

ŽILINSKÁ UNIVERZITA V ŽILINE

Fakulta riadenia a informatiky

BAKALÁRSKA PRÁCA

PAVOL ŠURIN

Experimentálne porovnanie prioritných frontov

Vedúci práce: Ing. Jankovič Peter, PhD

Registračné číslo: 1237/2020

Žilina, 2021

ŽILINSKÁ UNIVERZITA V ŽILINE

Fakulta riadenia a informatiky

BAKALÁRSKA PRÁCA

ŠTUDIJNÝ ODBOR: INFORMATIKA

PAVOL ŠURIN

Experimentálne porovnanie prioritných frontov

Žilinská univerzita v Žiline

Fakulta riadenia a informatiky

Školiace pracovisko Žilina

Žilina, 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 27.4.2021

.....

Pavol Šurin

Pod'akovanie

Na tomto mieste by som chcel poďakovať vedúcemu bakalárskej práce Ing. Petrovi Jankovičovi, PhD za cenné pripomienky a odborné rady, ktorými prispel k vypracovaniu tejto bakalárskej práce.

ABSTRAKT V ŠTÁTNOM JAZYKU

ŠURIN, Pavol: Experimentálne porovnanie prioritných frontov. [Bakalárska práca]. – Žilinská Univerzita v Žiline. Fakulta riadenia a informatiky; Katedra informatiky. – Školiteľ/Vedúci: Ing. Peter Jankovič, PhD – Mesto: FRI UNIZA, 2021. Počet strán 48

Cieľom bakalárskej práce je popis prioritných frontov, ich implementácii a testov. Práca sa primárne zaoberá binárnou haldou, binomickou haldou, Fibonacciho haldou, párovacou haldou a úrovňovou párovacou haldou.

Kľúčové slová: Halda, Prioritný front

ABSTRAKT V CUDZOM JAZYKU

ŠURIN, Pavol: Experimental comparisons of priority queues. [Bachelor's thesis]. – University of Žilina. Faculty of management and informatics; Supervisor: Ing. Peter Jankovič, PhD – City: FRI UNIZA, 2021. Number of pages 48

The aim of the thesis is to describe priority queues, their implementations, and test their performance. The main focus of this thesis is binary heap, binomial heap, fibonacci heap, pairing heap and rank pairing heap.

Key words: Heap, Priority queue

Obsah

Zoznam obrázkov	8
Zoznam tabuliek	9
Zoznam skratiek.....	10
Úvod.....	11
1 Prioritný front.....	12
1.1 Binárna halda	13
1.2 Lenivá binomická halda	15
1.3 Binomická halda.....	18
1.4 Fibonacciho halda	20
1.5 Párovacia halda na základe úrovni	21
1.6 Párovacia halda	22
2 Implementácie prioritného frontu	24
2.1 Triedy reprezentujúce prvky v prioritnom fronte	24
2.1.1 PriorityQueueItem	24
2.1.2 BinaryTreeItem.....	25
2.1.3 DegreeBinaryTreeItem	25
2.1.4 FibonacciHeapItem.....	25
2.2 Triedy reprezentujúce prioritný front	26
2.2.1 PriorityQueue.....	26
2.2.2 BinaryHeap	26
2.2.3 ExplicitHeap	27
2.2.4 LazyBinomialHeap	29
2.2.5 BinomialHeap	31
2.2.6 FibonacciHeap	33
2.2.7 RankPairingHeap	34
2.2.8 PairingHeap	28
3 Testy.....	36
3.1 Testovacia sada 1	37
3.2 Testová sada 2	41
Záver.....	46
Zoznam použitej literatúry	47

Zoznam príloh.....	48
--------------------	----

Zoznam obrázkov

Obr. 1. Binárna halda zobrazená ako explicitný binárny strom.....	13
Obr. 2. Binárna halda z obr. 1. zobrazená pomocou poľa a vzťahy jednotlivých prvkov k ich priamym potomkom v rámci neho	13
Obr. 3. Vloženie prvku do binárnej haldy	14
Obr. 4. Zlúčenie dvoch binomických stromov prvého stupňa	15
Obr. 5. Rôzne zobrazenia binomického stromu 2. stupňa	16
Obr. 6. Zlučovanie prvkov použitím viacprechodovej stratégie.....	17
Obr. 7. Zlučovanie prvkov použitím jednoprechodovej stratégie	17
Obr. 8. Zvýšenie priority prvku 19	21
Obr. 9. Zvýšenie priority prvku 12 a sériový rez nad prvkom 11.....	21
Obr. 10. Zvýšenie priority prvku 14 a úprava úrovne prvku 12	22
Obr. 11. Diagram prvkov prioritného frontu	24
Obr. 12. Diagram triedy PriorityQueue	26
Obr. 13. Diagram triedy BinaryHeap.....	27
Obr. 14. Diagram triedy ExplicitPriorityQueue.....	28
Obr. 15. Diagram triedy LazyBinomialHeap.....	30
Obr. 16. Diagram tried BinomialHeap.....	32
Obr. 17. Diagram triedy FibonacciHeap.....	33
Obr. 18. Diagram triedy LazyBinomialHeap.....	35
Obr. 19. Diagram tried PairingHeap	28
Obr. 20. Prostredie MicroProfiler	36
Obr. 21. Prostredie memory profileru.....	37
Obr. 22. Vývin binárnej haldy v pamäti pre scenár A	38

Zoznam tabuliek

Tabuľka 1. Tabuľka scenárov testovacej sady 1.....	37
Tabuľka 2. Tabuľka pamäťovej nročností	38
Tabuľka 3. Tabuľka priemerných časov pre vykonanie operácie.....	39

Zoznam skratiek

Q	prioritný front
K	priorita prvku
X	dáta
I	prvok obsahujúci dáta a im priradenú prioritu
size	počet prvkov v prioritnom fronte
BH	binárna halda
BNVP	viacprechodová binomická halda
BNJP	jednoprechodová binomická halda
FH	fibonacciho halda
PHVP	viacprechodová párovacia halda
PHJP	jednoprechodová párovacia halda
UPH	úrovňová párovacia halda

Úvod

Cieľom bakalárskej práce je experimentálne porovnať implementácie prioritných frontov. Prioritné fronty sa používajú všade tam, kde je potrebné efektívne pracovať s dátami, ktoré musia byť zoradené podľa určitého kritéria. Sú používané pri plánovaní úloh v operačnom systéme, efektívnom implementovaní grafových algoritmov až po umelú inteligenciu. Nakoľko existuje množstvo implementácií, kde každá ma svoje prednosti, je nutné ich porovnať, keďže ani ich teoretické zložitosti nepodávajú presný prehľad o ich skutočnej efektívnosti.

Prvá časť práce sa zaoberá problematikou prioritných frontov a rôznych spôsobov ich implementácie.

Ďalšia časť sa zaoberá rozborom spôsobov implementovania jednotlivých prioritných frontov. Tieto boli implementované v jazyku C++ podľa popisu spomenutého v prvej časti.

Ďalšia kapitola popisuje spôsob testovania štruktúr a výsledky z testov. Výsledky boli namerané použitím profileru a spracované pomocou aplikácie Microsoft Excel.

Prínosom tejto práce bude dôkladné porovnanie údajových štruktúr spolu s uvedením možnosti ich využitia.

1 Prioritný front

V tejto časti sa budeme venovať prioritnému frontu ako abstraktnej údajovej štruktúry a jeho implementáciám. Cieľom tejto časti je predstaviť si a popísať rôzne spôsoby implementovania prioritného frontu. Taktiež priradíme k jednotlivým operáciám ich amortizované a worst-case časové náročnosti.

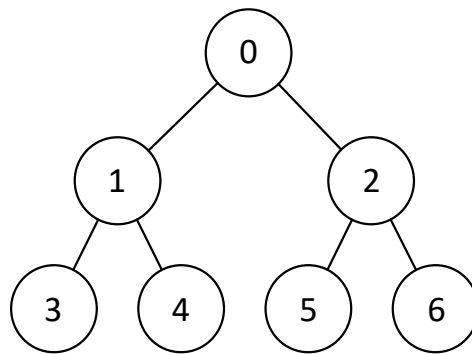
Prioritný front je údajová štruktúra, ktorá uchováva dáta spolu s im priradenou prioritou. Nad prioritným frontom sú definované nasledovné operácie:

- **Operácia vlož(K, X)** - vloží do prioritného frontu prvok I, ktorý obsahuje dáta X a k ním priradenú prioritu K.
- **Operácia nahliadni minimum** - vráti z prioritného frontu hodnotu dát X, ktoré majú priradenú najvyššiu prioritu.
- **Operácia vyber minimum** - odstráni z prioritného frontu prvok I, ktorý obsahoval dáta X s najvyššou priradenou prioritou a vráti hodnotu dát X.
- **Operácia vyber prvok(I)** - odstráni z prioritného frontu prvok I a vráti hodnotu dát X, ktoré obsahoval.
- **Operácia spoj(Q)** – pripojí k prioritnému frontu prioritný front Q.
- **Operácia zmeň prioritu(I, K)** - zmení prioritu prvku I na hodnotu K.

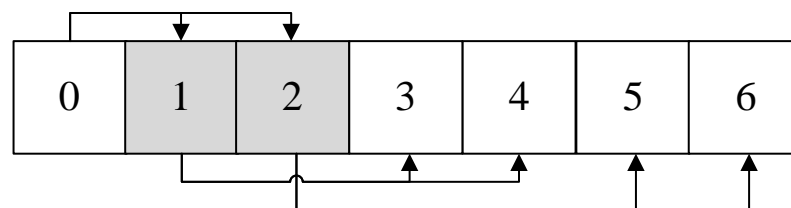
Prioritný front býva často nesprávne označovaný ako halda, z čoho vzniklo delenie prioritných frontov podľa priority, a to na minimálnu (ang. *min-heap*) a maximálnu haldu (ang. *max-heap*). Budeme však namiesto týchto označení používať označenia minimálny a maximálny prioritný front. Minimálny prioritný front predstavuje prioritný front, kde s rastúcou hodnotou priority klesá priorita prvku, teda prvok s najvyššou prioritou má najnižšiu hodnotu priority. Pri maximálnom prioritnom fronte, s rastúcou hodnotou priority priorita, prvku rastie, čo znamená, že prvok s najvyššou prioritou má najvyššiu hodnotu priority. Všetky nami skúmané prioritné fronty sú implementované ako minimálne prioritné fronty, avšak všetky nami popisované vlastnosti platia aj pre maximálne implementácie jednotlivých prioritných frontov.

1.1 Binárna halda

Ide o implementáciu prioritného frontu pomocou haldy, ktorú predstavil J. W. J. Williams (1, s. 347). Pri binárnom strome halda predstavuje kompletný binárny strom, ktorý dodržiava haldové usporiadanie: priorita prvku je nižšia alebo nanajvýš rovná priorite otca. Prvok s najvyššou prioritou predstavuje koreň binárneho stromu. Haldové usporiadanie je udržiavanie pomocou výmen prvku s jeho priamym predkom, alebo priamymi potomkami. Takúto haldu je možné efektívne implementovať použitím implicitného zoznamu, kde prvok na indexe i predstavuje priameho predka prvkom na indexoch $2i + 1$ a $2i + 2$. To však znamená, že celý prioritný front musí byť uložený v súvislej pamäti a jednotlivé operácie prioritného frontu sú len tak efektívne, ako je efektívny zoznam.



Obr. 1. Binárna halda zobrazená ako explicitný binárny strom



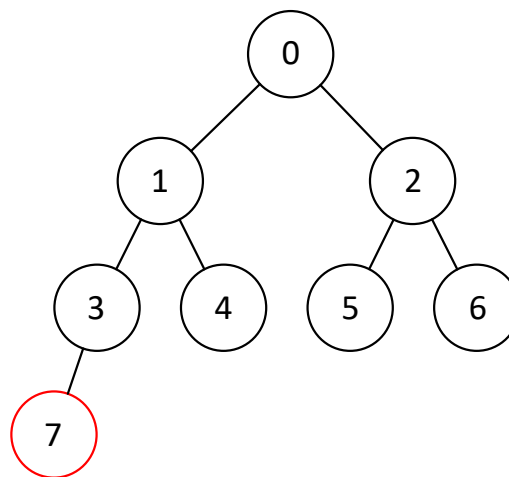
Obr. 2. Binárna halda z obr. 1. zobrazená pomocou poľa a vzťahy jednotlivých prvkov k ich priamym potomkom v rámci neho

Operácia vlož(K, X) - dáta X sú vložené spolu s priradenou prioritou K do binárneho stromu tak, aby bola dodržaná kompletnosť stromu: budú tvoriť prvok v poslednej úrovni čo najviac vľavo. Tento prvok je následne vymieňaný s jeho priamym predkom, dokiaľ nebude priorita priameho predka vyššia ako priorita prvku. Keďže maximálny počet predkov prvku, s ktorými môžeme prvok vymeniť je $\lfloor \log_2 size \rfloor$ a časová náročnosť výmeny je $O(1)$, amortizovaná časová náročnosť samotnej operácie je $O(\log n)$. Pri implementácii implicitným zoznamom môže počas vkladania prvku dôjsť k rozšíreniu

1. FORSYTHE, George E. Algorithm 232: Heapsort. 1964

zoznamu, čo by zvýšilo časovú náročnosť operácie v najhoršom prípade na $O(n + \log n) = O(n)$.

Operácia vyber minimum - prvok s najvyššou prioritou, ktorý tvorí koreň binárneho stromu je vymenený s prvkom, ktorý sa nachádza v poslednej úrovni čo najviac vpravo a je odstránený. Následne, pokiaľ nebude priorita prvku vyššia ako priorita oboch priamych potomkov, sa bude prvok tvoriaci koreň stromu postupne vymieňať s tým priamym potomkom, ktorý má vyššiu prioritu. Nakoniec je vrátená hodnota dát odstráneného prvku. Časová náročnosť tejto operácie je $O(\log n)$, keďže maximálny počet výmen s potomkami prvku je $\lfloor \log_2 \text{size} \rfloor$.



Obr. 3. Vloženie prvku do binárnej haldy

Operácia vyber prvok(I) - prvok I sa vymení s prvkom, ktorý sa nachádza v poslednej úrovni čo najviac vpravo a je odstránený. Ďalší postup a časové zložitosti sú identické ako pri operácií vyber minimum.

