Sprawozdanie Projekt 1 - ADT		
Rafał Jasiński 259384	02.04.2022r	Dr hab. inż. Andrzej Rusiecki
Zadania na ocenę bdb (5.0)		
Link do repozytorium: `https://gitlab.com/JasinskiR259384/pamsi-2022\`		

1 Wstęp

Celem zadania było zaprojektowanie i zaimplementowanie odpowiedniej abstrakcyjnej struktury danych (ADT) do przechowywania oraz sortowania danych składających się z pary: liczby porządkowej oraz wiadomości (typ nie został określony). Przeanalizować czas działania - złożoność obliczeniową proponowanego rozwiązania.

2 Podejście do problemu

Ze względu na fakt, że dane, które otrzymujemy nie są posortowane, pojawia się problem jaką strukturę należy wybrać, aby łatwo się móc przemieszczać pomiędzy jej elementami w celu posortowania/wypisania posortowanej wiadomości.
Na typ etapie ograniczyłem znane mi abstrakcyjne struktury danych do:

- BST - binary search tree - binarne drzewo poszukiwań
- hash table (hash map)
- heap - kopiec

Postanowiłem rozważać moje rozwiązanie problemu na nich ze względu na wstępną możliwość posortowania dodawanych danych.
Na wstępie założyłem, aby mój algorytm miał jak najmniejszą złożoność obliczeniową.

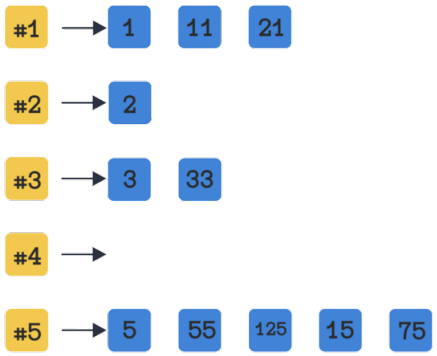
2.1 Hash map

Wykorzystanie hash map'y było moim pierwszym pomysłem na podejście do danego problemu. Jednak napisanie własnej klasy działającej na zasadzie **std::map** jest dość nieefektywne. Pojawia się tutaj problem hashowania naszego klucza, który jest liczbą porządkową. Najprostszym sposobem było by użycie

```
struct MyKeyHash {
    unsigned long (const uint32_t &key) const {
        return key % 10;
    }
};
```

Listing 1: Hash function

Jednakże pojawia się tutaj problem z przyszłym poszukiwaniem naszego konkretnego elementu. Musielibyśmy przejść po tablicach z odpowiednio zahashowanym kluczem a następnie dostać się do naszego elementu. Znalezienie konkretnego elementu w tym momencie możemy wyznaczyć jako ilość kluczy hashujących (przeszukanie $\mathbf{O}(n)$) - **n** w tym przypadku jest zależne od funkcji **mod**) a następnie w danym kluczu przeszukanie **n** elementów. Co ostatecznie daje nam $\mathbf{O}(n^2)$.
Problem ten występuje w przypadku kiedy naszym kluczem hashującym jest liczba. Można spróbować zmniejszyć złożoność tego algorytmu stosując $\sqrt{key}$.
Dlatego na tym etapie porzuciłem pomysł hash map'y.



Rysunek 1: Reprezentacja przechowywania elementów w hash map'ie (jako tablica dwu wymiarowa)

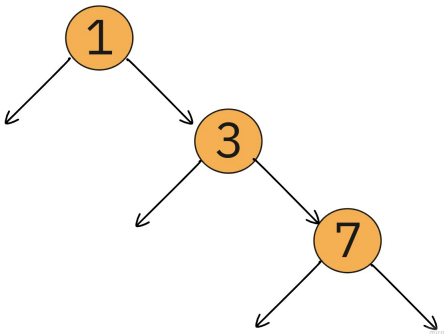
2.2 BST - binary search tree - binarne drzewo poszukiwań

W przypadku tej struktury udało mi się napisać już działający algorytm, który jako wejście przyjmował parę (liczba porządkowa, wiadomość)

```
1 std::pair<uint32_t, message>
```

