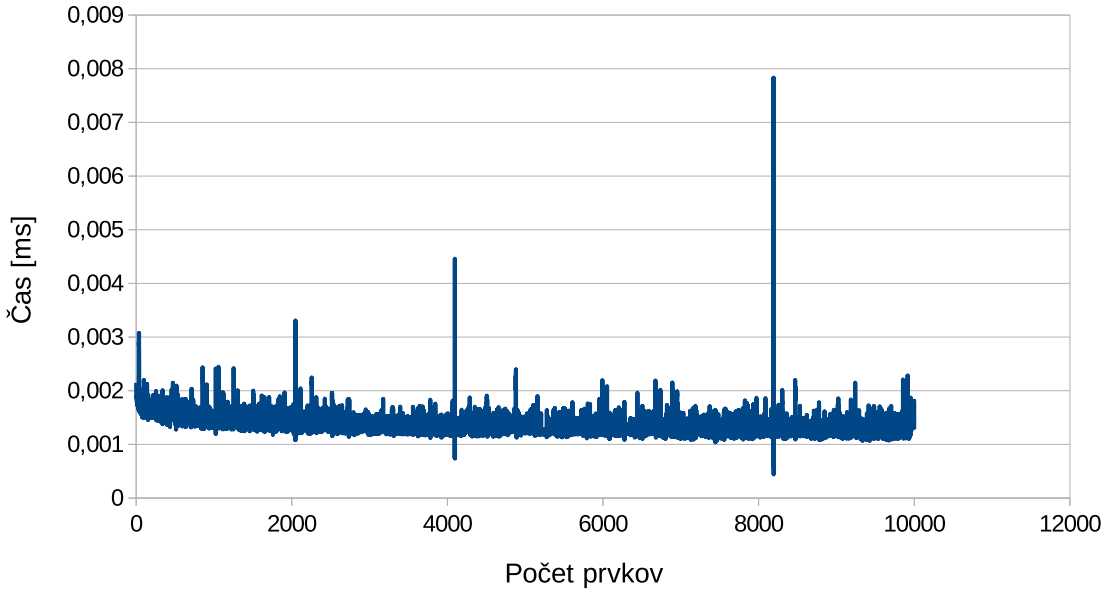
Obr. 4.4: Nameraný čas trvania operácie vloženia prvku do kalendára typu štandardný zoznam v závislosti na počte prvkov v kalendári



Obr. 4.5: Nameraný čas trvania operácie výberu prvku z kalendára typu štandardný zoznam v závislosti na počte prvkov v kalendári

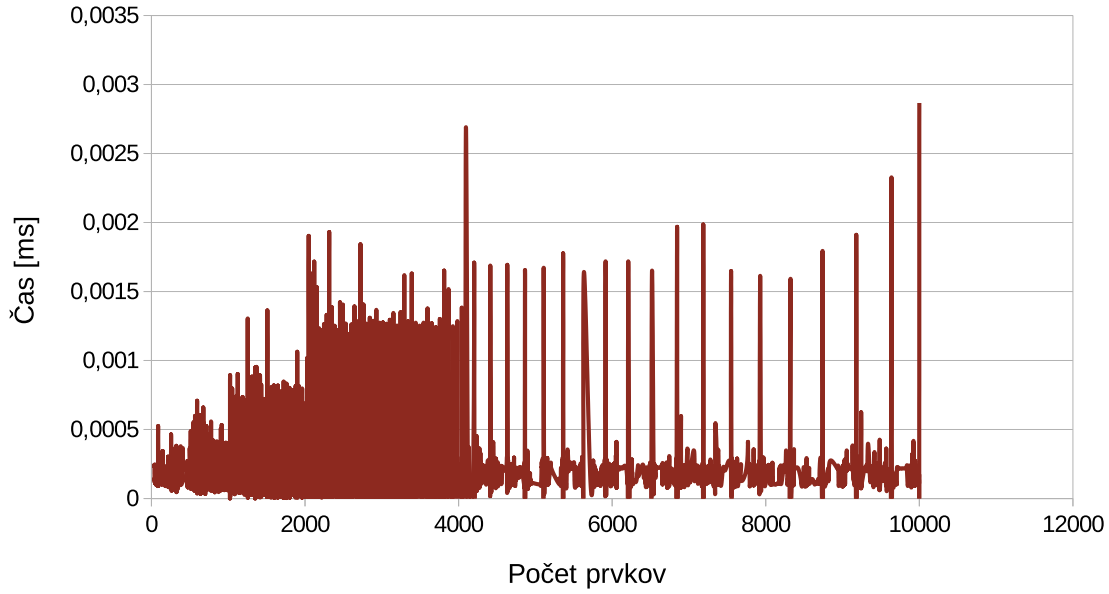
4.3.2 Calendar queue

Calendar queue podľa teórie z kapitoly 2.3 dosahuje konštantnú zložitosť $O(1)$. Na obrázku 4.6 je vidieť, že priebeh je skutočne konštantný, až na pár špičiek. Tieto špičky sú spôsobené hlavne zmenou veľkosti v kalendári. Keďže zmena veľkosti je operácia zo zložitosťou $O(n)$, tak,na grafe môžeme pozorovať rastúcu amplitúdu špičiek. Keďže počet prvkov je nutné vždy zdvojnásobiť na vykonanie zmeny veľkosti je vidieť, že vzdialenosť medzi špičkami sa tiež zdvojnásobuje. Z tohto vyplýva, že zmena veľkosti sa uskutočňuje dvakrát menej často, ale jej čas trvá dvakrát dlhšie.



Obr. 4.6: Nameraný čas trvania operácie vloženia prvku do kalendára typu Calendar queue v závislosti na počte prvkov v kalendári

Výber prvku je o niečo horší od lineárneho zoznamu, ale táto operácia je pomerne rýchla. Priebeh operácie je možné vidieť na obrázku 4.7. Namerané výsledky odpovedajú amortizovanej zložitosti, ktorá je uvedená v kapitole 2.3. Špičky sú spôsobené zmenou veľkosti, ako aj rozložením udalostí v zozname. V najhoršom prípade dochádza pri výbere v prezretí $2n$ kýblikov, takže v najhoršom prípade môže byť zložitosť lineárna $O(n)$.



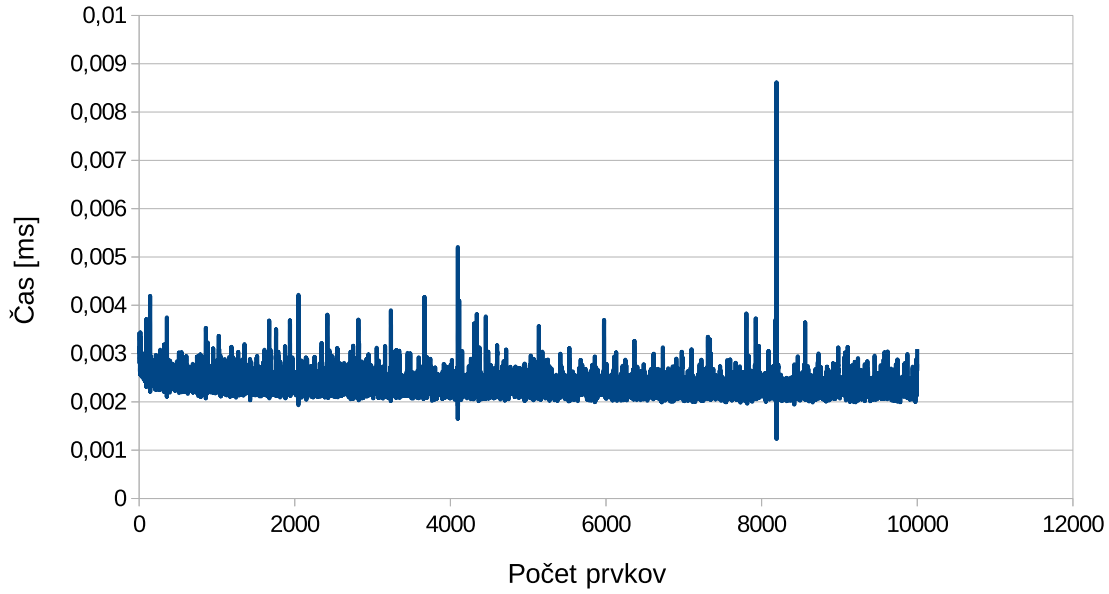
Obr. 4.7: Nameraný čas trvania operácie výberu prvku z kalendára typu Calendar queue v závislosti na počte prvkov v kalendári

4.3.3 Dynamic calendar queue

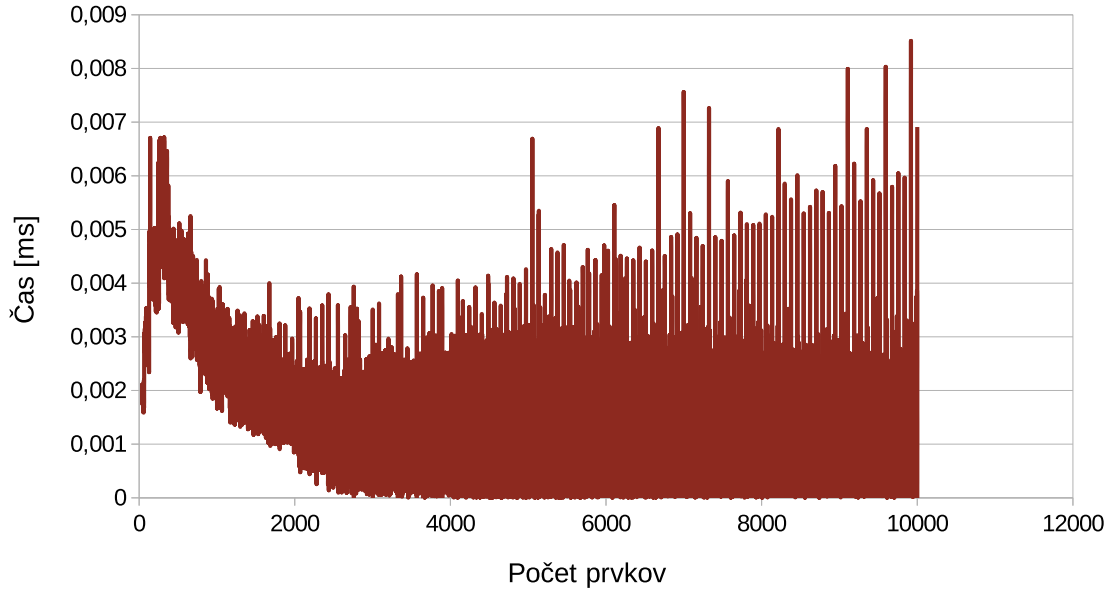
Dynamic calendar queue sa v zložitosti vloženia prvku podobá veľmi calendar queue. Priebeh je možné vidieť na obrázku 4.8. Na obrázku pozorujeme špičky spôsobené zmenou veľkosti, ale aj to že priebeh je v podstate konštantný, zložitosť $O(1)$, ako je uvedené v kapitole 2.4.