Operácia spoj(Q) - jednotlivé prvky z binárnej haldy Q sú pripojené k prioritnému frontu. Táto halda je následne utriedená, aby dodržiavala haldové usporiadanie použitím Floydovho treesort 3 algoritmu (2, str. 701). Od predposlednej úrovne, každý prvok je vymieňaný s jeho priamymi potomkami, pokiaľ jeho priorita nie je vyššia ako priorita oboch priamych potomkov. Časová náročnosť tohto algoritmu, a teda aj celej operácie je $O(n)$ (3, str. 85).

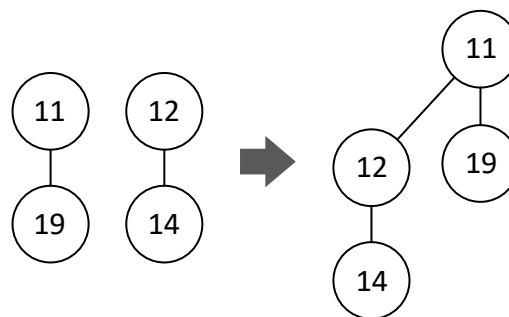
2. FLOYD, R. W. Algorithm 245: Treesort 1964

3. SUCHENEK, M. A. Elementary Yet Precise Worst-Case Analysis of Floyd's Heap-Construction Program. 2012

Operácia zmeň prioritu (X, K) - 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 X sa postupne vymieňa s tým z jeho potomkov, ktorý má najvyššiu prioritu, pokiaľ nebude splnená vlastnosť haldy. V oboch prípadoch je maximálny počet výmen $\lfloor \log_2 n \rfloor$, čo nám dáva časovú zložitosť $O(\log n)$.

1.2 Lenivá binomická halda

Lenivá binomická halda je prioritný front, ktorý vznikol ako relaxácia binomickej haldy navrhnuť J. Vullein (4). Aj napriek tomu, že nie je priamo súčasťou nami skúmaných implementácií, je ju dobré spomenúť, nakoľko tvorí spoločný základ pre niekoľko nami skúmaných hald. Lenivá binomická halda je tvorená lesom binomických stromov, kde binomický strom predstavuje explicitný usporiadaný viaccestný strom, ktorý dodržiava haldové usporiadanie. To znamená, že koreň binomického stromu tvorí prvok s najvyššou prioritou. Každý prvok stromu si udržiava počet jeho priamych potomkov, čo nazývame stupňom prvku. Stupňom, a podobne aj prioritou binomického stromu rozumieme stupeň a prioritu prvku, ktorý tvorí koreň stromu. Binomický strom stupňa k vzniká prepojením dvoch stromov stupňa $k - 1$, kde strom stupňa 0 je tvorený samostatným prvkom. Prepojenie prebieha tak, že binomický strom s nižšou prioritou sa stane prvým potomkom stromu s vyššou prioritou, čím sa zvýši jeho stupeň. Celý princíp lenivej binomickej haldy a jej variant je založený na zlučovaní binomických stromov po operáciách vybratia prvku, prípadne iných operácií, ako si spomenieme pri binomickej halde.

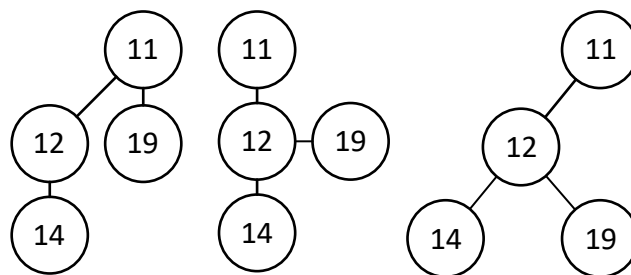


Obr. 4. Zlúčenie dvoch binomických stromov prvého stupňa

Pre ďalší popis je nutné si definovať pojem pravej chrbtice prvku. Je to postupnosť prvkov v binárnom strome, kde každý prvok tvorí pravého priameho potomka predchádzajúceho prvku v postupnosti.

4. VUILLEMIN, Jean. A data structure for manipulating priority queues. 1978

Zobrazenie binomickej haldy - les binomických stromov môžeme zobrazit' ako zret'azený zoznam binomických stromov, kde prvý strom je ten s najvyššou prioritou. Takáto reprezentácia ale používa zbytočne veľa referencií, keďže každý prvok si musí pamätať všetkých svojich potomkov, čo však nie je vždy potrebné. Toto sa dá vyriešiť tým, že každý prvok si bude pamätať referenciu na predka, ľavého súrodenca, pravého súrodenca a prvého potomka. Dá sa však efektívne zobrazit' aj ako binárny strom, kde koreň tvorí prvok s najvyššou prioritou, ako to navrhol J. Vuillemin (4, str. 313) použitím binárnej reprezentácie viaccestného stromu (5, str. 334). V takejto reprezentácii predstavuje ľavý potomok prvku jeho prvého potomka vo viaccestnom zobrazení a pravý potomok jeho pravého súrodenca. Avšak oproti jeho návrhu, v našej implementácii je pravá chrbtica koreňa tvorená binomickými stromami, ktoré nemajú referenciu na priameho predka a posledný prvok v postupnosti ukazuje svojím pravým potomkom na koreň binárneho stromu. Pravá chrbtica koreňa teda tvorí cyklický jednostranne zret'azený zoznam, čím umožňuje konštantné vkladanie do lesa binomických stromov. Výhoda takejto reprezentácie je, že pri niektorých variáciách lenivej binomickej haldy nie je potrebné pristupovať vo viaccestnom zobrazení k priamemu predkovi prvku a postačuje nám, aby si priameho predka pamätal len prvý potomok, čo nám uvoľňuje nutnosť referovať priameho predka v každom prvku. Ak je však potrebná táto referencia, stačí ju pridať. Pri binárnej reprezentácii dochádza k zmene haldového usporiadania, nakoľko postačuje, aby bola priorita prvku vyššia alebo rovná priorite každého prvku v pravej chrbtici jeho ľavého potomka.



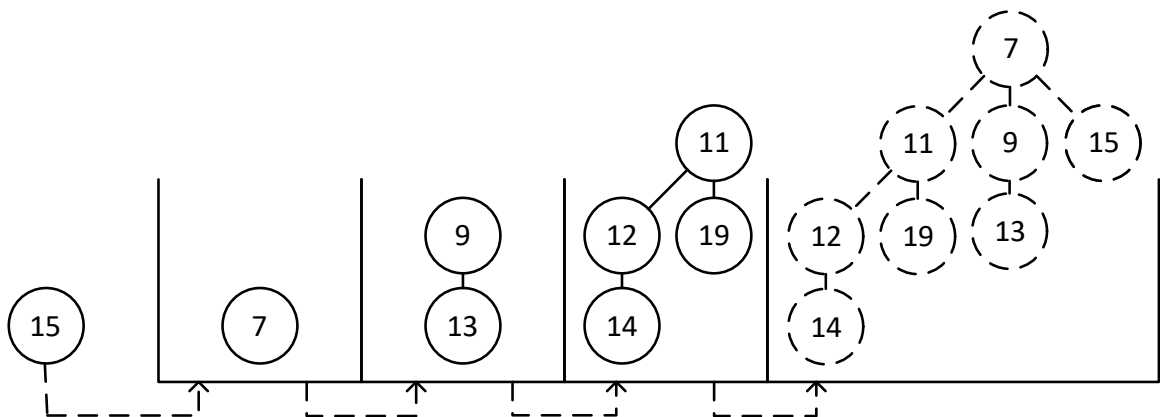
Obr. 5. Rôzne zobrazenia binomického stromu 2. stupňa

Zlučovanie - spočíva v prepájaní binomických stromov rovnakého stupňa. Budeme popisovať a porovnávať dve stratégie zlučovania: viacprechodovú (ang. *multipass*) stratégiu a jednoprechodovú (ang. *one-pass*) stratégiu, ktorá tvorí lenivejšiu verziu lenivej binomickej haldy (6, str. 1470). Obe tieto verzie vznikli ako alternácie algoritmu navrhnutého T. H.

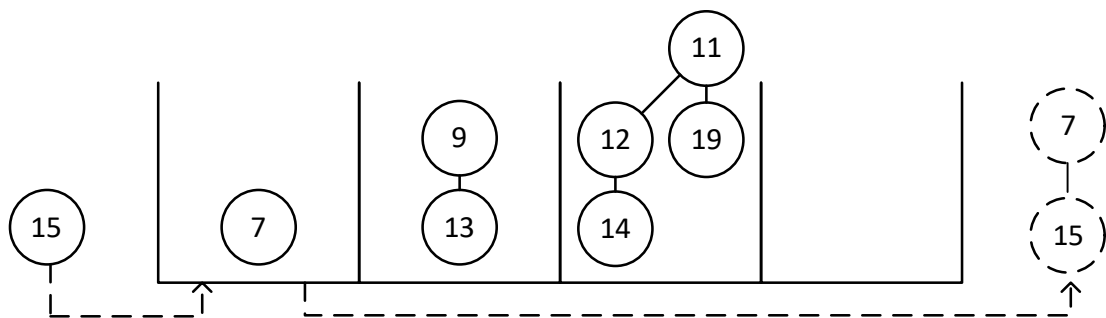
5. KNUTH, D. E. The Art of Computer Programming, Vol. 1: Fundamental Algorithms. 1997

6. HAEUPLER, Bernhard - TARJAN, Robert E. Rank-Pairing Heaps. 2011

Cormenom (7, str. 516). V oboch verziách sú binomické stromy prepájané s využitím poľa. Binomické stromy, ktoré sa nachádzajú v pôvodnom lese/lesoch sú postupne vkladné do poľa na index rovný ich stupňu. Ak už je na danom indexe binomický strom, tieto stromy sú prepojené. Pri viacprechodovej verzii je následne takto prepojený binomický strom vložený do poľa na ďalší index, kdežto pri jednoprechodovej verzii je vložený do finálneho lesa binomických stromov. Ak už nie je v pôvodnom lese žiadny strom, všetky stromy z poľa sú pridané do finálneho lesa binomických stromov. Počas zlučovania je výhodne vybrať aj binomický strom s najvyššou prioritou, nakoľko prechádzame všetkými stromami. Časová náročnosť zlučovania je v amortizovanom čase $O(\log n)$, ale v najhoršom prípade, keď nastáva zlučovanie zriedka, môže ísť až o $O(n)$.



Obr. 6. Zlučovanie prvkov použitím viacprechodovej stratégie



Obr. 7. Zlučovanie prvkov použitím jednoprechodovej stratégie

Operácia vlož (K , X) - je vytvorený nový binomický strom stupňa 0, ktorý je tvorený jedným prvkom s dátami X a prioritou K . Tento strom je pridaný do lesa binomických stromov. Časová náročnosť tejto operácie je závislá len od časovej náročnosti vloženia prvku do lesa binomických stromov, čo má konštantnú časovú zložitosť $O(1)$.

Operácia vyber minimum - v lese binomických stromov je vybraný ten binomický strom, ktorého koreň ma najväčšiu prioritu. Tento strom je z lesa odstránený, pričom vzniknú dva lesy binomických stromov: pôvodný les binomických stromov bez odstráneného stromu a potomkovia koreňa izolovaného stromu. Tieto stromy sú následne zlúčené a je vrátená hodnota dát odstráneného prvku. Nakoľko má odstránenie z lesa binomických stromov konštantnú časovú náročnosť $O(1)$, časová náročnosť operácie sa odvíja len od zlučovania, čiže je $O(\log n)$ v amortizovanom a $O(n)$ v najhoršom prípade .

Operácia vyber prvok (I) - prvok I je odstránený z binomického stromu, prípadne, ak tvoril koreň binomického stromu, z lesa binomických stromov. Les, ktorý tvoria potomkovia prvku I je pripojený k pôvodnému lesu binomických stromov. Ak mal prvok I najvyššiu prioritu, stromy v lese binomických stromov sú zlúčené. Časová náročnosť je teda $O(\log n)$.

Operácia spoj (Q) - k lesu binomických stromov sú pripojené binomické stromy lenivej binomickej haldy Q, čo je možné vykonať v konštantnom čase. Náročnosť tejto operácie je teda $O(1)$.

Operácia zmeň prioritu (I, K) - prvku I je zmenená priorita. Ak sa priorita zvýšila, prvok je vystrihnutý a pridaný do lesa binomických stromov. Toto má časovú náročnosť $O(1)$. V prípade, že sa priorita znížila, postupne sú potomkovia prvku, ktorí majú vyššiu prioritu ako tento prvok, vystrihovaní a pripájaní k lesu binomických stromov. Toto nám dáva časovú náročnosť $O(\log n)$.

Pri zmene priority a výbere prvku nastáva značná deformácia binomického stromu, čo ma za následok postupné zníženie efektívnosti zlučovania a tým celej údajovej štruktúry.

1.3 Binomická halda

Binomická halda je, rovnako ako jej lenivá variácia, tvorená lesom binomických stromov, kde je vždy najviac jeden strom s daným stupňom a po každej operácii meniacej počet prvkov v halde sú stromy v lese zlúčené. To nám dáva logaritmické časové zložitosti na väčšinu operácií. Pôvodná verzia binomickej haldy, ktorú navrhol J. Vuillemin (4), využíva les binomických stromov zoradený vzostupne podľa ich stupňa. Tato verzia požaduje, aby pri operáciách boli prvky vkladane na správne miesto v lese, čo by vyžadovalo

dva ukazovatele do lesa, jeden na prvý binomický strom a druhý na prvok s najvyššou prioritou. Nami popisovaná štruktúra predstavuje variáciu lenivej binomickej haldy, čo uvoľňuje od požiadavky pôvodnej verzie na nutné poradie binomických stromov, a tým aj ukazovateľa na začiatok lesa.

Operácia vlož (K, X) - je vytvorený nový binomický strom stupňa 0, ktorý je tvorený jedným prvkom s dátami X a prioritou K. Tento strom je zlúčený s lesom binomických stromov. Časová náročnosť tejto operácie je závislá len od časovej náročnosti zlúčenia, čo je $O(\log n)$.

Operácia vyber minimum - keďže je operácia vyber minimum identická tej v lenivej binomickej halde, nebude tu popísaná. Keďže však vykonávame zlučovanie po každej operácii meniacej počet prvkov, časová náročnosť tejto operácie v najhoršom prípade klesá na $O(\log n)$.

