Projektowanie algorytmów i metody sztucznej inteligencji.

Projekt 1 - wersja na ocene 5.0

26.03.2023

Nazwisko i imię	Wojciech Bukała
Termin zajęć:	Poniedziałek 18:55
Kod grupy projektowej	Y01-27h
Data wykonania ćwiczenia	23.03.2023
Prowadzący kurs	Dr Marek Bazan

1 Treść polecenia.

Załóżmy, że Jan chce wysłać przez Internet wiadomość W do Anny. Z różnych powodów musi podzielić ją na n pakietów. Każdemu pakietowi nadaje kolejne numery i wysyła przez sieć. Komputer Anny po otrzymaniu przesłanych pakietów musi poskładać je w całą wiadomość, ponieważ mogą one przychodzić w losowej kolejności.
Państwa zadaniem jest zaprojektowanie i zaimplementowanie odpowiedniego rozwiązania radzą- cego sobie z tym problemem. Należy wybrać i zaimplementować zgodnie z danym dla wybranej struktury ADT oraz przeanalizować czas działania - złożoność obliczeniową proponowanego rozwiązania. W sprawozdaniu (3-5 strony + strona tytułowa) należy opisać rodzaj wybranej struktury danych wraz z uzasadnieniem wyboru oraz opisać proponowane rozwiązanie problemu. Należy także opisać sposób analizy złożoności obliczeniowej i podać jej wynik w notacji dużego O.

2 Zaproponowana struktura danych.

2.1 Kolejka priorytetowa.

Do rozwiązania postawionego problemu postanowiłem wykorzystać dynamiczną strukturę danych zwaną kolejką priorytetową (ang. priority queue). Jest to struktura, która pozwoli na porządkowanie obieranych pakietów według przypisanych im priorytetów.

- ***Node*** - Kolejka składa się z pojedynczych elementów, zdefiniwanych w moim programie za pomocą klasy "Node", co tłumaczone z angielskiego oznacza węzeł. Pojedynczy węzeł zawiera pola chronione:
  - ***element*** - pole typu zadeklarowanego przez użytkowanika przy tworzeniu instancji klasy, zawierające informację użyteczną.
  - ***key*** - klucz, czyli priorytet elementu ze względu na, który będzie on pozycjonowany w kolejce. Jest to zmienna typu całkowitego i tylko takie wartości może przyjmować klucz.
  - ***next*** - zmienna wskaźnikowa, która w trakcie działania kolejki powinna wskazywać na poprzedni element kolejki (jeśli taki istnieje).
  - ***prev*** - zmienna wskaźnikowa, wskazująca na element następny, oczywiście jeśli taki istnieje.

  Klasa Node posiada trzy konstruktory. W przypadku wywoałania bezparametrycznego wszyskie wskaźniki ustawiane są na wartość NULL, tak samo się dzieje jeśli podczas tworzenia obiektu klasy node zdecydyjemy się na zainicjowanie pól element oraz key określonymi wartościami. Istnieje również możliwość wywołania wraz z podaniem adresów next i prev.
  Klasa Node posiada również zestaw metod służących jako interface do obsługi jej pól i zmiany ich wartości.
- ***PQueue*** - drugą częścią tej ADT jest klasa realizująca funkcjonalności koljeki priorytetowej. Dziedziczy ona publicznie po klasie Node co zapewnia jej dostęp do interface'u pozwalającego na tworzenie i obsługę pojedynczych węzłów. Klasa *PQueue* zawiera parę kluczowych pól i metod:
  - ***head*** - jest to zmienna wskaźnikowa, wskazująca na Node. Jeśli kolejka nie jest pusta to wskazuje ona na pierwszy elemnt listy, ten z najwyższym priorytetem.
  - ***tail*** - w przeciwieństwie do head'a, jest to wskaźnik na ostatni elemnet koljeki. Jeśli kolejka jest pusta, podobnie jak head, wartość waskaźnika jest równa NULL.
  - ***empty()*** - kluczowa metoda, pozwalająca reszcie meotd działać w prawidłowy sposób. Zwraca wartość *true* jeśli kolejka jest pusta, w innym przypadku zwraca wartość *false*.
  - ***push()*** - metoda, pozwalająca na dodawanie elementów do kolejki, zgodnie z podanym priorytetem. Złożoność obliczeniowa algorytmu pozwalającego na sortowanie elementów ze względu na klucz zostanie opisana w dalszym punkcie.
  - ***removeMax()*** - metoda usuwająca i zwracająca element o największym priorytecie.
  - ***removeMin()*** - metoda usuwająca i zwracająca element o najniższym priorytecie.

Pomocniczo zdefiniowana została też struktura realizująca funkcjonalność stosu. Nie jest ona wykorzystywana do rozszyfrowania widomości. Została storzona z racji braku możliwości używania kontenerow STL w programie.