Listing 2: Para danych

A na wyjściu przedstawiał nam już uporządkowaną wiadomość. Złożoność umieszczenia elementu na drzewie w przypadku tego algorytmu wynosi $\mathbf{O}(\log(n))$. Jest to jednak wartość oczekiwana i nie gwarantuje, że zawsze będzie tyle wynosić. W najgorszym przypadku możemy otrzymać złożoność $\mathbf{O}(n)$ - dzieje się tak w przypadku kiedy otrzymujemy na wejście już posortowane dane - w dowolną stronę. Obiekty wtedy są umieszczane na jednej gałęzi.

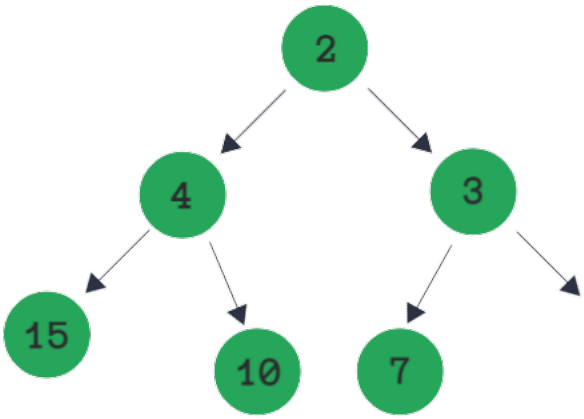


Rysunek 2: Najgorszy możliwy przypadek - otrzymujemy na wejście posortowane dane

Ze względu na te przypadki postanowiłem poszukać jeszcze innego rozwiązania - kopiec.

2.3 Heap - kopiec oparty na tablicy

Elementy w przypadku tej struktury są umieszczane od lewej do prawej, schodząc kolejno w dół do kolejnych "pokoleń". Gwarantuje nam to że rozmiar naszego kopca zawsze będzie $\mathbf{O}(\log(n))$. Złożoność umieszczenia elementu na kopiec również wynosi $\mathbf{O}(\log(n))$. Ze względu na wypełnianie od lewej do prawej nie mamy przypadków, w których zapełniana jest tylko jedna gałąź, co niweluje nam przypadki możliwe do wystąpienia w przypadku drzewa BST.



Rysunek 3: Na kopcu nie obowiązuje też zależność lewego - prawego dziecka. Przykład posortowanego kopca dla podanych danych

2.4 Wybrana struktura danych

Podsumowując zalety i wady rozważanych przeze mnie abstrakcyjnych struktur danych postanowiłem rozwiązać problem przedstawiony w zadaniu wykorzystując kopiec oparty na tablicy.

3 Rozwiązanie

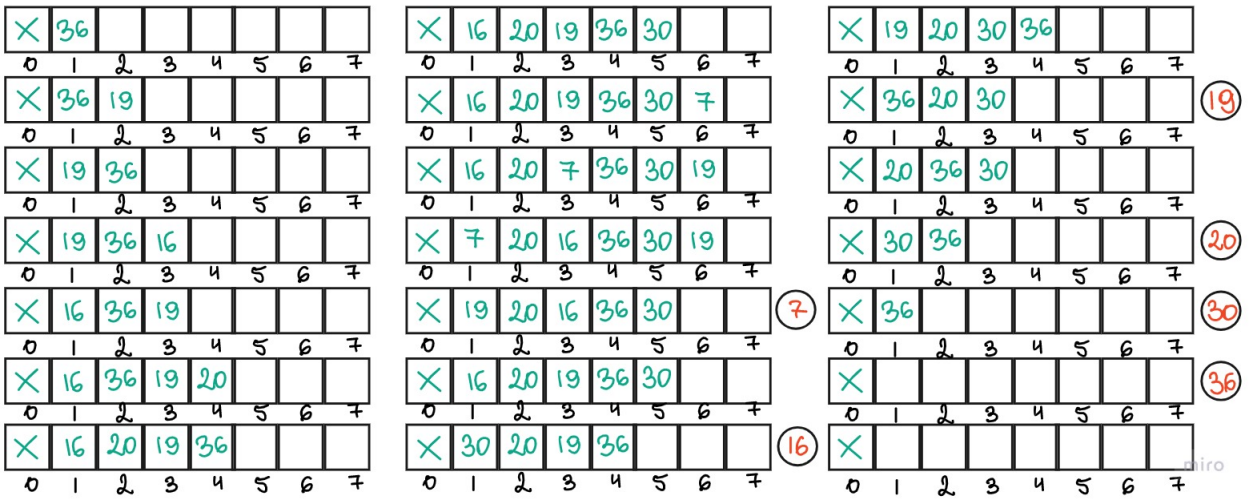
Aby móc w jakikolwiek sposób testować algorytm posłużyłem się dodatkową klasą *Packets*.
Działa ona w dwóch trybach:

1. generuje parę liczb (liczba porządkowa, liczba - która jest równa $n-lp$ [gdzie n ilość liczb, lp. liczba porządkowa]), a następnie wykonywana jest operacja pomieszania danych.

2. wczytuje dane z pliku, który musi zostać przygotowany w odpowiedni sposób tj. liczba porządkowa wiadomości, dowolny separator będący znakiem białym oraz sama wiadomość do znaku enter "\0"

3.1 Zasada działania

Dane te są następnie wczytywane przez funkcję klasy *Heap*, która czyta każdą parę z osobna i wykonuje operację dodawania elementu do struktury. Zazwyczaj w kopcu elementy największe będą znajdować się na samej górze. Jednak w moim przypadku postanowiłem zrobić odwrotny kopiec, a więc to wartość najmniejsza będzie na szczycie. Sortowanie kopca polega na tak zwanym kopcowaniu - heapifying, czyli zamianie elementu aktualnie dodawanego z jego rodzicem do momentu, aż nie natkniemy się na wartość mniejszą (działa na zasadzie rekurencji). Kiedy zakończy się proces dodawania elementów na kopiec, odczytanie posortowanych wartości odbywa się poprzez odczytanie pierwszego elementu (który w tym przypadku będzie najmniejszy), a następnie zamiany pierwszego elementu z ostatnim. Wyzerowaniu/usunięcie ostatniego elementu. A następnie odbywa się operacja "spychania" naszego największego elementu na dół. Po tej operacji nasza nowa najmniejsza wartość znowu pojawi się na pierwszym miejscu.



Rysunek 4: Pokazanie zasady działania algorytmu kopca (dla przykładowych danych)

3.2 Przechowywanie danych

Przechowywanie danych wykorzystując dynamiczną alokacje pamięci może wydawać się trochę mało optymalne obliczeniowo. Ponieważ algorytm wykorzystuję operacje na tablicy posiada dostęp do *i'tego* elementu (w naszym przypadku będziemy wykorzystywać głównie operacje *i/2*). Gdybyśmy chcieli zadbać o optymalizację pamięciową moglibyśmy użyć listy dwukierunkowej z iteracją by móc przemieszczać się po elementach. Jednak stwarza to problem potrzeby przemieszczania się po liście, w celu dostępu do naszego elementu. Dlatego postanowiłem zastosować operację alokacji statycznej z automatyczną opcją rozszerzania tablicy (podwojenie dostępnej pamięci)

3.2.1 Początek iteracji od 1 - tab[1]

Dane, które są dodawane na kopiec są zapisywane w tablicy dopiero od pierwszego indeksu w celu łatwiejszej implementacji dostępu do rodziców oraz dzieci.