Operácia vyber prvok (I) - nad prvkom I je zavolaná operácia zmeň prioritu, ktorá mu nastaví prvku I najvyššiu prioritu a urobí ho koreňom binomického stromu. Následne je zavolaná operácia vyber minimum. Keďže obe operácie majú časovú náročnosť $O(\log n)$, má ju aj operácia

Operácia spoj (Q) - les binomických stromov prioritného frontu Q je zlúčený s lesom binomických stromov. Časová náročnosť tejto operácie je závislá len od časovej náročnosti zlúčenia, čo je $O(\log n)$.

Operácia zmeň prioritu (I, K) - prvku I je zmenená priorita. Ak sa priorita zvýšila, prvok je vymieňaný so svojím priamym predkom, pokiaľ nebude priorita jeho priameho predka vyššia ako jeho priorita, alebo sa nestane koreňom binomického stromu. Ak sa priorita prvku znížila, prvok je postupne vymieňaný s tým z jeho potomkov, ktorý má najvyššiu prioritu, pokiaľ nie je jeho priorita vyššia ako priorita všetkých jeho priamych potomkov. V oboch prípadoch je maximálny počet výmen rovný stupňu koreňa binomického stromu v ktorom sa nachádza prvok I, čo je najviac $\lfloor \log_2 \text{size} \rfloor$. To nám dáva časovú $O(\log n)$. Takýto spôsob implementácie navrhol D. B. Johnson (8).

8. JOHNSON, D. B. Priority queues with update and finding minimum spanning trees. 1975

1.4 Fibonacciho halda

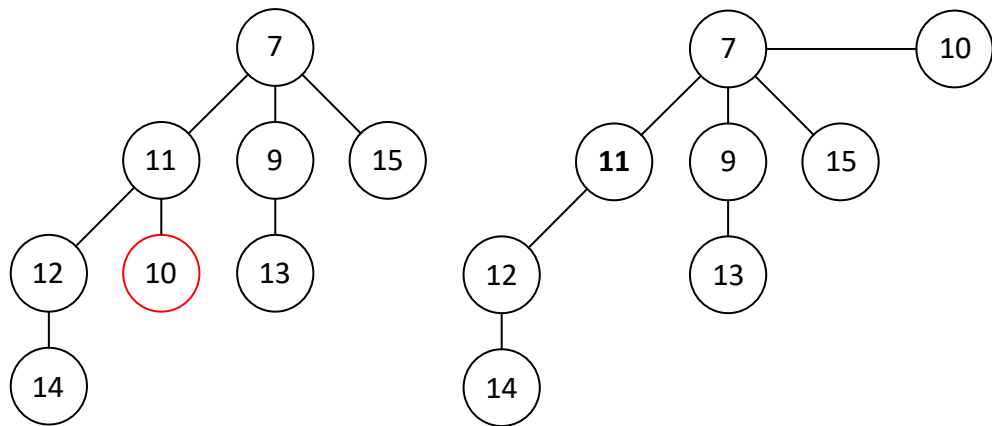
Fibonacciho halda je implementácia prioritného frontu, ktorá vznikla ako rozšírenie binomickej haldy (9). Ide o variáciu lenivej binomickej haldy, ktorá upravuje vystrihovanie prvku z binomického stromu tak, aby sa zabránilo prílišnej deformácii haldy. Toto je dosiahnuté sériou rezov nad predkami po vystrihnutí prvku (9, str. 603). Tieto rezy obmedzujú, aby ktorýkoľvek prvok v binomickom strome, okrem koreňa toho stromu, nestratil viac jedného potomka, inak je aj on sám vystrihnutý. Pre implementáciu takého rezu je nutné, aby každý prvok mal značku o tom, či už mal niekedy odstráneného potomka. Nakoľko potrebuje každý prvok prístup k svojmu potomkovi je nutné, aj keď je použitá binárne zobrazenie binomickej haldy, aby mal každý prvok referenciu na svojho priameho predka. Keďže operácie vlož, vyber minimum a spoj sú identické operáciám lenivej binomickej haldy, nebudú popísané.

Rez vo fibonacciho halde - prvok je vystrihnutý zo stromu, odznačený a pridaný do lesa binomických stromov. Ak priamy predok tohto prvku nebol označený, je označený, inak je nad týmto prvkom vykonaný rez. Takúto postupnosť rezov nazývame sériovým rezom.

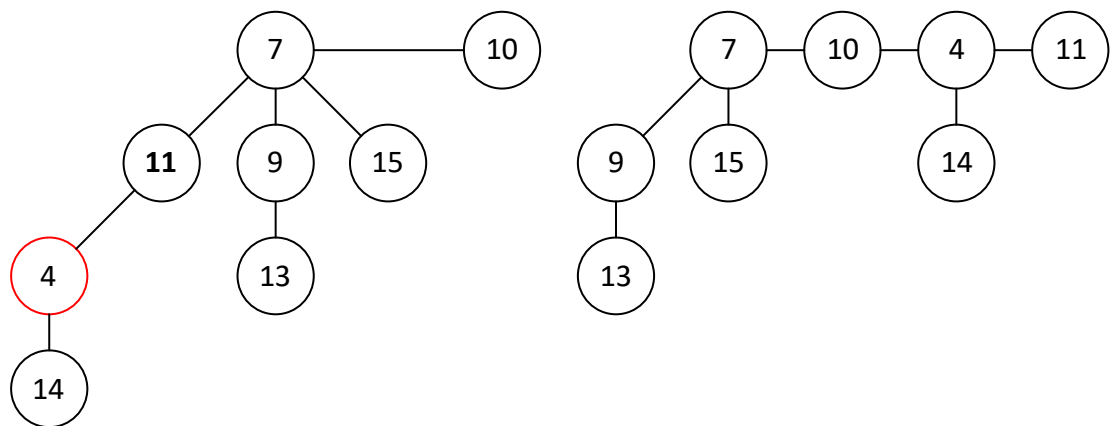
Operácia vyber prvok(I) - operácia funguje rovnako ako pri binomickej halde, ale nad prvkom I je vykonaný rez. Časová náročnosť v najhoršom prípade sa zvyšuje na $O(\log n)$, no v amortizovanom čase je to $O(1)$.

Operácia zmeň prioritu(I, K) - operácia prebieha podobne ako pri lenivej binomickej halde. Prvku I je zmenená priorita. Ak sa priorita zvýšila, je nad prvkom vykonaný rez. Toto má časovú náročnosť $O(\log n)$, ale v amortizovanom čase $O(1)$. V prípade, že sa priorita znížila, je postupne nad potomkami prvku, ktorí majú vyššiu prioritu ako tento prvok, vykonaný rez. Tento postup je možné brať, ako keby sme zvýšili prioritu všetkým potomkom prvku I, čo nám dáva časovú náročnosť $O(\log n)$. Táto náročnosť bude $O(\log n)$ aj v najhoršom prípade, nakoľko je po druhom reze jeho potomka vykonaný rez aj nad samotným prvkom, čím sa zabráni ďalším sériovým rezom.

9. FREDMAN, Michael. L., TARJAN, Robert E. Fibonacci Heap and Their Uses in Improved Network Optimization Algorithms 1987



Obr. 8. Zvýšenie priority prvku 19



Obr. 9 Zvýšenie priority prvku 12 a sériový rez nad prvkom 11

1.5 Úrovňová párovacia halda

Párovacia halda na základe úrovni je variácia lenivej binomickej haldy, ktorá upravuje úrovne jednotlivých binomických stromov po vystrihnutí prvku, čím sa zachováva efektívnosť zlučovania. Táto halda je efektívne implementovaná binárnou reprezentáciou, nakoľko nie je potrebný prístup k priamemu predkovi.

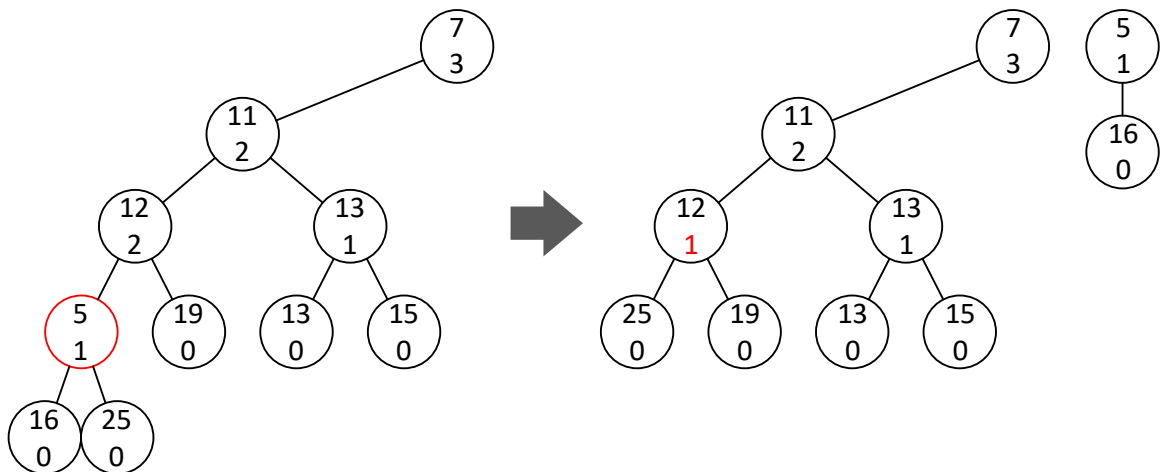
Pre pochopenie zmeny priority je nutné si zaviesť pojem i, j - prvku (5). Táto definícia berie do úvahy binárne zobrazenie haldy. Nakoľko prvok nazývame i -potomkom, ak je rozdiel medzi jeho stupňom a stupňom jeho priamym predka i . i, j - prvok je potom taký prvok, ktorého ľavý a pravý potomok majú stupne i a j .

Stupňové pravidlo - Haupler (6) definoval dva stupňové pravidla: pri stupňovom pravidle typu jeden je potrebné, aby koreň stromu bol i -prvok a každý iný prvok bol 1, 1 - prvok alebo 0, i - prvok, kde $i > 1$. Stupňové pravidlo typu dva je podobné, ale navyše povoľuje 1, 2-prvky.

Operácie vlož, vyber minimum a spoj sú identické lenivej binomickej halde, a preto nebudú popísané.

Operácia vyber prvok(I) - prvok I je vystrihnutý a jeho potomkovia sú pridaný do lesa binomických stromov. Od predka tohto prvku je následne nastolené stupňové pravidlo. Ak má prvok najvyššiu prioritu, stromy v lese binomických stromov sú zlúčené. Časová náročnosť je teda $O(\log n)$, no v amortizovanom čase je to $O(1)$ (5).

Operácia zmeň prioritu(I, K) - prvku I je zmenená priorita. Ak sa priorita zvýšila, prvok je vystrihnutý a pridaný do lesa binomických stromov. Pre všetkých predchodcov prvku je opätovne nastavené stupňové pravidlo. Toto má časovú náročnosť $O(\log n)$, no v amortizovanom čase $O(1)$. Ak sa táto priorita zmenšila, sú postupne všetci potomkovia z pravej chrbtice ľavého syna vystrihnutý a pridaný do lesa binomických stromov, pričom sa zapamätá predok naposledy vystrihnutého prvku. Od tohto predka je následne nastolené stupňové pravidlo. Toto má časovú náročnosť $O(\log n)$.



Obr. 10. Zvýšenie priority prvku 14 a úprava úrovne prvku 12

1.6 Párovacia halda

Párovacia halda predstavuje implementáciu prioritného frontu, ktorý je reprezentovaný viaccestným stromom, ktorý dodržiava vlastnosť haldy. Ide o samovyrovnávaciu haldu (10). Túto implementáciu je možné brať ako variáciu lenivej binomickej haldy, kde je umožnené neférové prepájanie binomických stromov. Túto štruktúru je možné efektívne reprezentovať binárnym stromom. Rovnako ako pri lenivej binomickej halde, aj pri párovacej halde je potrebné zlučovanie stromov po vybratí prvku.

Zlúčovanie - podľa spôsobu zlučovania môžeme rozdeliť párovaciu haldu na viacprechodovú (ang. *multipass*) a dvojprechodovú (ang. *two-pass*) párovaciu haldu (10). Pri viacprechodovej párovacej halde je les stromov zlúčený tak, že sa po sebe idúce prvky v lese v pároch prepájajú, pokiaľ nevznikne jeden strom. Pri dvojprechodovej párovacej halde sa pri prvom prechode po sebe idúce prvky v lese stromov v pároch prepoja a pri druhom sa prepájajú s posledným stromom v lese, čím vznikne nový strom. Prepájanie prebieha identicky na základe priority ako pri lenivej binomickej halde.

Operácia vlož(K, X) - je vytvorený nový prvok, ktorý obsahuje dáta X s prioritou K. Tento prvok je prepojený s koreňom stromu. Časová náročnosť tejto operácie je $O(1)$.

Operácia vyber minimum - prvok s najvyššou prioritou tvorí koreň stromu. Po odstránení koreňa vznikne les stromov, ktorý tvorili jeho potomkov. Tento les je zlúčený a vytvorený strom tvorí haldu. Časová náročnosť tejto operácie je $O(n)$ ak je strom deformovaný, ale v amortizovanom čase ide o $O(\log n)$.

Operácia vyber prvok(I, K) - prvok I je vystrihnutý a les stromov tvorený jeho potomkami je zlúčený. Tento strom je následne pripojený na pôvodné miesto prvku I a je vrátená hodnota tohto prvku. prioritou. Takáto operácia nám dáva časovú náročnosť $O(n)$ v najhoršom prípade a $O(\log n)$ v amortizovanom.

Operácia spoj(Q) - koreň haldy prepojíme koreňom haldy Q a takto vzniknutý strom stanovíme ako haldu. Keďže časová náročnosť operácie je závislá len od rýchlosti prepojenia, ktoré je vykonané konštantné, tak aj celková náročnosť prepojenia je $O(1)$.

