Struktury Cache-Oblivious – Kolejki priorytetowe

Paweł Ledwoń

16 stycznia 2011

Spis treści

1 Wstęp

1.1 Kolejki priorytetowe

Kolejką priorytetową nazywamy strukturę danych służącą do przechowywania elementów ze zbioru, na którym określona jest pewna relacja porządku.

Na kolejce możemy wykonywać następujące operacje:

- *insert* – wstawienie elementu do kolejki,
- *delete_min* – wybranie i usunięcie z kolejki najmniejszego elementu.

Najprostsza implementacja kolejki priorytetowej wykorzystuje kopiec binarny, osiągając dla obu operacji złożoność obliczeniową $O(\log n)$ w modelu RAM. Niestety, kopce nie były projektowane z myślą o współczesnej architekturze sprzętowej, w związku z czym poszukiwane są efektywniejsze struktury danych.

1.2 Model I/O

Dostęp do danych w pamięci RAM jest operacją znacznie kosztowniejszą niż odczyt wiersza z cache procesora. Schodząc w dół hierarchii, mamy do czynienia z co raz wolniejszymi pamięciami. Tymczasowo jednak założymy, że dysponujemy tylko pamięcią główną i podręczną (cache).

Model I/O narzuca na algorytmy następujące ograniczenia:

- pamięć jest podzielona na **wiersze**, każdy o długości B słów,
- pamięć podręczna ma określony rozmiar M słów,
- operacje na danych wykonywane są w miarę możliwości w cache,
- jeśli danych nie ma w pamięci podręcznej, zachodzi **cache miss**,
- cache miss skutkuje pobraniem **całego bloku** z pamięci głównej do cache.

Złożoność algorytmu w modelu I/O mierzymy przez ilość *cache miss*. Możemy wciąż liczyć złożoność obliczeniową, lecz liczba wykonanych transferów w pamięci jest zwykle lepszą metryką.

Znane są ograniczenia na złożoność wielu algorytmów, z których wyróżniamy:

- $scan(N) = \Theta(\frac{N}{B})$ – odczytanie N ciągłych elementów,
- $sort(N) = \Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ – sortowanie N elementów.

Model I/O nie ogranicza nas do dwupoziomowej hierarchii pamięci – możemy ustalać dodatkowe poziomy o różnych parametrach, aby zbliżyć się do architektury rzeczywistego sprzętu.

1.3 Model Cache-Oblivious

Mimo tego, że kilkuwarstwowy model I/O dość dobrze przedstawia dzisiejszą architekturę sprzętową, jego zastosowanie przy konstruowaniu algorytmów jest dość niewygodne. Mając trzy warstwy pamięci podręcznej procesora, pamięć RAM, dysk SSD (jako cache) i dysk magnetyczny, musimy uwzględnić interakcje między sześcioma warstwami pamięci o różnych parametrach.

Model Cache-Oblivious rozszerza model I/O o następujące ograniczenia:

- pamięć ma dwa poziomy – główną oraz cache,
- cache jest **w pełni asocjacyjny**,
- **tall cache assumption**: rozmiar cache jest rzędu $M = \Omega(B^2)$ słów,
- polisa wywłaszczania w cache jest optymalna (offline),
- algorytmy nie mają wiedzy o parametrach cache.

Wynika z tego, że algorytm cache-oblivious pozostaje optymalny dla dowolnych parametrów pamięci. Mniej oczywistym wnioskiem jest fakt, że algorytm zachowuje swoją optymalność dla wielopoziomowych hierarchii pamięci, co czyni go atrakcyjnym narzędziem do projektowania algorytmów.

2 Podejście pierwsze – Arge, Bender, Demaine

Pierwszą optymalną kolejkę priorytetową przedstawiono w 2002 roku w pracy *Cache-Oblivious Priority Queue and Graph Algorithm Applications* autorstwa L. Arge, M. Bendera, E. Demaine, B. Holland-Minkleya i I. Munro.

Wspierane przez nią operacje *insert*, *delete* i *delete_min* działają ze zamortyzowaną ilością $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ transferów z pamięci. Aby osiągnąć tę granicę, wykorzystane zostały znane algorytmy cache-oblivious, struktura **buffer tree** czy **schemat M/B-drożnego łączenia** list.

