ŽILINSKÁ UNIVERZITA V ŽILINE

Fakulta riadenia a informatiky

Bakalárska práca

Študijný odbor: **Informatika**

**Pavol Šurin**

**Experimentálne porovnanie prioritných frontov**

Vedúci práce: **Ing. Peter Jankovič PhD.**

Reg.č. X/2021 Máj 2021

**Čestné vyhlásenie**

Prehlasujem, že som bakalársku prácu *Experimentálne porovnanie prioritných frontov* vypracoval samostatne pod vedením Ing. Peter Jankovič PhD, a uviedol v nej všetky použité literárne a iné odborné zdroje v súlade s právnymi predpismi, vnútornými predpismi Žilinskej univerzity a vnútornými aktmi riadenia Žilinskej univerzity a Fakulty riadenia a informatiky.

V Žiline, dňa .............. ..........................................

Meno Priezvisko

**Poďakovanie**

Na tomto mieste by som chcela poďakovať vedúcemu diplomovej práce.... za cenné pripomienky a odborné rady, ktorými prispel k vypracovaniu tejto XXXXX práce. Taktiež ďakujem môjmu ... . Zároveň ďakujem mojej rodine a priateľom za ich nekonečnú podporu a trpezlivosť.

**ABSTRAKT V ŠTÁTNOM JAZYKU**

PRIEZVISKO, Meno: Názov témy záverečnej práce. [Bakalárska práca]. – Žilinská Univerzita v Žiline. Fakulta riadenia a informatiky; Katedra informatiky. – Školiteľ/Vedúci: Ing. Oľga Chovancová – Stupeň odbornej kvalifikácie: magister. – Mesto: FRI UNIZA, 20xx. Počet strán

(napr. 35 s.)

Cieľom bakalárskej práce je ...

**Kľúčové slová**: kľúčové slovo 1, kľúčové slovo 2, kľúčové slovo 3, ... kľúčové slovo x.

**ABSTRAKT V CUDZOM JAZYKU**

The aim of the thesis is to .......

**Key words:** key word 1, key word 2, key word 3, ... , key word x.

**Obsah**

Obsah

[Zoznam obrázkov v](#_Toc67307248)

[Zoznam tabuliek vi](#_Toc67307249)

[Zoznam skratiek vii](#_Toc67307250)

[Úvod 1](#_Toc67307251)

[1 Teoretická implementácia prioritného frontu 2](#_Toc67307252)

[1.1 Abstraktná údajová štruktúra prioritný front 2](#_Toc67307253)

[1.2 Spôsoby implementácie prioritného frontu 3](#_Toc67307254)

[Binárna halda 3](#_Toc67307255)

[Binomiálna halda 4](#_Toc67307256)

[Fibonacciho halda 6](#_Toc67307257)

[Párovacia halda 6](#_Toc67307258)

[1.3 Zhrnutie teoretických časových náročnosti operácií 7](#_Toc67307259)

[2 Praktická implementácia prioritného frontu 8](#_Toc67307260)

[2.1 Triedy reprezentujúce prvky v prioritnom fronte 8](#_Toc67307261)

[2.1.1 PriorityQueueItem<Priority, Data> 8](#_Toc67307262)

[2.1.2 BinaryTreeItem<Priority, Data> 9](#_Toc67307263)

[2.1.3 BinaryTreeItemWithAncestor<Priority, Data> 11](#_Toc67307264)

[2.2 Triedy reprezentujúce prioritný front 11](#_Toc67307265)

[2.2.1 PriorityQueue<Priority, Data> 11](#_Toc67307266)

[2.2.2 LazyBinomialHeap<Priority, Data> 12](#_Toc67307267)

[2.2.3 BinomialHeap<Priority, Data> 14](#_Toc67307268)

[2.2.4 FibonacciHeap<Priority, Data> 16](#_Toc67307269)

[3 Návrh testov pre overenie výkonnosti implementácií prioritných frontov 17](#_Toc67307270)

[4 Vyhodnotenie výsledkov testov implementácií prioritných frontov 18](#_Toc67307271)

[Zoznam použitej literatúry 19](#_Toc67307272)

[Zoznam príloh 20](#_Toc67307273)

# Zoznam obrázkov

# Zoznam tabuliek

# Zoznam skratiek

# Úvod

Cieľom bakalárskej/diplomovej práce je ...... Náplňou prvej časti práce je oboznámenie sa s problematikou ............... Ďalšia časť sa zaoberá rozborom existujúcich ............... Ďalšia kapitola popisuje spôsob implementácie ................. Posledná časť práce ............................

Napísať o tom čo to zlepší, alebo aký je zmysel toho čo robíte alebo aký to ma dopad.

Postup práce:

1. Oboznámenie sa s problematikou .........
2. Rozbor existujúcich .......
3. Implementácia v....
4. Experimentálne porovnanie .....

# Teoretická implementácia prioritného frontu

V tejto časti sa budeme venovať prioritnému frontu ako abstraktnej údajovej štruktúry a jeho jednotlivým implementáciám. Cieľom tejto časti je popísanie rôznych spôsobov implementovania prioritného frontu, rovnako ako aj časové náročnosť ich jednotlivých operácií.

Prioritný front predstavuje abstraktnú údajovú štruktúru, ktorá uchováva dáta spolu s im priradenou prioritou. Jednotlivé dáta sú z prioritného frontu vyberané na základe priority, kde prvé sa vyberú dáta s najvyššou prioritou. Podporovanými operáciami a funkčnosťou sa prioritný front podobá údajovým štruktúra front a zásobník.

Prioritný front je možné na základe určovania priority implementovať dvoma spôsobmi, a to ako min-heap alebo max-heap. Min-heap prioritný front predstavuje prioritný front, kde s rastúcou hodnotou priority klesá priorita prvku. Pri max-heap prioritnom fronte s rastúcou hodnotou priority rastie priorita prvku.

Nasledujúce návrhy a implementácie pracujú s min-heap prioritnými frontami.

Každý popisovaný prioritný front podporuje nasledovné operácie:

**Operácia vlož(Q, K, X) -> void**

Vloží do prioritného frontu Q dáta X, ku ktorým je priradená priorita K

**Operácia vyber(Q) -> X**

Vyberie a odstráni z prioritného frontu Q dáta X, ktoré mali priradenú najvyššiu prioritu

**Vráť-minimum(Q) -> X**

Vráti z prioritného frontu Q dáta X, ktoré majú priradenú najvyššiu prioritu

**Spoj(Q1, Q2) -> Q**

Spojí prioritné fronty Q1 a Q2 do jedného frontu Q, ktorý následne vráti

**Zmeň-prioritu(Q, X, K) -> void**

Zmení prioritu prvku X na hodnotu K.

