Politechnika Wrocławska

Projektowanie i analiza algorytmów

Projekt 1

Freddy-Ms

8 maja 2024

1 Wstęp

Rozważając problem przechowywania danych o różnych priorytetach, gdzie szybki dostęp do elementów o określonym priorytecie jest istotny, stosuje się strukturę danych znana jako kolejka priorytetowa. Kolejka priorytetowa umożliwia efektywne zarządzanie elementami poprzez utrzymanie ich w odpowiedniej kolejności zgodnie z nadanymi priorytetami.

W kontekście rozwiązania tego problemu, stosowanie kopca jako struktury danych wydaje się być jednym z najlepszych podejść. Kopiec jest strukturą, w której każdy rodzic ma dwójkę dzieci, z których obydwoje są młodsze od rodzica. W przypadku kopca binarnego, najwyższy priorytet znajduje się na szczycie kopca, a każde kolejne poziomy są coraz mniejsze w porównaniu ze swoimi rodzicami.

Złożoności czasowe operacji na kopcu, w kontekście kolejki priorytetowej, są następujące:

- Wstawienie elementu do kopca: O(log(n))
- Usunięcie elementu kopca: O(log(n))

W przypadku rozbiórki całego kopca (usunięcie wszystkich elementów), złożoność czasowa wynosi O(n log(n)).

2 Opis algorytmu

Algorytm działa w następujący sposób: Najpierw użytkownik dostarcza liczbę elementów, co prowadzi do utworzenia tablicy o określonym rozmiarze. Następnie, dla wszystkich elementów, tworzony jest kopiec, w którym największa wartość jest na szczycie kopca. Kolejno, kopiec jest "rozbierany"w sposób, w którym szczyt kopca zamieniany jest miejscami z ostatnim elementem, co powoduje, że element o najniższym priorytecie znajduje się na ostatniej pozycji w tabeli. Należy zaznaczyć, że priorytet 1 jest najwyższym priorytetem i powinien znajdować się na pierwszym miejscu w posortowanej już tablicy.

2.1 Make Heap

Operacja ta polega na iteracyjnym porównywaniu wartości priorytetu każdego elementu z jego rodzicem poprzez przesuwanie go w górę drzewa binarnego, reprezentującego kopiec. Jeśli wartość elementu jest większa niż wartość rodzica, elementy zostają zamienione miejscami, a proces ten kontynuuje się rekurencyjnie w kierunku korzenia kopca. Proces ten kończy się, gdy wartość rodzica jest większa lub równa wartości dziecka, lub gdy proces osiąga korzeń kopca.

```
1     void PriorityQueue::make_heap(){
2     int child,parent;
3     for(int i = 1;i<actualsize;i++)
4     {
5         child = i;
6         parent = (child - 1) / 2;
7         while((child > 0) && (tab[parent].priority < tab[child].priority
              ))
8         {
9             swap(tab[child],tab[parent]);
10            child = parent; parent = (child - 1) / 2;
11        }
12    }
13 };
```

2.2 Heap Sort

Operacja zdejmowania w kopcu binarnym polega na przestawieniu korzenia kopca z ostatnim elementem i następnie rekurencyjnie naprawieniu właściwości kopca, aby przywrócić porządek kopcowy. Priorytet każdego węzła w kopcu jest określany przez jego wartość, a w kopcu maksymalnym największe wartości mają najniższy priorytet.

Operacja zdejmowania jest realizowana poprzez porównywanie priorytetów węzła z priorytetami jego dzieci. Jeśli priorytet któregoś dziecka jest większy niż priorytet rodzica, następuje zamiana miejscami tych elementów. Proces ten jest kontynuowany rekurencyjnie dla przesuniętego dziecka, aby zachować właściwość kopca.

```
1     void PriorityQueue::heap_sort(){
2     int parent, lchild, rchild, to_swap;
3     for(int i = actualsize-1;i > 0;i--){
4             swap(tab[0],tab[i]);
5             parent = 0;
6             lchild = 1;
7             rchild = 2;
8             to_swap = parent;
9             while(1){
10                if(lchild < i && tab[lchild].priority > tab[to_swap].
                      priority){
11                    to_swap = lchild;
12                }
13                if(rchild < i && tab[rchild].priority > tab[to_swap].
                      priority){
14                    to_swap = rchild;
15                }
16                if(parent != to_swap){
17                    swap(tab[parent],tab[to_swap]);
18                    parent = to_swap;
19                    lchild = parent * 2 + 1;
20                    rchild = parent * 2 + 2;
21                } else {break;}
22            }
23    }
24 }
```

2.3 Funkcja generująca wiadomość z losowymi wartościami priorytetu

Pamiętajmy że priorytet najmniejszy powinien być pierwszy w "tabeli"po przesortowaniu, czyli powinniśmy uzyskać posortowaną tabele, aby jej elementy był rosnące.

```
1     void PriorityQueue::generate_mssg(int n){
2     {
3     for(int i = 0; i < n; i++){
4         tab[actualsize].data = rand() % 10;
5         tab[actualsize].priority = rand() % 10000000 + 1;
6         actualsize++;
7     };
```

2.4 Funkcja generująca wiadomość posortowaną

```
1     void PriorityQueue::generate_mssg_sorted(int n){
2     int j = n;
3     for(int i = 0; i < n; i++)
4     {
5         tab[actualsize].data = rand() % 10;
6         tab[actualsize].priority = j;
7         actualsize++;
8         j--;
9     }
10    }
```