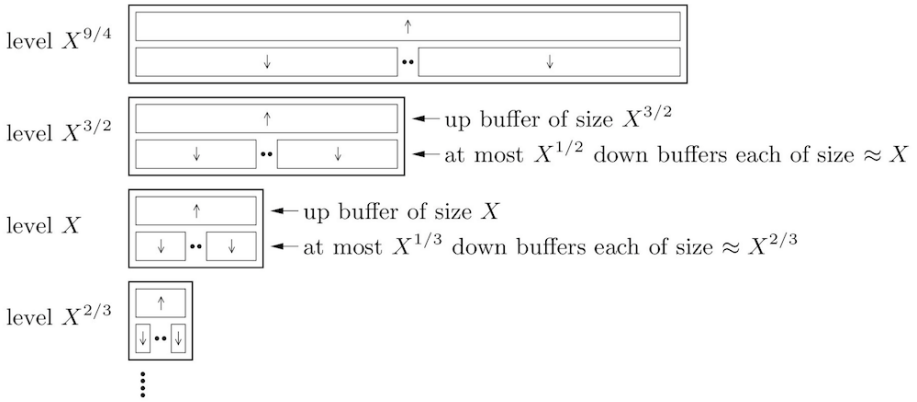
2.1 Struktura

2.1.1 Poziomy

Dane w kolejce przechowywane są na $\Theta(\log\log N)$ poziomach o rozmiarach równych od N do wyznaczonej z góry stałej c. Rozmiar poziomu odpowiada asymptotycznie liczbie elementów, które mogą być na nim przechowywane.

Poziom o numerze i od góry ma rozmiar $N^{(2/3)^{i-1}}$. Dla wygody poziomy oznaczamy rozmiarami, a nie indeksami. Zgodnie z przyjętą konwencją, idąc od góry, kolejne poziomy to: N, $N^{2/3}$, $N^{4/9}$, $\ldots$, $X^{3/2}$, X, $X^{2/3}$, $\ldots$, $c^{3/2}$ oraz c.

Intuicyjnie, niższe poziomy przechowują mniejsze elementy. Operacje wykonywane są, w miarę możliwości, na najniższych poziomach.



Rysunek 1: Poziomy i bufory w kolejce priorytetowej.

2.1.2 Bufory

Każdy poziom posiada określoną liczbę buforów, w których przechowywane są elementy kolejki. Poziom X posiada jeden **górny bufor** u^X o rozmiarze X oraz do $X^{1/3}$ **dolnych buforów** $d_1^X, \ldots, d_{X^{1/3}}^X$ przechowujące od $\frac{1}{2}X^{2/3}$ do $2X^{2/3}$ elementów. Wynika z tego, że na poziomie X może być przechowywane maksymalnie $3X$ elementów. Można również zauważyć, że górny bufor u^X ma (z dokładnością do stałej) rozmiar dolnego bufora $d^{X^{3/2}}$.

2.1.3 Własności buforów

Elementy w buforach muszą spełniać następujące niezmienniki:

Niezmiennik 1. *Na poziomie X elementy są posortowane **między** buforami.*

Ściśle mówiąc, elementy bufora d_i^X mają klucze mniejsze niż elementy d_{i+1}^X. Porządek wewnątrz poszczególnych buforów jest natomiast nieokreślony.

Niezmiennik 2. *Elementy w buforach d^X są mniejsze niż w buforze u^X.*

Niezmiennik 3. *Elementy w buforach d^X są mniejsze niż w buforach $d^{X^{3/2}}$.*

Powyższe własności gwarantują, że idąc w dół struktury będziemy napotykać mniejsze elementy. Dodatkowo, zachowany jest porządek pomiędzy dolnymi i górnymi buforami na danym poziomie. Nie istnieje jednak porządek pomiędzy elementami bufora u^X a dolnymi buforami $d^{X^{3/2}}$ na wyższym poziomie.

Intuicyjnie, elementy dolnych buforów są kandydatami do przeniesienia na niższy poziom. Analogicznie, elementy z górnego bufora mają szansę być przeniesione wyżej. Najmniejszy element w kolejce znajduje się w buforze d_1^c.

2.1.4 Układ pamięci

Dane przechowywane są w ciągłym obszarze pamięci, zaczynając od najniższego poziomu. Dane na każdym poziomie zapisywane są również w sposób ciągły. Poziom X rezerwuje miejsce na $3X$ elementów – X na bufor u^X oraz po $2X^{2/3}$ na każdy z $X^{1/3}$ dolnych buforów. Górny bufor przechowywany jest pierwszy, dolne bufory w dowolnej kolejności, lecz muszą zostać zorganizowane w posortowaną listę łączoną.

