W. 5
Wyszukiwanie
oraz
Tablice haszujące

Przeszukiwanie nieposortowanej listy

Idea

- Jeżeli elementy w strukturze danych są ułożone losowo, bez określonego porządku, to musimy przejrzeć każdy element struktury do momentu znalezienia interesującego nas elementu.
- Złożoność obliczeniowa:
  - Najlepsza O(1) - od razu znajdujemy element.
  - Najgorsza O(n) - szukany element jest ostatnim elementem który przejrzymy (n – liczba elementów).
  - Średnia O(n).
- Złożoność jest więc taka jak dla listy.

Przykład

- int szukaj (int tab[n], int x)
- {
- int i :
- for(i =0; (i < n) && (tab[i] ! = x); i ++);
-
- return i :
- }

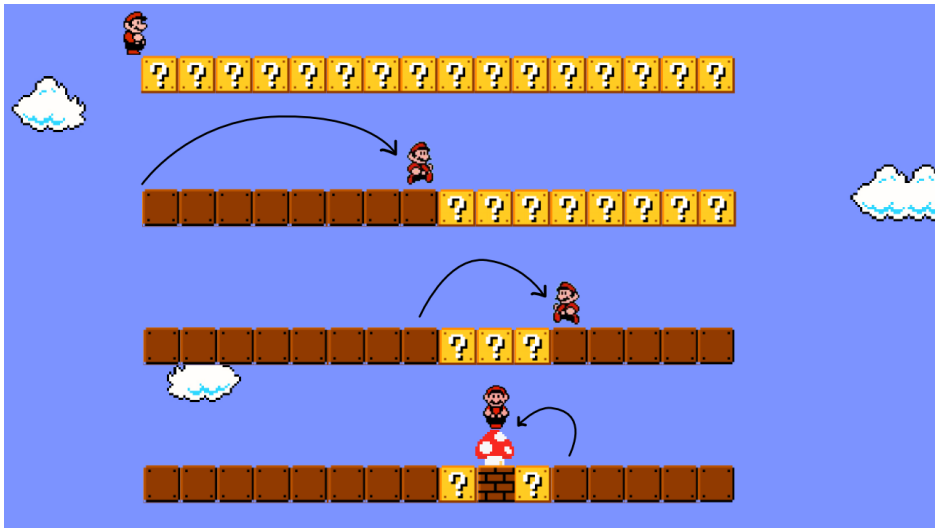
- Zauważ, że pętla for nie ma ciała – sprawdzanie tab[i] != x znajduje się w pętli for!!!

Przeszukiwanie binarne
(Binary search)

Idea

- Jeżeli mamy do czynienia ze strukturą w której elementy są posortowane, to możemy wykorzystać uporządkowanie, żeby przyśpieszyć wyszukiwanie.
- W przepadku uporządkowanej listy liczb od najmniejszego do największego, szukanie binarne polega na potraktowaniu listy jak drzewa, gdzie:
  - Elementy o indeksach większych od danego mają większe wartości.
  - Elementy i indeksach mniejszych od danego mają wartości mniejsze.

Algorytm



- Dopóki nie znajdziesz elementu lub nie przejrzysz całej tablicy:
  - Rozpocznij od środka tablicy:
    - Jeżeli element szukany ma wartość większą od środka, to rozpocznij od prawej połowy tablicy
    - Jeżeli element szukany jest mniejszy od środka to rozpocznij od lewej części tablicy.

Implementacja rekurencyjna

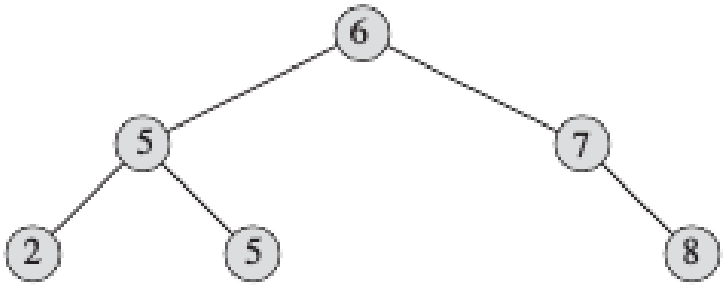
```
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
     int mid = l + (r - l) / 2;

        if (arr[mid] == x)
            return mid;

        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);

        // Else
        return binarySearch(arr, mid + 1, r, x);
    }

    // We reach here when element is not present in array
    return -1; }
```

Implementacja iteracyjna

```
int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r) {
        int m = l + (r - l) / 2;

        if (arr[m] == x)
            return m;

        if (arr[m] < x)
            l = m + 1;
        else
            r = m - 1;
    }

    // if we reach here, then element was not present
    return -1; }
```

