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<K, T> 8](#_Toc67307262)

[2.1.2 BinaryTreeItem<K, T> 9](#_Toc67307263)

[2.1.3 BinaryTreeItemWithAncestor<K, T> 11](#_Toc67307264)

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

[2.2.1 PriorityQueue<K, T> 11](#_Toc67307266)

[2.2.2 LazyBinomialHeap<K, T> 12](#_Toc67307267)

[2.2.3 BinomialHeap<K, T> 14](#_Toc67307268)

[2.2.4 FibonacciHeap<K, T> 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.

## Abstraktná údajová štruktúra prioritný front

Prioritný front predstavuje abstraktnú údajovú štruktúru, ktorá uchováva dáta zoradené na základe priority, ktorá bola dátam priradená. 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.

Podľa označenia priorít rozdeľujeme prioritný front do dvoch skupín:

Min-heap – s rastúcim označením priority, klesá priorita prvku

Max-heap – s rastúcim označením priority, rastie priorita prvku

Kvôli jednoduchosti budú všetky implementácie prioritných frontov typu min-heap.

Operácie prioritného frontu

Prioritný front musí podporovať prinajmenšom nasledovné operácie:

Vlož(Q, K, X) -> void

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

Vyber(Q) -> X

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

Nad prioritným frontom môžeme ďalej zadefinovať nasledovné operácie:

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

## Spôsoby implementácie prioritného frontu

### Binárna halda

Jedná sa o implementáciu prioritného frontu pomocou úplného binárneho stromu, kde predok prvku nazývame rodičom a potomkov synmi. Tento strom musí dodržiavať vlastnosť haldy[[1]](#footnote-2). Spomenutý strom je možne efektívne implementovať pomocou poľa.

Operácie

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 otcom, pokiaľ nie je splnená vlastnosť haldy. Keďže maximálny počet predkov prvku, s ktorými môžeme prvok vymeniť je , 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 .

Vyber(Q) -> X

Prvok X s najväčšou prioritou K, ktorý sa nachádza na začiatku zoznamu je vymenený s posledným prvkom a odoberieme ho zo zoznamu. Následne je prvok na začiatku zoznamu vymieňaný s jeho potomkami, pokiaľ nebude splnená vlastnosť haldy. Časová náročnosť tejto implementácie je , keďže maximálny počet postupných výmen s potomkami prvku je .

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 .

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, aby bola 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).

Využitie

Triedenie prvkov pomocou heapsort.

### Binomiálna halda

Ide o prioritný front implementovaný pomocou zreťazeného zoznamu binomiálnych stromov[[2]](#footnote-3) zoradených vzostupne podľa ich indexu tak, že v zozname je vždy najviac jeden strom s daným indexom. Ide o front ktorý umožňuje efektívne spájanie frontov, preto ho môžeme považovať aj za spojiteľný prioritný front[[3]](#footnote-4).

Operácie

Vlož(Q, K, X) -> void

Vytvoríme novú binomiálnu haldu Q1, ktorý obsahuje len jeden prvok X s prioritou K. Tento front následne spojíme s frontom Q. Vidíme, že časová náročnosť tejto operácie je závislá len od časovej náročnosti spájania. Ide teda o časovú náročnosť .

Vyber(Q) -> X

V zozname binomiálnych stromov nájdeme ten binomiálny strom, ktorého koreň ma najväčšiu prioritu. Minimálny strom je izolovaný a potomkovia koreňa tohto stromu sú následne spojené so zoznamom binomiálnych stromov. Minimálny prvok 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 Spoj tento smerník aktualizovať. Samotné spájanie má časovú náročnosť . Celková časová náročnosť operácie je teda .

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 .

