Mateusz Flieger, 375150

Kolejka priorytetowa

na przykładzie

kopca Fibonacciego oraz kopca Binarnego

Spis treści

Opis rozwiązywanego problemu 3

Algorytmy użyte do napisania programu wraz z pseudokodami 4

Kopiec Binarny 4

Wstęp 4

Minimum 4

Extract-minimum 4

Decrease-key 5

Insert 5

Union 5

Delete 5

Kopiec Fibonacciego 6

Wstęp 6

Make-heap 7

Insert 7

Minimum 7

Extract-min 7

Union 9

Decrease-key 9

Delete 10

Implementacja 10

Testy wydajnościowe 10

Specyfikacja techniczna 10

Środowisko programistyczne 10

Parametry techniczne sprzętu 11

Wyniki 11

# Opis rozwiązywanego problemu

Przedmiotem niniejszej dokumentacji jest program implementujący kolejkę priorytetową. Kolejka priorytetową to abstrakcyjny typ danych służący do przechowywania elementów zbioru, na którym określono relację porządku[[1]](#footnote-1). Ma ona zastosowanie tam, gdzie obiekty są przetwarzane w kolejności od obiektu o najwyższym priorytecie do obiektu o najniższym priorytecie. My skupimy się na kolejce typu min – najmniejszy klucz ma największy priorytet.   
Na kolejkach priorytetowych można wykonać następujące metody:

1. **make-heap** – stworzenie kolejki(kopca)
2. **insert** – wstawienie elementu o danym kluczu do kolejki
3. **minimum** – podanie elementu o najmniejszym kluczu w danej kolejce
4. **extract-min** – usunięcie elementu o najmniejszym kluczu w danej kolejce oraz wypisanie go użytkownikowi
5. **union** – połączenie dwóch kolejek priorytetowych
6. **decrease-key** – zmniejszenie klucza danego elementu kolejki do podanej wartości
7. **delete** – usunięcie wybranego elementu ze wskazanej kolejki

Istnieje kilka możliwych podejść do implementacji tej struktury. Między innymi są to: kopce binarne, kopce Fibonacciego, tablice, listy. Zajmiemy się pierwszymi dwiema. Opiszemy algorytmy wykorzystywane na tych strukturach , implementacje, różnice w złożonościach obliczeniowych a na koniec testy na różnego rodzaju danych.

Poniżej zamieszczam dodatkowe informacje o rozwiązaniach przyjętych przy implementacji opisywanego problemu:

1. Po włączeniu programu wybieramy jeden rodzaj kopca i na nim działamy aż do ponownego uruchomienia programu.
2. Do kopca dodajemy liczby naturalne większe od zera (liczby ujemne są zarezerwowane błędom które możemy napotkać wprowadzając niepoprawne dane)
3. Przy operacji złączania kolejek usuwamy te z których go łączyliśmy.
4. Przy operacji zmniejszania klucza:
   1. Dla kopców binarnych podajemy najpierw miejsce w tablicy (liczymy od 1) , które zmieniamy, a następnie wartość na która zmieniamy.
   2. Dla kopców Fibonacciego podajemy najpierw wartość klucza który zmieniamy (jest on wyszukiwany w głąb w każdym poddrzewie – wybieramy pierwszego który ma wartość podaną przez użytkownika), a następnie wartość na która zmieniamy.
5. Przy operacji usuwania klucza:
   1. Dla kopców binarnych podajemy miejsce w tablicy (liczymy od 1)
   2. Natomiast dla kopców Fibonacciego podajemy wartość klucza (sposób pobierania wartości – patrz podpunkt 4.b)
6. Wypisywanie kopca Fibonacciego odbywa się w następującym porządku:
   1. Pierwszy wiersz to lista korzeni
   2. Każdy kolejny wierszy to synowie na danym poziomie, są oni oddzielani znakiem | oraz zawsze jest napisane kto jest ojcem danego syna, jeśli syn jest oznaczony będzie przy nim literka ‘m’

# Algorytmy użyte do napisania programu wraz z pseudokodami

## Kopiec Binarny

### Wstęp

Zakładam, że czytelnik jest zaznajomiony z kopcami binarnymi, opisuję tylko ogólną ideę.

Kopiec binarny to tablicowa struktura danych, która możemy rozpatrywać jak drzewo binarne. Każdy węzeł odpowiada pewnemu elementowi tablicy. Struktura ta musi zachowywać tzw. własność kopca tj. dla każdego węzła A[Parent(i)] ≤ A[i]. Podstawowymi operacjami są: Parent, Left, Right, Min-Heapify, Build-Min-Heap.