## Binárna halda

Jedná sa o implementáciu prioritného frontu pomocou úplného binárneho stromu. Tento strom musí dodržiavať vlastnosť haldy[[1]](#footnote-2). Binárny strom je možne efektívne implementovať pomocou implicitného zoznamu.



**Operácia vlož(Q, K, X) -> void**

Prvok X je vložený spolu s priradenou prioritou K na prvé voľne miesto v binárnom strome, následne je vymieňaný s priamym predchodcom, pokiaľ nie je splnená vlastnosť haldy. Keďže maximálny počet predkov prvku, s ktorými môžeme prvok vymeniť je a časová náročnosť výmeny je , amortizovaná časová náročnosť samotnej operácie je . Ak je strom implementovaný pomocou poľa, môže počas vkladania prvku dôjsť k rozšíreniu poľa, čo by zvýšilo časovú náročnosť operácie v najhoršom prípade na . 

**Operácia vyber(Q) -> X**

Prvok X s najväčšou prioritou K, ktorý sa nachádza na začiatku zoznamu je vymenený s posledným prvkom a odobraný zo zoznamu. Následne je prvok na začiatku zoznamu postupne vymieňaný s jeho potomkami, pokiaľ nebude splnená vlastnosť haldy. Prvok sa vymieňa vždy s tým potomkom, ktorý ma najvyššiu prioritu. Nakoniec sa odstráni posledný prvok v zozname a je vrátená hodnota jeho dát. Časová náročnosť tejto implementácie je , keďže maximálny počet postupných výmen s potomkami prvku je .



**Operácia vráť-minimum(Q) -> X**

Vráti hodnotu prvku X s najväčšou prioritou K, ktorý sa nachádza na začiatku zoznamu. Prístup k tomuto prvku je konštantný, preto časová náročnosť danej operácie je .

**Operácia spoj(Q1, Q2) -> Q**

Jednotlivé prvky z oboch háld sú vložené do novej haldy a obe haldy sú zničené. Nová halda je následne utriedená, aby dodržiavala vlastnosť haldy. Časová náročnosť utriedenia prvku tak, aby dodržal vlastnosť haldy je , čo pre n prvkov znamená, že časová náročnosť operácie Spoj je . Dá sa však dokázať, že amortizovaná časová náročnosť je (viz. Cormen a kol., 2009, s. 156).



**Operácia zmeň-prioritu(Q, X, K) -> void**

Prvku X je zmenená priorita na hodnotu K. Ak bola jeho priorita zvýšená, je vymieňaný s jeho priamym predchodcom, pokiaľ nie je splnená vlastnosť haldy. Ak bola znížená, prvok I sa postupne vymieňa s tým z jeho potomkov, ktorý ma najvyššiu prioritu, pokiaľ nebude splnená vlastnosť haldy. V oboch prípadoch je maximálny počet výmen, čo nám dáva časovú zložitosť .

## Lenivá binomiálna halda

Jedná sa o prioritný front, ktorý vznikol ako relaxácia binomiálnej haldy a tvorí základ variácie binomiálnej haldy, Fibonacciho haldy a párovacej haldy na základe priority. Využíva cyklický zoznam binomiálnych stromov, kde každý strom vzniká prepojením dvoch stromov s rovnakým stupňom. Binomálne stromy môžu byť reprezentované ako mnohonásobné stromy alebo ako binárne stromy. To ma za následok, že pri binárnom strome je upravené pravidlo tak, že priorita prvku je väčšia alebo nanajvýš rovná priorite ľubovoľného prvku, ktorý sa nachádza na pravej chrbtici ľavého potomka tohto prvku. Táto verzia binomiálnej haldy nerekonštruuje zoznam binomiálnych stromov za každou operáciou meniacou prioritný front, ale až po odstránení minimálneho prvku, čo značne zlepšuje časy operácií vlož a spoj.

Rekonštrukcia spočíva v prepojovaní binomiálnych stromov rovnakého stupňa použitím implicitného poľa. Dva binárne stromy sa prepoja tak, že binárny strom, ktorého koreň ma nižšiu prioritu sa stane prvým potomkom prvku s vyššou prioritou, ktorému sa zvýši stupeň. Ak sa pri rekonštrukcii spájajú prvky pokiaľ neexistuje dvojica prvkov s rovnakým stupňom, hovoríme o multipass lenivej binomiálnej halde. Ak sa dvojica prvkov spojí a už sa ďalej nespája, ide o singlepass lenivú binomiálnu haldu.



**Operácia vlož(Q, K, X) -> void**

Je vytvorená nová binomiálna halda Q1, ktorá obsahuje len jeden prvok X s prioritou K. Táto halda je následne spojená s prioritným frontom Q. Časová náročnosť tejto operácie je závislá len od časovej náročnosti pripojenia prvku X do zoznamu, čo má konštantnú časovú zložitosť. Ide teda o časovú náročnosť .

**Operácia vyber(Q) -> X**

V zozname binomiálnych stromov je vybraný ten binomiálny strom, ktorého koreň ma najväčšiu prioritu. Tento strom je izolovaný a potomkovia koreňa tohto stromu sú spojení so zoznamom binomiálnych stromov a prioritný front je zrekonštruovaný. Prvok s najvyššou je následne odstránený. Hľadanie minimálneho prvku má časovú náročnosť , to však môžeme urýchliť vedením smerníka k minimálnemu prvku, čo nám zefektívni operáciu na , budeme však musieť pri každej operácií upravujúcej prioritný front tento smerník aktualizovať. Spájanie má časovú náročnosť a rekonštrukcia . Celková časová náročnosť operácie je teda .







**Operácia spoj(Q1, Q2) -> Q**

K zoznamu binomiálnych stromov Q1 sú pripojené binomiálne stromy prioritného frontu Q2, čo je možné vykonať v konštantnom čase. Náročnosť tejto operácie je teda .

**Operácia zmeň-prioritu(Q, X, K) -> void**

Prvku X je zmenená priorita. Ak sa priorita zvýšila, prvok je vystrihnutý a pridaný k zoznamu binomiálnych stromov. Toto má časovú náročnosť . V prípade, že sa priorita znížila, postupne sú potomkovia prvku, ktorí majú vyššiu prioritu ako tento prvok, odstraňovaní. Tento postup nám dáva časovú náročnosť . Pri zmene priority ale nastáva značná deformácia binomiálneho stromu, čo ma za následok postupné zníženie efektívnosti rekonštrukcie a tým aj operácie vyber.