3 Działanie programu.

Projekt został podzielony na dwa programy. Pierwszy program nazwany *sender* realizuje wysyłanie wiadmości pobranej z pliku w pakietach o określonej przez użytkownika wielkości. Drugi program *reveicer* pobiera z buffora pakiety wysłane w losowej kolejności i odszyfrowuje wiadmość wyświtlając ją w terminalu.

- ***Sender*** - Program po uruchomieniu pyta użytkowanika o wielkość pojedynczego pakietu. Następnie oczekuje podania nazwy pliku z którego będzie pobierał pakiety. Do testowania funkcjonalności wykorzystałem plik *lista.xlsx* zawierającego losowe dane wpisane losowo w komórki pliku excelowego. Program w celu nadania widmości wykonuje następujące kroki:

  1. Otwarcie pliku z widomością do nadania.
  2. Do czasu napotkania końca pliku, pobieranie paczek o określonej objętości oraz wrzucanie ich na tymczasowy stos.
  3. Zamknięcie pliku z widomością do nadania.
  4. Stworzenie tablicy "Rand"o wielkości równej ilości pakietów do nadania.
  5. Pomieszanie tablicy w losowy sposób.
  6. Kolejne "ściąganie"elementów ze stosu oraz wpisywanie go do tablicy na miejsce określone przez losową sekwencję tablicy "Rand", równolegle z wpisywaniem priorytetu na to samo miejsce w drugiej tablicy.
  7. Utworzenie folderu *buffer*.
  8. Pierwszy plik folderu to inforamcje o transmisji, ile pakietów należy się spodziewać przy odbieraniu.
  9. Tworzenie i wpisywanie do następnych plików folderu *buffer* elementów w kolejności pomiesznej losowo wraz z ich priorytetami oddzielonych spacją.
  10. Wyświetlanie wiadomości *Sending.* oznaczającej prawidłowe nadanie widomości.

- ***Receiver*** - Program powinien być uruchamiany po wysłaniu wiadmości przez program Sender, w innym przypadku możliwy jest brak folderu *buffer* lub zapisana może w nik być inna wiadomość. Program *Receiver* wykorzystuje przygotowaną wcześniej strukturę ADT - kolejkę priorytetową. Do odszyforwania wiadomości algorytm wykonuje następujące kroki:

  1. Otwieranie następnych plików z folderu *buffer*.
  2. Pobranie informacji o transmisji z pierwszego pliku.
  3. Pobieranie danych z następnych plików znak po znaku.
  4. Rozdzielanie pobranych danych na wiadmość użyteczną i priorytet.
  5. Wpisywanie do kolejki priorytetowej następnych elementów wraz z kluczem.
  6. Zamknięcie pliku z folderu *buffer*.
  7. Powtórzenie odczytu dla następnych plików znajdujących się w folderze *buffer*
  8. Usuwanie za pomocą meotdy *RemoveMax()* elemntów o najwyższym priorytecie i wypisywanie ich w terminalu.

4 Testy.

4.1 Testy jednostkowe.

Testy jednostkowe wykonane zostały za pomocą framework'a google test. Aby, możliwe była wykorzystanie tej technologii w pliku CMakeLists.txt zadeklarowane zostało użycie gtesta oraz utworzone pliki testowe. Wykonane zostały następujące grupy testów:

- Testy konstruktorów - testowwane jest czy konstruktor prawidłowo inicjalizuje instancje klasy *PQueue*, tzn. czy kolejka początkowo jest pusta oraz czy jej rozmiar róny jest 0. Testy wykonano dla inicjalizacji obiaktu dla różnych typów elementów.
- Testy *RemoveMax()* - testy poprawnego usuwania elementów oraz zachowania się kolejki w przypadkach granicznych (gdy, siągamy wszystkie elementy lub próbujemy ściągnać ich wręcz zbyt dużo).

Wszystkie testy zakończyły się powodzeniem.

4.2 Testy wydajnosciowe.

Testy wydajnościowe wykonane zostały z wykorzystaniem biblioteki c++ *chrono*. Przetestowałem działanie kolejki przy wpisywaniu i usuwaniu 100, 1.000 i 100.000 elementów, dla przypadku najgorszego i najlepszego.

- **Przypadek najgorszy** - jest to przypdake w którym do kolejki wpisujemy zestaw elementow posortowny w przeciwny sposób, to znaczy pierwszy element ma największy priorytet, zaś ostani wpisywany element ma priorytet najniższy.
- **Przypadek najlepszy** - to przypadek kiedy wpisywany zestaw elementów jest uporządkowony zgodnie z sekwencją wpisywania do kolejki, tzn. element o najniższym priorytecie wchodzi pierwszy.