### Minimum

Podaje element o aktualnie najmniejszym kluczu. Jej złożoność obliczeniowa jest O(1). Wygląda ona następująco:

**Heap-Minimum**(A)  
1. return A[1]

### Extract-minimum

Jest to procedura usuwająca z kopca oraz zwracająca element o aktualnie najmniejszym kluczu. Jej złożoność obliczeniowa to: Θ (lg n). Wynika to z operacji Min-Heapify. Ogólną idea jest usunięcie z kopca (przeniesienie poza heapsize) minimalnej wartości, zmniejszenie wielkości kopca o 1 oraz naprawienie jego własności. Wygląda ona następująco:

**Heap-Extract-Minimum**(A):  
1. if A.heap-size <1  
2. error “kopiec pusty”  
3. min = A[1]  
4. A[1] = A[A.heap-size]  
5. A.heapsize = A.heapsize -1   
6. Min-Heapify(A, 1)  
7. Return min

Jak widać algorytm sprawdza przypadki krańcowe (brak elementów w kopcu).

### Decrease-key

Procedura ta sprawdza, czy dany klucz jest mniejszy niż aktualny. Jeśli tak wstawia go w miejsce gdzie był element i. Następuje wtedy naprawa kopca “w górę” tzn. sprawdzamy czy rodzic ma większą wartość niż jego syn, jeśli tak wymieniamy węzły. Złożoność obliczeniowa tej procedury to Θ (lg n).   
Tę procedurę opisuje pseudokod:

**Heap-Decrease-Key**(A, i, key)  
1. if key < A[i]  
2. error “Nowy klucz jest większy niż aktualny”  
3. A[i] = key  
4. while i > 1 i A[Parent(i)] > A[i]  
5. zamień A[i] z A[Parent(i)]  
6. i = Parent(i)

### Insert

Wstawienie klucza polega na powiększeniu rozmiaru kopca o 1, wstawienie w nowe miejsce elementu o wartości ∞, a następnie zmniejszeniu elementu na tym kluczu do wartości podanej przez użytkownika. Złożoność jest równa Θ (lg n) (wynika z decrease-key). Wygląda to następująco:

**Min-Heap-Insert**(A, key)  
1. A.heap-size = A.heap-size + 1  
2. A[A.heap-size] = ∞  
3. Heap-Decrease-key(A, A.heap-size, key)

### Union

Złączenie kopców polega na połączeniu tablic a następnie wykonaniu na nich operacji zbudowania kopca binarnego typu min. Złożoność obliczeniowa tej operacji jest równa Θ (n) – wynika z operacji budowania kopca.

**Min-heap-union**(A, B)  
1. C = make-heap()  
2. Złacz tablicę A I B oraz przypisz do C.  
3. Build-heap(C)  
4. return C

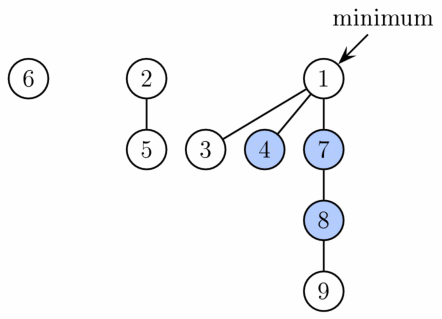
### Delete

Usuwanie klucza jest operacją bardzo podobną do usuwania minimum. Najpierw sprawdzamy czy dany kopiec zawiera ten element, następnie wymieniamy ten element z ostatnim I wykonujemy   
Min-heapify aby naprawić własność kopca. Złożoność to Θ (lg n) – wynika z Min-heapify.

**Min-heap-delete**(A, i)  
1. if i < A.heap\_size  
2. error „Kopiec nie ma tego elementu”  
3. A[i] = A[A.heap-size]  
4. A.heap-size = A.heap-size -1  
5. Min-heapify(A,i)

## Kopiec Fibonacciego

### Wstęp

Kopiec Fibonacciego jest zbiorem drzew ukorzenionych, z których każde ma własność kopca typu min (Parent(i) < i). Każdy węzeł zawiera wskaźnik do ojca oraz do któregokolwiek ze swoich synów. Synowie węzła są powiązani ze sobą cykliczną listą dwukierunkową (lista synów). Każdy syn ma wskaźnik na lewego jak i prawego brata (x.left, x.right). Kolejność synów jest dowolna. Przykładowy kopiec Fibonacciego wygląda tak[[2]](#footnote-2):  


Gdzie 6, 2 oraz 1 są korzeniami. 2 ma jednego syna jest nim 5, natomiast 1 ma 3 synów: 3, 4, 7.  
Kopiec Fibonacciego przechowuje takie dane jak: liczba węzłów w parametrze n, oraz wskaźnik na minimalny węzeł.