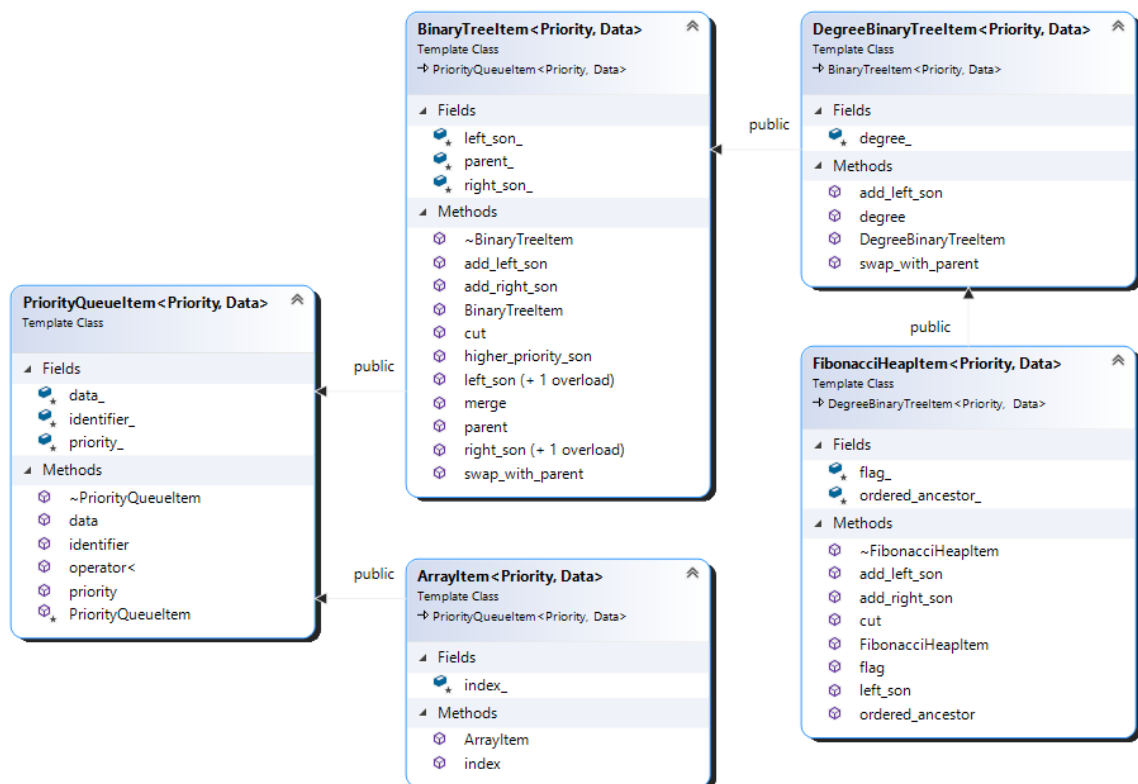
Operácia zmeň prioritu(I, K) - prvku I je zmenená priorita. Ak sa priorita zvýšila, prvok je vystrihnutý a prepojený s koreňom stromu. Toto má časovú náročnosť $O(1)$. V prípade, že sa priorita znížila, je vytvorený les stromov z prvku I a jeho potomkov a tento les je zlúčený. Vytvorený strom je pripojený na pôvodné miesto prvku I. Toto nám dáva časovú náročnosť $O(\log n)$ v amortizovanom čase a $O(n)$ v najhoršom prípade.

2 Implementácie prioritného frontu

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

2.1 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.



Obr. 11. Diagram prvkov prioritného frontu

2.1.1 PriorityQueueItem

Generická trieda, tvoriaca spoločného predka pre jednotlivé prvky v prioritnom fronte, obsahujúca prioritu, dáta a jedinečný identifikátor prvku.

bool operator<(PriorityQueueItem* node) - vráti pravdivostnú hodnotu, či je priorita inštalácie väčšia ako priorita prvku node, alebo v prípade, že sú priority rovné, či má nižší identifikátor.

2.1.2 BinaryTreeItem

Generická trieda, ktorá predstavuje prvok explicitného binárneho stromu. Potomok triedy PriorityQueueItem.

BinaryTreeItem* cut() – vystrihne inštanciu z binárneho stromu spolu s jej ľavým potomkom. V priamom predkovi je nahradený ukazovateľ na inštanciu pravým potomkom inštancie. Následne je v pravom potomkovi aktualizovaný ukazovateľ na priameho predka. Všetky ukazovatele v inšanciách sú nastavené na nulové hodnoty. Ako návratová hodnota je vrátená inštancia.

BinaryTreeItem* merge(BinaryTreeItem* node) – prepojí prvok node s inštanciou na základe ich priority. Očakáva sa, že oba prvky tvoria korene polovičných binárnych stromov. Prvku s nižšou prioritou je nastavený ukazovateľ na pravého potomka ako ukazovateľ na ľavého potomka 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 ako ľavého potomka inštancie. Očakáva sa, že prvok node tvorí koreň polovičného binárneho stromu. Prvku node je nastavený pravý potomok ako ľavý potomok inštancie. Ľavý potomok inštancie je upravený ako ukazovateľ na prvok node. Zvýši sa stupeň inštancie o jeden. Návratovou hodnotou je ukazovateľ na inštanciu.

2.1.3 DegreeBinaryTreeItem

Generická trieda, ktorá rozširuje triedu BinaryTreeItem o stupeň prvku.

2.1.4 FibonacciHeapItem

Generická trieda rozširujúca triedu DegreeBinaryTreeItem o označenie a ordinálneho predchodcu. Táto trieda je využitá pri Fibonacciho halde.

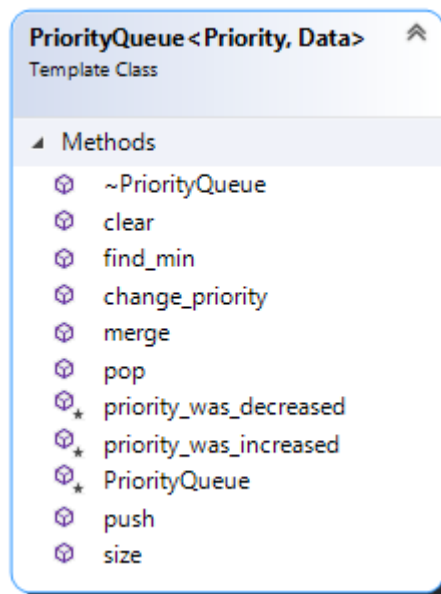
2.1.5 ArrayItem

Generická trieda rozširujúca PriorityQueueItem. Obsahuje v sebe svoj index v implicitnom zozname.

2.2 Triedy reprezentujúce prioritné fronty

2.2.1 PriorityQueue

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



Obr. 12. Diagram triedy PriorityQueue

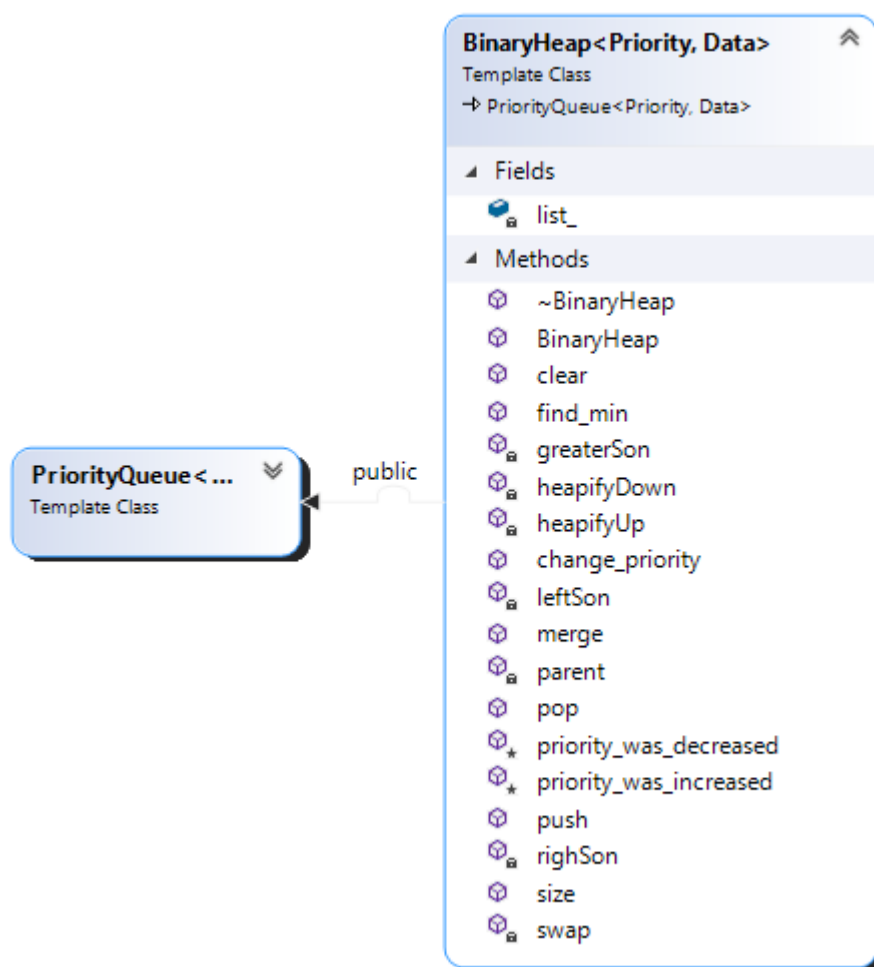
2.2.2 BinaryHeap

Generická trieda, ktorá implementuje binárnu haldu. Je implementovaná implicitným zoznamom prvkov. Jednotlivé prvky binárnej haldy sú implementované inštanciami triedy ArrayItem. Implementácia sa nachádza v súbore BinaryHeap.h.

heapify_down() – vymieňa prvok s tým z jeho potomkov, ktorý ma vyššiu prioritu, pokiaľ nie je dodržané haldové usporiadanie - priorita inštalácie je vyššia ako priorita oboch potomkov.

push(int identifier, K priority, T data, PriorityQueueItem*& data_item) - vytvorí nový prvok ako inštanciu ArrayItem, vloží jeho adresu do parametru data_item a zaradí ho na koniec zoznamu. Tento prvok je následne vymieňaný s jeho priamym predkom, pokiaľ nebude priorita prvku väčšia ako priorita prvu. Táto výmena prebieha metódou heapify_up().

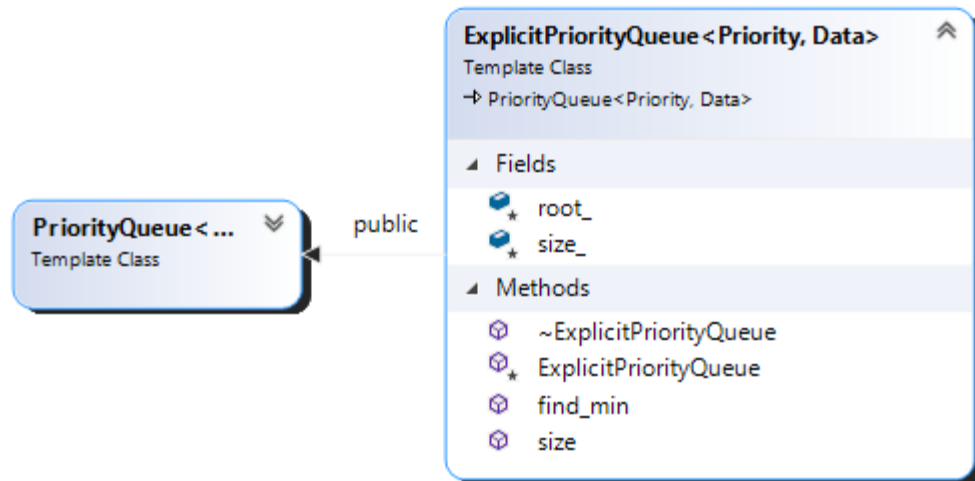
merge(PriorityQueue* other_heap) – pripojí k zoznamu prioritných prvkov prvky zoznamu patriace other_heap. Tento zoznam je upravený tak, aby každý prvok dodržiaval obojstranné pravidlo haldy. To je dosiahnuté tak, že každý prvok od polovice zoznamu je vymieňaný s jeho potomkom metódou heapifyDown(). Algoritmus je spomenutý v časti 1.3.



Obr. 13. Diagram triedy BinaryHeap

2.2.3 ExplicitHeap

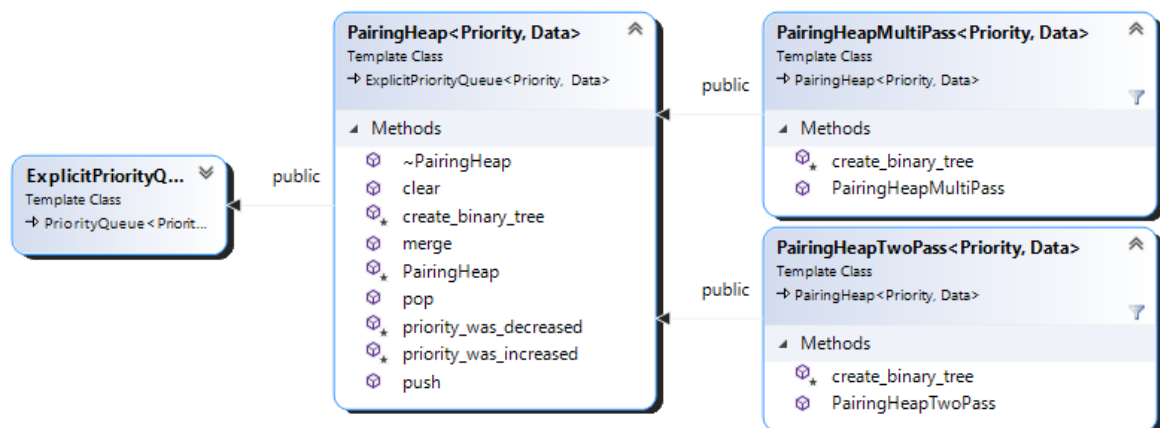
Abstraktná generická trieda, ktorá je predchodcom implementáciám prioritného frontu implementovanými použitím explicitných binárnych stromov. Prvky binárneho stromu sú tvorené inštanciami triedy `BinaryTreeItem` a jej potomkami. Haldové usporiadanie v týchto stromoch je dodržiavané tak, že priorita prvku musí byť väčšia ako priorita ľubovoľného prvku v pravej chrbtici ľavého potomka. Implementovaná je v súbore `ExplicitHeap.h`.