Złożoność obliczeniowa

- Tablica
  - [2,5,5,6,7,8]



- W algorytmie przeszukiwania binarnego tablica o n elementach jest traktowana jako drzewo o wysokości $\log_2(n)$.
- Zatem w najgorszym przypadku musimy przejść w głąb całe drzewo startując od korzenia (środkowy element tablicy):
  - $O(\log_2(n))$

Tablice z haszowaniem
(Hash table)

Idea

- „Spłaszczamy” tablicę grupując elementy podobne w mniejsze listy.
- Grupowanie odbywa się przy pomocy funkcji, która dla danego elementu zwraca indeks listy w której dany element zapisać. Taką funkcję nazywamy **funkcją haszującą** lub **funkcją skrótu.**
  - Funkcja skrótu powinna w miarę jednorodnie rozdzielać dane z zakresu który zamierzamy magazynować w tablicy haszującej.

Funkcja skrótu/haszująca (hash function)

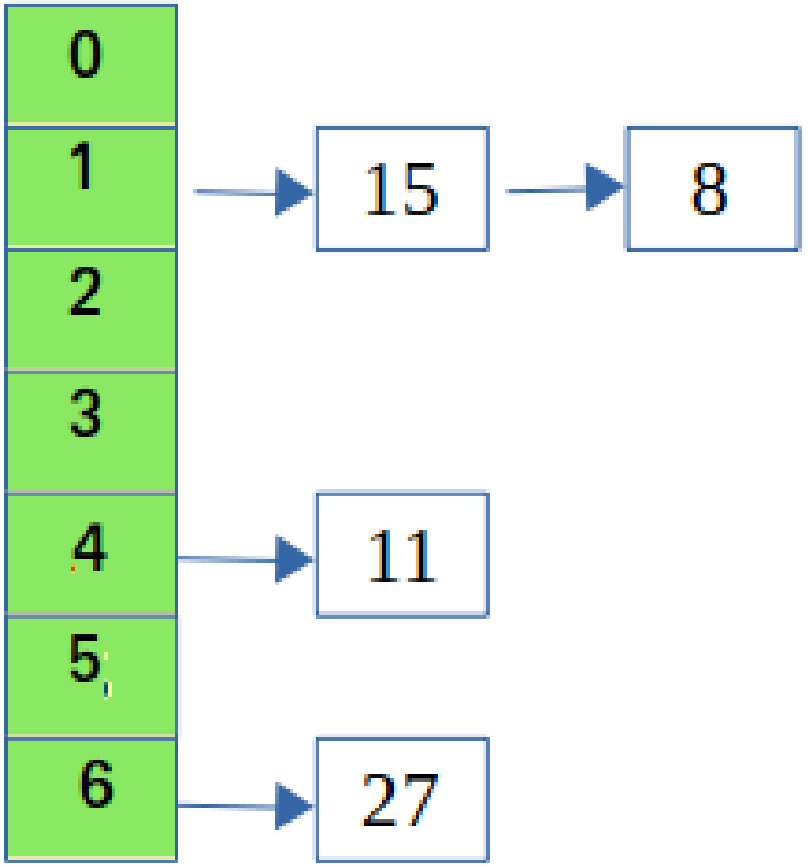
- Funkcja sumująca modulo Max.
- W przypadku napisów sumujemy kody ASCII liter w napisie, a następnie sumę dzielimy modulo Max.
- Metoda ta umożliwia rozdzielenie napisów w tablicy o Max komórkach.
  - Uwaga: Mogą zachodzić kolizje – dwa napisy mogą mieć taką samą wartość funkcji haszującej. Problem rozwiążemy tworząc tablice haszujące.

- int **hash**(const string &key, int max)
- { int hashVal = 0;
- for(int i = 0; i<key.length(); i++)
- { **hashVal += key[i];** }
- **hashVal %= max;**
- return hashVal;}

Praktyczne algorytmy haszujące

- MD5 – Message Digest – zwraca 128-bitową liczbę binarną będącą podsumowaniem danego ciągu znaków/pliku.
  - https://en.wikipedia.org/wiki/MD5
- SHA-1 (Secure Hash Algorithm) – zwraca 20 bajtową funkcję skrótu.
  - https://en.wikipedia.org/wiki/SHA-1

Tablica haszująca



- Tablica haszująca dla liczb całkowitych zawiera max kubełków (0-6).
- Każdy kubełek ma indeks będący wartością funkcji haszującej i %max.
- Kubełek zawiera wskaźnik na listę elementów które mają taką samą wartość funkcji haszującej.
- Wyszukiwanie elementu x polega na dotarciu do kubełka dla elementu o hashu x, a następnie analizie zawartości kubełka.