Testy wydajnościowe pokazały następujące czasy działania algorytmu:

- Dla 100 elementów:
  1. Czas wpisywania w najgorzym przypadku : 156 us
  2. Czas usuwania w najgorszym przypadku : 11 us
  3. Czas wpisywania w najlepszym przypdaku: 47 us
  4. Czas usuwania w najlepszym przypadku : 10 us
- Dla 1.000 elementów:
  1. Czas wpisywania w najgorzym przypadku : 7070 us
  2. Czas usuwania w najgorszym przypadku : 32 us
  3. Czas wpisywania w najlepszym przypdaku: 148 us
  4. Czas usuwania w najlepszym przypadku : 31 us
- Dla 100.000 elementów:
  1. Czas wpisywania w najgorzym przypadku : 12982139 us
  2. Czas usuwania w najgorszym przypadku : 2957 us
  3. Czas wpisywania w najlepszym przypdaku: 14103 us
  4. Czas usuwania w najlepszym przypadku : 2956 us

5 Złożoność obliczeniowa.

Złożoność obliczeniowa wykorzystanej w programie kolejki priorytetowej zostanie obliczona na podstawie analizy ilości operacji podstawowych wykonanych przez metody *push()* i *RemoveMax()* wykorzystywanych w programie odbierającym.

5.1 Metoda *push()*

```
template <typename T>
void PriorityQueue<T>::push(T new_elem, int new_key)
{
    Node<T>* temp = new Node<T>(new_elem,new_key);
    Node<T>* temp_pointer = NULL;

    if(empty())
    {
        head = temp;
        tail = temp;
        queue_size = 1;
    }
    else if(!empty()) //adding on the end of the queue
    {

        temp_pointer = head;
        bool inserted = 0;

        while(temp_pointer != tail->getNext())
        {
            if(new_key > temp_pointer->getKey())
            {
                // adding node after the head
                if(temp_pointer->getPrev() == NULL)
                {
                    temp->setNext(head);
                    head = temp;

                    head->setPrev(NULL);
                    temp->getNext()->setPrev(temp);
                }
                // adding node between two others nodes
                else
                {
                    temp->setPrev(temp_pointer->getPrev());
                    temp->setNext(temp_pointer);

                    temp_pointer->setPrev(temp);
                    (temp->getPrev())->setNext(temp);
                }
                inserted = 1;
                queue_size++;
                break;
            }

            temp_pointer = temp_pointer->getNext();
        }

        // adding node before tail
        if(!inserted)
        {
            temp->setPrev(tail);
            tail = temp;

            tail->setNext(NULL);
            temp->getPrev()->setNext(temp);

            queue_size++;
        }

    }
}
```

W najgorszym przypadku (czyli przypadku, gdy dane wejściowe będą odwrotnie posortowane względem priorytetów) pętla while wykona się n razy. Wszytkie pozostałem operacje w tym fragmencie kodu źródłowego to

operacje stałoczasowe. **Złożoność obliczeniowa metody push() : O(n).**

5.2 Metoda *removeMax()*

```
template <typename T>
Node<T> PriorityQueue<T>::removeMax()
{
    try
    {
        if(!empty())
        {
            Node<T> temp = *head;
            Node<T>* temp_ptr = head;

            head = head->getNext();

            if(head != NULL) head->setPrev(NULL);

            temp_ptr->setNext(NULL);
            temp_ptr->setPrev(NULL);

            queue_size--;

            return temp;
        }
        else
        {
            throw 2;
        }
    }
    catch(int error_num)
    {
        std::cout << "KOLEJKA JEST PUSTA NIE MOZNA USUNAC ELEMENTOW." << std::endl;

    }
}
```

W metodzie removeMax() występuje parę operacji stało czasowych: sprawdzenie czy kolejna nie jest pusta, utworzenie węzła tymczasowego *temp*, ustawienie wskaźnika *head* na następny element kolejki, w przypadku, gdy head nie jest już ostatnim elementem kolejki, ustawiony zostaje wskaźnik prev następnego elementu na NULL, ustawienie wskaźników *next* i *prev* na NULL. Brak w tym fragmencie operaci nie będącymi stałoczasowymi. **Złożoność obliczeniowa metody removeMAX() : O(1).**

5.3 Wykorzystanie w programie odbierającym.

W programie odbierającym *receiver*, pętla która wpisuje koljno pakiety do pliku wyjściowego wykona się maksymalnie n razy. Ma więc złożoność obliczeniową **O(n)**.

Osatecznie sumując wszystkie fragmenty algorytmu relizującego funkcjonolność koljeki priorytetowej otrzymujemy: **Złożonośc finalna: O(n^2).**