## Binomiálna halda

Ide o prioritný front implementovaný pomocou zreťazeného zoznamu binomiálnych stromov[[2]](#footnote-3). Pôvodná verzia binomiálnej haldy, ktorú navrhol J. Vulleimin[4], využíva zoznam binomiálnych stromov zoradený vzostupne podľa ich stupňa tak, že v zozname je vždy najviac jeden strom s daným stupňom. Táto variácia požaduje, aby pri operáciách boli prvky vkladané na správne miesto v zozname. Nami popisovaná štruktúra predstavuje lenivú variáciu binomiálnej haldy, čo uvoľňuje od požiadavky pôvodnej verzie na nutnú zoradenosť binomiálnych stromov.

**Operácia vlož(Q, K, X) -> void**

Je vytvorená nová binomiálna halda Q1, ktorá obsahuje len jeden prvok X s prioritou K. Táto halda je následne spojená s prioritným frontom Q a prioritný front je rekonštruovaný. Vidíme, že časová náročnosť tejto operácie je závislá len od časovej náročnosti rekonštrukcie, keďže pripojenie prvku X do zoznamu má konštantnú časovú zložitosť. Ide teda o časovú náročnosť .

**Operácia vyber(Q) -> X**

Táto operácia je rovnaká ako pri lenivej binomiálnej halde.

**Operácia vráť-minimum(Q) -> X**

V zozname binomiálnych stromov nájdeme a vrátime koreň binomiálneho stromu, ktorý ma najväčšiu prioritu. Táto operácia má časovú náročnosť , čo však môže byť urýchlené vedením smerníka k minimálnemu prvku, čo nám zefektívni operáciu na časovú náročnosť .

**Operácia spoj(Q1, Q2) -> Q**

Vytvoríme novú binomiálnu haldu Q, ktorá v sebe obsahuje zoznam vytvorený spojením zoznamov binomiálnych stromov z binomiálnych háld Q1 a Q2, čo je možné vykonať v konštantnom čase. Túto binomiálnu haldu následne zrekonštruujeme. Náročnosť tejto operácie je teda .

**Operácia zmeň-prioritu(Q, X, K) -> void**

Prvku X je zmenená priorita. Ak sa priorita zvýšila, prvok je vymieňaný so svojím priamym predchodcom pokiaľ nebude platiť pravidlo haldy. Ak sa priority prvku znížila, prvok je postupne vymieňaný s tým z jeho potomkov, ktorý má najvyššiu prioritu, pokiaľ nie je splnené pravidlo haldy. V oboch prípadoch je maximálny počet výmen rovný stupňu koreňa binomiálneho stromu v ktorom sa nachádza prvok X, čo je najviac To nám dáva časovú zložitosť .





## Fibonacciho halda

Fibonacciho halda je implementácia prioritného frontu, ktorá vznikla rozšírením lenivej variácie binomiálnej haldy o efektívne zníženie priority v amortizovanom čase. Toto je dosiahnuté sériou rezov po úprave priority prvku, čo zmierni deformáciu stromu[2]. Rovnako ako pri binomiálnej halde je fibonacciho halda reprezentovaná zoznamom binomiálnych stromov.

**Operácia vlož(Q, K, X) -> void**

Prvok X s prioritou K vložíme na ľubovoľné miesto v zozname. Ak sa jedná o prvok s najvyššou prioritou, aktualizujeme smerník k prvku s najvyššou prioritou. Časová náročnosť tejto operácie je teda .

**Operácia vyber(Q) -> X**

V zozname binomiálnych stromov je vybraný binomiálny strom s najvyššou prioritou. Tento strom je vyrezaný a potomkovia koreňa tohto stromu sú spojení so zoznamom binomiálnych stromov a prioritný front je zrekonštruovaný. Prvok s najvyššou je následne odstránený. Spájanie má časovú náročnosť a rekonštrukcia . Celková časová náročnosť operácie je teda .

**Vráť-minimum(Q) -> X**

V zozname binomiálnych stromov je vybraný koreň binomiálneho stromu s najvyššou prioritou. Operácia má časovú náročnosť .

**Spoj(Q1, Q2) -> Q**

Vytvoríme novú Fibonacciho haldu Q, ktorá v sebe obsahuje prepojený zoznám stromov z oboch háld, pričom haldy Q1 a Q2 zaniknú. Časová náročnosť operácie je závislá výlučne od rýchlosti prepojenia zoznamov oboch háld, čo je pri zreťazenom zozname .

**Operácia zmeň-prioritu(Q, X, K) -> void**

Prvku X je zmenená priorita. Ak sa priorita zvýšila, prvok je vystrihnutý, odznačený a pridaný k zoznamu binomiálnych stromov. Ak bol priamy predok prvku označený, bol odrezaný a postup opakujeme. Ak prvok nie je koreňom binomiálneho stromu a nebol označený, je označený. Toto má časovú náročnosť , ale v amortizovanom čase . V prípade, že sa priorita znížila, postupne sú potomkovia prvku, ktorí majú vyššiu prioritu ako tento prvok, odstraňovaní a je vykonávaný rovnaký postup s označeniami ako pri zvyšovaní prvku. Tento postup je možné brať, ako keby sme zvýšili prioritu všetkým potomkom prvku X, čo nám dáva časovú náročnosť .





## Párovacia halda

Párovacia halda predstavuje implementáciu prioritného frontu, ktorý je reprezentovaný viacuzlovým stromom, ktorý dodržuje vlastnosť haldy. Ide o samo-vyrovnávaciu alternatívu Binomiálnej haldy. Túto implementáciu je možne brať ako variáciu lenivej binomiálnej haldy, kde je umožnené neférové prepájanie binomiálnych stromov.

Podľa rekonštrukcie môžeme rozdeliť párovaciu haldu na multipass a twopass párovaciu haldu. Pri multipass párovacej halde je použitá metóda rekonštrukcie taká, že sa po sebe idúce prvky v zozname v pároch prepájajú, pokiaľ nevznikne jeden strom. Pri twopass párovacej halde sa rekonštrukcia vykoná tak, že sa po sebe idúce prvky spárujú a následne sa postupne prepoja do jedného stromu.

**Operácia vlož(Q, K, X) -> void**

Vytvoríme novú párovaciu haldu Q1, ktorej koreňom bude prvok X s prioritou K. Túto haldu prepojíme s haldou Q. Časová náročnosť tejto operácie je .