Lemat 1. *Kolejka priorytetowa wykorzystuje pamięć rzędu $O(N)$.*

Dowód. Zgodnie z powyższym opisem, rozmiar kolejki to:

$$\sum_{i=0}^{\log_{3/2}\log_c N} 3N^{(2/3)^i} = O(N)$$

□

2.2 Operacje

Wstawienie elementu do kolejki jest jednoznaczne ze wstawieniem go na najniższy poziom, natomiast pobranie minimum sprowadza się do usunięcia najmniejszego elementu z pierwszego dolnego bufora na poziomie c.

Gdy po operacji wstawienia bufory zostaną przepełnione, nadmiar elementów należy „przepchnąć" poziom wyżej. W przypadku operacji usunięcia minimum, dolne bufory mogą zawierać zbyt mało elementów, w związku z czym będą musiały zostać zapełnione elementami z wyższych poziomów.

Implementacja wymaganych do tego procedur została opisana w następnych podrozdziałach.

2.2.1 Push

Operacja *push* wstawia wybrane X elementów na poziom $X^{3/2}$ z zachowaniem niezmienników opisanych w rozdziale 2.1.3.

Pierwszym krokiem algorytmu jest posortowanie listy X elementów. Sortowanie możemy wykonać wykorzystując optymalny algorytm cache-oblivious wykorzystujący $O(1 + \frac{X}{B}\log_{M/B}\frac{X}{B})$ trasferów z pamięci.

Mając uporządkowaną listę wejściową, możemy dołączyć jej elementy do dolnych buforów $d^{X^{3/2}}$. Operacja łączenia polega na sekwencyjnym dodawaniu elementów do kolejno odwiedzanych buforów. Do następnego bufora przechodzimy, gdy wstawiany element jest większy od piwota (największy element) obecnego bufora. Elementy większe od piwota ostatniego dolnego bufora wstawiane są do bufora górnego.

Przeskanowanie X posortowanych elementów kosztuje $O(1 + X/B)$ transferów. Dołączając elementy, może zaistnieć potrzeba odwiedzenia każdego z $X^{1/2}$ buforów, nawet jeśli nie zmieniamy ich zawartości. Sumaryczny koszt dołączania nowych elementów to $O(X^{1/2} + \frac{X}{B})$.

W trakcie wstawiania elementów mogą wystąpić trzy przypadki wymagające podjęcia dodatkowych działań.

Przypadek 1. *Bufor $d_i^{X^{3/2}}$ zawiera $2X$ elementów.*

Jeśli bufor jest pełny, należy podzielić go na dwie równe części. Wpierw szukamy mediany, co kosztuje $O(1 + X/B)$ transferów z pamięci. Następnie przenosimy elementy do drugiego bufora liniowo przechodząc przez listę w pierwszym buforze (pamiętając, że elementy wewnątrz bufora nie muszą być uporządkowane).

Całkowity koszt tej operacji to $O(1 + X/B)$. Jako że od ostatniego podziału bufora musiało zostać do niego dodane X elementów, zamortyzowany koszt tej operacji to $O(1/X + 1/B)$.

Przypadek 2. *Liczba buforów przed podziałem to $X^{1/2}$.*

Żeby zwolnić miejsce do podziału, usuwamy ostatni dolny bufor, a jego elementy (mniej niż $2X$) przenosimy do bufora górnego. Ponownie, możemy wykonać to wykorzystując $O(1 + X/B)$ transferów.

Przypadek 3. *Bufor górny ulega przepełnieniu.*

W tym przypadku przenosimy wszystkie jego elementy (razem z nadmiarem) poziom wyżej. Wykonujemy w tym celu rekurencyjnie operację *push*. Zignorujmy na ten czas koszt tego wywołania.

Lemat 2. *Koszt zamortyzowany operacji push to $O(X^{1/2} + \frac{X}{B} \log_{M/B} \frac{X}{B})$.*

2.2.2 Pull

Operacja *pull* przenosi X najmniejszych elementów z poziomu $X^{3/2}$ w dół.

Przypadek 1. *Dolne bufory zawierają co najmniej $\frac{3}{2}X$ elementów.*

Bufory $d_1^{X^{3/2}}$, $d_2^{X^{3/2}}$ i $d_3^{X^{3/2}}$ zawierają w sumie od $\frac{3}{2}X$ do $6X$ elementów. Sortujemy ich zawartość i wybieramy X najmniejszych elementów. Pozostałe elementy (od $\frac{1}{2}X$ do $5X$) rozprowadzamy, w zależności od ich liczby, do jednego, dwóch lub trzech buforów. Koszt konstrukcji nowych buforów to $O(1+X/B)$, co razem z sortowaniem daje $O(1 + \frac{X}{B} \log_{M/B} \frac{X}{B})$ transferów w pamięci.