Obr. 14. Diagram triedy ExplicitPriorityQueue

2.2.4 PairingHeap

Abstraktná generická trieda implementujúca párovaciu haldu. Potomok triedy ExplicitHeap. Prvky binárneho stromu sú tvorené inštanciami triedy BinaryTreeItem. Je implementovaná v súbore PairingHeap.h.



Obr. 15. Diagram tried PairingHeap

priority_was_decreased(BinaryTreeItem* node, K priority) – zapamätá sa priamy predok 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 inštalácie consolidate(node) a vytvorený prvok je pripojený na predchádzajúce miesto prvku node.

push(int identifier, K priority, T data, PriorityQueueItem*& data_item) – je vytvorený nový prvok ako inštancia BinaryTreeItem, s atribútmi identifier, 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_.

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.

T pop(int& identifier) – vyberie z prioritného frontu prvok tvoriaci atribút `root_` a jeho identifikátor vloží do parametra `identifier`. Ak bol atribút `root_` prázdny, je vytvorená výnimka. Inak sa prvok izoluje metódou prvku `cut()` a ako `root_` sa nastaví binárny strom, ktorý vznikne funkciou `create_binary_tree(BinaryTreeItem* node)`. Je vrátená hodnota vybraného prvku jeho funkciou `data()`.

PairingHeapTwoPass

Implementácia párovacej haldy zlučovacou metódou `two-pass`. Trieda je implementovaná v súbore `PairingHeap.h`.

create_binary_tree(BinaryTreeItem* node) – vytvorí z pravej chrbtice prvku binárny strom, ktorý dodržiava haldové usporiadanie. Každé dva prvky v pravej chrbtici sú navzájom prepojené metódou prvku `merge(node)` a vzniknutý prvok je pridaný do zásobníka. Vyberie sa prvý prvok zo zásobníka. Postupne sa zo zásobníka vyberajú zvyšné prvky a spájajú sa s prvým vybraným prvkom. Vrátený je koreň vytvoreného binárneho stromu.

PairingHeapMultiPass

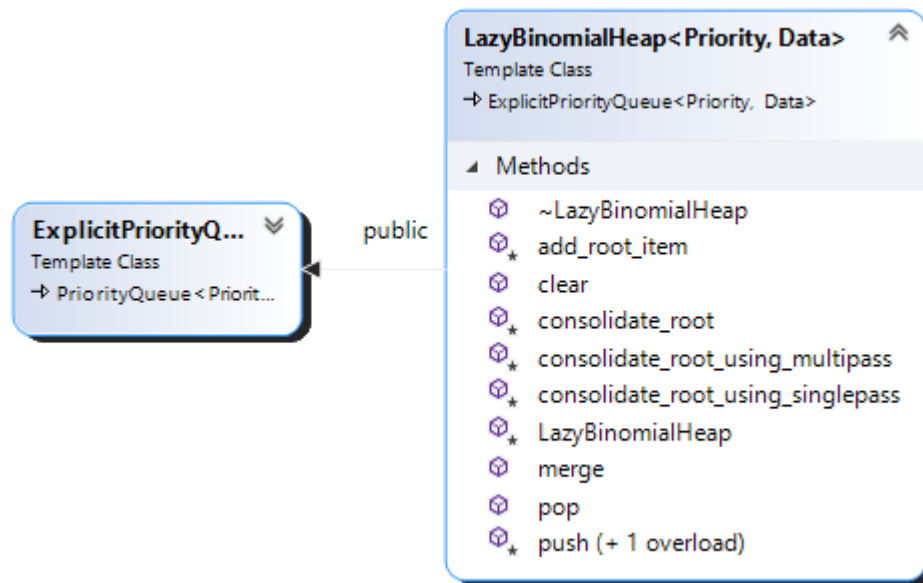
Implementácia párovacej haldy zlučovacou metódou `multipass`. Trieda je implementovaná v súbore `PairingHeap.h`.

create_binary_tree(BinaryTreeItem* node) – vytvorí z pravej chrbtice prvku binárny strom, ktorý dodržiava haldové usporiadanie. Každé dva prvky v pravej chrbtici sú navzájom prepojené metódou prvku `merge(node)` a vzniknutý prvok je pridaný do frontu. Prvky z frontu sa vyberajú po dvojiciach, spájajú dohromady a vzniknuté prvky sú zaradené na koniec frontu, dokiaľ sa vo fronte nenachádza jediný prvok. Tento prvok, ktorý tvorí koreň binárneho stromu, je vrátený.

2.2.5 LazyBinomialHeap

Abstraktná generická trieda, ktorá tvorí predchodcu prioritným frontom implementovaných lesom binomických stromov. Jednotlivé binomické stromy sú implementované ako binárne stromy, kde prvky binárneho stromu sú tvorené inštanciami triedy `BinaryTreeItem` a jej potomkami. Les binomických stromov predstavuje pravú

chrbticu atribútu `root_`. Ide o cyklický jednostranne zreťazený zoznam prvkov, kde pravý potomok ukazuje na ďalší koreň binomického stromu v lese. Implementovaná je v súbore `PriorityQueue.h`.



Obr. 16. Diagram triedy `LazyBinomialHeap`

`add_root_item(BinaryTreeNode* node)` - pripojí k pravej chrbtici atribútu `root_` pravú chrbticu prvku `node`. Nastaví atribút `root_` na prvok s najvyššou prioritou.

`PriorityQueueItem* push(BinaryTreeNode* node)` – funkcia, ktorá pridá do prioritného frontu nový prvok `node`. Tento prvok je pripojený k prvku `root_` ako pravý syn pomocou 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.

`T pop(int& identifier)` – vyberie z prioritného frontu prvok tvoriaci atribút `root_` a jeho identifikátor vloží do parametra `identifier`. Ak je atribút prázdny, je vytvorená výnimka. Prvok tvoriaci atribút `root_` si nastaví odkaz na priameho predka tak, aby ukazoval na seba. Toto je znak, že prvok má byť odstránený. Binomické stromy sú zlúčené pomocou metódy `consolidate_root(node)`, kde parameter `node` je tvorený ľavým potomkom atribútu `root_`. Počet prvkov v prioritnom fronte je znížený o jeden, pôvodný prvok `root_` je odstránený a je vrátená hodnota metódou `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 atribút `root_` z `other_heap`. Atribút

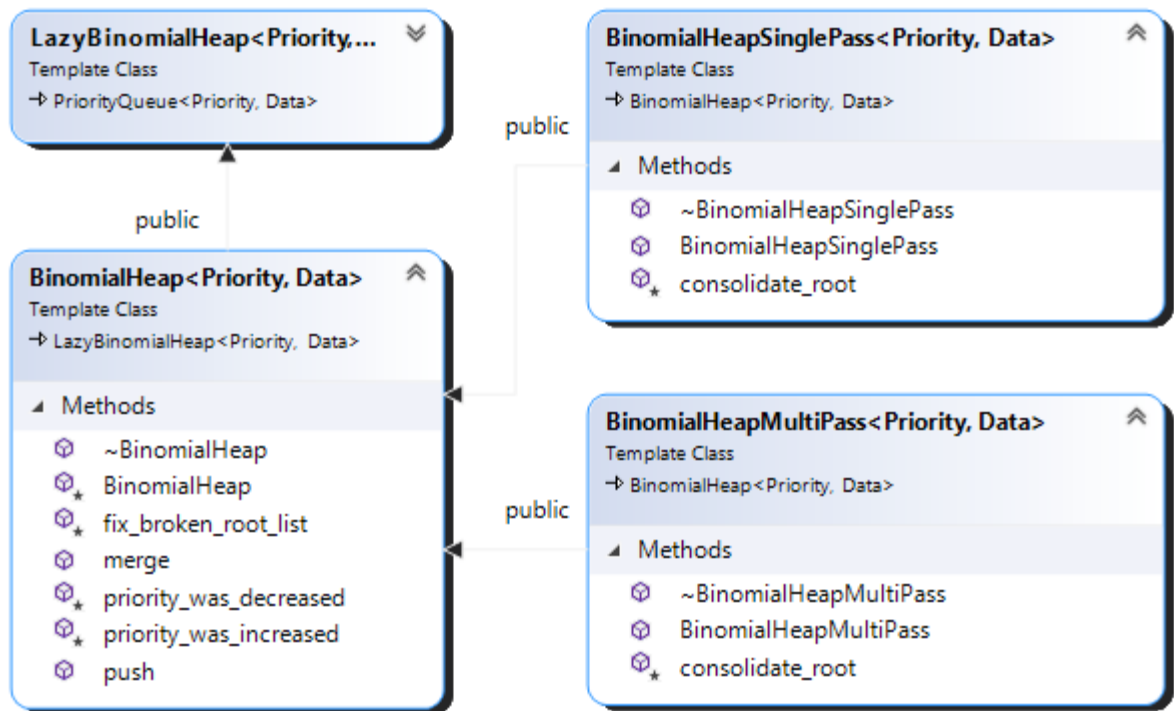
root_ sa nastaví na ten z pôvodných atribútov root_, ktorý mal vyššiu prioritu a zvýši sa počet prvkov o počet prvkov v prioritnom fronte other_heap. Ak nebol nastavený atribút root_, inštancia preberie atribúty z prioritného frontu other_heap. V oboch prípadoch sú na konci atribúty other_heap nastavené na nulovú hodnotu a prioritný front je odstraný.

consolidate_root_using_multipass(BinaryTreeItem* node, int array_size) - spravodlivo zlúči jednotlivé binomické stromy prioritného frontu stratégiou multipass. Je vytvorené pole o veľkosti array_size. Postupne prechádzame prvky pravej chrbtice parametru node, a potom atribútu root_. Ak má root_ nastaveného predka, preskočíme ho. 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. Tento algoritmus je vytvorený ako variácia algoritmu navrhnutého Thomas H. Cormenom (5, str. 516).

consolidate_root_using_singlepass(BinaryTreeItem* node int array_size) – spravodlivo zlučuje jednotlivé binomické stromy prioritného frontu stratégiou singlepass. Je vytvorené pole o veľkosti array_size. 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_ pomocou 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.

2.2.6 BinomialHeap

Abstraktná generická trieda, ktorá implementuje binomiálnu haldu. Jednotlivé prvky binomických stromov sú implementované inštanciami triedy BinaryTreeItem. Je implementovaná v súbore BinomialHeap.h.



Obr. 17. Diagram tried BinomialHeap

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 binomického stromu, ktorého je súčasťou, je prvok vymieňaný s jeho usporiadaným predchodcom pomocou metódy prvku node swap_with_ancestor_node(). 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 má najvyššiu prioritu pomocou metódy swap_with_ancestor_node(). Najmenší prvok je možné nájsť pomocou funkcie prvku node find_minimal_son().

push(int identifier, K priority, T data, PriorityQueueItem*& data_item) – pridá do prioritného frontu nový prvok tvorený inštanciou DegreeBinaryTreeItem. Prvok je pridaný tak, že je zlúčený s binárnymi stromami prioritného frontu pomocou metódy consolidate_root(node), kde node je novovytvorený prvok. Adresa vytvoreného prvku je priradená do parametra data_item.

PriorityQueueItem* merge(PriorityQueue* other_heap) – jednotlivé binomické stromy sú zlučené dohromady pomocou metódy `consolidate_root(node)`, kde `node` je atribút `root_` z `other_heap`. `Other_heap` je následne odstránený.

BinomialHeapMultiPass

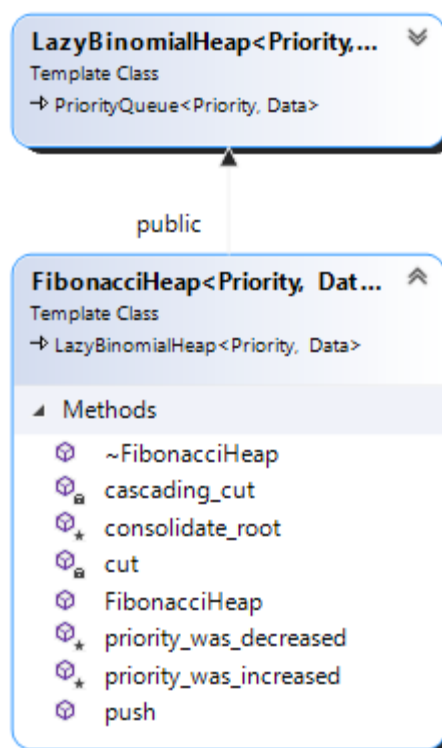
Generická trieda, ktorá rozširuje triedu `BinomialHeap` tak, aby používala stratégiu zlučovania binomických stromov `multipass`. Je implementovaná v súbore `BinomialHeap.h`.

BinomialHeapSinglePass

Generická trieda, ktorá rozširuje triedu `BinomialHeap` tak, aby používala stratégiu zlučovania `singlepass`. Je implementovaná v súbore `BinomialHeap.h`.

2.2.7 FibonacciHeap

Generická trieda, implementujúca Fibonacciho haldu, ktorá rozširuje triedu `LazyBinomialHeap`. Prvky binomického stromu sú tvorené inštanciami triedy `FibonacciHeapItem`. Trieda je implementovaná v súbore `FibonacciHeap.h`



Obr. 18. Diagram triedy `FibonacciHeap`

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 usporiadaný predchodca označí. Ak je priorita prvku `node` vyššia ako priorita minimálneho prvku, atribút `root_` sa nastaví na prvok `node`. (9)

`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 rezov pomocou metódy `cascading_cut(node)`. (9)

`cascading_cut(BinaryTreeItem* node)` – ak je prvok `node` označený, je vystrihnutý a pridaný 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. (9)

2.2.8 RankPairingHeap

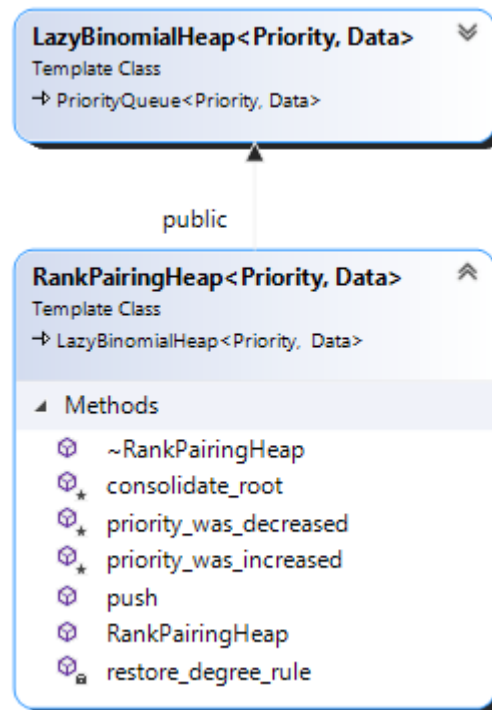
Generická trieda, ktorá implementuje párovaciu haldu na základe priority. Ide o rozšírenie triedy `LazyBinomialHeap`. Binomické stromy sú tvorené z inštancií triedy `BinaryTreeItem`. Trieda je implementovaná v súbore `RankPairingHeap.h`.