Ta struktura jest o wiele bardziej skomplikowana w stosunku do zwykłych kopców. Motywacją implementacji i testowania jej są złożoności obliczeniowe dla podstawowych operacji na kolejkach priorytetowych. Okazuję się, że kopce Fibonacciego wypadaj znacznie lepiej, co prezentuje tabela:

|  |  |  |
| --- | --- | --- |
| Procedura | Kopiec Binarny  (koszt pesymistyczny) | Kopiec Fibonacciego  (koszt zamortyzowany) |
| Make-Heap | Θ(1) | Θ(1) |
| Insert | Θ(lg n) | Θ(1) |
| Minimum | Θ(1) | Θ(1) |
| Extract-Min | Θ(lg n) | O(lg n) |
| Union | Θ(n) | Θ(1) |
| Decrease-Key | Θ(lg n) | Θ(1) |
| Delete | Θ(lg n) | O(lg n) |

Stąd powinniśmy spodziewać się, że kopce Fibonacciego będą działać znacznie szybciej niż kopce binarne dla określonych operacji.

Opiszę teraz wszystkie procedury na kopcu Fibonacciego.

### Make-heap

Polega na stworzeniu kopca binarnego. Rezerwujemy dla niego pamięć, ustawiamy wskaźnik elementu minimalne na NIL a liczbę wierzchołków ustawiamy na 0. Jak widać złożoność to Θ(1).

**MakeHeap(**H)  
1. Zarezerwuj pamięć na H  
2. H.min = Nil  
3. H.n = 0

### Insert

Na początku ustawiamy nowemu węzłowi x wartości degree – liczba synów = 0, wskaźnik na syna child NIL, oraz oznaczenie wierzchołka mark na false. Jeśli jest to pierwszy węzeł – tworzymy listę korzeni zawierających drzewa i ustawiamy ten element jako minimalny w przeciwnym przypadku wstawiamy po lewej stronie od minimum na listę korzeni oraz jeśli ten element jest mniejszy niż aktualny zmieniamy wskaźnik kopca na nowe minimum. Złożoność to Θ(1) ponieważ tworzenie listy korzeni jak i wstawianie nowego elementu jest Θ(1).

**Insert**(H, x)  
1. x.degree = 0  
2. x.child = NIL  
3. x.p = NIL  
4. x.mark = FALSE  
5. if H.min == NIL  
6. utwórz listę korzeni dla H zawierającą tylko węzeł x  
7. H.min = x  
8. else  
9. wstaw x na listę korzeni H  
10. if x.key < H.min.key  
11. H.min = x  
12. H.n = H.n + 1

### Minimum

Dostanie się do minimum odbywa się w bardzo prosty sposób – poprzez wskaźnik na ten element zapamiętany w kopcu. Złożoność to Θ(1).

**Minimum**(H)  
1. return H.min

### Extract-min

Operacja ta jest najbardziej skomplikowanym działaniem na kopcu. Na samym początku zapamiętujemy wskaźnik do minimalnego elementu. Następnie przenosimy wszystkich synów tego węzła na listę korzeni. Wybieramy węzeł na prawo od starego minimalnego jako nowy element minimalny. Jeśli nie jest jedynym elementem na liście korzeni dokonujemy procedury konsolidacji. Do tej pory mieliśmy listę korzeni o takim samym degree. Aby usprawnić działanie kopca łączymy za każdym razem dwa węzły o tym samym degree i w ten sposób powstaje nowa lista korzeni. Konsolidacja korzysta z funkcji link – która usuwa jeden węzeł z listy korzeni, a następnie czyni go synem innego węzła, musi więc również zwiększyć mu degree o 1.   
Złożoność całej procedury jest O(lg n). [[3]](#footnote-3)