**Operácia vyber(Q) -> X**

Prvok tvoriaci koreň stromu je prvok s najvyššou prioritou. Po odstránení koreňa vznikne zoznam stromov, ktorí tvorili jeho potomkov. Tento zoznam je zrekonštruovaný a vytvorený strom je požitý ako nový strom. Časová náročnosť tejto operácie je ak je strom deformovaný, ale v amortizovanom čase ide o .

**Operácia vráť-minimum(Q) -> X**

Vráti prvok, ktorý sa nachádza v koreni stromu. Časová náročnosť tejto operácie je .

**Operácia spoj(Q1, Q2) -> Q**

Korene párovacích háld Q1 a Q2 prepojíme4 a vzniknutý koreň stanovíme ako koreň novej haldy Q. Keďže časová náročnosť operácie je závislá len od rýchlosti prepojenia, ktoré je vykonané v čase, tak aj celková náročnosť prepojenia je .

**Operácia zmeň-prioritu(Q, X, K) -> void**

Prvku X je zmenená priorita. Ak sa priorita zvýšila, prvok je vystrihnutý a spojený s koreňom stromu. Toto má časovú náročnosť . V prípade, že sa priorita znížila, je vytvorený zoznam z prvku X a jeho potomkov a tento zoznam je zrekonštruovaný. Vytvorený strom je pripojený na pôvodné miesto prvku X. Toto nám dáva časovú náročnosť v amortizovanom čase alebo v najhoršom prípade.

Príklady:





## Párovacia halda na základe úrovní

Párovacia halda na základe úrovní je vytvorená ako variácia lenivej binomiálnej haldy, ktorá upravuje úrovne jednotlivých binomiálnych stromov po zmene priority prvku tak, aby bola zachovaná efektívnosť prepájania.

Je nutné zaviesť pojem i, j – prvku[5]. Nazvime prvého potomka prvku ako ľavého potomka, pravého súrodenca ako pravého potomka a ľavého súrodenca, alebo ak ten neexistuje tak predchodcu ako priameho predchodcu. Takto sme zobrazili binomiálny strom pomocou binárneho stromu. Prvok nazývame i-potomkom, ak je rozdiel medzi jeho stupňom a stupňom jeho priamym predchodcom i. i, j-prvok je potom taký prvok, ktorého ľavý a pravý potomok majú stupne i a j. Pre zachovanie efektívnosti musia prvky prioritného frontu dodržiavať pravidlo stupňa. Pri pravidle stupňa typu jeden je potrebné, aby koreň stromu bol i-prvok a každý iný prvok bol   
1, 1-prvok alebo 0, i-prvok, kde i > 1. Pravidlo stupňa typu dva je podobné, ale umožňuje navyše 1, 2-prvky[5].

**Operácia vlož(Q, K, X) -> void**

Prvok X s prioritou K vložíme na ľubovoľné miesto v zozname. Ak sa jedná o prvok s najvyššou prioritou, aktualizujeme smerník k prvku s najvyššou prioritou. Časová náročnosť tejto operácie je teda .

**Operácia vyber(Q) -> X**

V zozname binomiálnych stromov je vybraný binomiálny strom s najvyššou prioritou. Tento strom je vyrezaný a potomkovia koreňa tohto stromu sú spojení so zoznamom binomiálnych stromov a prioritný front je zrekonštruovaný. Prvok s najvyššou je následne odstránený. Spájanie má časovú náročnosť a rekonštrukcia . Celková časová náročnosť operácie je teda .

**Vráť-minimum(Q) -> X**

V zozname binomiálnych stromov je vybraný koreň binomiálneho stromu s najvyššou prioritou. Operácia má časovú náročnosť .

**Spoj(Q1, Q2) -> Q**

Vytvoríme novú Párovaciu haldu Q, ktorá v sebe obsahuje prepojený zoznám stromov z oboch háld, pričom haldy Q1 a Q2 zaniknú. Časová náročnosť operácie je závislá výlučne od rýchlosti prepojenia zoznamov oboch háld, čo je pri zreťazenom zozname .

**Operácia zmeň-prioritu(Q, X, K) -> void**

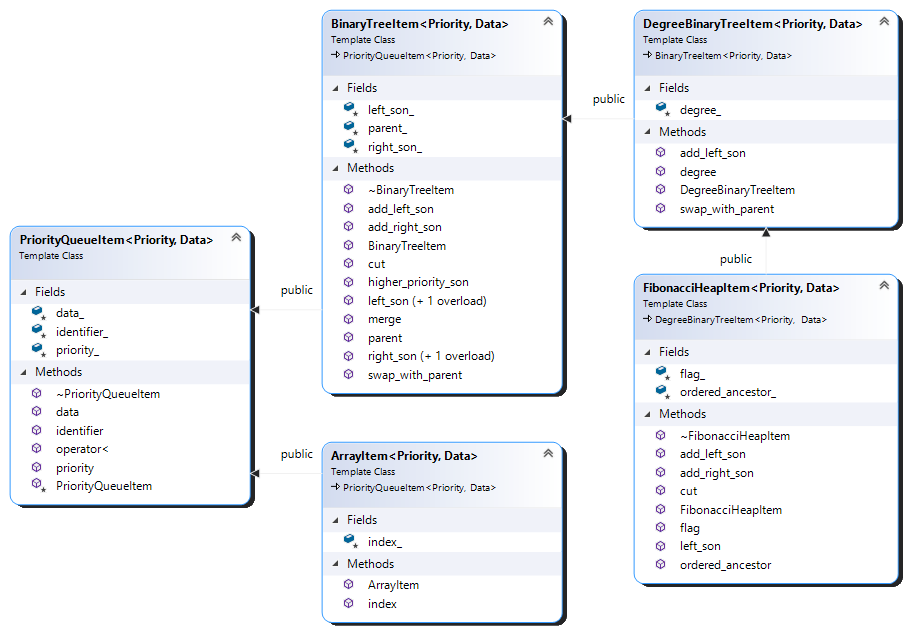
Prvku X je zmenená priorita. Ak sa priorita zvýšila, prvok je vystrihnutý a pridaný k zoznamu binomiálnych stromov. Pre všetkých predchodcov prvku je opätovne nastavené pravidlo stupňa.

# Praktická implementácia prioritného frontu

Pre implementáciu prioritných frontov bol použitý jazyk C++. Bol vybraný kvôli umožneniu manuálnej správy pamäte. Jednotlivé triedy je možné rozdeliť na dve skupiny - triedy, ktoré reprezentujú prvky v prioritnom fronte a triedy reprezentujúce implementáciu prioritného frontu.