Przypadek 2. *Dolne bufory zawierają mniej niż $\frac{3}{2}X$ elementów.*

Aby wypełnić bufory, rekurencyjnie wykonujemy operację *pull* otrzymując posortowaną listę $X^{3/2}$ nowych elementów. Nie wiemy jednak, które z elementów powinny znaleźć się w górnym buforze, a które w dolnych.

Załóżmy, że bufor $u^{X^{3/2}}$ zawiera U elementów. Sortujemy jego zawartość i łączymy liniowo z poprzednio otrzymaną listą. Największe U elementów wstawiamy ponownie do górnego bufora, resztę (od $X^{3/2}$ do $X^{3/2} + \frac{1}{2}X$) rozprowadzamy do dolnych buforów. Zauważmy, że kolejne wywołanie rekurencyjne nastąpi po usunięciu kolejnych $X^{3/2}$ elementów.

Na koniec odnajdujemy i zwracamy minimalne X elementów tak, jak w pierwszym przypadku. Koszt dołączania elementów (jedno sortowanie i jedno skanowanie) jest zdominowany przez koszt rekurencyjnego wywołania.

Koszt operacji *pull*, ponownie ignorując wywołanie rekurencyjne, to zamortyzowane $O(1 + \frac{X}{B} \log_{M/B} \frac{X}{B})$ transferów danych.

3 Funnel Heap – Brodal, Fagerberg

Ciekawszą strukturę kolejki zaprezentowali G. S. Brodal i R. Fagerberg w pracy *Funnel Heap – A Cache Oblivious Priority Queue*. Ich pomysł wykorzystuje **binary mergery**, na których bazuje Funnel Sort – optymalny algorytm sortowania cache-oblivious.

3.1 Binary merger

Binary merger przyjmuje jako wejście dwa strumienie posortowanych elementów i łączy je w jeden posortowany strumień wyjściowy. Początki strumieni przechowywane są w osobnych buforach o ograniczonym rozmiarze. Bufor jest zwykłą tablicą elementów, przy której dodatkowo zapisujemy jej pojemność oraz wskaźniki na pierwszy i ostatni element.

Binary mergery mogą być łączone w **binary merge tree**, w którym bufory wyjściowe jednych mergerów mogą być wejściem mergerów na wyższym poziomie drzewa. Liście zawierają strumienie wejściowe, które chcemy połączyć, wierzchołki wewnętrzne są mergerami, a krawędzie buforami.

Wywołanie mergera jest procedurą wypełniającą w całości jego bufor wyjściowy. Jeśli bufor wejściowy jest pusty, rekurencyjnie zostaje wywołana procedura wypełniająca mergera, którego wyjściem jest ten bufor. Jeżeli oba bufory wejściowe zostały opróżnione, a na wyjściu nie ma żadnych elementów, oznaczamy bufor również jako opróżniony.

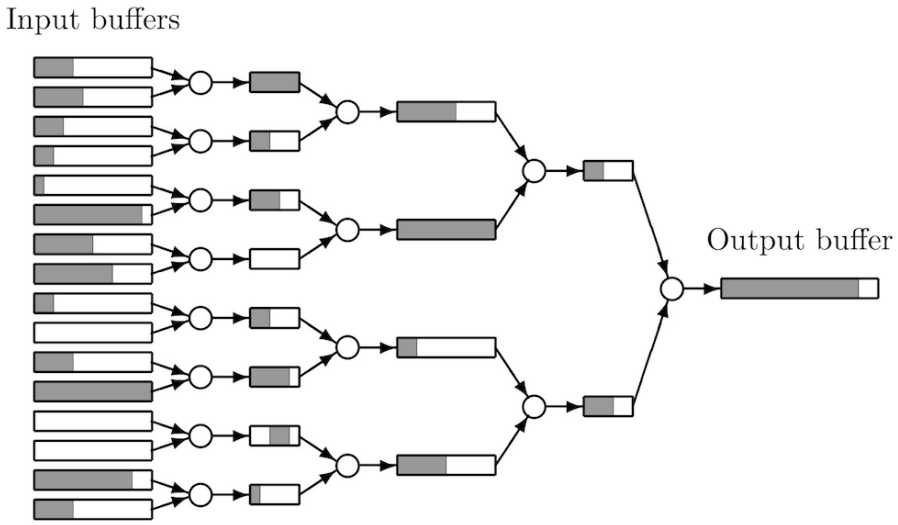
3.1.1 k-merger

Jedną z odmian merge tree jest **k-merger**. Ustalmy $k = 2^i$ dla pewnej liczby dodatniej i. k-merger jest pełnym drzewem binarnym łączącym k strumieni wejściowych przy użyciu $k-1$ binary mergerów. Bufor wyjściowy w jego korzeniu ma rozmiar k^3.