Priebeh výberu prvku je možné vidieť na obrázku 4.9. Priebeh je viac rozkmitaný ako calendar queue, ale taktiež je možne vidieť konštantnosť priebehu až na rastúce špičky, ktoré sa v tomto prípade nachádzajú častejšie ako pri calendar queue. To môže byť spôsobené napríklad nerovnomerným rozložením udalostí vo fronte alebo chybou merania. Ale aj napriek tomu je možné povedať že ide o konštantnú zložitosť $O(1)$.

Samozrejme táto implementácia je o niečo pomalšia ako calendar queue, keďže dochádza k sledovaniu cien vloženia a výberu, a teda k sledovaní viacerých podmienok ako pri calendar queue. Taktiež nemusí zmena veľkosti nastať len z dôvodu dosiahnutia príslušnej hranice.



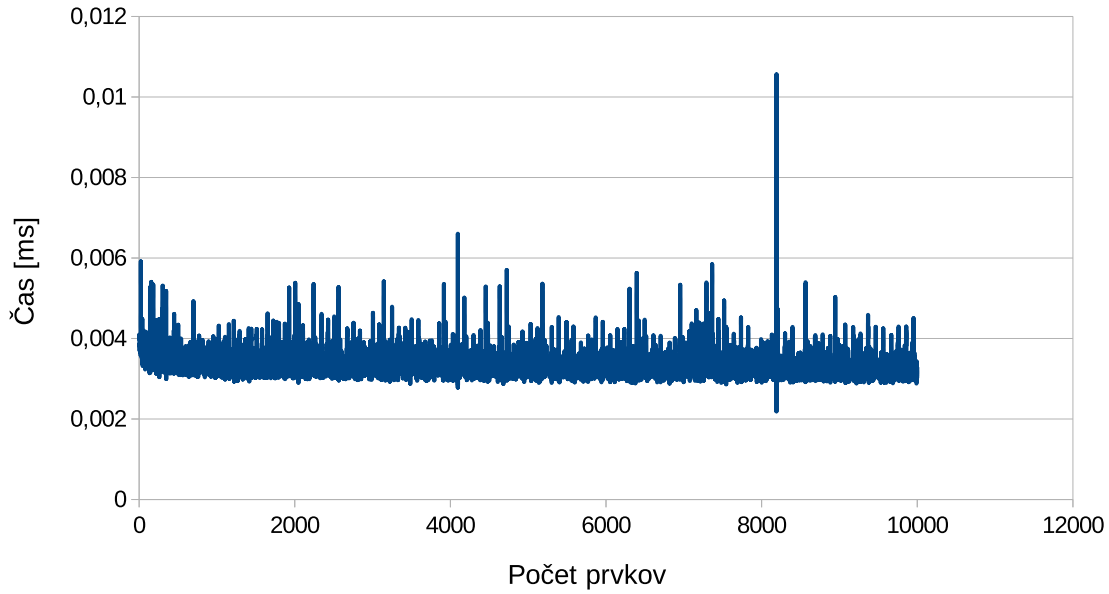
Obr. 4.8: Nameraný čas trvania operácie vloženia prvku do kalendára typu dynamic calendar queue v závislosti na počte prvkov v kalendári



Obr. 4.9: Nameraný čas trvania operácie výberu prvku z kalendára typu dynamic calendar queue v závislosti na počte prvkov v kalendári

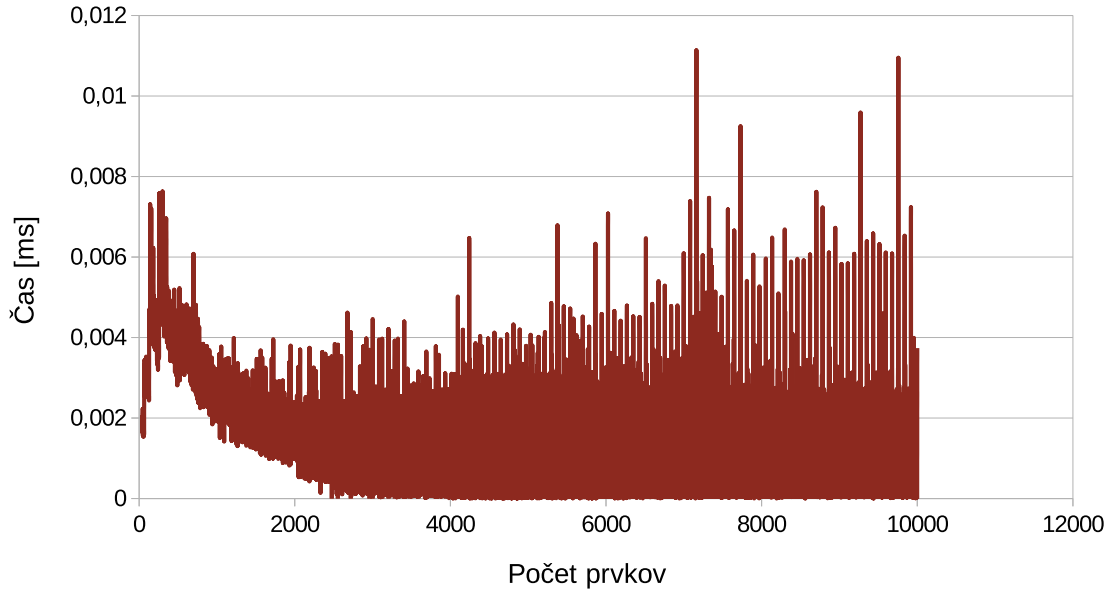
4.3.4 SNOOPy calendar queue

SNOOPy calendar queue je skoro totožný s dynamic calendar queue. Priebeh vloženia prvku je možné vidieť na obrázku 4.10. Priebeh obsahuje konštantnú zložitosť $O(1)$, ako je aj uvedené v kapitole 2.5, so špičkami pri hraniciach na zmenu veľkosti.



Obr. 4.10: Nameraný čas trvania operácie vloženia prvku do kalendára typu SNOOPy calendar queue v závislosti na počte prvkov v kalendári

Výber prvku je taktiež skoro totožný s dynamic calendar queue. Priebeh je možné vidieť na obrázku 4.11. Zložitosť je v priemere konštantná $O(1)$. Tento kalendár je vo svojej podstate totožný s dynamic calendar queue, až na rozdiel, že výpočet šírky kýblikov je viac matematický ako pri dynamic calendar queue, ktorý túto šírku počíta na základe priemeru z viacerých udalostí v okolí najplnšieho kýblika.

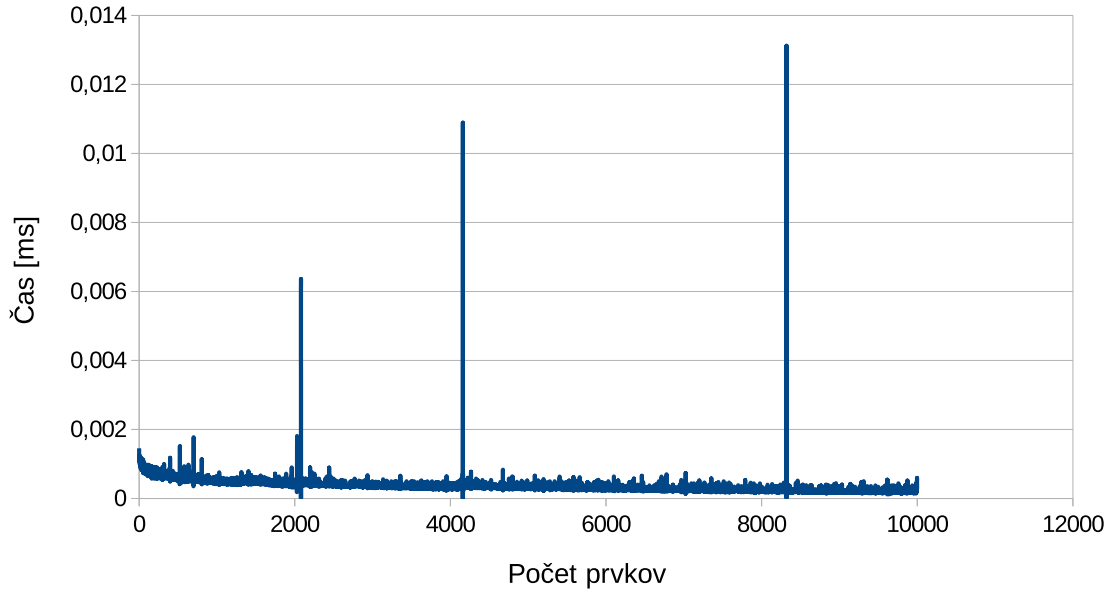


Obr. 4.11: Nameraný čas trvania operácie výberu prvku z kalendára typu SNOOPy calendar queue v závislosti na počte prvkov v kalendári

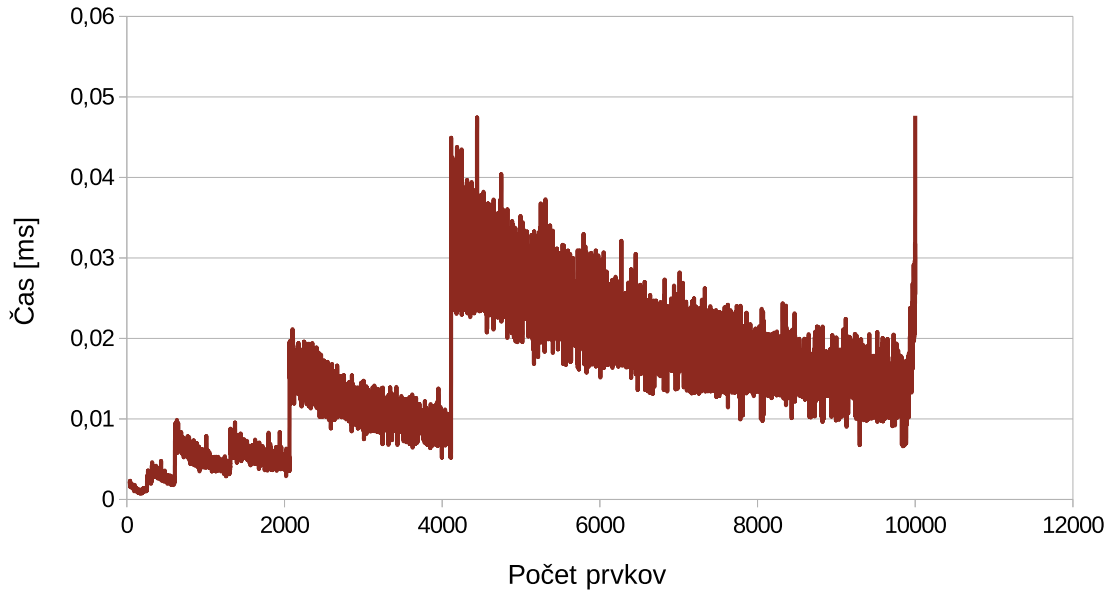
4.3.5 Lazy queue

Lazy queue dosahuje pre vloženie prvku konštantnú zložitosť, ako tvrdí aj kapitola 2.8, čo potvrdzuje aj obrázok 4.12. Na obrázku je možné vidieť aj špičky spôsobené zmenou veľkosti. Tieto špičky sú o hodne výraznejšie od bežného fungovania, ale nedochádza k ním často.