## Triedy reprezentujúce prvky v prioritnom fronte

Tieto triedy predstavujú základné jednotky, ktoré sú použité pre uloženie dát spolu s prioritou do prioritného frontu. Tieto prvky môžu zároveň aj v sebe obsahovať vzťahy s inými prvkami. Jednotlivé implementácie sa nachádzajú v súbore PriorityQueueItems.h.



### PriorityQueueItem<Priority, Data>

Generická trieda, tvoriaca spoločného predka pre jednotlivé prvky v prioritnom, ktorá v sebe obsahuje prioritu a dáta.

### BinaryTreeItem<Priority, Data>

Generická trieda, ktorá predstavuje prvok binárneho stromu. Je to potomok triedy PriorityQueueItem<Priority, Data>.

#### Operácie

**BinaryTreeItem\* cut()** – osamostatní inštanciu z binárneho stromu spolu s jej ľavým synom. V priamom predchodcovi je nahradený ukazovateľ na prvok, ľavý alebo pravý potomok, pravým potomkom prvku. Následne je v pravom potomkovi aktualizovaný ukazovateľ na priameho predchodcu. Všetky ukazovatele v inštancií sú nastavené na nulové hodnoty. Ako návratová hodnota je vrátený izolovaný prvok.

**BinaryTreeItem\* merge(BinaryTreeItem\* node)** – prepojí dva prvky spolu na základe ich priority. Očakáva sa, že oba prvky tvoria korene polovičných binárnych stromov[[3]](#footnote-4). Prvku s nižšou prioritou je nastavený pravý syn ako ukazovateľ na ľavého syna prvku s vyššou prioritou. Následne je v prvku s vyššou prioritou nastavený ukazovateľ na ľavého syna na prvok s menšou prioritou. Stupeň prvku s vyššou prioritou sa zväčší o jeden. Návratovou hodnotou je ukazovateľ na prvok s vyššou prioritou.

**BinaryTreeItem\* add\_left\_son(BinaryTreeItem\* node)** – pridá prvok node do binárneho stromu ako ľavého potomka inštancie. Očakáva sa, že prvok node tvorí koreň polovičného binárneho stromu5. Prvku node je nastavený pravý syn na ľavého syna inštancie. Ľavý syn inštancie je upravený ako ukazovateľ na prvok node. Zvýši sa stupeň inštancie o jeden. Návratovou hodnotou je ukazovateľ na inštanciu.

### DegreeBinaryTreeItem<Priority, Data>

Generická trieda, ktorá rozširuje triedu BinaryTreeItem<Priority, Data> o stupeň prvku.

### FibonacciHeapItem<Priority, Data>

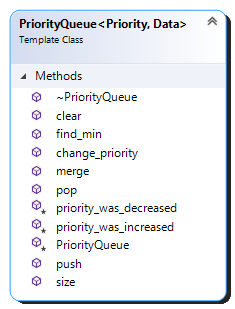
Generická trieda rozširujúca triedu DegreeBinaryTreeItem <Priority, Data> o označenie a ordinálneho predchodcu.

## Triedy reprezentujúce prioritný front

### PriorityQueue<Priority, Data>

Abstraktná generická trieda, ktorá slúži ako všeobecný predchodca jednotlivým implementáciám prioritného frontu. Implementácia sa nachádza v súbore PriorityQueue.h.

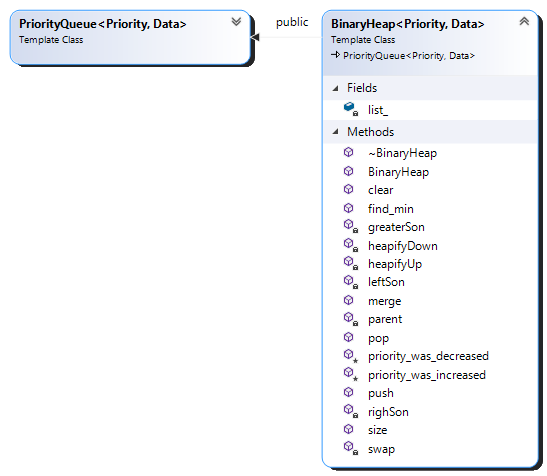
#### Diagram triedy



### BinaryHeap<Priority, Data>

Generická trieda, ktorá implementuje binárnu haldu. Je implementovaná ako implicitný zoznam prvkov. Jednotlivé prvky binárnej haldy sú implementované inštanciami triedy ArrayItem<Priority, Data>. Implementácia sa nachádza v súbore BinaryHeap.h.

#### Diagram triedy



#### Operácie:

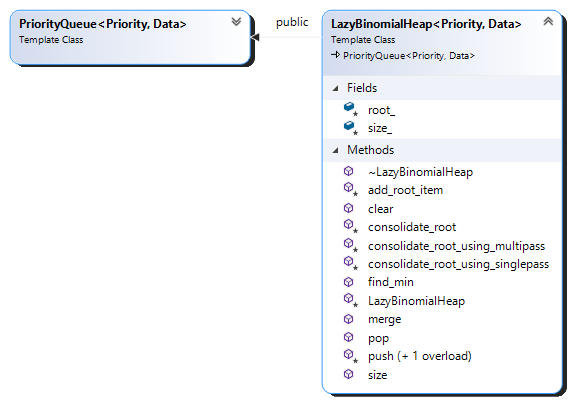
**heapifyDown()** – vymieňa prvok s tým z jeho potomkov, ktorý ma vyššiu prioritu, pokiaľ nie je dodržané obojstranné pravidlo haldy.

**merge(PriorityQueue\* other\_heap)** – pripojí k zoznamu prioritných prvkov prvky zoznamu patriace other\_heap. Tento zoznam je upravený tak, aby každý prvok dodržiaval obojstranné pravidlo haldy. To je dosiahnuté tak, že každý prvok od polovice zoznamu je vymieňaný s jeho potomkom metódou heapifyDown()

### LazyBinomialHeap<Priority, Data>

Abstraktná generická trieda, ktorá je predchodcom jednotlivým implementáciám prioritného frontu, ktoré sú implementované binomiálnym stromom, alebo lesom binomiálnych stromov. Jednotlivé binomiálne stromy sú implementované ako binárne stromy, kde prvky binárneho stromu sú tvorené inštanciami triedy BinaryTreeItem alebo jej potomkami. Binomiálne stromy sú uložené v pravej chrbtici atribútu root\_. Haldové usporiadanie v týchto binárnych stromoch je potom dodržiavané tak, že priorita prvku musí byť väčšia ako priorita ľubovoľného prvku v pravej chrbtici ľavého potomka. Implementovaná je v súbore PriorityQueue.h.