`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)`.

`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.



Obr. 19. Diagram triedy LazyBinomialHeap

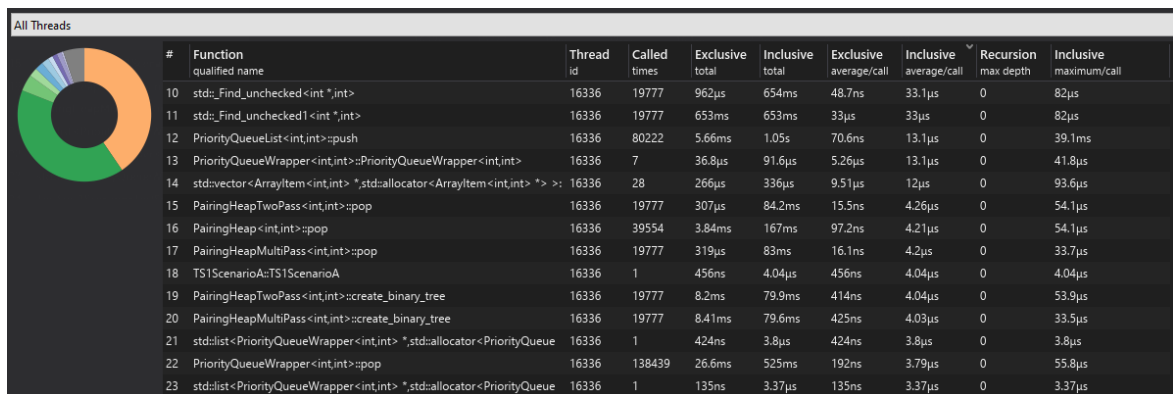
3 Testy

V tejto časti navrhujeme a vykonáme sady testov, pomocou ktorých bude možné overiť efektívnosť implementácie prioritných frontov.

Na prioritných frontoch sú testované nasledovné operácie:

- **vlož(K, X, I)** – do všetkých prioritných frontov vloží prvok X s prioritou K a identifikátorom I
- **vyber minimum()** – vyberie a odstráni zo všetkých prioritných frontov prvok s najvyššou prioritou
- **zmeň prioritu(I, K)** – všetkým prioritným frontom zmení prioritu prvku s identifikátorom I na K.

Pre odmeranie času trvania operácií bolo použité rozšírenie MicroProfiler pre Microsoft Visual Studio 2019. Pre odmeranie veľkosti použitej pamäte bol použitý vstavaný memory profiler. Nakoľko rozšírenie MicroProfiler nedokázalo správne označiť všetky volania operácií v konečnej verzii programu, boli sme nútený aplikáciu skompilovať a spustiť ako inštrumentizovanú aplikáciu, čo malo dopad na rýchlosti jednotlivých operácií.



#	Function qualified name	Thread id	Called times	Exclusive total	Inclusive total	Exclusive average/call	Inclusive average/call	Recursion max depth	Inclusive maximum/call
10	std::Find_unchecked<int *,int>	16336	19777	962µs	654ms	48.7ns	33.1µs	0	82µs
11	std::Find_unchecked1<int *,int>	16336	19777	653ms	653ms	33µs	33µs	0	82µs
12	PriorityQueueList<int,int>::push	16336	80222	5.66ms	1.05s	70.6ns	13.1µs	0	39.1ms
13	PriorityQueueWrapper<int,int>::PriorityQueueWrapper<int,int>	16336	7	36.8µs	91.6µs	5.26µs	13.1µs	0	41.8µs
14	std::vector<ArrayItem<int,int> *,std::allocator<ArrayItem<int,int> *> >::	16336	28	266µs	336µs	9.51µs	12µs	0	93.6µs
15	PairingHeapTwoPass<int,int>::pop	16336	19777	307µs	84.2ms	15.5ns	4.26µs	0	54.1µs
16	PairingHeap<int,int>::pop	16336	39554	3.84ms	167ms	97.2ns	4.21µs	0	54.1µs
17	PairingHeapMultiPass<int,int>::pop	16336	19777	319µs	83ms	16.1ns	4.2µs	0	33.7µs
18	TS1ScenarioAsTS1ScenarioA	16336	1	456ns	4.04µs	456ns	4.04µs	0	4.04µs
19	PairingHeapTwoPass<int,int>::create_binary_tree	16336	19777	8.2ms	79.9ms	414ns	4.04µs	0	53.9µs
20	PairingHeapMultiPass<int,int>::create_binary_tree	16336	19777	8.41ms	79.6ms	425ns	4.03µs	0	33.5µs
21	std::list<PriorityQueueWrapper<int,int> *,std::allocator<PriorityQueue	16336	1	424ns	3.8µs	424ns	3.8µs	0	3.8µs
22	PriorityQueueWrapper<int,int>::pop	16336	138439	26.6ms	525ms	192ns	3.79µs	0	55.8µs
23	std::list<PriorityQueueWrapper<int,int> *,std::allocator<PriorityQueue	16336	1	135ns	3.37µs	135ns	3.37µs	0	3.37µs

Obr. 20. Prostredie MicroProfiler

Native Heap

View Settings filters are applied (Just My Code)

Identifier	Count	Size (Bytes)	Module
Heap	844 498	43 833 835	
std::Hash<std::Umap_traits<int,PriorityQueueItem<int,int> *,std::Uhash_compare<...>	422 058	13 505 856	PrioritneFrontyTest.exe
std::Hash_vec<std::allocator<std::List_unchecked_iterator<std::List_val<std::List_si...	7	7 340 305	PrioritneFrontyTest.exe
BinomialHeap<int,int>::push	120 588	6 752 928	PrioritneFrontyTest.exe
PairingHeap<int,int>::push	120 588	5 788 224	PrioritneFrontyTest.exe
FibonacciHeap<int,int>::push	60 294	4 341 168	PrioritneFrontyTest.exe
RankPairingHeap<int,int>::push	60 294	3 376 464	PrioritneFrontyTest.exe
BinaryHeap<int,int>::push	60 294	1 929 408	PrioritneFrontyTest.exe
std::vector<ArrayItem<int,int> *,std::allocator<ArrayItem<int,int> *> >::Eplace_rea...	1	491 615	PrioritneFrontyTest.exe
std::vector<int,std::allocator<int> >::Eplace_reallocate<int const &>	1	245 827	PrioritneFrontyTest.exe
[Unknown frame from ntdll.dll]	324	58 264	ntdll.dll
PriorityQueueList<int,int>::PriorityQueueList<int,int>	25	2 712	PrioritneFrontyTest.exe
PriorityQueueWrapper<int,int>::PriorityQueueWrapper<int,int>	14	672	PrioritneFrontyTest.exe
std::basic_string<char,std::char_traits<char>,std::allocator<char> >::assign	7	288	PrioritneFrontyTest.exe
main	2	72	PrioritneFrontyTest.exe
Scenario::Scenario	1	32	PrioritneFrontyTest.exe

Obr. 21. Prostredie memory profileru

3.1 Testovacia sada 1

Nad jednotlivými prioritnými frontami je spustený n náhodných operácií. Počet n je vybraný tak, aby bol počet operácií vlož a vyber minimum medzi scenármi A, B a C, D približne rovnaký. Tento postup je zopakovaný 10 krát pre každý scenár.

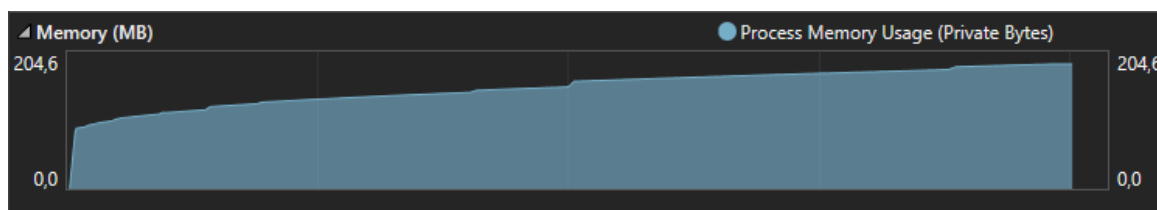
Tabuľka 1. Tabuľka scenárov testovacej sady 1

	Scenár A	Scenár B	Scenár C	Scenár D	Scenár E
Počet operácií	1000000	1250000	1000000	1666667	1000000
Vlož	87%	70%	67%	40%	50%
Vyber minimum	13%	10%	33%	20%	40%
Zmeň prioritu	0%	20%	0%	40%	10%
Očakávané početnosti	740000	750000	340000	320000	100000

Tabuľka 2. Tabuľka pamäťovej nročnosti

	Štruktúra	Scenár A, B	Scenár C, D	Scenár E
Počet prvkov		740806	334008	100159
Pamäťová nročnosť (kB)	BH	32104.7	19087.2	11604
	BNVP	41485.1	18704.4	5608.9
	BNJP	41485.1	18704.4	5608.9
	FH	53338	24048.5	7211.4
	PHVP	35558.6	16032.3	4807.6
	PHDP	35558.6	16032.3	4807.6
	UPH	41485.1	18704.4	5608.9

Všetky prvky v haldách založených na binomickej halde potrebujú minimálne tri referencie na ostatné prvky a musia poznať svoj stupeň. Pamäťovo najnáročnejšia je Fibonacciho halda. To je spôsobené tým, že každý prvok v halde si musí ešte navyše pamätať referenciu na svojho predka a informáciu o tom či už mal niekedy odstráneného potomka. Pamäťovo najúspornejšie sú párovacia a binárna halda. Párovacia halda nepoužíva stupne prvkov, preto jej postačujú len tri referencie pre prvky. Binárna halda využíva pri našej implementácii implicitný zoznam prvkov, kde každý prvok obsahuje navyše index.



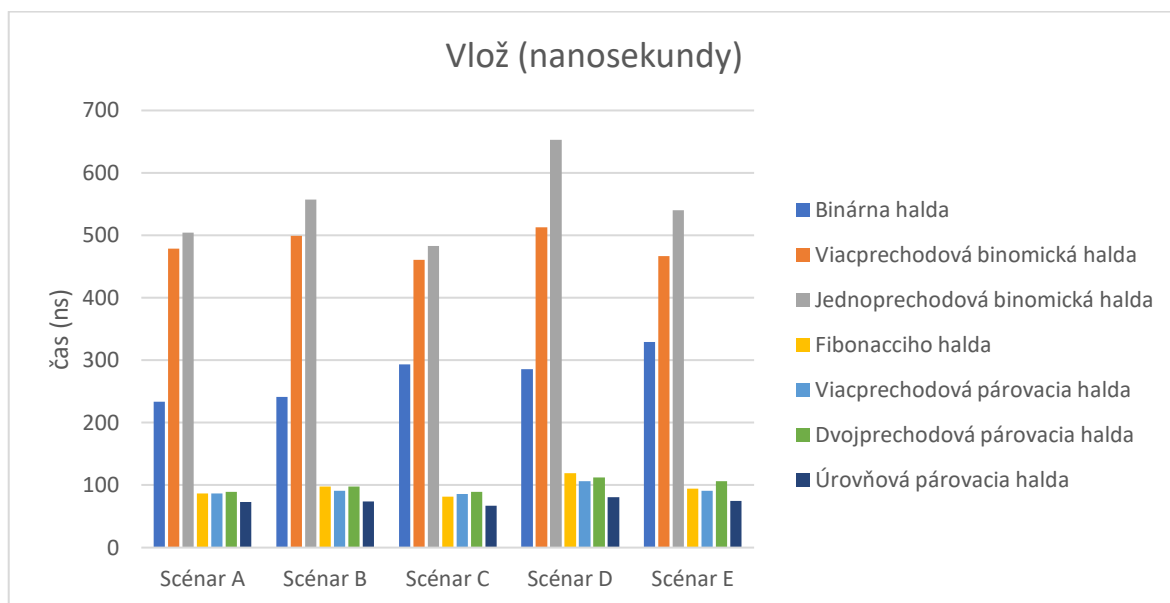
Obr. 22. Vývin binárnej haldy v pamäti pre scenár A

Použitie implicitného poľa ale znamená, že je potrebné alokovať väčšiu časť pamäte dopredu. To je znázornené skokmi v alokovanej pamäti na predchádzajúcom obrázku.