U výberu prvku môžme na obrázku 4.13 pozorovať mierne rastúci priebeh. Ten je spôsobený predovšetkým v rátaní počtu udalostí v hornej polovici ďalekej budúcnosti. Pre zrýchlenie by bolo vhodné toto počítanie upraviť, aby bola dosiahnutá konštantná zložitosť $O(1)$. V aktuálnom stave dochádza k miernej lineárnej zložitosti, ktorá má ale pomerne nízky sklon.



Obr. 4.12: Nameraný čas trvania operácie vloženia prvku do kalendára typu lazy queue v závislosti na počte prvkov v kalendári

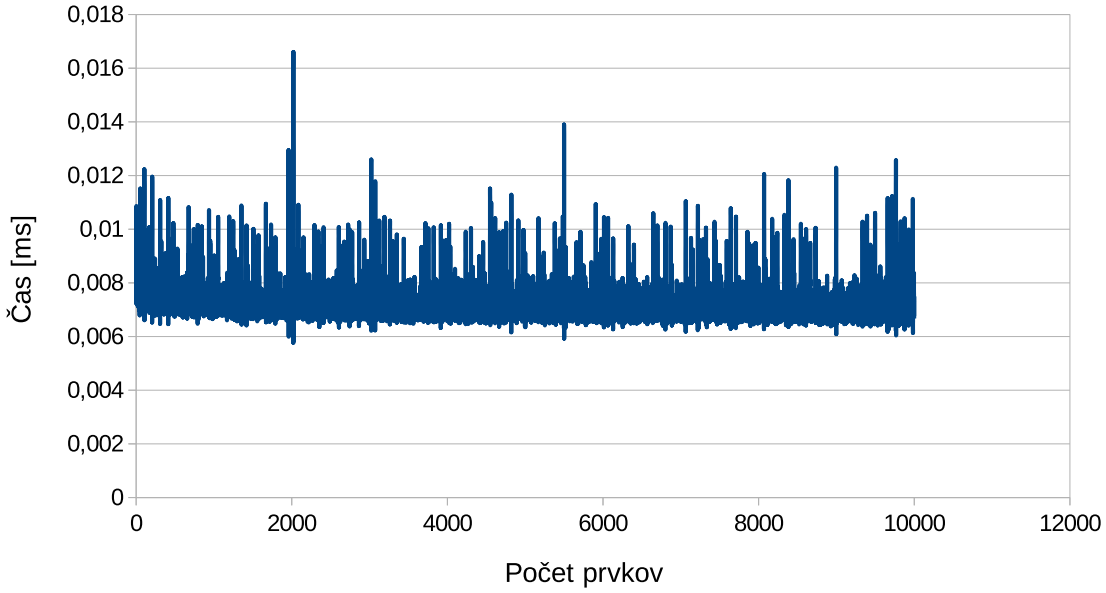


Obr. 4.13: Nameraný čas trvania operácie výberu prvku z kalendára typu lazy queue v závislosti na počte prvkov v kalendári

4.3.6 Ladder queue

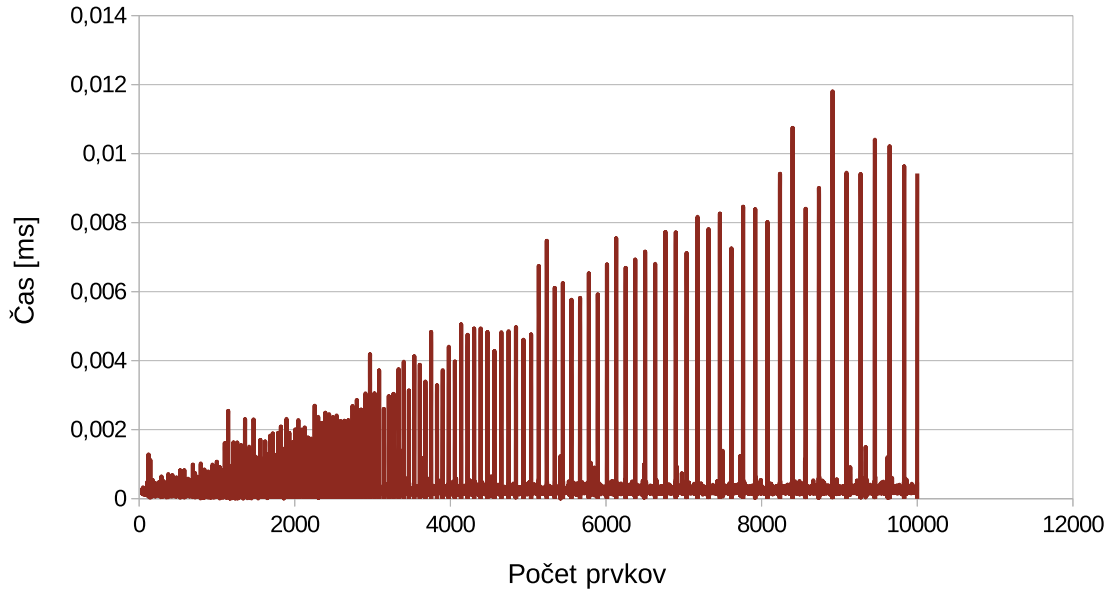
Na obrázku 4.14 je možné vidieť priebeh času pre vloženie prvku pri implementácii ladder queue. Priebeh skutočne poukazuje na to, že sa jedná o konštantnú zložitosť $O(1)$, ako je uvedené v kapitole 2.9. Implementácia dosahuje lineárnu zložitosť $O(n)$, len ak dochádza

k vkladaniu udalostí do spodného usporiadaného zoznamu. Ale keďže tento zoznam má obmedzenú veľkosť, tak sú aj tieto vloženia rýchle. Najdlhšie trvá situácia, keď po pridaní udalosti do spodného zoznamu dôjde k prekročeniu hranice a následná tvorba novej priečky.



Obr. 4.14: Nameraný čas trvania operácie vloženia prvku do kalendára typu ladder queue v závislosti na počte prvkov v kalendári

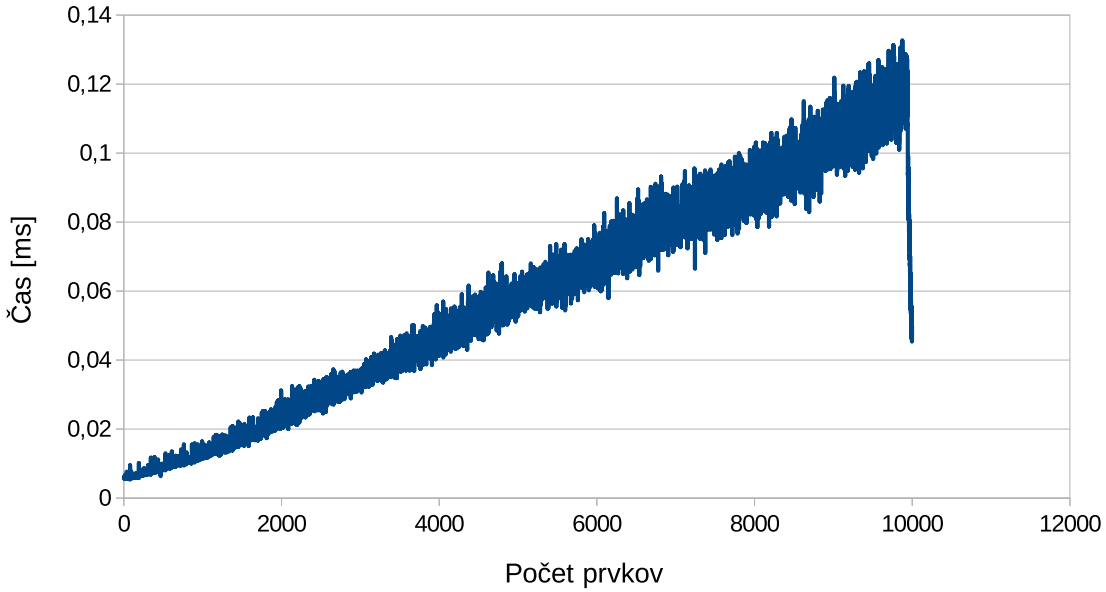
Na obrázku 4.15 je možné vidieť čas potrebný na výber prvku. Priebeh tejto operácie je relatívne konštantný až na situácie tvorby nových priečok a následnému presúvaniu udalostí. Tieto situácie je možné vidieť ako špičky na grafe, ktoré dosahujú teoreticky lineárnu zložitosť $O(n)$. Najdlhší výber prvku pozostáva z vytvorenia prvej priečky, presunutia všetkých udalostí do priečky a následnej tvorby nového spodného zoznamu.