Rozmiary buforów wyznaczane są rekurencyjnie. *Górnym drzewem* nazywamy poddrzewo zawierające wszystkie wierzchołki z najwyżej $\lceil i/2 \rceil$ poziomu. Drzewa zawierające wierzchołki z poziomów $\lceil i/2 \rceil + 1$ i wyższych nazywamy *drzewami dolnymi*. Bufory pomiędzy wierzchołkami z poziomów $\lceil i/2 \rceil$ i $\lceil i/2 \rceil + 1$ mają pojemność $\lceil k^{3/2} \rceil$ elementów.

Jak w większości struktur danych cache-oblivious, aby osiągnąć efektywność I/O, układ k-mergera jest określony rekurencyjnie. Cały k-merger zajmuje ciągły obszar pamięci, wpierw przechowywane jest górne drzewo, potem bufory ze środka drzewa, a na koniec dolne drzewa. Układ ten jest wykorzystywany rekurencyjnie w drzewach o mniejszym rozmiarze.

Lemat 3. *Wywołanie k-mergera wykorzystuje $O(k + \frac{k^3}{B} \log_{M/B} k^3)$ transferów. Rozmiar k-mergera jest rzędu $O(k^2)$, nie licząc danych z wejścia i wyniku.*



Rysunek 2: 16-merger. Szary kolor przedstawia zapełniony obszar bufora.

3.2 Struktura kolejki

Funnel heap składa się z listy k-mergerów, gdzie k rośnie podwójnie wykładniczo. Pomiędzy k-mergerami znajdują się bufory i binary mergery. Całość została przedstawiona na rys. 3.

Zdefiniujmy następująco liczby k_i oraz s_i:

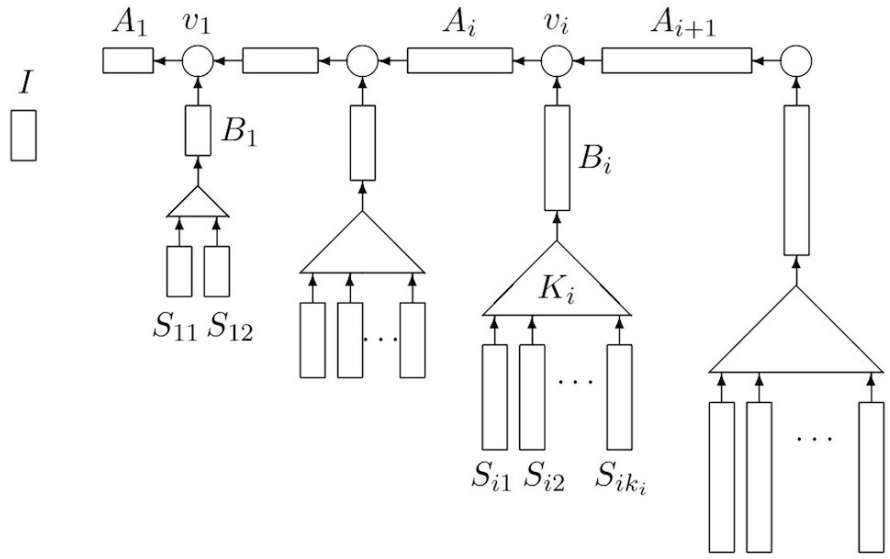
$$\begin{aligned} s_1 &= 8 \\ k_1 &= 2 \\ s_{i+1} &= s_i(k_i+1) \\ k_{i+1} &= \lceil\lceil s_{i+1}^{1/3}\rceil\rceil \end{aligned}$$

gdzie $\lceil\lceil x\rceil\rceil = 2^{\lceil \log x\rceil}$.

Ogniwo i-te kolejki składa się z binary mergera v_i, buforów A_i i B_i oraz k_i-mergera K_i z buforami wejściowymi $S_{i,1}, \ldots, S_{i,k_i}$. Wyjściem v_i jest bufor A_i, natomiast jego wejściami bufory B_i oraz A_{i+1}. Bufor B_i jest jednocześnie wyjściem mergera K_i. Rozmiar buforów A_i oraz B_i to k_i^3, natomiast $S_{i,j}$ ma rozmiar s_i. Elementy we wszystkich buforach są posortowane.

Razem ze strukturą i-tego ogniwa trzymamy licznik c_i, początkowo równy 1. Jego wartość oznacza, że bufory $S_{i,c_i}, \ldots, S_{i,k_i}$ są puste.