Tabuľka 3. Tabuľka priemerných časov pre vykonanie operácie

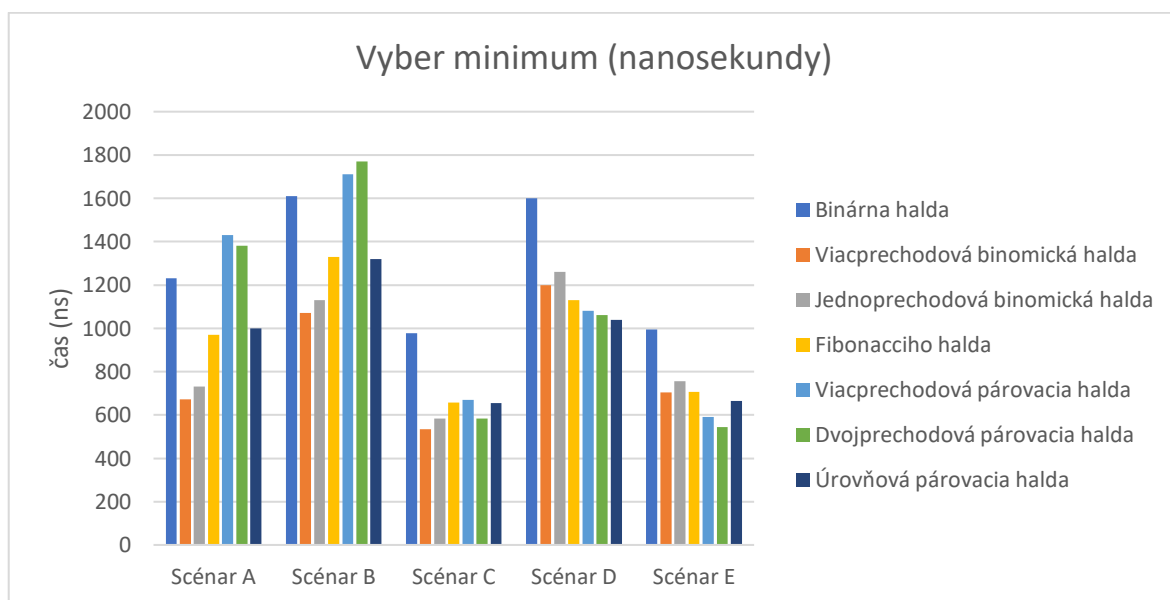
	Štruktúra	Scenár A	Scenár B	Scenár C	Scenár D	Scenár E
Priemerný čas operácie vlož (ns)	BH	234	241	293	286	329
	BNVP	479	499	461	513	467
	BNJP	504	557	483	653	540
	FH	86,5	97,7	81,8	119	94,2
	PHVP	87	91,3	85,8	106	91,2
	PHDP	89,1	97,6	89,1	112	106
	UPH	73,1	74,3	66,9	80,3	74,4
Priemerný čas operácie vyber minimum (ns)	BH	1230	1610	977	1600	995
	BNVP	671	1070	535	1200	705
	BNJP	730	1130	583	1260	757
	FH	969	1330	657	1130	706
	PHVP	1430	1710	669	1080	591
	PHDP	1380	1770	584	1060	544
	UPH	1000	1320	654	1040	665
Priemerný čas operácie zmenš prioritu (ns)	BH		607		567	633
	BNVP		565		668	925
	BNJP		612		699	1000
	FH		386		372	415
	PHVP		366		325	255
	PHDP		446		375	318
	UPH		902		783	796

Graf 1. Graf priemerných časov operácie vlož pre sadu testov 1



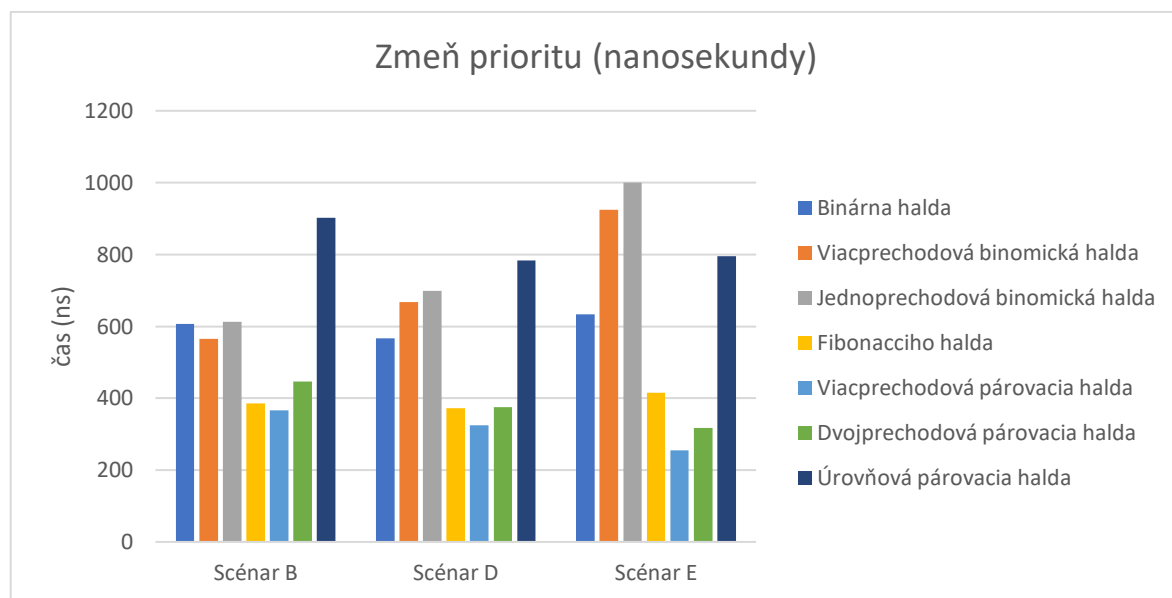
Z výsledkov pre vkladanie je možné vidieť, že časová náročnosť pre vkladanie je najvyššia pri binomickej halde, čo je spôsobené, ako už bolo spomenuté, zlučovaním štruktúry po vložení prvku. Binárna halda v sebe po vložení vymieňa prvky, avšak táto operácia je časovo menej náročná ako zlučovanie. Tento rozdiel je spôsobený tým, že pri zlučovaní je nutné prejsť všetky binomické stromy v zozname, kdežto pri binárnej halde sa prvky vymieňajú len dovtedy, pokiaľ nie je splnené haldové usporiadanie. Vkladanie do zvyšných hald je časovo nenáročné, nakoľko postačuje len upraviť ukazovateľov, ako bolo spomenuté pri popise implementácií.

Graf 2. Graf priemerných časov operácie vyber minimum pre sadu testov 1



Pri operácií vyber minimum je možné spozorovať výkyv efektívnosti párovacej haldy medzi scenármi A a C, ktoré je spôsobené tvarom haldy. Táto halda je tým rýchlejšia, čím je podobnejšia binárnemu stromu. Nato však musí dochádzať k častému zlučovaniu, ktoré prebieha výlučne pri operácií vyber minimum. To znamená, že zlučovanie je tým pomalšie, čím je vyšší pomer operácií vlož a vyber minimum.

Graf 3. Graf priemerných časov operácie zmeň prioritu pre sadu testov 1



Pri operácií zmeň prioritu nastáva v binomickej halde pri nižších hodnotách skok, čo je spôsobené našou implementáciou tejto operácie, kde ak sa chce prvok stať koreňom binomického stromu, musí byť celý les binomických stromov prejdený, aby boli správne zachované referencie na korene týchto stromov. Pri nižšom počte prvkov má obnovenie lesa binomických stromov väčší vplyv na trvanie operácie.

3.2 Testová sadá 2

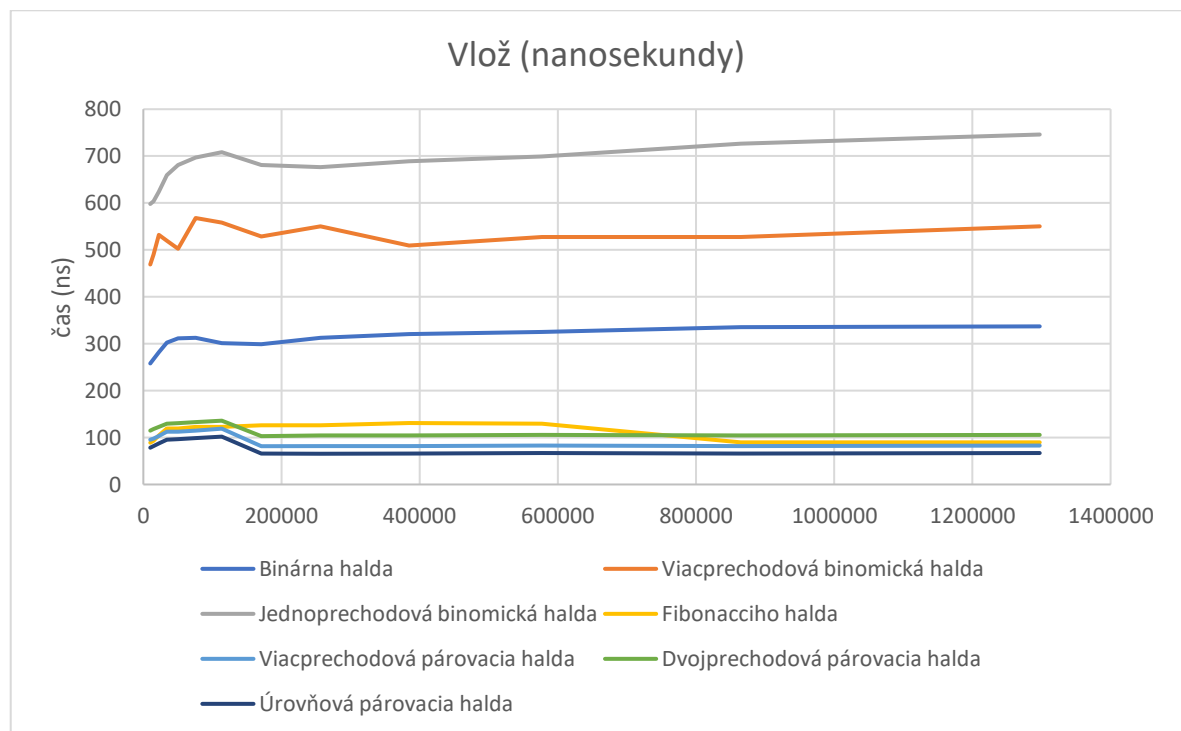
Scenár A - do jednotlivých prioritných frontov je postupne vkladáných n prvkov, kde $n_i = n_{i-1} * \frac{3}{2}$, $n_0 = 10000$. Po každej sérii vkladania sa vykoná 100 operácií vlož, vyber minimum a zmeň prioritu v náhodnom poradí, kde sa zmeria priemerný čas týchto operácií. Po vykonaní sú prioritné fronty vyčistené - sú z nich vymazané všetky prvky. Týmto scenárom sledujeme výkon operácií v najhoršom prípade, keď v niektorých implementáciách nedochádza dlhú dobu k zlučovaniu prvkov.

Scenár B - do prioritných frontov je vkladáných n prvkov tak, že po každých dvoch vložených prvkoch je vybraný minimálny prvok. Počet prvkov v prioritných frontoch je po

každej sérii vkladania rovnaký ako pri scenári A. Následne sa vykoná 10000 iterácií, kde každá iterácia spočíva zo 100 operácií vlož, vyber minimum a zmen prioritu v náhodnom poradí. Je zmeraný priemerný čas operácií cez všetky iterácie. Po skončení iterácií sú prioritné fronty vyprázdnené. Týmto scenárom sledujeme výkon operácií v amortizovanom prípade, keď je pomer operácií vlož a vyber minimum rovnaký.

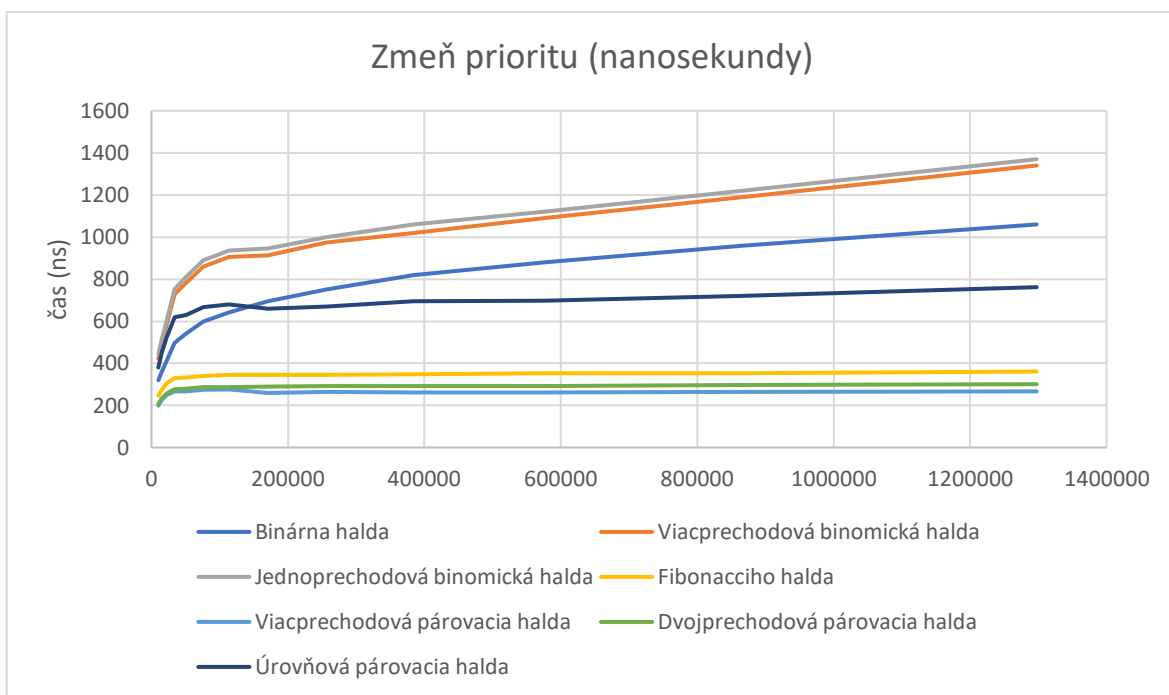
Nakoľko výkon operácií vlož a zmen prioritu silno nezáleží od stavu stromu, výsledné časy týchto operácií budú znázornené len grafom v amortizovanom prípade.