4 Analiza złożoności obliczeniowej oraz pamięciowej

```
template <typename message>
void Heap<message>::heapify(uint32_t node) {
  if (node == 1 || packetArr[node / 2] < packetArr[node]) return;      // O(1)
  swap(packetArr[node / 2], packetArr[node]);                          // O(1)
  heapify(node / 2);                                                   // O(log(n))
}

template <typename message>
std::pair<uint32_t, message> Heap<message>::minValue() {
  return packetArr[1];                                                 // O(1)
}

template <typename message>
void Heap<message>::pushDown(uint32_t node) {
  if (node * 2 > size) return;                                         // O(1)
  if (node * 2 + 1 > size) {                                           // O(1)
    if (packetArr[node * 2] < packetArr[node]) {                       // O(1)
      swap(packetArr[node], packetArr[node * 2]);                      // O(1)
      this->pushDown(node * 2);                                        // O(log(n))
    }
    return;                                                            // O(1)
  }
  if (packetArr[node * 2] < packetArr[node * 2 + 1]) {                 // O(1)
    if (packetArr[node * 2] < packetArr[node]) {                       // O(1)
      swap(packetArr[node], packetArr[node * 2]);                      // O(1)
      this->pushDown(node * 2);                                        // O(log(n))
    }
```

```
28      return;                                                          // O(1)
29    }
30    if (packetArr[node * 2 + 1] < packetArr[node]) {                   // O(1)
31      swap(packetArr[node], packetArr[node * 2 + 1]);                  // O(1)
32      this->pushDown(node * 2 + 1);                                    // O(log(n))
33    }
34  }
35
36  template <typename message>
37  void Heap<message>::pop() {
38    swap(packetArr[1], packetArr[size]);                               // O(1)
39    packetArr[size--] = std::make_pair(NULL, NULL);                    // O(1)
40    this->pushDown(1);                                                 // O(log(n))
41  }
42
43  template <typename message>
44  void Heap<message>::insertKey(std::pair<uint32_t, message> element) {
45    if (size + 1 >= sizeOfTab) {                                       // O(1)
46      sizeOfTab *= 2;                                                  // O(1)
47      resize();                                                        // O(1) - amortyzowany
48    }
49    packetArr[++size] = element;                                       // O(1)
50    heapify(size);                                                     // O(log(n))
51  }
52
53  template <typename message>
54  void Heap<message>::resize() {
55    std::pair<uint32_t, message> *tmp =
56        new std::pair<uint32_t, message>[sizeOfTab];                   // O(1)
57    for (uint32_t i = 0; i <= size; ++i) {
58      tmp[i] = packetArr[i];                                           // O(1)
59    }                                                                  // O(n)
60    delete[] packetArr;                                                // O(1)
61    packetArr = tmp;                                                   // O(1)
62  }                                                                    // O(n)
```

Listing 3: Kod z analizą złożoności

Powiększanie tablicy odbywa się poprzez podwojenie dostępnej pamięci. Początkowo funkcja ta będzie wykonywać się znacznie częściej, niż gdybyśmy ustawili stały przyrost. Jednakże wraz z rosnącą liczbą elementów ilość wykonywanych operacji powiększania maleje. Jej liczba wywołań wyniesie więc $\mathbf{O}(\log(n))$. Natomiast liczba wszystkich przepisań nie przekroczy $2n$. Stąd amortyzowana złożoność wywołania funkcji *resize()* w funkcji *insertKey(element)* będzie $\mathbf{O}(1)$. Podsumowując, złożoność całego algorytmu dla n elementów wyniesie $\mathbf{O}(n\log(n))$

5 Wnioski

1. Patrząc na przedstawione algorytmy, nie istnieje szybsza metoda "posortowania" wiadomości niż $\mathbf{O}(n\log(n))$
2. Biorąc pod uwagę koszt zamortyzowany funkcji *resize()* pozwala to na osiągnięcie końcowej złożoności *insertKey(element)* jako $\mathbf{O}(\log(n))$
3. Złożoność obliczeniowa $\mathbf{O}(\log(n))$ jest na tyle wydajna czasowo, że przy próbie sporządzenia wykresu nawet przy ilości elementów wynoszących 50 000 000, charakterystyka czasowa niewiele odbiegała od funkcji liniowej.
4. Porównując rozważane przeze mnie abstrakcyjne struktury, kopiec wydaje się mieć jak najmniej wad.