Obr. 4.15: Nameraný čas trvania operácie výberu prvku z kalendára typu ladder queue v závislosti na počte prvkov v kalendári

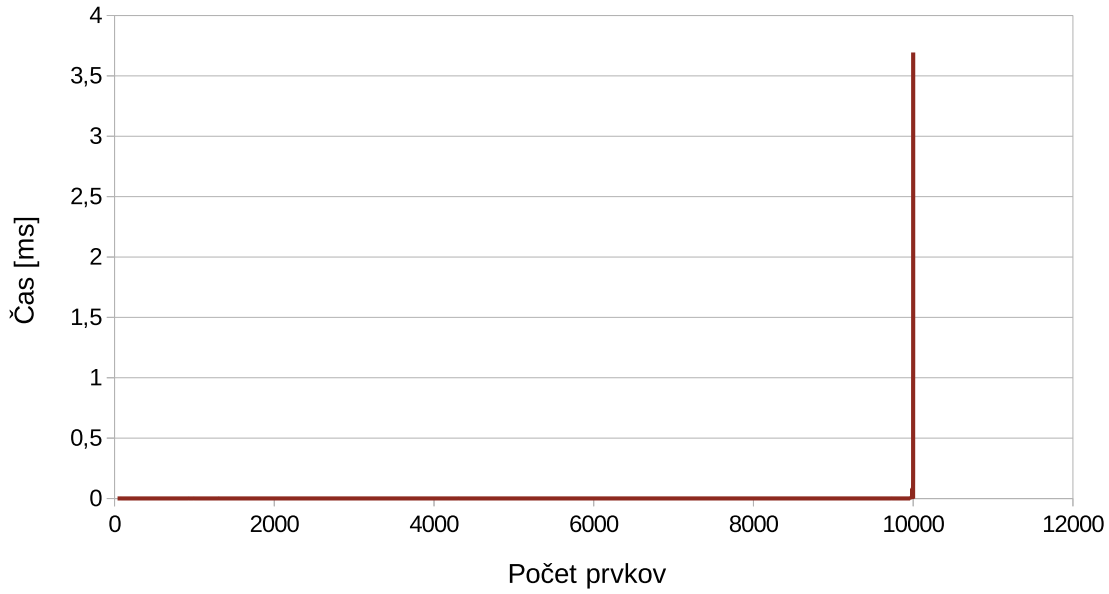
4.3.7 Flex queue

Flex queue dosahuje podľa kapitoly 2.7 lineárnej zložitosti $O(n)$, ako je možné vidieť na obrázku 4.16. Táto zložitosť je spôsobená tým, že počet kýblikov je konštantný, a preto s rastúcim počtom udalostí v štruktúre spôsobuje nutnosť umiestnenia viacerých udalostí do toho istého kýblika. Tento typ testu nie je úplne vhodný na testovanie výkonnosti flex queue, keďže v prvom momente dochádza k naplneniu štruktúry 10000 udalosťami. Keďže šírka kýblikov vo flex queue sa nastavuje vždy iba po vyprázdnení niektorého pola kýblikov môže dôjsť k tomu, že štruktúra môže mať dlhý čas nevhodne zvolenú šírku kýblika.



Obr. 4.16: Nameraný čas trvania operácie vloženia prvku do kalendára typu flex queue v závislosti na počte prvkov v kalendári

Na obrázku 4.17 je zase vidieť priebeh času pre výber prvku. Špička na konci je spôsobená tým, že je zo začiatku nutné prispôsobiť šírku kýblika, ku ktorej dôjde pri prvých výberoch zo štruktúry. Keďže počiatočná šírka je nastavená na pomerne nízke číslo, tak je väčšina prvých udalostí odložená do zoznamu pre udalosti nepatriace do aktuálneho horizontu. Po vyprázdnení aktuálneho horizontu následne musí dôjsť k presunu veľkého množstva udalostí do pola kýblikov po zmene jeho veľkosti. Okrem tejto špičky dosahuje priebeh konštantnú zložitosť $O(1)$.

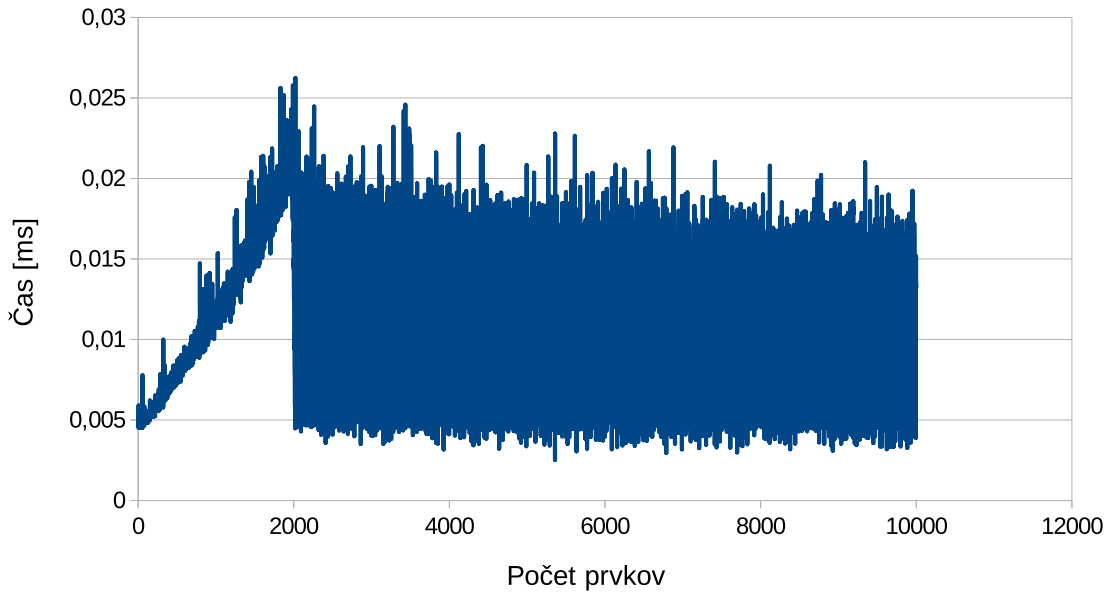


Obr. 4.17: Nameraný čas trvania operácie výberu prvku z kalendára typu flex queue v závislosti na počte prvkov v kalendári

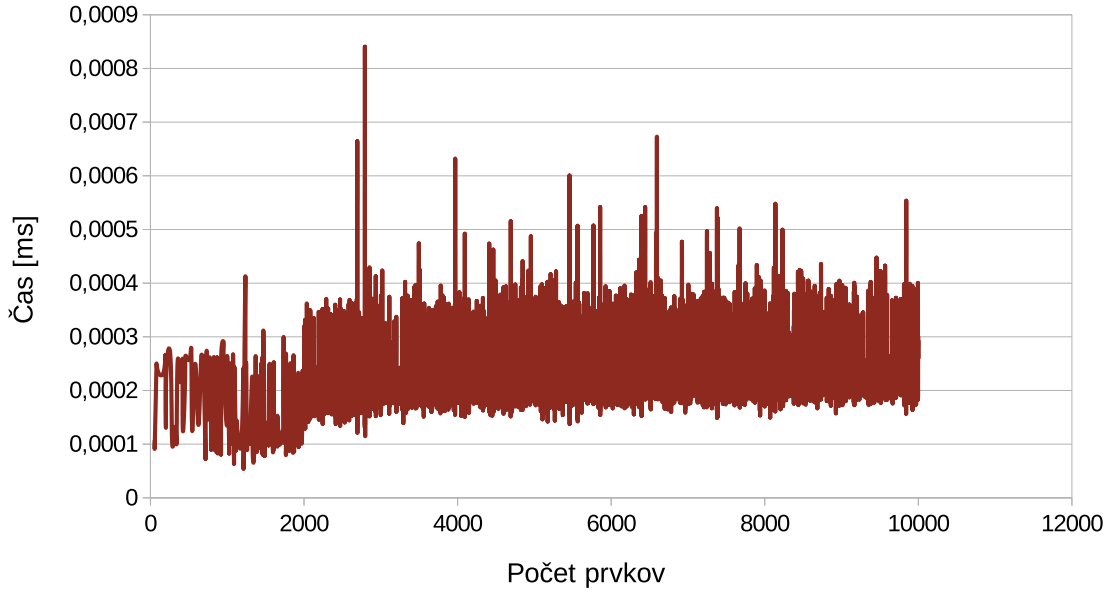
4.3.8 Sluggish calendar queue

Na obrázku 4.18 je vidieť priebeh času potrebného pre vloženie prvku do sluggish calendar queue. Je zrejmé, že sa skutočne jedná o konštantnú zložitosť $O(1)$, ako tvrdí aj kapitola 2.6. Čas potrebný na vloženie je daný prevažne počtom udalostí v laickom zozname, ako aj od časovej značky práve vkladanej udalosti. V najhoršom prípade dochádza k prejdeniu 2α udalostí, a následnému vloženiu laického zoznamu do calendar queue. Táto skutočnosť spôsobuje vyššie kmitanie v okolí strednej hodnoty. Keďže udalosti do calendar queue nie sú pridávané tak často ako pri bežnej implementácií calendar queue a zároveň ich nie je tak veľa, nie je zreteľná špička spôsobená zmenou veľkosti calendar queue časti, ktoré sa budú nachádzať až pri oveľa väčšom počte udalostí v kalendári.

Čas trvania operácie výberu prvku je vidieť na obrázku 4.19. Taktiež je možné si všimnúť konštantnú zložitosť $O(1)$. Calendar queue časť neznižuje svoju veľkosť pri tých istých hraniciach ako pri bežnom calendar queue, taktiež je nutné vždy kontrolovať a vyberať z dvoch udalostí a to z čela laického zoznamu a calendar queue časti.



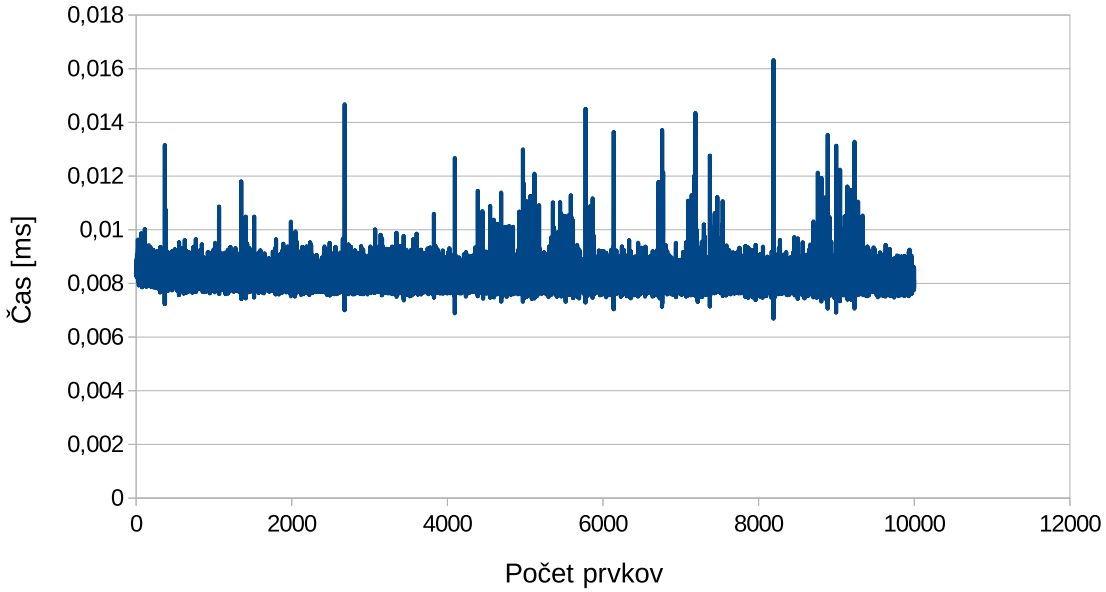
Obr. 4.18: Nameraný čas trvania operácie vloženia prvku do kalendára typu sluggish calendar queue v závislosti na počte prvkov v kalendári