Spoj(Q1, Q2) -> Q

Vytvoríme novú binomiálnu haldu Q, ktorá v sebe obsahuje jednotlivé binomiálne stromy z binomiálnych háld Q1 a Q2 zoradených vzostupne podľa indexu. Následne sú všetky binomiálne stromy s rovnakým indexom prepojené[[4]](#footnote-5). Táto operácia je ekvivalentná k binárnemu sčítaniu. Maximálny počet prepojení je m. Časová náročnosť tejto operácie je teda (viz. Brown., 1978, s. 301).

### Fibonacciho halda

Fibonacciho halda je implementácia prioritného frontu, ktorá vznikla rozšírením Binomiálnej haldy. Rovnako ako pri Binomiálnej halde, aj tu použitý zreťazený zoznam binomiálnych stromov, neplatí však obmedzenie na počet stromov s rovnakým indexom a nutnosť zoradenia. Taktiež ide o spojiteľný prioritný front.

Operácie

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 .

Vyber(Q) -> X

Vyberieme prvok s najväčšou prioritou.

Vráť-minimum(Q) -> X

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 .

### Párovacia halda

Párovacia halda predstavuje implementáciu prioritného frontu, ktorý je reprezentovaný viacuzlovým stromom, ktorý dodržuje vlastnosť haldy. Ide o samo-vyrovnávaciu alternatívu Binomiálnej haldy.

Operácie

Vlož(Q, K, X) -> void

Vytvoríme novú párovaciu haldu Q1, ktorej koreňom bude prvok X s prioritou K. Túto haldu prepojíme s haldou Q. 4asová náročnosť tejto operácie je .

Vyber(Q) -> X

Implementácia two-pass metódou

Po odstránení koreňa nám vznikne zoznam stromov. V zozname sú jednotlivé prvky navzájom v pároch prepojené. Následne sú jednotlivé páry postupne prepájané, čím vznikne nový viacuzlový strom. Tento strom predstavuje nový stav haldy. Takáto implementácia haldy nám však dáva najhoršiu možnú časovú zložitosť .

Vráť-minimum(Q) -> X

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

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 .

Párovacia halda na základe úrovní

## Zhrnutie teoretických časových náročnosti operácií

# 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<K, T>

Generická trieda, ktorá je použitá ako základná jednotka v prioritnom fronte, ktorá v sebe obsahuje prioritu a dáta.

Atribúty:

K priority\_ - obsahuje prioritu, ktorá bola priradená k dátam.

T data\_ - uložené dáta.

Konštruktor:

PriorityQueueItem(K& priority, T& data) – vytvorí inštanciu triedy, ktorej priradí prioritu z parametra priority a dáta z parametra data.

Metódy:

K& priority(), T& data() – vrátia odkaz na atribút prvku. Slúžia na prístup k atribútu pre čítanie alebo jeho zmenu.

### BinaryTreeItem<K, T>

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

Atribúty:

BinaryTreeItem\* parent\_ - priamy predchodca prvku. V prípade, že je prvok koreňom stromu, čiže nemá predchodcu, je táto hodnota nastavená na nulovú hodnotu.

BinaryTreeItem\* left\_son\_ - ľavý potomok prvku.

BinaryTreeItem\* right\_son\_ - pravý potomok prvku.

size\_t size\_ - stupeň prvku.

Konštruktor:

BinaryTreeItem(K& priority, T& data) – zavolá konštruktor predchodcu a všetky svoje atribúty nastaví na nulovú hodnotu.

Metódy:

BinaryTreeItem\* parent(), BinaryTreeItem\* left\_son(), BinaryTreeItem\* right\_son() - sprístupni prvok uložený v atribúte inštancie triedy.

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[[5]](#footnote-6). 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.

BinaryTreeItem\* add\_right\_son(BinaryTreeItem\* node) - pridá prvok node do binárneho stromu ako pravého potomka inštancie. Očakáva sa, že prvok node tvorí koreň polovičného binárneho stromu5. Prvku node je nastavený pravý syn na pravého syna inštancie. Pravý syn inštancie je upravený ako ukazovateľ na prvok node. Návratovou hodnotou je ukazovateľ na inštanciu.