#### Diagram triedy



#### Operácie

**PriorityQueueItem\* push(BinaryTreeItem\* node)** – metoda, ktorá pridá do prioritného frontu nový prvok node. Tento prvok je pripojený k prvku root\_ ako pravý syn pomoc metódy prvku add\_root\_item(node). Ak má nový prvok vyššiu prioritu ako starý prvok, je aktualizovaný atribút root\_, tak aby ukazoval na nový prvok. Počet prvkov v prioritnom fronte sa zvýši o jeden. Vrátená je adresa vytvoreného prvku. ***Pridať citáciu***

**T pop()** – vyberie z prioritného frontu prvok s najvyššou prioritou a vráti ho. Metóda zoberie atribút root\_. Ak je atribút prázdny, je vytvorená výnimka. Inak sa prvok izoluje metódou prvku cut(), a ako root\_ sa nastaví jeho pravý syn. Následne sú binomiálne stromy spravodlivo zlúčené[[4]](#footnote-5) pomocou metódy consolidate\_root(node), kde parameter node je tvorený ľavým synom pôvodného prvku root\_. Počet prvkov v prioritnom fronte je znížený o jeden, pôvodný prvok root\_ je odstránený a je vrátená hodnota metodou data() nad daným prvkom.

**merge(PriorityQueue\* other\_heap)** – spojí dva prioritné fronty spolu. Ak je nastavený atribút root\_, pripojí sa k nemu root\_ z prioritného frontu other\_heap pomocou metódy add\_root\_item(node), kde parameter node je nastavený na root\_ z other\_heap. Potom sa atribút root\_ nastaví na ten z pôvodných atribútov root\_, ktorý mal vyššiu prioritu a zvýši sa počet prvkov o počet prvkov v prioritnom fronte other\_heap. Ak nebol nastavený atribút root\_, inštancia preberie atribúty z prioritného frontu other\_heap. V oboch prípadoch sú na konci atribúty other\_heap nastavené na nulovú hodnotu a prioritný front je odstranený.

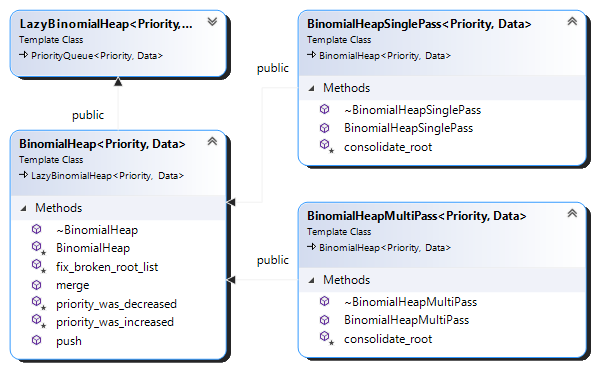
**consolidate\_root\_using\_multipass(BinaryTreeItem\* node, int array\_size)** - spravodlivo zlúči jednotlivé binomiálne stromy prioritného frontu stratégiou multipass. Je vytvorené pole o veľkosti array\_size. Ak nie je tento parameter nastavený, jeho hodnota je . Postupne prechádzame prvky pravej chrbtice parametra node a potom atribútu root\_. Nad jednotlivými prvkami sa vykoná nasledujúci algoritmus – Prvok sa pokúsime vložiť do poľa na index, ktorý sa rovná stupňu prvku. Ak sa na danom indexe v poli už nachádza prvok, je spojený s pridávaným prvkom pomocou jeho funkcie merge(node). Tento spojený prvok bude mať stupeň o jeden väčší ako spájané prvky a zopakujeme s ním algoritmus. Ak sa v poli na indexe nenachádza prvok, vložíme ho do poľa a pokračujeme ďalším prvkom. Po prejdení všetkých prvkov, sú prvky v poli navzájom prepojené metódou add\_root\_item(node). Atribút root\_ je nastavený na prvok s najvyššou prioritou. ***Pridať citáciu***

**consolidate\_root\_using\_singlepass(BinaryTreeItem\* node int array\_size)** – spravodlivo zlučuje jednotlivé binomiálne stromy prioritného frontu stratégiou singlepass. Je vytvorené pole o veľkosti array\_size. Ak nie je tento parameter nastavený, jeho hodnota je . Postupne prechádzame prvky pravej chrbtice parametra node a potom atribútu root\_. Nad jednotlivými prvkami sa vykoná nasledujúci algoritmus – Prvok sa pokúsime vložiť do poľa na index, ktorý sa rovná stupňu prvku. Ak sa na danom indexe v poli už nachádza prvok, je spojený s pridávaným prvkom pomocou jeho funkcie merge(node). Tento spojený prvok je potom pridaný do pravej chrbtice atribútu root\_ pomoc metódy add\_root\_item(node). Ak sa v poli na indexe nenachádza prvok, vložíme ho do poľa. Pokračujeme ďalším prvkom. Po prejdení všetkých prvkov, sú prvky v poli postupne pripojené metódou add\_root\_item(node). Atribút root\_ je nastavený na prvok s najvyššou prioritou. ***Pridať citáciu***

### BinomialHeap<Priority, Data>

Abstraktná generická trieda, ktorá implementuje binomiálnu haldu. Jednotlivé prvky binomiálnych stromov sú implementované inštanciami triedy BinaryTreeItemWithOrderedAncestor<Priority, Data>. Je implementovaná v súbore BinomialHeap.h.

#### Diagram triedy



#### Operácie

**priority\_was\_increased(BinaryTreeItem\* node)** – obnoví haldové usporiadanie prioritného frontu po zvýšení priority prvku node. Pokiaľ bude priorita prvku node vyššia ako priorita usporiadaného predchodcu, alebo sa nestane koreňom binomiálneho stromu, ktorého je súčasťou, je prvok vymieňaný s jeho usporiadaným predchodcom pomocou metódy prvku node swap\_with\_ordered\_ancestor(). Ak je priorita prvku node vyššia ako priorita minimalného prvku, atribút root\_ sa nastaví na prvok node.

**priority\_was\_decreased(BinaryTreeItem\* node)** – metóda, ktorá po znížení priority prvku node obnoví haldové usporiadanie prioritného frontu. Pokiaľ je priorita prvku node menšia ako priorita ľubovoľného prvku, ktorý sa nachádza v pravej chrbtici ľavého potomka prvku node, prvok node je vymenený s tým prvkom z pravej chrbtice ľavého potomka, ktorý ma najvyššiu prioritu pomocou metódy swap\_with\_ordered\_ancestor(). Najmenší prvok je možné nájsť pomocou funkcie prvku node find\_minimal\_son().