Obr. 4.19: Nameraný čas trvania operácie výberu prvku z kalendára typu sluggish calendar queue v závislosti na počte prvkov v kalendári

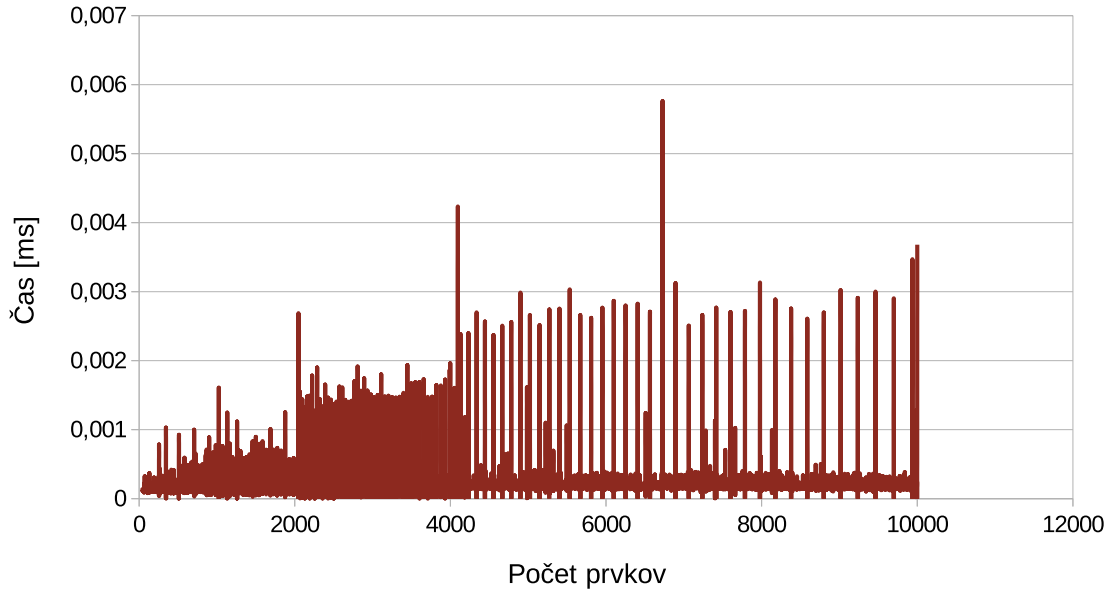
4.3.9 FELT

Obrázok 4.20 ukazuje priebeh trvania operácie vloženia prvku pre implementáciu FELT. Je možné vidieť že priebeh sa veľmi podobá priebehu calendar queue, až na hornú polovicu grafu, ktorá nedosahuje tak isto veľké špičky pri zmene veľkosti kalendára, keďže FELT časť zaisťuje, že kalendár pri rastúcom trende nedosahuje takú veľkosť ako bez FELT časti. Môžme vidieť, že zložitosť je konštantná $O(1)$ až na niekoľko špičiek, ktoré v tomto prípade nedosahujú lineárny rast práve preto, že kalendár nedosahuje také isté rozmery, ale iba presúva udalosti do FELT časti.



Obr. 4.20: Nameraný čas trvania operácie vloženia prvku do kalendára typu FELT v závislosti na počte prvkov v kalendári

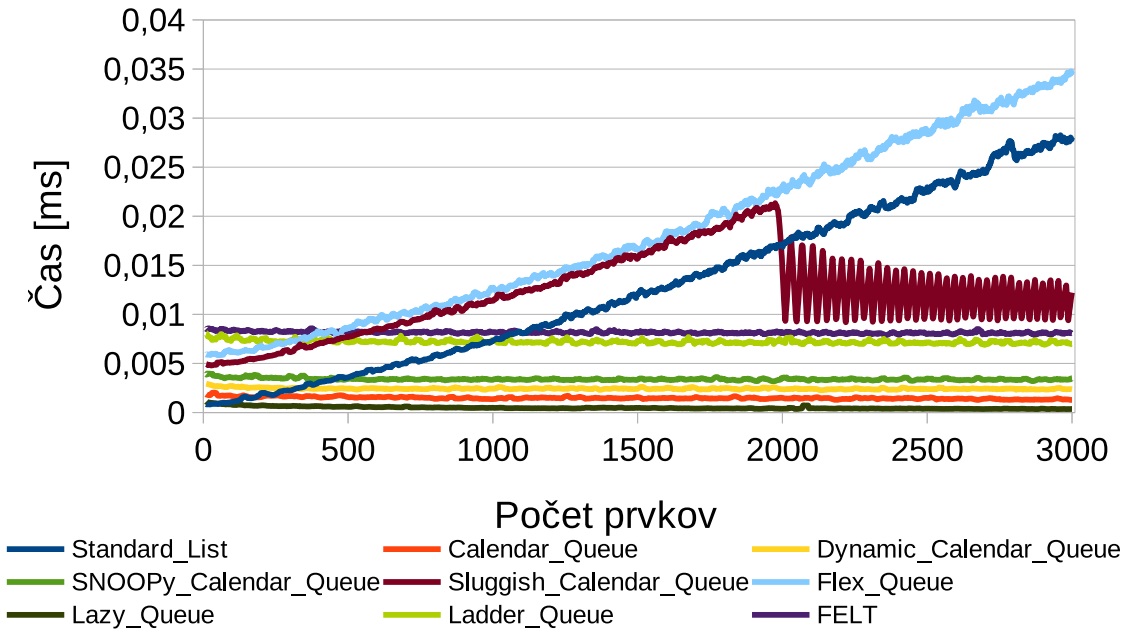
Na obrázku 4.21 vidíme zase priebeh trvania operácie výberu prvku u implementácií FELT. Táto zložitosť je taktiež konštantná, až na špičky spôsobené pravdepodobne presúvaním udalostí z FELT štruktúry do calendar queue.



Obr. 4.21: Nameraný čas trvania operácie výberu prvku z kalendára typu FELT v závislosti na počte prvkov v kalendári

4.3.10 Porovnanie efektivity jednotlivých implementácií

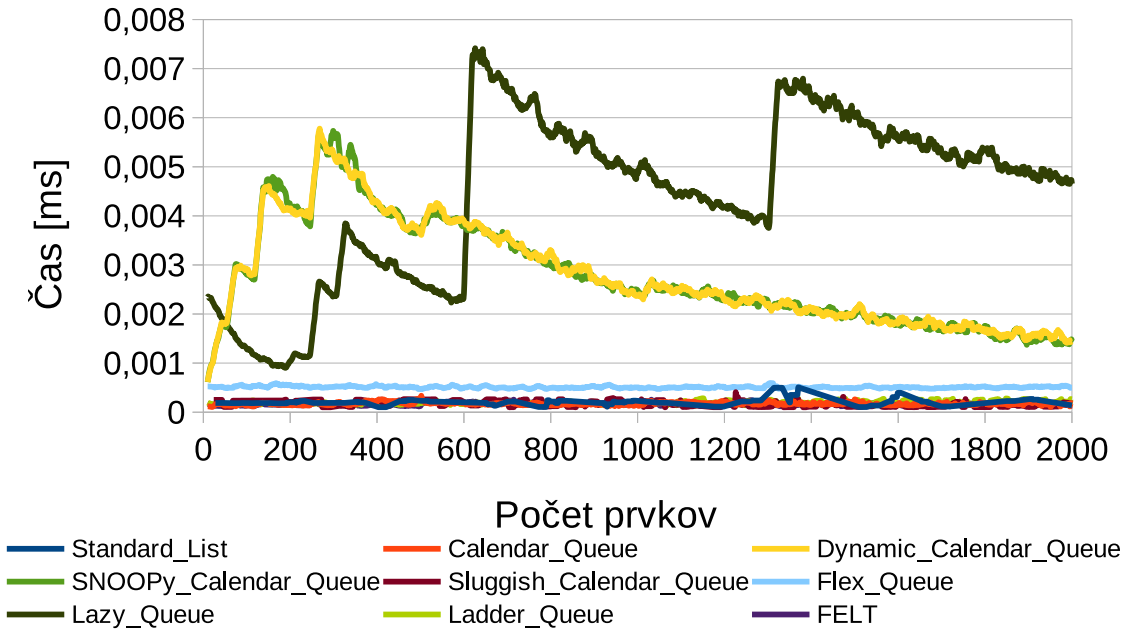
Na obrázku 4.22 je možné vidieť priebeh trvania operácie vloženia prvku pre všetky implementácie. Pre prehľadnosť tohto grafu bolo nutné dáta upraviť pomocou kĺzavého priemeru. Je možné vidieť, že väčšina implementácií skutočne dosahuje konštantnú zložitosť vloženia prvku, až na implementácie flex queue a štandardného zoznamu.



Obr. 4.22: Porovnanie času operácie vloženia prvku pre jednotlivé implementácie po vyhladení pomocou kĺzavého priemeru

Z porovnania jednotlivých implementácií je možné povedať, že najlepšie implementácie sú lazy queue a calendar queue. Hneď po nich nasledujú implementácie dynamic calendar queue a SNOOPy calendar queue. O niečo málo horšie sú následne ladder queue, FELT a sluggish calendar queue. Ale keďže rozdiely medzi týmito implementáciami sú pomerne malé, je možné predpokladať aj istý vplyv chyby merania. Preto sa dá tvrdiť, že tieto implementácie sú rovnocenné čo sa týka času potrebného na vloženie jedného prvku. Dôležité je ale podotknúť, že tieto merania prebehli za používania rovnomerného generovania udalostí.

Na obrázku 4.23 je možné vidieť priebeh trvaniaoperácie výberu prvku pre všetky implementácie. Je možné sledovať konštantnú zložitosť pre všetky implementácie s výnimkou implementácie lazy queue. Tá má lineárnu zložitosť preto, lebo kontroly vykonávané na zistenie nutnosti zmeny veľkosti prechádzajú hornú polovicu ďalekej budúcnosti, ktorá sa pochopiteľne s počtom udalostí zväčšuje. Hranice zmeny veľkosti je možné vidieť na obrázku ako skokové zmeny času vykonávania. Taktiež bolo nutné pre prehľadnosť tohto grafu odstrániť poslednú hodnotu u flex queue, keďže spôsobovala príliš veľkú hodnotu a zatienila zvyšok grafu, viz obrázok 4.17.