Graf 4. Graf vývinu časov operácie vlož pre sadu testov 2



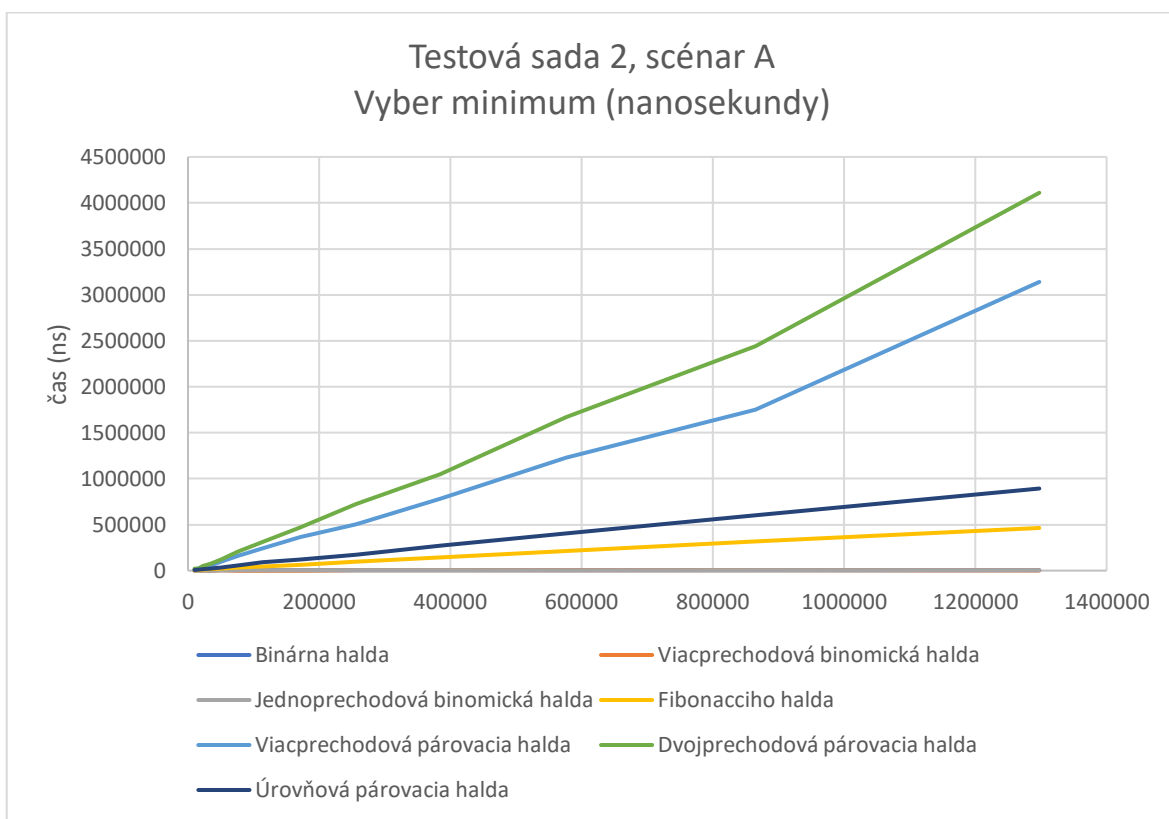
Pri operácií vlož sú viditeľné konštantné rýchlosti vkladania do Fibonacciho haldy, úrovňovej párovacej haldy a do párovacích hald. Binomická halda zlučuje prvky po vložení, čo má za následok vysoké časy pre vkladanie. Binárna halda vkladá prvky na poslednú úroveň a tie potom vymieňa smerom ku koreňu. Keďže nie je potrebné aby sa vymieňali až po koreň, priemerný čas pre vloženie je nižší ako pri binomickej halde.

Graf 5. Graf priemerných časov operácie zmeň prioritu pre sadu testov 2

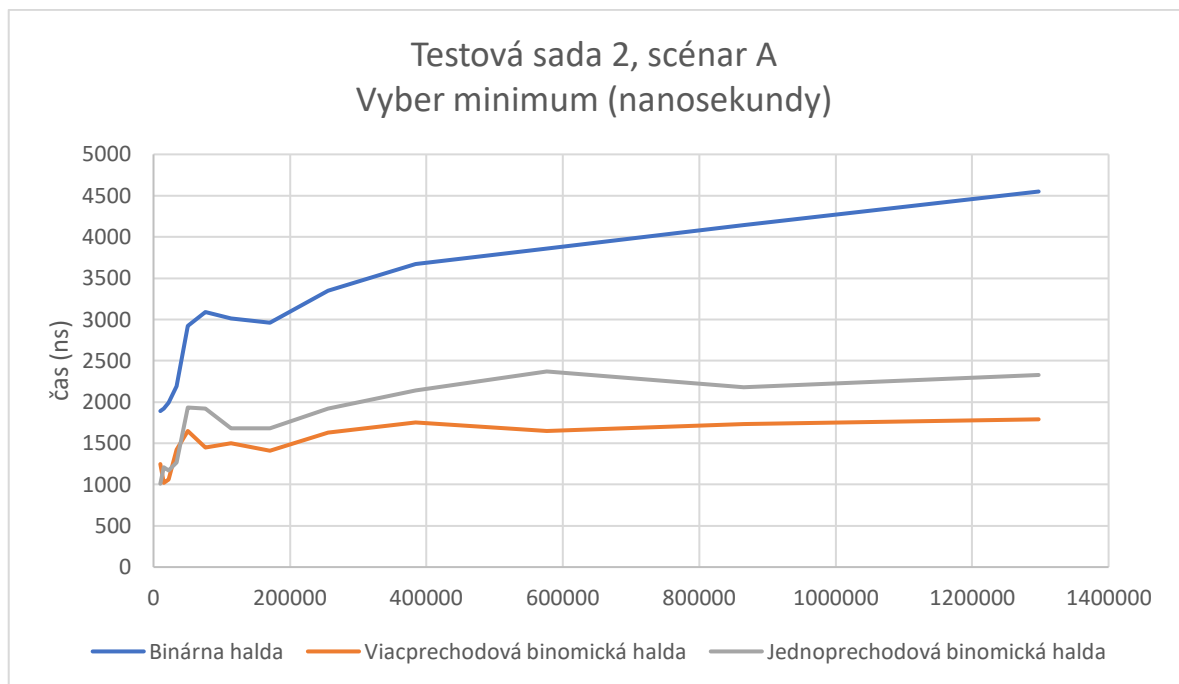


Pri párovacích haldách, Fibonacciho halde a úrovňovej párovacej halde sa nám podarilo potvrdiť konštantu amortizovanú rýchlosť operácie zmeň prioritu, avšak pri úrovňovej párovacej halde má operácia zmeň prioritu značne vyšší konštantný faktor.

Graf 6. Graf vývinu časov operácie vyber minimum pre sadu testov 2, scenár A

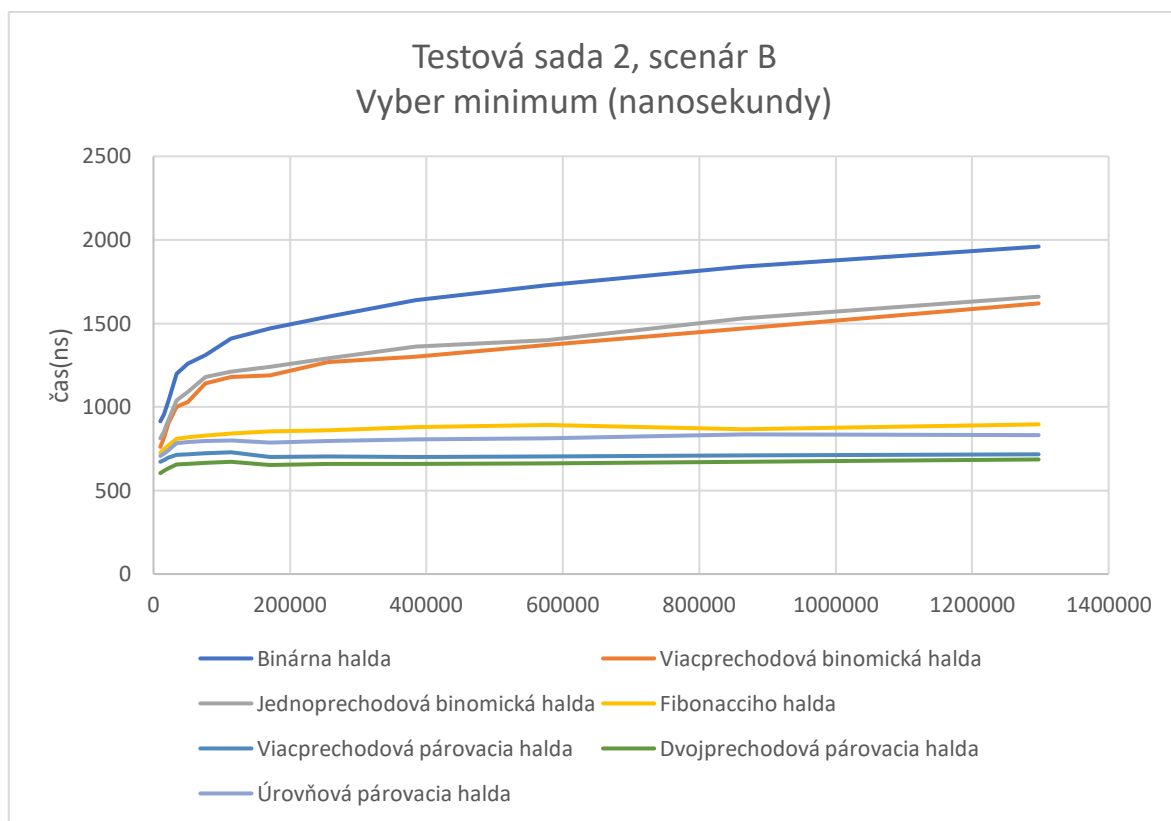


Graf 7. Graf vývinu časov operácie vyber minimum pre sadu testov 2, scénár A



Pri operácií vyber minimum mal veľký vplyv na výkon štruktúr počiatočný stav haldy. Fibonacciho halda, úrovňová párovacia halda a párovacie haldy dosahujú v prípade, že je v štruktúre množstvo nezlúčených prvkov, časovú náročnosť $O(n)$, čo môže nastať pri častom vkladaní do štruktúry a zriedkavom výskyte operácie vyber minimum. Avšak prvky Fibonacciho a úrovňová párovacia haldy sa už po prvej operácií vyber minimum zlúčia do binomických stromov, čo má za následok značne nižšiu priemernú časovú náročnosť počas prvých 100 operácií vyber minimum, ako párovacia halda, kde dochádza k postupnému zlúčeniu. Binomické haldy a binárna halda si zachovali logaritmické časy výberu.

Graf 8. Graf vývinu časov operácie vyber minimum pre sadu testov 2, scenár B



Čím je počet operácií vlož a vyber minimum podobnejší, tým výkon párovacích hálď narastá. Výkon Fibonacciho a úrovňovej párovacej haldy by mal byť podobný viacprechodovej binomickej halde, avšak to sa nám nepodarilo dokázať.

Záver

Ako bolo spomenuté v úvode, hlavným cieľom práce bolo porovnať prioritné fronty. Začali sme zhromaždením informácií o prioritných frontoch, vymedzením spoločných vlastností jednotlivých implementácií prioritných frontov a vytvorením teoretického modelu. Tento model bol popísaný v prvej časti bakalárskej práce a implementovaný v druhej časti.

Na základe testov sme dospeli k záveru, že nie je štruktúra, ktorá by bola objektívne najlepšia. Fibonacciho halda je síce časovo efektívna, avšak je komplikovaná na programovanie a zároveň je pamäťovo najnáročnejšia. Binárna halda je veľmi efektívna pre menšie množstvá dát, ale s rastúcim počtom prvkov dopláca na jej implicitnú implementáciu tým, že musí byť uložená v súvislej pamäti a jej nutnosťou používať stratégie rozširovania, ktoré zbytočne zaberajú pamäť, ktorú momentálne nepotrebuje. Je však vhodná na triedenie prvkov, nakoľko zostaviť binárnu haldu je možné v $O(n)$, ako sme si uviedli v časti 1.1. Párovacia halda sa síce ukázala ako najlepšia v amortizovanom prípade, to však silne závisí od tvaru haldy, čiže počtu operácií vkladania a vyberania prvku. Je ich vhodné použiť, ak sú počty oboch operácií podobné. Binomická halda sa ukázala efektívna v prípadoch, keď je nutné časté vkladanie a zriedkavý vyber. V našich testoch sa podarilo preukázať, že viacprechodová verzia je efektívnejšia ako jednoprechodová, aj keď musí zlučovať prvky viackrát. To je spôsobené tým, že aj keď sa v jednoprechodovej verzii spájajú prvky zriedkavejšie, les binomických stromov je väčší ako pri viacprechodovej. Úrovňová párovacia halda dosiahla podobné výsledky pri operáciách vyber minimum a vlož ako Fibonacciho halda, avšak pri operácií zmen prioritu malá vyšší, aj keď v amortizovanom prípade konštantný, priemerný čas vykonania operácie. Je však pamäťovo menej náročná ako Fibonacciho halda.

Zoznam použitej literatúry

- [1] FORSYTHE, George E. Algorithm 232: Heapsort. In *Communications of the ACM*, 1964, roč. 7, č. 6, s. 347-348. doi:10.1145/512274.512284
- [2] FLOYD, R. W. Algorithm 245: Treesort. In *Communications of the ACM*, 1964, roč. 7, č. 12, s. 701. doi:10.1145/355588.365103
- [3] SUCHENEK, M. A. Elementary Yet Precise Worts-Case Analysis of Floyd's Heap-Construction Program. In *Fundamenta Informaticae*, 2012, roč. 120, č. 1, s. 75-92, doi: 10.3233/FI-2012-751
- [4] VUILLEMIN, Jean. A data structure for manipulating priority queues. In *Communications of the ACM*, 1978, roč. 21, č. 4, s. 309-315. doi: 10.1145/359460.359478
- [5] KNUTH, D. E. The Art of Computer Programming Volume 1: Fundamental Algorithms. Addison Wesley Longman Publishing Co., Inc., 1997. 650 s.
- [6] HAEUPLER, Bernhard - TARJAN, Robert E. Rank-Pairing Heaps. In *SIAM Journal on Computing*, 2011 roč. 40 č. 6, s. 1463–1485. doi: 10.1137/100785351
- [7] CORMEN, Thomas H. - STEIN Clifford. Introduction to algorithm. London: The MIT Press, 2009. 1271 s.
- [8] JOHNSON, Donald B. Priority queues with update and finding minimum spanning. In *Information Processing Letters*, 1975, roč. 4, č. 3, s. 53-57. doi: 10.1016/0020-0190(75)90001-0
- [9] FREDMAN, Michael L., TARJAN Robert E. Fibonacci heaps and their uses in improved network optimization algorithms. In *Journal of the ACM*, 1987, roč. 34, č. 3, s. 596–615. doi: 10.1145/28869.28874
- [10] FREDMAN M. L. - TARJAN R. E. The Pairing Heap : A New Form of Self-Adjusting Heap. In *Algorithmica*, 1986, roč. 1, s. 111-129. doi: 10.1007/BF01840439
- [11] doc. Mgr. KLÍMA Valent, CSc., Prednášky k predmetu Údajové štruktúry 2 - Párovacia halda, Fibonacciho halda.

Zoznam príloh

Príloha A : DVD so zdrojovým kódom