12 marca 2021
Maciej Knapik

Abstrakcyjny Typ Danych (ADT) Set
Z wykorzystaniem tablicy haszującej

URUCHOMIENIE

1. Aby skompilować program testujący należy w terminalu wpisać polecenie.
   g++ set.cpp
2. Aby uruchomić program należy wpisać polecenie według jednego z dwóch sposobów.
   a. nazwa_pliku typ_danych elementy_do_wstawienia
   ./a.out integer 1 2 3 4
   ./a.out string ala ma kota kot ma ale
   b. nazwa_pliku elementy_typu_int_do_wstawienia
   ./a.out 1 2 3 4

IMPLEMENTACJA

1. Atrybuty klasy

```
    std::list<T> *table;
    int table_size;
```

- *table* - wskaźnik do tablicy przechowującej listy związane z danym indeksem,
- table_size - rozmiar tablicy,

2. Konstruktor

```
template<typename T>
Set<T>::Set(int set_size) {
        table_size = set_size;
        table = new std::list<T>[table_size];
}
```

- przekazanie rozmiaru tablicy,,
- inicjalizacja tablicy list,

3. Funkcja haszująca - zwraca indeks do odpowiedniej komórki w tablicy

```
int Set<T>::hash_function(T element) {

    std::ostringstream temp;
    temp << element;
    std::string str_el = temp.str();

    int sum = 0;
    for(int i = 0; str_el[i] != '\0'; i++) {
        sum += str_el[i];
    }

    return sum%table_size;
}
```

- funkcja sumuje kody ASCII, a następnie wylicza modulo z rozmiaru tablicy,

4. Sprawdzanie czy zbiór zawiera dany element

```
bool Set<T>::set_is_member(T element) {

    int index = hash_function(element);
    for(T el : table[index])
        if(el == element)
            return true;

    return false;
}
```

- Funkcja wylicza indeks komórki za pomocą funkcji haszującej,
- Funkcja przechodzi przez elementy, które trafiły do listy na skutek kolizji,
- Średnia złożoność wyszukiwania jest złożonością liniowego wyszukiwania elementu na liście i zależy od współczynnika wypełnienia listy,
- w pesymistycznym przypadku należy sprawdzić całą listę tablica[hash(element)],
- w optymistycznym przypadku, przy dobrej funkcji haszującej złożoność mogłaby wynosić nawet O(1),

5. Wstawienie elementu

```
void Set<T>::set_insert(T element) {

    if(!set_is_member(element)) {
        int index = hash_function(element);
        table[index].push_back(element);
    }
}
```

- funkcja sprawdza, czy dany element już istnieje w zbiorze,
- wyliczany jest indeks komórki za pomocą funkcji haszującej,
- element jest dodawany na koniec listy elementów pod wyliczonym indeksem,
- złożoność wyniosłaby O(1), gdyby nie fakt, iż należy sprawdzić, czy element nie znajduje się już w zbiorze,

6. Usuwanie elementu

```
T Set<T>::set_remove(T element) {

    int index = hash_function(element);

    for(auto it = table[index].begin();
        it != table[index].end(); it++)
        if(*it == element) {
            table[index].erase(it);
            return element;
        }

    return element;
}
```

- w pesymistycznym przypadku usunięcie elementu wymaga przejścia całej listy pod danym indeksem,

7. Różnica zbiorów - zwraca zbiór będący różnicą zbiorów przekazanych w argumentach

```
Set<T> Set<T>::set_difference(Set set1, Set set2) {


    for(int index = 0; index < set1.table_size; index++) {

        for(auto it1 = set1.table[index].begin();
            it1 != set1.table[index].end(); it1++) {

            if(set2.set_is_member(*it1)) {
                T temp = *it1;
                it1--;
                set1.set_remove(temp);
            }

        }
    }

    return set1;
}
```

- funkcja przechodzi po elementach zbioru pierwszego,
- jeśli element znajduje się również w zbiorze drugim, funkcja usuwa go ze zbioru pierwszego,
- funkcja zwraca zmodyfikowany zbiór pierwszy, dzięki czemu nie zajmuje nowego miejsca w pamięci,
- funkcja przechodzi po wszystkich elementach zbioru, więc złożoność jest liniowa,

8. Suma zbiorów - zwraca zbiór będący sumą zbiorów przekazanych w argumentach

```
Set<T> Set<T>::set_union(Set set1, Set set2) {

    for(int i = 0; i < set2.table_size; i++) {

        for(auto it2 : set2.table[i])
            if(!set1.set_is_member(it2))
                set1.set_insert(it2);
    }

    return set1;
}
```

- funkcja przechodzi przez wszystkie elementy zbioru drugiego,
- jeśli element nie znajduje się w zbiorze pierwszym, funkcja dodaje go do zbioru pierwszego,
- funkcja zwraca zmodyfikowany zbiór pierwszy, dzięki czemu nie zajmuje nowego miejsca w pamięci,
- funkcja przechodzi po wszystkich elementach zbioru, więc złożoność jest liniowa,

9. Przecięcie zbiorów - zwraca zbiór będący iloczynem zbiorów przekazanych w argumentach

```
Set<T> Set<T>::set_intersection(Set set1, Set set2) {

    for(int index = 0; index < set1.table_size; index++) {

        for(auto it1 = set1.table[index].begin();
            it1 != set1.table[index].end(); it1++) {

            if(!set2.set_is_member(*it1)) {
                T temp = *it1;
                it1--;
                set1.set_remove(temp);
            }

        }
    }

    return set1;
}
```

- funkcja przechodzi przez wszystkie elementy zbioru pierwszego,
- jeśli element nie znajduje się w zbiorze drugim, funkcja usuwa go ze zbioru pierwszego,
- funkcja zwraca zmodyfikowany zbiór pierwszy, dzięki czemu nie zajmuje nowego miejsca w pamięci,
- funkcja przechodzi po wszystkich elementach zbioru, więc złożoność jest liniowa,