Obr. 4.23: Porovnanie času operácie výberu prvku pre jednotlivé implementácie po vyhladení pomocou kĺzavého priemeru

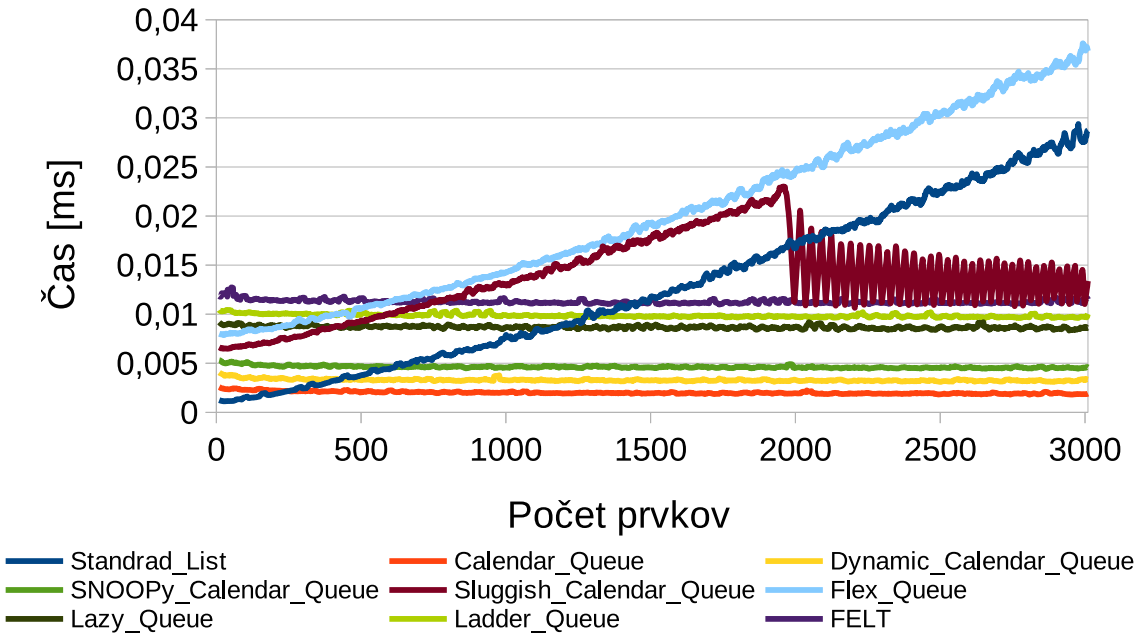
Pre lepšie porovnanie efektivity prioritných front s konštantnou zložitosťou je v tabuľke 4.4 možné vidieť priemerné trvanie operácie vloženia a výberu udalosti. Operácie bez konštantnej zložitosti nemajú v tabuľke uvedenú číselnú hodnotu.

Implementácia	Trvanie operácie vloženia prvku [μs]	Trvanie operácie výberu prvku [μs]
Lineárny zoznam	N/A	0,128
Calendar queue	1,373	0,170
Dynamic calendar queue	2,363	1,151
SNOOPy calendar queue	3,344	1,162
Sluggish calendar queue	11,036	0,216
Flex queue	N/A	0,890
Lazy queue	0,337	N/A
Ladder queue	7,092	0,242
FELT	8,100	0,196

Tabuľka 4.4: Priemerné trvanie operácií vloženia a výberu prvku pre jednotlivé implementácie pri použití rovnomerného generovania udalostí.

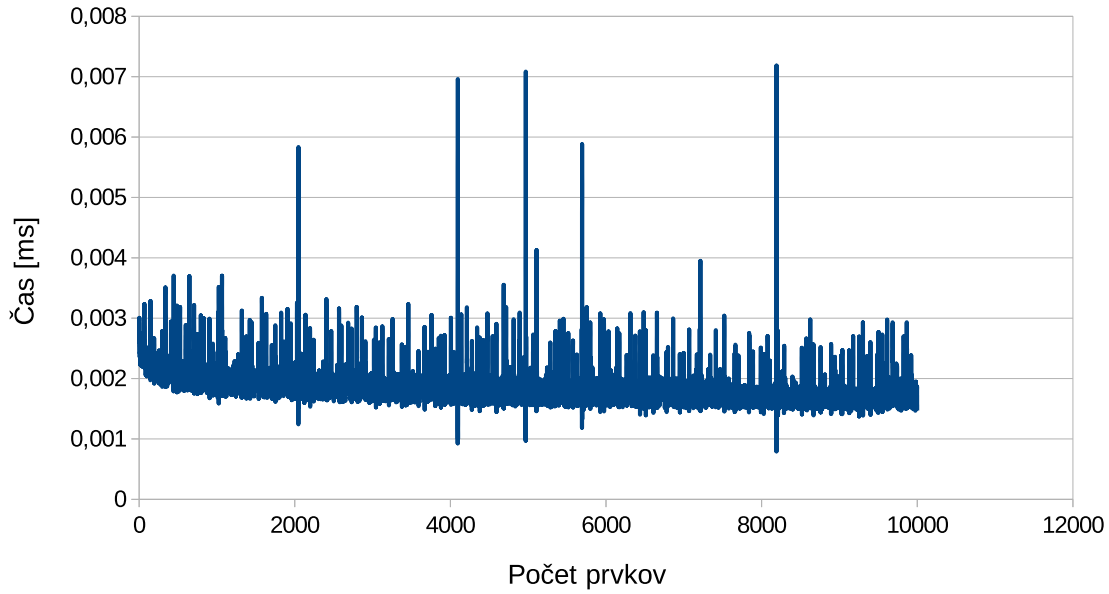
4.3.11 Porovnanie efektivity implementácií - trojuholník

Všetky grafy doposiaľ uvedené vo vyšších sekciách boli namerané pri použití rovnomerného generovania udalostí. Viacej implementácií je citlivých na udalosti, ktoré nie sú rovnomerne rozdelené v čase. Na obrázku 4.24 je možné vidieť graf trvania operácie vloženia prvku pre všetky implementácie pri použití generátora typu trojuholník. Na prvý pohľad je možné vidieť, že nedošlo k závažnej zmene od rovnomerného rozloženia. Najviditeľnejšou zmenou je mierne zhoršenie implementácie lazy queue, ale ostatné implementácie nedosahujú vážnejšie zmeny.



Obr. 4.24: Porovnanie času operácie vloženia prvku pre jednotlivé implementácie po vyhladení pomocou kĺzavého priemeru za použitia generátoru typu trojuholník

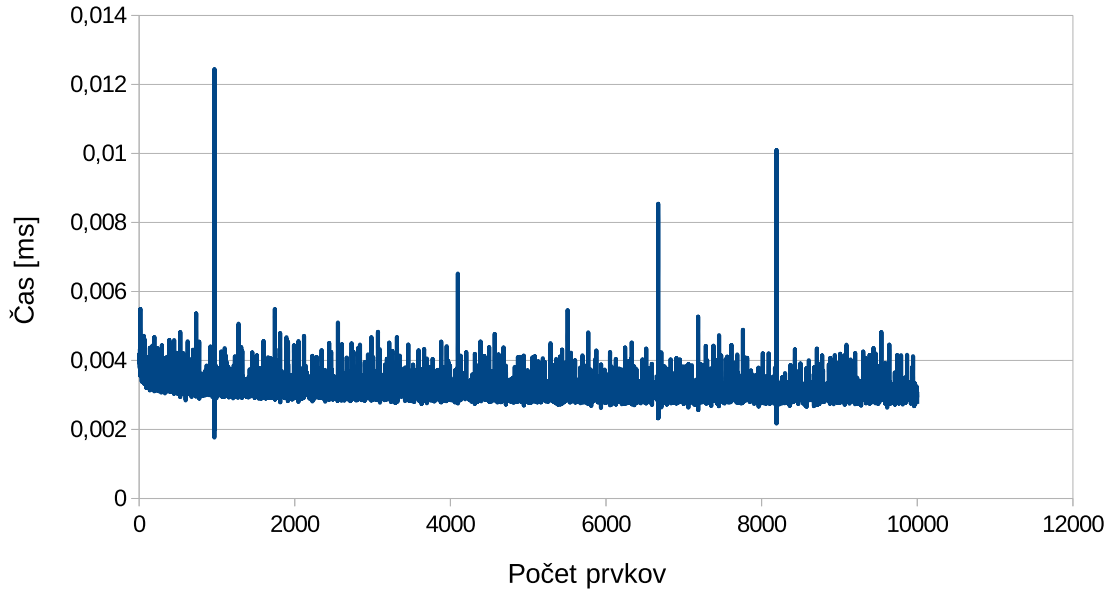
Jedna z viac citlivých implementácií na nerovnomerné rozloženie je práve základná verzia calendar queue. Na obrázku 4.25 je možné vidieť trvanie operácie vloženia prvku tejto implementácie za použitia generátora typu trojuholník. Skutočne je možné postrehnúť, že na rozdiel od priebehu na obrázku 4.6 obsahuje tento priebeh viac špičiek. To je práve spôsobené vkladaním udalostí do plnších kýblikov, keďže vloženie udalosti do kýblika má lineárnu zložitosť.



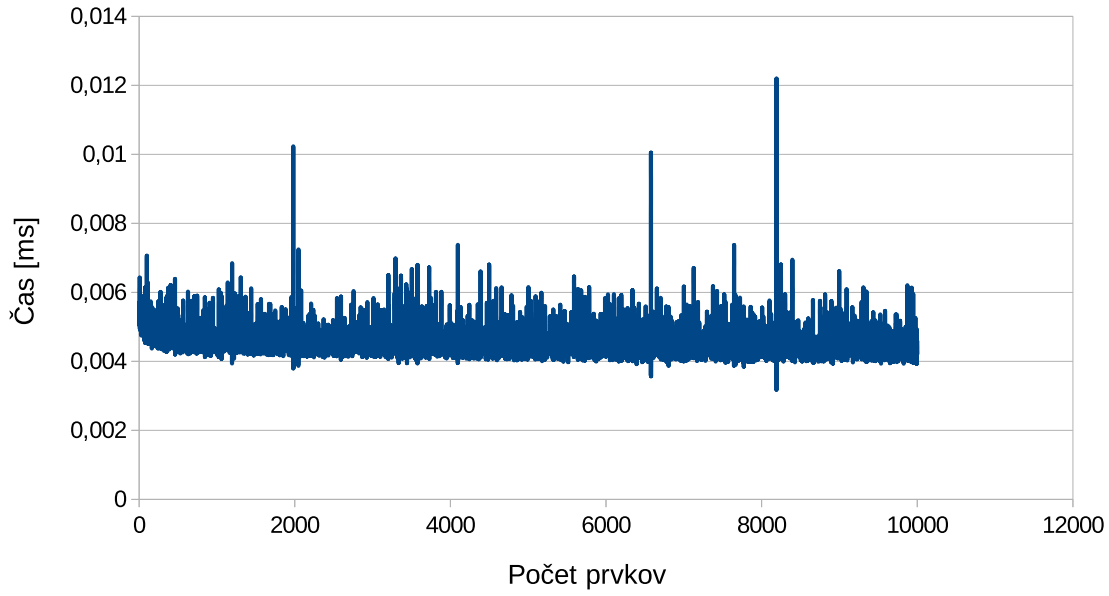
Obr. 4.25: Nameraný čas trvania operácie vloženia prvku do kalendára typu calendar queue v závislosti na počte prvkov v kalendári pri použití generátora typu trojuholník

Dynamic calendar queue sa snaží vylepšiť problém základnej implementácie calendar queue. Priebeh trvania vloženia prvku tejto implementácie je možné vidieť na obrázku 4.26. Na prvý pohľad je špičiek menej a zároveň sú menšie. Na základe tohto grafu je možné povedať, že tento problém je čiastočne zredukovaný, ale lepšie výsledky dosahuje dynamic calendar queue v rovnomernom rozložení udalostí.