Kolejka posiada dodatkowo **insertion buffer** I o rozmiarze s_1. Bufory przechowywane są w ciągłym fragmencie pamięci, wpierw bufor I, a następnie lista ogniw wchodzących w skład kolejki. Każde z nich składa się kolejno z buforów A_i, B_i, K_i, $S_{i,1}, \ldots, S_{i,k_i}$.



Rysunek 3: Funnel heap.

Powyżej opisana lista tworzy drzewo binarne T, którego korzeniem jest v_1, a krawędzie są posortowanymi listami elementów kolejki. Drzewo jest posortowane zgodnie z porządkiem zachodzącym w kopcu – idąc w stronę korzenia napotykamy elementy w porządku malejącym.

3.3 Operacje

3.3.1 *delete_min*

Aby usunąć najmniejszy element, sprawdzamy najpierw, czy bufor A_1 jest pusty. Jeśli tak, wywołujemy merger v_1, który powoduje wypełnienie A_1. Następnie usuwamy i zwracamy minimum z buforów A_1 oraz I.

3.3.2 *insert*

Nowy element wstawiamy najpierw do bufora I zachowując w nim porządek. Jeśli bufor nie jest pełny, kończymy procedurę wstawiania.

Gdy bufor I zostanie zapełniony, wykonujemy procedurę *zamiatania* z parametrem i, który przyjmuje minimalną wartość, dla której $c_i \leqslant k_i$. Jej celem jest przeniesienie zawartości ogniw $1, \ldots, i-1$ do bufora S_{i,c_i}.

Pierwszym krokiem procedury jest zliczenie ilości elementów każdego bufora na ścieżce p z A_1 do S_{i,c_i}. Następnie tworzone są dwa posortowane strumienie danych:

σ_1 – elementy buforów na ścieżce z A_i do S_{i,c_i},

σ_2 – elementy wszystkich buforów z ogniw $1, \ldots, i-1$ oraz bufora I.

Strumień σ_2 powstaje przez oznaczenie bufora A_i jako wyczerpany i ciągłe wywoływanie procedury *delete_min*.

Strumienie σ_1 i σ_2 łączone są w strumień σ, którego elementy umieszczane są w kolejnych buforach na ścieżce p z zachowaniem ilości elementów sprzed momentu wywołania procedury wstawiania. Pozostała część σ trafia do S_{i,c_i}.

Na koniec dla $j = 1, 2, \ldots, i-1$ przypisujemy $c_j = 1$, a c_i inkrementujemy.

3.4 Analiza poprawności

Poprawność *delete_min* wynika z porządku ustalonego na drzewie mergerów. Musimy pokazać więc, że procedura *insert*:

- zachowuje porządek w drzewie,
- nie przepełnia bufora S_{i,c_i} przy zamiataniu.

Po wykonaniu *insert*, elementy na ścieżce p są najmniejszymi elementami ze strumienia σ, więc nie są większe od elementów, które na niej były przed wykonaniem procedury. Wynika z tego, że porządek w drzewie jest zachowany.

Oznaczmy B_l, K_l i bufory $S_{l,j}$ jako dolną część ogniwa l. Za każdym razem, gdy licznik c_l jest resetowany, dolna część l jest opróżniania, więc nie zawiera więcej elementów niż zostało wstawione do buforów $S_{l,j}$ od ostatniego zerowania licznika.

Możemy udowodnić indukcyjnie, że podczas zamiatania z parametrem i, do bufora S_{i,c_i} wstawiono co najwyżej $s_1 + \sum_{j=1}^{i-1} k_j s_j = s_i$ elementów.

3.5 Analiza kosztu

Chcemy udowodnić, że koszt wędrówki elementu z bufora wejściowego K_i w górę drzewa T to $O(\frac{1}{B} \log_{M/B} s_i)$. Zakładamy, że w pamięci podręcznej trzymane jest możliwie dużo małych ogniw kolejki. Optymalna polisa wymiany stron może przyczynić się jedynie do zmniejszenia ilości operacji I/O.

Oznaczmy przez Δ_i ilość komórek pamięci zajmowaną przez ogniwa $1, \ldots, i$. Zauważmy, że zachodzi $s_i^{1/3} \leqslant k_i \leqslant 2s_i^{1/3}$, więc bufory $S_{i,1}, \ldots, S_{i,k_i}$ zajmują $\Theta(s_i k_i) = \Theta(k_i^4)$ komórek pamięci, co dominuje rozmiar ogniwa i. Natomiast z nierówności $s_i^{4/3} < s_{i+1} < 3s_i^{4/3}$ wynika, że zarówno s_i jak i k_i rosną podwójnie wykładniczo względem i. Na podstawie tych dwóch spostrzeżeń możemy wywnioskować, że $\Delta_i = \Theta(k_i^4)$, gdyż rozmiar jest zdominowany przez ogniwo i. Oznaczmy więc przez i_M największe i, dla którego $\Delta_i \leqslant M$. Zakładamy, że pierwsze i_M ogniw przechowujemy zawsze w cache.