**PriorityQueueItem\* push(K& priority, T& data)** – pridá do prioritného frontu nový prvok. Prvok je pridaný tak, že je spravodlivo zlúčený s binárnymi stromami prioritného frontu pomocou metody consolidate\_root(node), kde node je novovytvorený prvok. Vrátená je adresa vytvoreného prvku.

**PriorityQueueItem\* merge(PriorityQueue\* other\_heap)** – jednotlivé binomiálne stromy sú spravodlivo zlučené dohromady pomocou metody consolidate\_root(node), kde node predstavuje root\_ other\_heap. Other\_heap je následne odstránený.

#### BinomialHeapMultiPass<Priority, Data>

Generická trieda, ktorá rozširuje triedu BinomialHeap<Priority, Data> tak, aby používala stratégiu zlučovania binomiálnych stromov multipass. Je implementovaná v súbore BinomialHeap.h.

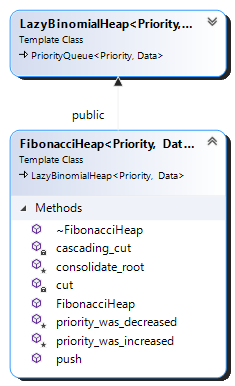
#### BinomialHeapSinglePass<Priority, Data>

Generická trieda, ktorá rozširuje triedu BinomialHeap<Priority, Data> tak, aby používala stratégiu zlučovania singlepass. Je implementovaná v súbore BinomialHeap.h.

### FibonacciHeap<Priority, Data>

Generická trieda, implementujúca Fibonacciho haldu, ktorá rozširuje triedu LazyBinomialHeap<Priority, Data>. Prvky binomiálneho stromu sú tvorené inštanciami triedy FibonacciHeapItem<Priority, Data>. Trieda je implementovaná v súbore FibonacciHeap.h

#### Diagram triedy



#### Operácie

**priority\_was\_increased(BinaryTreeItem\* node)** – upraví prioritný front po zvýšení priority prvku node. Ak má prvok node usporiadaného predchodcu a priorita prvku node je vyššia ako priorita jeho usporiadaného predchodcu, prvok node je vystrihnutý a pridaný do pravej chrbtice atribútu root\_ metódou inštancie cut(node). Následne ak je jeho usporiadaný predchodca označený, je nad ním a jeho usporiadanými predchodcami spustená séria rezov pomocou metódy cascading\_cut(node), inak sa usporidaný predchodca označí. Ak je priorita prvku node vyššia ako priorita minimálneho prvku, atribút root\_ sa nastaví na prvok node. ***Pridať citáciu***

**priority\_was\_decreased(BinaryTreeItem\* node)** - upraví prioritný front po znížení priority prvku node. Pre každý prvok, ktoré sa nachádzajú na pravej chrbtici ľavého potomka prvku node, ak je priorita prvku vyššia ako priorita prvku node, prvok osamostatníme a pripojíme do pravej chrbtice atribútu root\_ metódou inštancie cut(node). Ak je bol prvok node označený, spustíme postupne nad ním a jeho usporiadanými predchodcami sériu rezou pomocou metódy cascading\_cut(node). ***Pridať citáciu***

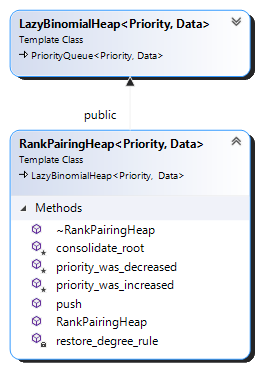
**cut(BinaryTreeItem\* node)** – osamostatní prvok node z binárneho stromu funkciou prvku cut() a pridá ju do pravej chrbtice atribútu root\_ metódou add\_root\_item(node). Ak mál prvok node usporiadaného predchodcu, je mu znížený stupeň o jeden. ***Pridať citáciu***

**cascading\_cut(BinaryTreeItem\* node)** – ak je prvok node označený, je osamostatnený a pridaný do pridá ju do pravej chrbtice atribútu root\_ metódou inštancie cut(node). Následne je zavolaná táto metóda nad jeho usporiadaným predchodcom. Ak prvok nebol označený, je označený a metóda končí. ***Pridať citáciu***

### RankPairingHeap<Priority, Data>

Generická trieda, ktorá implementuje párovaciu haldu na základe priority. Ide o rozšírenie triedy LazyBinomialHeap<Priority, Data>. Binomiálne stromy sú tvorené z inštancií triedy BinaryTreeItem<Priority, Data>. Trieda je implementovaná v súbore RankPairingHeap.h.

#### Diagram triedy



#### Operácie

**priority\_was\_increased(BinaryTreeItem\* node)** - upraví prioritný front po zvýšení priority prvku node. Prvok je vystrihnutý metódou prvku cut() a pridaný do pravej chrbtice atribútu root\_ metódou add\_root\_item(node). Stupeň tohto prvku je nastavený o jeden väčší ako stupeň jeho ľavého potomka. Následne je opravené stupňové pravidlo od priameho predchodcu prvku pomocou metódy inštancie restore\_degree\_rule(node). ***Pridať citáciu***

**priority\_was\_decreased(BinaryTreeItem\* node)** - upraví prioritný front po znížení priority prvku node. Všetky prvky tvoriace pravú chrbticu ľavého potomka prvku node, ktorých priorita je vyššia ako priorita prvku node, sú vystrihnuté a pripojené k pravej chrbtici atribútu node. Stupeň tohto prvku je nastavený o jeden väčší ako stupeň jeho ľavého potomka. Následne je opravené stupňové pravidlo pomocou metódy inštancie restore\_degree\_rule(node), kde parameter node predstavuje priameho predka prvku, ktorý tvoril pôvodného pravého potomka posledného vystrihnutého prvku.