**Extract-Min**(H)  
1. z = H.min  
2. if z ≠ NIL  
3. for każdy syn x węzła z  
4. dodaj x do listy korzeni kopca H  
5. x.p = NIL  
6. usuń z z listy korzeni kopca H  
7. if z == z.right  
8. H.min = NIL  
9. else H.min = z.right  
10. CONSOLIDATE(H)  
11. H.n = H.n – 1  
12. return z

**Consolidate**(H)  
1. niech A[0… D(H.n)] będzie nową tablicą  
2. for i = 0 to D(H.n)  
3. A[i] = NIL  
4. for każdy węzeł w na liście korzeni H  
5. x = w  
6. d = x.degree  
7. while A[d] ≠ NIL  
8. y = A[d]  
9. if x.key > y.key  
10. zamień x z y  
11. Link(H, y, x)  
12. A[d] = NIL  
13. d = d +1  
14. A[d] = x  
15. H.min = NIL  
16. for i = 0 to D(H.n)  
17. if A[i] ≠ NIL  
18. utwórz listę korzeni kopca H zawierającego tylko A[i]  
19. H.min = A[i]  
20. else wstaw A[i] na listę korzeni kopca H  
21. if A[i].key < H.min.key  
22. H.min = A[i]

**Link**(H, y, x)  
1. usuń y z listy korzeni kopca H  
2. uczyń y synem x i zwiększ x.degree o 1  
3. y.mark = FALSE

### Union

Łączenie kopców odbywa się poprzez stworzenie nowego kopca, przypisania listy korzeni jednego do drugiego oraz ustawieniem nowego mina. Złożoność to Θ(1).

**Union**(H1, H2)  
1. H = Make-Heap()  
2. H.min = H1.min  
3. doklej listę korzeni kopca H2 do listy korzeni kopca H  
4. if (H1 == NIL) lub (H2.min ≠ NIL I H2.min.key < H1.min.key)  
5. H.min = H2.min  
6. H.n = H1.n + H2.n  
7. return H

### Decrease-key

Przy zmienianiu wartości klucza musimy zadbać o własność kopca typu min. Stąd wykonujemy następujące kroki: sprawdzamy czy nowy klucz nie jest za duży, jeśli tak informujemy o błędzie. W przeciwnym przypadku dokonujemy kaskadowego odcięcia. Tam właśnie korzystamy z oznaczania węzłów. Dany węzeł jest oznaczany gdy traci swojego syna. Procedura kaskadowego odcięcia polega na „wędrowaniu w górę” drzewa oraz odcinaniu mu wierzchołków do czasu aż nie znajdzie jakiegoś który jest nieoznaczony. Następnie uaktualniamy wartość minimalną jeśli nowa wartość klucza jest mniejsza od poprzedniego minimum. Złożoność całej tej procedury to O(lg n). [[4]](#footnote-4)

**Decrease-Key**(H, x, k)  
1. if k > x.key  
2. error “nowa wartość jest większa niż bieżąca”  
3. x.key = k  
4. y = x.p  
5. if y ≠ NIL i x.key < y.key  
6. Cut(H, x. y)  
7. Cascading-cut(H, y)  
8. if x.key < H.min.key  
9. H.min = x

**Cut**(H, x, y)  
1. usuń x z listy synówy I zmniejsz y.degree o 1  
2. dodaj x do listy korzeni kopca H  
3. x.p = NIL  
4. x.mark = False

**Cascading-cut**(H, y)  
1. z = y.p   
2. if z ≠ NIL  
3. if y.mark == FALSE  
4. y.mark = TRUE  
5. else  
6. Cut(H, y, z)  
7. Cascading-Cut(H, z)

### Delete

Jest to operacja usuwania klucza. Zmniejszamy podany węzeł do najmniejszej możliwej wartości oraz usuwamy minimum za pomocą Extract-Min. Złożoność tej operacji to O(lg n) – wynika z Decrease-Key oraz Extract-Min.

**Delete**(H, x)  
1. Decrease-key(H, x, -∞)  
2. Extract-Min(H)

### Implementacja

Do zaimplementowania tej struktury wykorzystałem strukturę pomocniczą – Node która ma wskaźniki do lewego węzła, prawego węzła, ojca oraz syna. Zdefiniowałem na niej operacje dodawania do listy z lewej strony, wstawienie nowego dziecka, wyszukanie elementu o danej wartości, usunięcie wierzchołka oraz połączenie list. Sam kopiec Fibonacciego jest klasą. Można na nim wykonywać operacje dotyczące kolejki priorytetowej. Dodatkowo jest zdefiniowana funkcja łącząca dwa kopce oraz zwracająca wskaźnik na nowy kopiec.