Tablica Haszująca

```
class Hash
{
int BUCKET; // No. of buckets
// Pointer to an array containing buckets
list<int> *table;

public:
Hash(int V); // Constructor

// inserts a key into hash table
void insertItem(int x);

// deletes a key from hash table
void deleteItem(int key);

// hash function to map values to key
int hashFunction(int x) { return (x % BUCKET); }

void displayHash();  };
```

- BUCKET – liczba kubełków;
- list<int> *table; - wskaźnik do tablicy kubełków;
- Funkcja haszująca modulo liczba kubełków:
  - int hashFunction(int x) { return (x % BUCKET); }

Konstruktor

- Hash::Hash(int b)
- {
- this->BUCKET = b;
- table = new list<int>[BUCKET];
- }

Wstawianie elementu

- void Hash::insertItem(int key)
- {
- int index = hashFunction(key);
- table[index].push_back(key);
- }

- Oblicz funkcję haszującą wstawianego elementu – key; To daje indeks kubełka.
- Wstaw element do kubełka.
- Metod push_back jest metodą kontenera list z biblioteki standardowej C++:
  - http://www.cplusplus.com/reference/list/list/

Usuwanie elementu

```
void Hash::deleteItem(int key)
{
int index = hashFunction(key);

list <int> :: iterator i;
for (i = table[index].begin();
i != table[index].end(); i++) {
if (*i == key) break; }

if (i != table[index].end())
table[index].erase(i); }
```

- Oblicz funkcję haszującą/indeks elementu do usunięcia.
- Iteruj po kubełku szukając indeksu elementu do usunięcia
- Jeżeli indeks odnaleziono, to usuń element – metoda erase z klasy list:
  - http://www.cplusplus.com/reference/list/list/

Wyświetlane zawartości listy

- void Hash::displayHash() {
- for (int i = 0; i < BUCKET; i++) {
- cout << i;
- for (auto x : table[i])
- cout << " --> " << x;
- cout << endl;
- }
- }

- Algorytm:
  - Iteruj po kubełkach:
    - Iteruj po liście wypisując elementy;
- Zauważ, że do wypisywania elementów tablicy użyto konstrukcji ze standardu C++11:
  - for (auto x : table[i]) cout << " --> " << x;
- Jest to konstrukcja for analogiczna jak w Pythonie do iteracji po liście. **Auto** jest zmienną o typie automatycznym – ustalanym na podstawie analizy typu prawej strony.
  - https://en.cppreference.com/w/cpp/language/range-for
  - https://en.cppreference.com/w/cpp/language/auto
- Program kompilujemy z flagą
  - **-std=c++11**

Program główny

- int main()
- { int a[] = {15, 11, 27, 8, 12};
- int n = sizeof(a)/sizeof(a[0]);
-
- Hash h(7);
-
- for (int i = 0; i < n; i++)
- h.insertItem(a[i]);
-
- h.deleteItem(12);
-
- h.displayHash();
-
- return 0; }

Złożoność obliczeniowa

- Jeżeli do tablicy haszującej wkładamy n elementów, to przy równym rozłożeniu elementów pomiędzy Max kubełków element znajdziemy w czasie:
  - O(n/Max) = O(n)
- Asymptotycznie złożoność obliczeniowa jest taka sama dla dużego n, jednak warto pamiętać, że czas przeszukiwania może być zredukowany o stały czynnik – Max – liczba kubełków.
- Aby elementy były w miarę jednorodnie rozłożone pomiędzy kubełkami należy wybrać funkcję haszującą dopasowaną do rozkładu danych.

Zadanie

- Napisz funkcję szukającą element w tablicy haszującej.
  - Wskazówka:
    - Wzoruj się na funkcji usuwającej element z tablicy haszującej.

Literatura dodatkowa

- Standard C++ 11:
  - https://en.wikipedia.org/wiki/C%2B%2B11
- Rozdział 7 - Algorytmy. Struktury danych i techniki programowania.
- Rozdziały 11 – T. Cormen, ‘Wprowadzenie do algorytmów’, PWN
- Rozdział 11 - Data Structures and Algorithms using Python.
- https://www.geeksforgeeks.org/c-program-hashing-chaining/
- http://www.cplusplus.com/reference/list/list/
- https://en.cppreference.com/w/cpp/language/range-for
- https://en.cppreference.com/w/cpp/language/auto

Koniec