Okrem dynamic calendar queue rieši tento problém aj SNOOPy calendar queue. Ten narozdiel od prezerania rozloženia medzi udalosťami, používa štatistický výpočet šírky kýblika. Priebeh je možné vidieť na obrázku 4.27. Na prvý pohľad je možné povedať, že táto implementácia dosahuje veľmi podobné výsledky ako dynamic calendar queue. Takže skutočne je možné povedať, že implementácie SNOOPy calendar queue a dynamic calendar queue dosahujú podstatne lepšie výsledky ako základná implementácia calendar queue pri situáciach, keď nie sú udalosti rovnomerne rozložené v čase.



Obr. 4.26: Nameraný čas trvania operácie vloženia prvku do kalendára typu dynamic calendar queue v závislosti na počte prvkov v kalendári pri použití generátora typu trojuholník

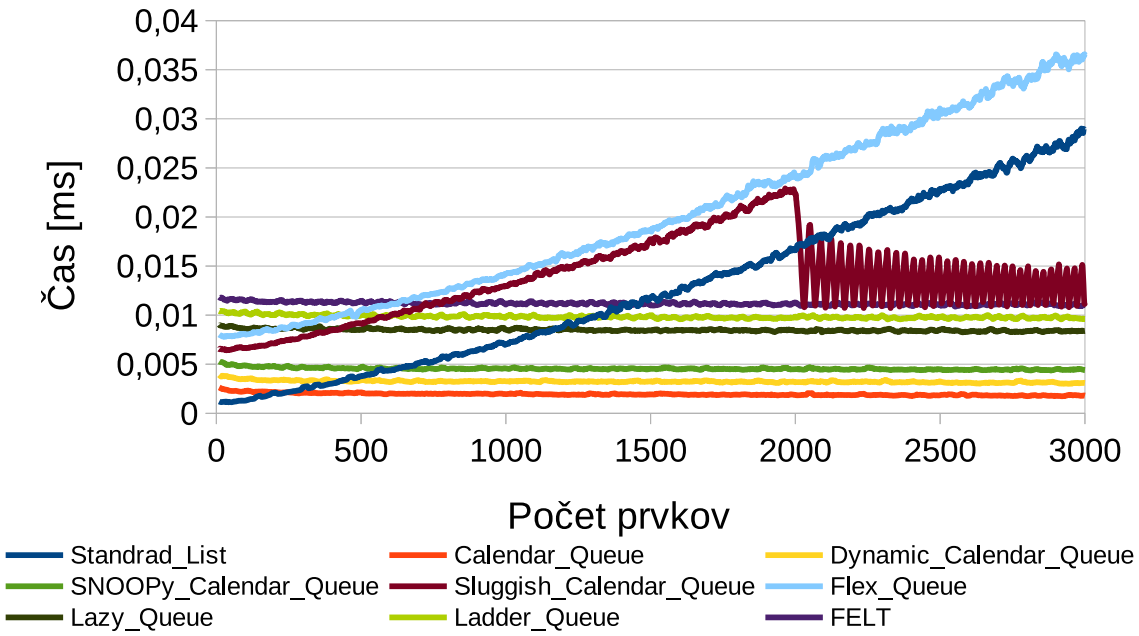


Obr. 4.27: Nameraný čas trvania operácie vloženia prvku do kalendára typu SNOOPy calendar queue v závislosti na počte prvkov v kalendári pri použití generátora typu trojuholník

4.3.12 Porovnanie efektivity implementácií - ťava

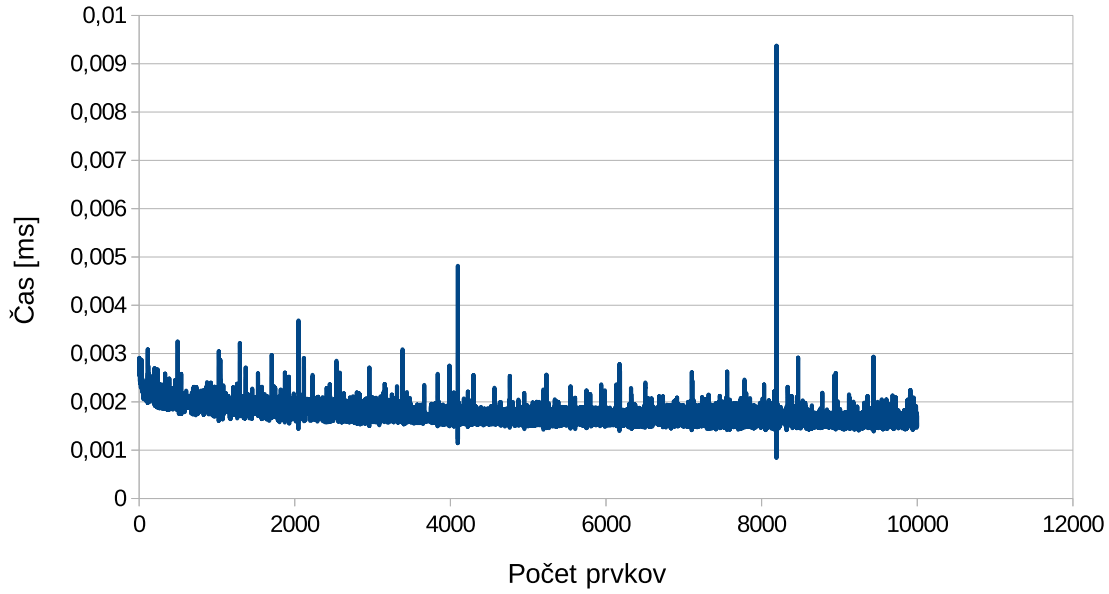
Priebeh trvania operácie vloženia prvku pri použití generátora typu ťava je možné vidieť na obrázku 4.28. Časová zložitosť je opäť veľmi podobná ako pri použití trojuholníkového rozloženia. Na základe tohto môžme povedať, že mnou zvolené rozloženia udalostí nemajú až

také rozličné vlastnosti v jednotlivých implementáciách. Všetky implementácie si zachovali svoje zložitosti aj pri použití rôznych generátorov udalostí.

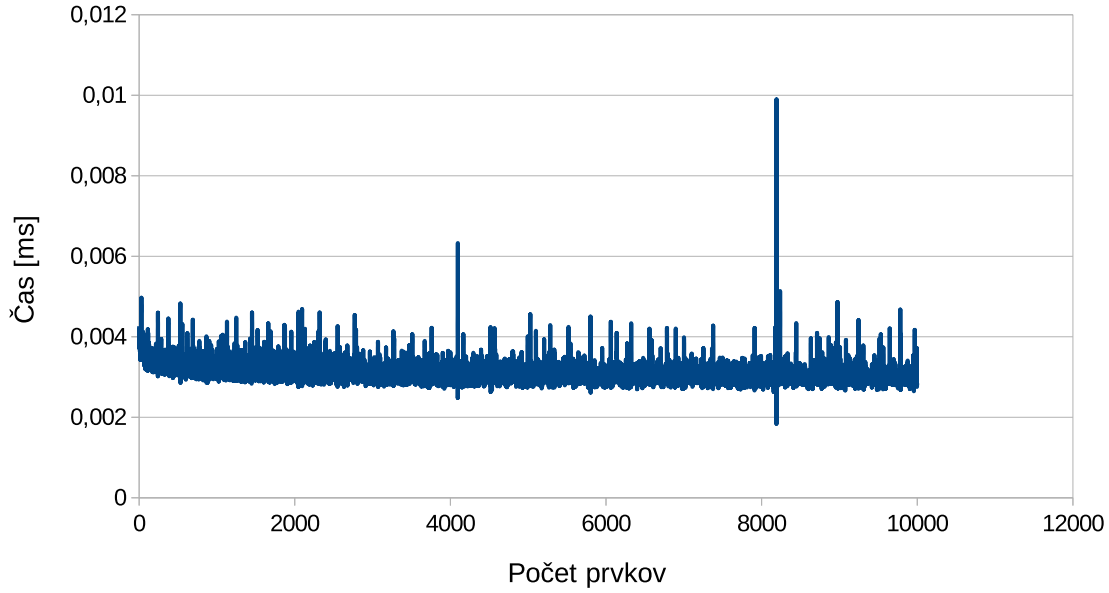


Obr. 4.28: Porovnanie času operácie vloženia prvku pre jednotlivé implementácie po vyhladení pomocou kĺzavého priemeru za použitia generátoru typu ťava