**restore\_degree\_rule(BinaryTreeItem\* node)** – zabezpečí opravenie stupňového pravidla v prioritnom fronte. Nuloví potomkovia majú stupeň -1. Ak je prvok koreňom, je mu nastavený stupeň o jeden vyšší ako stupeň ľavého potomka a metóda končí. Majme premennú stupeň. Ak je rozdiel medzi stupňami ľavého a pravého potomka väčší ako jeden, táto premenná sa nastaví na väčší z týchto dvoch stupňov, inak sa nastaví na hodnotu o jeden väčšiu ako väčší z týchto dvoch stupňov. Ak je stupeň prvku node menší ako premenná stupeň, metóda končí. Inak je prvku node nastavený stupeň rovný premennej stupeň a metóda je zavolaná nad priamym predchodcom prvku node. ***Pridať citáciu***

### PairingHeap<Priority, Data>

Abstraktná generická trieda implementujúca párovaciu haldu. Ide o potomka triedy LazyBinomialHeap<Priority, Data>, kde je poľavené pravidlo párovania binomiálnych stromov rovnakého stupňa. Prvky binárneho stromu sú tvorené inštanciami triedy BinaryTreeItem<Priority, Data>. Je implementovaná v súbore PairingHeap.h.

#### Operácie

**priority\_was\_increased(BinaryTreeItem\* node**) – prvok node je vystrihnutý metódou prvku cut() a spojený s atribútom root\_ funkciou merge(node). Prvok vzniknutý spojením je nastavený ako atribút root\_. ***Pridať citáciu***

**priority\_was\_decreased(BinaryTreeItem\* node)** – zapamätá sa priamy predchodca prvku node, prvok je vystrihnutý a z jeho ľavého potomka sa stane jeho pravý potomok. Z pravej chrbtice prvku node je vytvorený binárny strom tým, že je zavolaná funkcia inštancie consolidate(node) a vytvorený prvok je pripojený na predchádzajúce miesto prvku node.

**PriorityQueueItem\* push(K& priority, T& data)** – je vytvorený nový prvok s atribútmi priority a data. Ak je nastavený atribút root\_, tento prvok je s ním spojený funkciou prvku merge(node) a spojený prvok je nastavený ako atribút root\_. Inak je prvok nastavený ako atribút root\_. Zväčší sa počet prvok o jeden.

**merge(PriorityQueue\* other\_heap)** - sú prepojené atribúty root\_ oboch prioritných frontov pomocou metódy prvku merge(node). Vzniknutý prvok je nastavený ako atribút root\_ inštancie, zvýši sa veľkosť prvkov o počet prvkov other\_heap. Sú vynulované atribúty other\_heap a prioritný front je vymazaný.

#### PairingHeapTwoPass<Priority, Data>

Implementácia párovacej haldy zlučovacou metódou twopass. Trieda je implementovaná v súbore PairingHeap.h.

**Metódy a funkcie:**

consolidate(BinaryTreeItem\* node) – vytvorí z pravej chrbtice prvku binárny strom, ktorý dodržuje haldové usporiadanie. Každé dva prvky v pravej chrbtici sú navzájom prepojené metodou prvku merge(node) a vzniknutý prvok je pridaný do zásobníka. Následne sa postupne prvky v zásobníku vyberajú a spájajú dohromady. Vrátený je koreň vytvoreného binárneho stromu.

#### PairingHeapMultiPass<Priority, Data>

Implementácia párovacej haldy zlučovacou metódou multipass. Trieda je implementovaná v súbore PairingHeap.h.

**Metódy a funkcie:**

consolidate(BinaryTreeItem\* node) – vytvorí z pravej chrbtice prvku binárny strom, ktorý dodržuje haldové usporiadanie. Každé dva prvky v pravej chrbtici sú navzájom prepojené metodou prvku merge(node) a vzniknutý prvok je pridaný dofrontu. Následne sa postupne prvky z frontu vyberajú a spájajú dohromady vzniknutý prvok je zaradený na koniec frontu, pokiaľ sa vo fronte nenachádza jediný prvok. Vrátený je koreň vytvoreného binárneho stromu.

# Návrh testov pre overenie výkonnosti implementácií prioritných frontov

V tejto časti navrhneme sady testov, pomocou ktorých bude možné overiť efektívnosť implementácie prioritných frontov. Tieto testy budú navrhnuté tak, aby simulovali praktické použitie prioritných frontov.

Test Vytvor, Spoj

Budeme vytvárať dva prioritné fronty náhodnej veľkosti, ktoré budeme nasledovne spájať. Výkon implementácie prioritného frontu sa bude skúmať v dvoch prípadoch, a to:

Veľkosť oboch frontov bude podobná.

Veľkosť oboch frontov bude značne rozdielna.

Testy budú vykonané n krát pre každú veľkosť prioritného frontu. Veľkosť prioritného frontu sa bude pohybovať od 100 po 100000 prvkov.

Sledujeme

Čas potrebný na vytvorenie jednotlivých frontov, čas potrebný na spojenie oboch frontov.

Test náhodné operácie Vlož, Vráť-minimum, Vyber

Test fázové operácie Vlož, Vráť-minimum, Vyber

# Vyhodnotenie výsledkov testov implementácií prioritných frontov

# Zoznam použitej literatúry

1. Thomas H. Cormen, Charles E. Leiserson a Clifford Stein. “Introduction to algorithm“*.* London: The MIT Press, 2009
2. Michael L. Fredman, Robert E. Tarjan. “Fibonacci heaps and their uses in improved network optimization algorithms“. In: *Journal of the ACM,* 34.3(1987),s.596–615*.*  
   doi: 10.1145/28869.28874
3. Mark R. Brown. “Implementation and Analysis of Binomial Queue Algorithms“*. SIAM Journal on Computing,* 7.3 (1978), s. 298–319. doi: 10.1137/0207026
4. Jean Vuillemin. “A data structure for manipulating priority queues“*.* In: *Communications of the ACM* 21.4 (1978)*,* s. 309–315. doi: 10.1145/359460.359478
5. Bernhard Haeupler., Siddhartha Sen, a Robert E. Tarjan. “Rank-Pairing Heaps“. In: *SIAM Journal on Computing,* 40.6 (2011), s. 1463–1485*.* doi: 10.1137/100785351

# Zoznam príloh

**Príloha A :** Použivateľská príručka

**Príloha B :** Programatorská príručka

**Príloha C :** Obsah CD disku

**Prílohy**

1. Priorita každého prvku je menšia alebo rovná priorite otca [↑](#footnote-ref-2)
2. Každý strom je definovaný svojim indexom. Strom so stupňom 0 predstavuje prvok s prioritou. Strom stupňa k, je tvorený prepojením dvoch stromov so stupňom k-1. [↑](#footnote-ref-3)
3. prvok nemá predchodcu ani pravého syna [↑](#footnote-ref-4)
4. Sú spájané len tie binomiálne stromy, ktoré majú rovnakú úroveň, resp. ktorých prvky tvoriace korene majú rovnakú úroveň. [↑](#footnote-ref-5)