# Testy wydajnościowe

## Specyfikacja techniczna

### Środowisko programistyczne

Aplikacja została napisana w języku C++. Składa się ona z plików nagłówkowych:

1. **abstract.h** – zawiera klasę abstrakcyjną kolejki priorytetowej implementowanej kopcem
2. **binary.h** – zawiera deklarację klasy BinaryHeap, jej metod I funkcji działających na kopcach binarych
3. **fibonacci.h** – zawiera deklaracje klasy FibonacciHeap, jej metod I funkcji działających na kopcach Fibonacciego
4. **node.h** – plik zawierający deklaracje funkcji działających na strukturze Node za pomocą której implementuję kopiec fibonacciego

Oraz z plików źródłowych:

1. **main.cpp** – plik zawierający główną pętle interakcji z użytkownikiem.
2. **binary.cpp** – implementacja metod kopca binarnego wraz z dodatkowymi funkcjami związanymi z nim (np. union)
3. **fibonacci.cpp** – implementacja metod kopca Fibonacciego wraz z dodatkowymi funkcjami związanymi z nim (np. union)
4. **node.cpp** – implementacja operacji na liście dwukierunkowej kopca Fibonacciego.

Kod był kompilowany za pomocą g++ (MacPorts gcc48 4.8.3\_0) 4.8.3 – zbudowanego na MacPortach na komputerze z rodziny Mac.  
Kompilacja odbywa się w następujący sposób:  
g++ main.cpp binary.cpp fibonacci.cpp node.cpp -o *nazwa*

Gdzie *nazwa* jest nazwą pliku binarnego do którego ma być ten program skompilowany.

### Parametry techniczne sprzętu

1. Model komputera: MacBook Pro
2. System operacyjny: Mac OS X Version 10.9.5 (Build 13F34)
3. Procesor: 3 GHz Intel Core i7 (2 rdzenie)
4. Pamięć RAM: 8 GB 1600 MHz DDR3
5. Dysk Twardy: 256 GB SSD

## Wyniki

Wykonałem testy wydajnościowe na następujących danych: rosnących, malejących oraz losowych. Każdy test miał inna liczbę danych wejściowych: 1000, 10000, 100000 oraz 100000. Testowanymi operacjami były: Insert oraz Insert + Extract-Min. Wyniki dostępne są w pliku testy.xlsx dostępnym również w tym katalogu. Nie sprawdzałem operacji delete,ponieważ wymaga ona najpierw znalezienia wierzchołka który chcemy zmienić co zdecydowanie zwiększa prawdziwy koszt wykonania tej operacji. Ze wszystkich wyników które otrzymałem obie implementacje mają zbliżony czas wykonania, z korzyścią dla kopca binarnego. Należy pamiętać, że koszty które na początku wypisywałem są kosztami zamortyzowanymi.

Widać tutaj, że kopiec Fibonacciego jest bardzo dobrą strukturą w teorii jednak w praktyce lepiej jej nie stosować. Dopiero przy operacjach na ogromnych danych (jak się okazuje jeszcze większych niż testowany milion) implementacja tego kopca ma sens w praktyce.

Bibliografia

1. Kolejka Priorytetowa, pl.wikipedia.org, dostęp z dnia 9.12.2014, [link](http://pl.wikipedia.org/wiki/Kolejka_priorytetowa)
2. Fibnacci Heap, en.wikipedia.org, dostęp z dnia 9.12.2014, [link](http://en.wikipedia.org/wiki/Fibonacci_heap#mediaviewer/File:Fibonacci_heap.png ()
3. T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, „Wprowadzenie do algorytmów”, Wydawnictwo Naukowego PWN 2013, Warszawa, s. 151-167, 513 - 539

1. Definicja zaczerpnięta z [Wikipedii](http://pl.wikipedia.org/wiki/Kolejka_priorytetowa) [↑](#footnote-ref-1)
2. Źródło [Wikipedia.en](http://en.wikipedia.org/wiki/Fibonacci_heap#mediaviewer/File:Fibonacci_heap.png) [↑](#footnote-ref-2)
3. Dokładny dowód na złożoność można znaleźć w książce numer 3. w bibliografii [↑](#footnote-ref-3)
4. Dokładny dowód na złożoność można znaleźć w książce numer 3. w bibliografii [↑](#footnote-ref-4)