BinaryTreeItem\* add\_root\_item(BinaryTreeItem\* node) –

BinaryTreeItem\* left\_son(BinaryTreeItem\* node) – nastaví prvok node ako ľavého potomka inštancie. Aktualizuje prvku node ukazovateľ na priameho predchodcu za ukazovateľ na inštanciu.

BinaryTreeItem\* right\_son(BinaryTreeItem\* node) – nastaví prvok node ako pravého potomka inštancie. Aktualizuje prvku node ukazovateľ na priameho predchodcu za ukazovateľ na inštanciu.

BinaryTreeItem\* parent(BinaryTreeItem\* node) – nastaví prvok node ako priameho predchodcu inštancie.

size\_t& degree() – vráti odkaz na stupeň inštancie.

### BinaryTreeItemWithAncestor<K, T>

Generická trieda, ktorá rozširuje triedu BinaryTreeItem<K, T> o usporiadaného predchodcu[[6]](#footnote-7) [[7]](#footnote-8).

Atribúty:

BinaryTreeItemWithAncestor\* ordered\_ancestor\_ - ukazovateľ na usporiadaného predchodcu inštancie.

## Triedy reprezentujúce prioritný front

### PriorityQueue<K, T>

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

Metódy:

change\_priority(PriorityQueueItem\* node, K& priority) – zmení prioritu prvku node na hodnotu priority. Ak bola priorita zvýšená, je zavolaná metóda priority\_was\_increased(node). Ak bola znížená, je volaná metóda priority\_was\_decreased(node).

### LazyBinomialHeap<K, T>

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.

Atribúty:

BinaryTreeItem\* root\_ - ukazovateľ na prvok s najvyššou prioritou. Pravá chrbtica atribútu root\_ je tvorená koreňmi jednotlivých binomiálnych stromov a tvorí obojstranne spojený explicitný zoznam.

size\_t size\_ - počet prvkov v prioritnom fronte.

Konštruktor:

Všetky atribúty naplní nulovými hodnotami.

PriorityQueueItem\* push<I>(K& priority, T& data) – generická metoda, ktorá pridá do prioritného frontu nový prvok s dátovou hodnotou data a priradenou prioritou priority. Najskôr sa vytvorí nový prvok generického typu I[[8]](#footnote-9). Tento prvok je následne pripojený k prvku root\_ ako pravý syn pomoc metódy prvku add\_root\_item(node), kde parameter node tvorí novovytvorený prvok. Ak má nový prvok vyššiu prioritu ako starý prvok, je aktualizovaný atribút root\_, tak aby ukazoval na nový prvok. Počet prvok v prioritnom fronte sa zvýši o jeden. Vrátená je adresa vytvoreného prvku.

size\_t size() – vráti počet prvkov v prioritnom fronte.

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[[9]](#footnote-10) 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.

T& find\_min() – vráti hodnotu dát z prvku s maximálnou prioritou. Ak je atribút root\_ prázdny, je vytvorená výnimka. Inak sa vráti hodnota dát z prvku root\_ metódou data().

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.

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.

### BinomialHeap<K, T>

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

Metódy, funkcie:

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. Metóda najskôr zavolá generickú metódu predchodcu push<I>(priority, data), kde generický typ I je nahradený triedou BinaryTreeItemWithOrderedAncestor<K, T>, ktorá pridá nový prvok s parametrami priority a data do prioritného frontu. Nakoniec sú binomiálne stromy spravodlivo zlučené dohromady pomocou metody consolidate\_root(node), kde node je nulový prvok. Vrátená je adresa vytvoreného prvku.

PriorityQueueItem\* merge(PriorityQueue\* other\_heap) – sú prepojené binomiálne stromy oboch prioritných frontov pomocou metódy predchodcu merge(other\_heap). Jednotlivé binomiálne stromy sú spravodlivo zlučené dohromady pomocou metody consolidate\_root(node), kde node je nulový prvok.