Rozważmy wędrówkę elementu z K_i do bufora B_i dla $i > i_M$. Z lematu 3. wynika, że każde wywołanie K_i powoduje $O(k_i + \frac{k_i^3}{B} \log_{M/B} k_i^3)$ operacji I/O. Na podstawie założenia na i_M wnioskujemy, że $M < \Delta_{i_M+1}$, co daje oszacowanie $M = O(k_i^4)$. Przyjmując tall-cache assumption ($B^2 \leqslant M$), otrzymujemy rozmiar cache $B = O(k_i^2)$, z czego wynika, że $k_i = O(\frac{k_i^3}{B})$.

Nie licząc wywołań K_i, które nie wypełniają do końca B_i (na przykład, gdy merger zostanie wyczerpany), każdy element podczas drogi do B_i jest obciążony kosztem $O(\frac{1}{B}\log_{M/B} k_i^3) = O(\frac{1}{B}\log_{M/B} s_i)$ operacji I/O.

Element może zostać również obciążony podczas wstawiania do buforów A_j, dla $j = i_M, \ldots i$ (pozostałe są w cache). Wypełnienie A_j wymaga $O(1 + |A_j|/B)$ transferów. Skoro $B = O(k_{i_M+1}^2) = O(k_{i_M}^{8/3})$ i $|A_j| = k_j^3$, to koszt wstawienia jednego elementu jest równy $O(1/B)$ I/O. Na podstawie $M = O(k_{i_m+1}^4) = O(s_{i_M}^{16/9})$ otrzymujemy $i - i_M = O(\log\log_M s_i) = O(\log_M s_i)$. Korzystając z tall cache assumption otrzymujemy $i - i_M = O(\log_{M/B} s_i)$. Wynika z tego, że koszt wstawiania elementu do buforów A_j nie jest większy od kosztu wybrania go z K_i.

Do ukończenia dowodu pozostaje nam wliczenie kosztu operacji zamiatania. Możemy udowodnić indukcyjnie, że pomiędzy kolejnymi wywołaniami procedury zamiatania dla ogniwa i zostanie wstawione przynajmniej s_i elementów. Pozwalamy tym elementom pokryć koszt ich wędrówki w górę T, który jest równy $O(\frac{1}{B}\log_{M/B} s_i)$ na element, co udowodniliśmy powyżej. Będziemy również naliczać tym elementom koszt tworzenia i łączenia strumieni σ_1, σ_2 i σ oraz operacje I/O potrzebne do wypełnienia wyczerpanych buforów.

Budowa strumienia σ_1 polega na przejściu fragmentu ścieżki p od bufora A_i do S_{i,c_i}. Dzięki ustalonemu układowi danych w pamięci, wystarczy liniowo przeskanować fragment pamięci pomiędzy A_1 a końcem K_i, co kosztuje nas $O((\Delta_{i-1} + |A_i| + |B_i| + |K_i|)/B) = O(k_i^3/B) = O(s_i/B)$ operacji I/O. Budowa σ_2 została wliczona wcześniej w koszta operacji *insert*. Łączenie σ_1 i σ_2 w σ i rozmieszczenie elementów w pamięci kosztują najwyżej tyle, co przejście ścieżki p oraz bufora S_{i,c_i}, co daje $O(s_i/B)$ transferów.

Zauważmy, że bufor B_i (a co za tym idzie A_i) może zostać wyczerpany tylko raz pomiędzy zamiataniem ogniwa i. Obciążając operację zamiatania dodatkowym kosztem $\frac{s_i}{B}\log_{M/B} s_i$ pokryjemy koszt wypełnienia tych buforów, co wynika z tego, że $|A_i| = |B_i| = k_i^3 = \Theta(s_i)$.