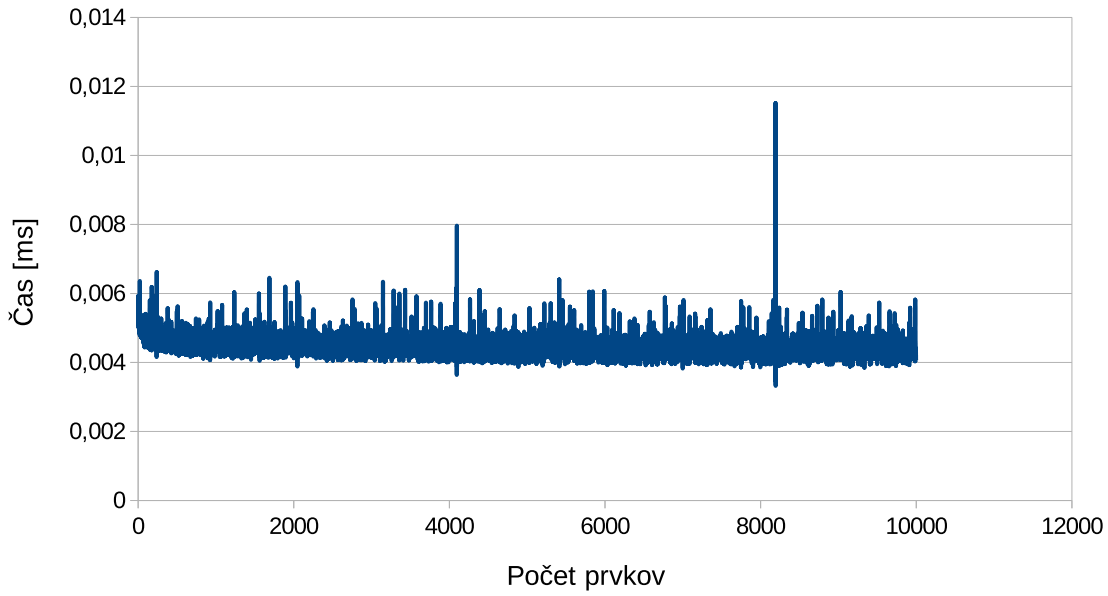
Na obrázku 4.29 vidíme priebeh implementácie calendar queue. Na obrázku 4.30 je zase vidieť priebeh implementácie dynamic calendar queue a na obrázku 4.31 je priebeh implementácie SNOOPy calendar queue. Všetky tieto priebehy sú namerané za použitia generátoru typu ťava. Priebehy sú skoro totožné s použitím rovnomerného rozloženia udalostí. Pri tomto rozložení udalostí ale dosahuje calendar queue o niečo horšie vlastnosti pri výbere udalostí, ako je možné vidieť na obrázku 4.32. Pri porovnaní s obrázkom 4.7 je možné vidieť väčší počet špičiek v priebehu získanom pri použití generátoru typu ťava. Ale aj napriek zvýšenému počtu špičiek sa dá povedať, že časová zložitosť implementácie zostáva konštantná.



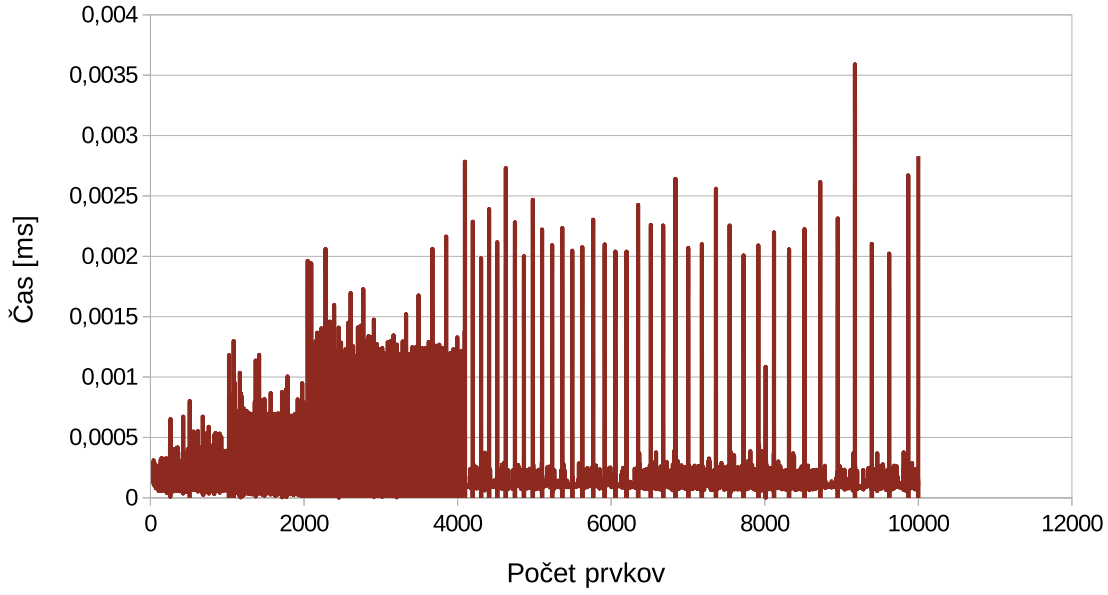
Obr. 4.29: Nameraný čas trvania operácie vloženia prvku do kalendára typu calendar queue v závislosti na počte prvkov v kalendári pri použití generátora typu ťava



Obr. 4.30: Nameraný čas trvania operácie vloženia prvku do kalendára typu dynamic calendar queue v závislosti na počte prvkov v kalendári pri použití generátora typu ťava



Obr. 4.31: Nameraný čas trvania operácie vloženia prvku do kalendára typu SNOOPy calendar queue v závislosti na počte prvkov v kalendári pri použití generátora typu ťava



Obr. 4.32: Nameraný čas trvania operácie výberu prvku z kalendára typu calendar queue v závislosti na počte prvkov v kalendári pri použití generátora typu ťava

4.4 Zhrnutie výsledkov

Z nameraných výsledkov je skutočne vidieť, že implementácie skutočne dosahujú zložitosti, ktoré sú uvedené v kapitole 2. Implementácia flex queue je porovnateľná zo štandardným zoznamom, ale dosahuje o niečo horšie výsledky. Implementácia flex queue by mohla dosahovať lepších výsledkov, pri použití iných parametrov, keďže táto implementácia je parametrizovateľná.

Sluggish calendar queue vyšiel z implementácií s konštantnou zložitosťou najhoršie. Štruktúra tejto implementácie nenarastá tak rýchlo ako iné ale vyžaduje o niečo viac času na vloženie udalosti. Po nej nasledujú implementácie FELT a Ladder queue. Čas trvania vloženia prvku u týchto implementácií je skoro totožný. Pre vloženie prvku dopadli najlepšie implementácie calendar queue, dynamic calendar queue, SNOOPy calendar queue a lazy queue. Keďže lazy queue dosahuje dosť zlých výsledkov pre výber prvku, tak môžme povedať že implementácie calendar queue, dynamic calendar queue a SNOOPy calendar queue dosahujú najlepšie výsledky zo všetkých zvolených implementácií.

Kapitola 5

Záver

Cieľom práce bolo navrhnúť, implementovať, otestovať a zhodnotiť knižnicu implementujúcu kalendár udalostí. Táto knižnica zahŕňa osem rôznych implementácií kalendára udalostí plus jednoduchú implementáciu pomocou lineárneho zoznamu. Rôzne spôsoby implementácie vychádzajú primárne z implementácie calendar queue, ktorú taktiež zahŕňa. Knižnica je navrhnutá tak, aby bola jednoducho a ľahko rozšíriteľná, a aby jej rozšírenie nespôsobilo žiadne zmeny v jej používaní. Knižnica bola riadne otestovaná pomocou testovacej aplikácie. V rámci testov boli vykonané merania výkonnosti jednotlivých implementácií.

V meraniach najlepšie dopadla implementácia calendar queue a jej modifikácie dynamic calendar queue a SNOOPy calendar queue. Podobne dobré výsledky dosahujú aj implementácie Ladder queue, FELT a Sluggish calendar queue. Lazy queue dosahuje dobré výsledky pri vložení prvku, ale pri výbere prvku dosahuje najhoršie výsledky. Štandardný zoznam a flex queue dosahuje dobré výsledky pre malí počet udalostí, ale s rastúcim počtom udalostí sa ich efektivita znižuje.

Knižnica je jednoduchá na použitie, avšak za cenu rýchlosti, keďže každé volanie metódy s prioritnej fronty je vykonané prostredníctvom virtuálnej tabuľky metód. Vo veľkej časti simulácií by to ale nemal byť vážny problém. Knižnica je vhodná na použitie vo výuke, keďže je možné jednoducho vytvoriť ukážkové príklady, ktoré môžu využívať rôzne implementácie kalendára bez nutnosti opätovného prekladu pri zmene implementácie.

Do budúcnosti by bolo možné pridať niektoré stromové implementácie kalendára udalostí. Následne by bolo vhodné optimalizovať reinicializáciu kalendára udalostí pre všetky kombinácie, aby dochádzalo k opätovnému využívaniu čo najväčšej časti alokovaných štruktúr, namiesto cyklu výberu zo starej prioritnej fronty a následnému vloženiu do novej. Taktiež by bolo možné umožniť skopírovať kalendár udalostí.

Literatúra

[1] Brown, R. Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem. *Communications of the ACM*. Október 1988, zv. 31, č. 10, s. 1220 – 1227.

[2] Guanhua Yan a Eidenbenz, S. Sluggish Calendar Queues for Network Simulation. In: *14th IEEE International Symposium on Modeling, Analysis, and Simulation*. 2006, s. 127–136. DOI: 10.1109/MASCOTS.2006.46.

[3] Honzík, J. M. *Algoritmy, Studijní opora*. FIT VUT v Brně, január 2018 [cit. 2021-05-10].

[4] Hui, T. C.-K. a Thng, I. L.-J. FELT: A Far Future Event List Structure Optimized for Calendar Queues. *SIMULATION*. 2002, zv. 78, č. 6, s. 343–361. DOI: 10.1177/0037549702078006573. Dostupné z: https://doi.org/10.1177/0037549702078006573.

[5] Kah Leong Tan a Li-Jin Thng. SNOOPy Calendar Queue. In: *2000 Winter Simulation Conference Proceedings (Cat. No.00CH37165)*. 2000, sv. 1, s. 487–495 vol.1. DOI: 10.1109/WSC.2000.899756.

[6] Oh, S. a Ahn, J. Dynamic Calendar Queue. In: *Proceedings of the 32nd Annual Simulation Symposium*. Apríl 1999, s. 20–25.

[7] Peringer, P. *Modelování a simulace, Studijní opora*. FIT VUT v Brně, december 2012 [cit. 2021-01-08].

[8] Ronngren, R., Riboe, J. a Ayani, R. Lazy Queue: an efficient implementation of the pending-event set. In: *Proceedings of the 24th Annual Simulation Symposium*. Los Alamitos, CA, USA: IEEE Computer Society, Apríl 1991, s. 194,195,196,197,198,199,200,201,202,203,204. DOI: 10.1109/SIMSYM.1991.151506. Dostupné z: https://doi.ieeecomputersociety.org/10.1109/SIMSYM.1991.151506.

[9] Ross, S. M. *Simulation, Fourth Edition*. USA: Academic Press, Inc., 2006. ISBN 0125980639.

[10] Tang, W., Goh, R. a Thng, I. Ladder queue: An O (1) priority queue structure for large-scale discrete event simulation. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*. Júl 2005, zv. 15, s. 175–204. DOI: 10.1145/1103323.1103324.

[11] Zhang, Yifan. *FlexQueue: Simple and Efficient Priority Queue for System Software*. UWSpace, 2018. Dostupné z: http://hdl.handle.net/10012/13316.