#### BinomialHeapMultiPass<K, T>

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

Metódy, funkcie:

consolidate\_root(BinaryTreeItem\* node) – zlúči binomiálne stromy prioritného frontu stratégiou multipass použitím metódy consolidate\_root\_using\_multipass(node, array\_size), kde array\_size je .

#### BinomialHeapSinglePass<K, T>

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

Metódy, funkcie:

consolidate\_root(BinaryTreeItem\* node) – metóda, ktorá prepisuje metódu predka tak, že použije metódu consolidate\_root\_using\_singlepass(node, array\_size), kde array\_size je .

### FibonacciHeap<K, T>

Generická trieda, ktorá rozširuje triedu LazyBinomialHeap<K, T>. Prvky binomiálneho stromu sú tvorené inštanciami triedy FibonacciHeapItem<K, T>. Trieda je implementovaná v súbore FiboancciHeap.h

Metódy, funkcie:

priority\_was\_increased(BinaryTreeItem\* node) – obnoví haldové usporiadanie prioritného frontu 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.

priority\_was\_decreased(BinaryTreeItem\* node) - obnoví haldové usporiadanie prioritného frontu 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).

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.

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.

PriorityQueueItem\* push(K& priority, T& data) - – pridá do prioritného frontu nový prvok zavolaním generickej metódy predchodcu push<I>(priority, data), kde generický typ I je nahradený triedou FibonacciHeapItem<K, T>, ktorá pridá nový prvok s parametrami priority a data do prioritného frontu. Vrátená je adresa vytvoreného prvku.

consolidate\_root(BinaryTreeItem\* node) – zlúči binomiálne stromy prioritného frontu stratégiou multipass použitím metódy consolidate\_root\_using\_multipass(node, array\_size), kde array\_size je .

# 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. Cormen, T. H., LEISERSON, C. E., & STEIN C. (2009). *Introduction to algorithm.* London: The MIT Press
2. BROWN, M. R., (1978). *Implementation and Analysis of Binomial Queue Algorithms. SIAM Journal on Computing, 7(3), 298–319.* doi:10.1137/0207026
3. Michel Goossens, Frank Mittelbach a Alexander Samarin. *The LATEX Companion*. Reading,

Massachusetts: Addison-Wesley, 1993.

1. Jiexun Li, Rong Zheng a Hsinchun Chen. “From fingerprint to writeprint”. In: *Communications of the ACM* 49.4 (2006), s. 76–82. issn: 00010782. doi: [10.1145/1121949.1121951.](http://dx.doi.org/10.1145/1121949.1121951)

url: [http://portal.acm.org/citation.cfm?doid=1121949.1121951.](http://portal.acm.org/citation.cfm?doid=1121949.1121951)

# 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 s indexom 0 predstavuje prvok s prioritou. Strom úrovne k, je tvorený prepojením dvoch stromov s indexom k-1. [↑](#footnote-ref-3)
3. Mergeable priority queue [↑](#footnote-ref-4)
4. Stromy s rovnakým indexom sú prepojené tak, že jeden strom sa stane ľavým synom druhého, pričom sa musí dodržať vlastnosť haldy. [↑](#footnote-ref-5)
5. prvok nemá predchodcu ani pravého syna [↑](#footnote-ref-6)
6. Prvok A je usporiadaným predchodcom prvku B, ak sa prvok B nachádza v pravej chrbtici ľavého potomka prvku A. [↑](#footnote-ref-7)
7. Pravá chrbtica prvku A predstavuje postupnosť prvkov, kde každý prvok je pravým potomkom predchádzajúceho prvku v postupnosti. [↑](#footnote-ref-8)
8. Tento typ si určí každý potomok triedy, ktorý bude potrebovať použiť túto implementáciu metódy. [↑](#footnote-ref-9)
9. Sú spájané len tie binomiálne stromy, ktoré majú rovnakú úroveň, resp. ktorých prvky tvoriace korene majú rovnakú úroveň. [↑](#footnote-ref-10)