Podsumowując, ostatnie s_i wstawionych elementów poprzedzających zamiatanie ogniwa i obciążamy kosztem $O(\frac{1}{B}\log_{M/B} s_i)$. Mając daną ciąg operacji na początkowo pustej kolejce, niech i_{max} będzie największym i, dla którego wykonywane jest zamiatanie w ogniwie i. Zauważmy, że $s_{i_{max}} \leqslant N$, gdzie N to liczba operacji *insert* w rozważanym ciągu. Wstawienie elementu może być obciążone kosztem jedynie raz dla każdego ogniwa $i \leqslant i_{max}$. Jako że s_i rośnie podwójnie wykładniczo, koszt wstawienia elementu to:

$$O\left(\sum_{k=0}^{\infty} \frac{1}{B}\log_{M/B} N^{(3/4)^k}\right) = O\left(\frac{1}{B}\log_{M/B} N\right).$$

3.6 Profile Adaptive Performance

Aby uzależnić złożoność operacji od ilości elementów N_l znajdujących się w danym momencie w kolejce, wprowadzamy nowe pole r_i, które przechowuje ilość elementów znajdujących się w dolnej części ogniwa i. Wartość r_i przechowywana jest w mergerze v_i i podlega zmianie jedynie przy usuwaniu elementu z B_i oraz operacji zamiatania.

Algorytm *insert* musi zostać zmodyfikowany, aby wykorzystać r_i do zmniejszenia złożoności. Podczas zamiatania zmieniamy kryterium wyszukiwania ogniwa – szukamy najmniejszego i dla którego zachodzi $c_i \leqslant k_i$ **lub** $r_i \leqslant k_i s_i/2$. Jeśli zajdzie pierwszy warunek, procedura wygląda dokładnie tak jak opisano w 3.3.

Jeśli zachodzi $c_i = k_i + 1$ oraz $r_i \leqslant k_i s_i/2$, chcemy wykorzystać ponownie jeden z buforów $S_{i,j}$. Gdy jeden z buforów jest pusty, wykonujemy na nim procedurę zamiatania. W przeciwnym wypadku, dwa najmniej zapełnione bufory S_{i,j_1}, S_{i,j_2} zawierają maksymalnie s_i elementów. Bez utraty ogólności, zakładamy, że $\min S_{i,j_1} \geqslant \min S_{i,j_2}$. Dołączamy zawartość bufora S_{i,j_1} do S_{i,j_2}, zaczynając od największych elementów, dzięki czemu wystarczy jeden odczyt i zapis dla każdego elementu. Na koniec wykonujemy operację zamiatania na buforze S_{i,j_1}.

3.6.1 Analiza

Dodatkowy koszt ponoszony przez zmodyfikowany algorytm zamiatania to $O(k_i + s_i/B)$ operacji I/O, który jest znacznie mniejszy od kosztu całej operacji zamiatania.

Chcemy udowodnić, że przy zamiataniu zostanie zebrane $\Omega(s_i)$ elementów z ogniw $1, \ldots, i-1$, które zostały wstawione od czasu ostatniego zamiatania w i. Dodatkowo, połowa tych elementów została wstawiona z $N_l = \Omega(s_i)$.

Nowy algorytm zachowuje następujący niezmiennik: dla każdego i, ogniwa $1, \ldots, i$ zawierają najwyżej $\sum_{j=1}^{i} |A_j|$ elementów, które zostały usunięte z bufora A_{i+1} przez merger v_i od odstatniej operacji zamiatania ogniwa $i+1$ lub większego. Po zamiataniu w ogniwie $i+1$ definiujemy wszystkie elementy z A_j jako usunięte z A_l dla $1 \leqslant j \leqslant l \leqslant i+1$.

Gdy element e jest przenoszony z A_{i+1} do A_i, wszystkie elementy z dolnej części ogniwa $i+1$ są na pewno większe od e. Natomiast wszystkie elementy usunięte po ostatnim zamiataniu przed e z A_{i+1} były mniejsze od e. Elementy te muszą być przechowywane w jednym z buforów $A_1, \ldots, A_i$, gdyż mogły zostać przekazane dalej przez jeden z mergerów v. Wynika z tego, że jest ich najwyżej $(\sum_{j=1}^{i-1} |A_j|) + |A_i| - 1$. Zatem niezmiennik pozostanie zachowany po przeniesieniu e do A_i.

Operacja zamiatania w ogniwie i utworzy listę składającą się przynajmniej z $s_1 + \sum_{j=1}^{i-1} k_j s_j/2 \geqslant s_i/2$ elementów. Powyższy niezmiennik zapewnia, że przynajmniej $t = s_i/2 - \sum_{j=1}^{i-1} k_j^3 = \Omega(s_i)$ elementów zostało wstawione od ostatniego zamiatania w ogniwie i. Kosztem zamiatania obciążamy więc nowe elementy. Złożoność operacji *insert* argumentujemy tak, jak w przypadku pierwotnej wersji struktury.