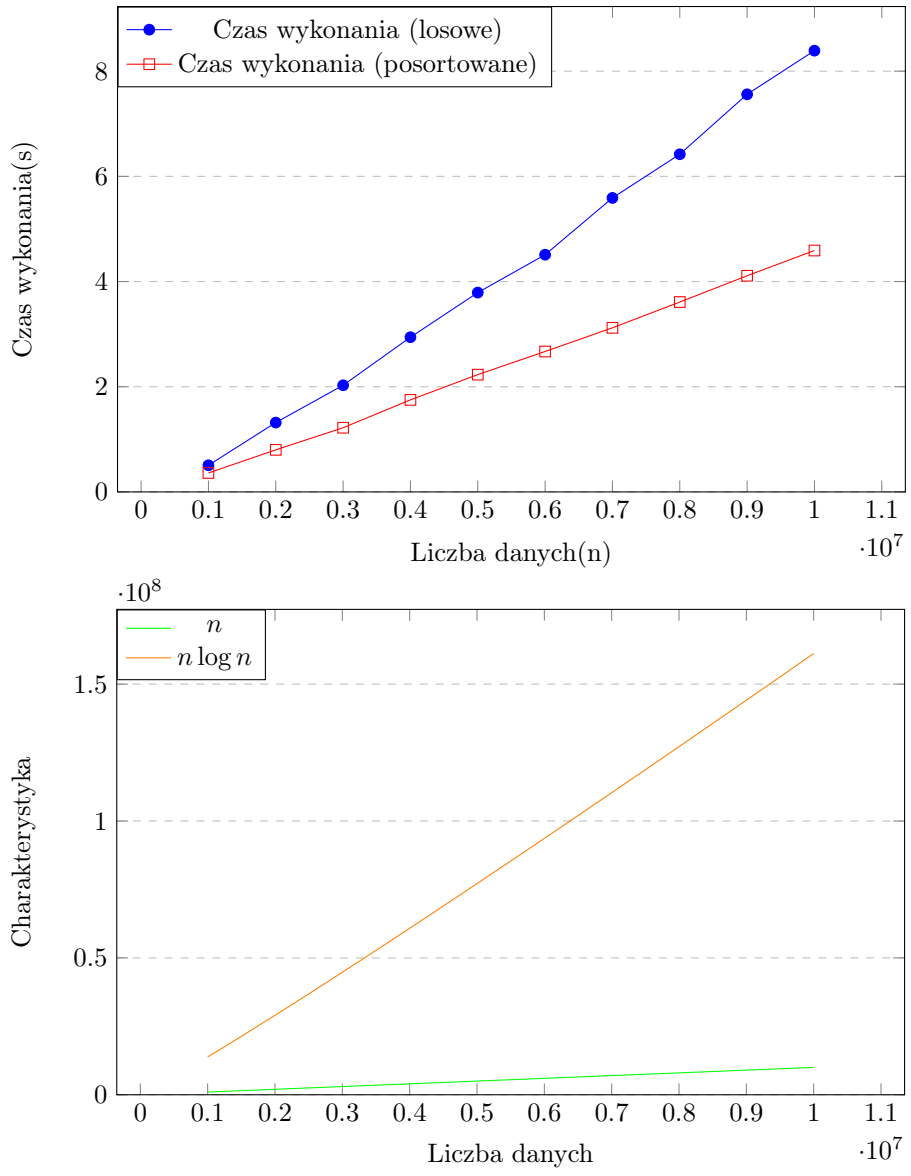
3 Badania

Plan eksperymentu zakładał użycie powyższych funkcji w celu wygenerowania pakietów odpowiednio: losowych oraz posortowanych. Następnie zostały wykonane pomiary czasu tworzenia oraz sortowania pakietów z użyciem kolejki priorytetowej.

Otrzymane wyniki zaokrąglone do dwóch miejsc po przecinku:

Tabela 1: Czasy wykonania dla danych losowych i posortowanych

Liczba danych	**Czas wykonania (losowe)s**	**Czas wykonania (posortowane)s**
1000000	0,50	0,36
2000000	1,31	0,8
3000000	2,02	1,22
4000000	2,9	1,75
5000000	3,79	2,23
6000000	4,51	2,67
7000000	5,59	3,12
8000000	6,42	3,61
9000000	7,56	4,11
10000000	8,39	4,59



Rysunek 1: Czasy wykonania dla danych losowych i posortowanych oraz charakterystyka n i $n \log n$

4 Wnioski

Analiza wyników eksperymentalnych wskazuje na skuteczność oraz efektywność proponowanego rozwiązania opartego na użyciu kolejki priorytetowej zarówno dla danych losowych, jak i dla danych już posortowanych. Obliczenia potwierdzają zgodność z teoretycznymi założeniami co do złożoności obliczeniowej, co dodatkowo umacnia przekonanie o poprawności implementacji. Dla zbiorów danych posortowanych o rozmiarze $n = 5 * 10^6$ osiągnięto czas wykonania równy 2.23 sekundy. Zwiększenie ilości danych dwukrotnie (do $n = 10 * 10^6$) skutkowało przewidywanym dwukrotnym wydłużeniem czasu wykonania, co potwierdza wyniki eksperymentu, gdzie uzyskano czas 4.59 sekundy, czyli dwukrotnie dłuższy niż dla mniejszej ilości danych.

5 Bibliografia

- Cormen T.; Leiserson C.E.; Rivest R.L.; Stein C.: Wprowadzenie do algorytmów, WNT
- Pat Morin, Open Data Structures.