データ構造とアルゴリズム: サーチ

2022/07/09 糸川倫太朗

リニアサーチとバイナリサーチ

- リニアサーチ
  - **「先頭から順番に探す値が見つかるまで探す方法」**配列を直線的（リニア）に探すので「リニアサーチ」と呼ばれる。値をひとつひとつ調べていくので、データが大量になると時間がかかってしまう。
- バイナリサーチ
  - **「調べる範囲を半分に絞りながら探す方法」**範囲を「二つ(バイナリ)に分けて」探すので「バイナリサーチ」と呼ばれる。データがばらばらに並んでいると規則性がないので、あらかじめデータを順番に並べておく必要がある。

> 出典: JavaScript リニアサーチとバイナリサーチ

探索の結果

		存在するか	探索回数	存在するか	探索回数	存在するか	探索回数	存在するか	探索回数	存在するか	探索回数
データ		20293		7789		4021		6586		30000	
リニアサーチ	10個	○	1	×	6	×	6	×	6	×	6
	100個	○	1	○	11	×	51	×	51	×	51
	1000個	○	1	○	11	○	101	×	501	×	501
	10000個	○	1	○	11	○	101	○	1001	×	5001
バイナリサーチ	10個	○	1	×	5	×	5	×	5	×	5
	100個	○	4	○	7	×	7	×	8	×	8
	1000個	○	10	○	9	○	3	×	11	×	11
	10000個	○	13	○	13	○	11	○	10	×	15

探索の考察

リニアサーチに比べてバイナリサーチのほうがデータ量が多くなっても探索回数が増えなかった。また各アルゴリズムの探索回数から計算流に従っていることも分かる。

自作したリニアサーチ、バイナリサーチのソースコード

cpp : linear.cpp

```cpp
#include <iostream>
#include <fstream>
#include <stdexcept>
#include <vector>
#include <string>

using std::cout;
using std::cin;
using std::endl;
using std::vector;
using std::string;

template<typename T>
```

```
int search(const vector<T> &arr, T &search_Element, int &counter) {
  int left;
  int length = arr.size();
  int position;
  int right = length - 1;
  for (left = 0; left <= right;) {
    counter++;
    if (arr[left] == search_Element) {
      position = left;
      return position + 1;
    }
    if (arr[right] == search_Element) {
      position = right;
      return position + 1;
    }
    left++;
    right--;
  }
  return 0;
}

int main(int argc, char *argv[]) {
  try {
    if (argc != 2) {
      throw std::out_of_range("out of range");
    }
    std::ifstream in(argv[1]);
    vector<int> vec;
    string value;
    if (in.fail()) {
      throw std::logic_error("file couldn't open");
    }
    while (!in.eof()) {
      in >> value;
      vec.push_back(std::stoi(value));
    }
    int count = 0;
    int a = 20293;
    int b = 7789;
    int c = 4021;
    int d = 6586;
    int e = 30000;
    cout << a << ":" << search(vec, a, count) << " " << "count: " << count <<
endl;
    count = 0;
    cout << b << ":" << search(vec, b, count) << " " << "count: " << count <<
endl;
    count = 0;
    cout << c << ":" << search(vec, c, count) << " " << "count: " << count <<
endl;
    count = 0;
    cout << d << ":" << search(vec, d, count) << " " << "count: " << count <<
endl;
    count = 0;
```

```
    cout << e << ":" << search(vec, e, count) << " " << "count: " << count <<
endl;
    return 0;
  }
  catch (std::exception &e) {
    cout << e.what() << endl;
    return -1;
  }
}
```

cpp : binary.cpp

```cpp
#include <iostream>
#include <fstream>
#include <stdexcept>
#include <vector>
#include <string>

using std::cout;
using std::cin;
using std::endl;
using std::vector;
using std::string;

template<typename T>
T partitionVec(vector<T> &vec, size_t start, size_t end) {
  T pivot = vec.at(start);
  auto lh = start + 1;
  auto rh = end;
  while (true) {
    while (lh < rh && vec.at(rh) >= pivot) {
      rh--;
    }
    while (lh < rh && vec.at(lh) < pivot) {
      lh++;
    }
    if (lh == rh) {
      break;
    }
    T tmp = vec.at(lh);
    vec.at(lh) = vec.at(rh);
    vec.at(rh) = tmp;
  }
  if (vec.at(lh) >= pivot) {
    return start;
  }
  vec.at(start) = vec.at(lh);
  vec.at(lh) = pivot;
  return lh;
}

template<typename T>
void sort(vector<T> &vec, size_t start, size_t end) {
  if (start >= end) {
    return;
```

```
  }
  auto boundary = partitionVec(vec, start, end);
  sort(vec, start, boundary);
  sort(vec, boundary + 1, end);
}

template<typename T>
void quickSort(vector<T> &vec) {
  sort(vec, 0, vec.size() - 1);
}

template<typename T>
int binarySearch(vector<T> &vec, T &item, int s1, int s2, int &counter) {
  counter++;
  if (s1 > s2) {
    return 0;
  }
  auto middle = (s1 + s2) / 2;
  if (item == vec.at(middle)) {
    return middle;
  }
  if (item > vec.at(middle)) {
    return binarySearch(vec, item, middle + 1, s2, counter);
  } else {
    return binarySearch(vec, item, s1, middle - 1, counter);
  }
}

template<typename T>
int search(vector<T> &vec, T &item, int &counter) {
  quickSort(vec);
  return binarySearch(vec, item, 0, vec.size() - 1, counter);
}


int main(int argc, char *argv[]) {
  try {
    if (argc != 2) {
      throw std::out_of_range("out of range");
    }
    std::ifstream in(argv[1]);
    vector<int> vec;
    string value;
    if (in.fail()) {
      throw std::logic_error("file couldn't open");
    }
    while (!in.eof()) {
      in >> value;
      vec.push_back(std::stoi(value));
    }
    int count = 0;
    int a = 20293;
    int b = 7789;
    int c = 4021;
    int d = 6586;
```

```
    int e = 30000;
    cout << a << ":" << search(vec, a, count) << " " << "count: " << count <<
endl;
    count = 0;
    cout << b << ":" << search(vec, b, count) << " " << "count: " << count <<
endl;
    count = 0;
    cout << c << ":" << search(vec, c, count) << " " << "count: " << count <<
endl;
    count = 0;
    cout << d << ":" << search(vec, d, count) << " " << "count: " << count <<
endl;
    count = 0;
    cout << e << ":" << search(vec, e, count) << " " << "count: " << count <<
endl;
    return 0;
  }
  catch (std::exception &e) {
    cout << e.what() << endl;
    return -1;
  }
}
```

ハッシュ法

ハッシュ探索は、O(1) という圧倒的に小さい計算量で探索を行える非常に優れたアルゴリズムです。

原理は、データを登録するときに、そのデータ自身の値を使って何らかの計算をおこなって、格納位置（通常、データ構造としては配列を使うので、その添字のこと）を決定します。

試しにC++で書くのならこのように書くと思われます。

cpp
```cpp
#include<bits/stdc++.h>
using namespace std;

class Hash {
private:
  int BUCKET;
  list<int> *table;
public:
  explicit Hash(int b);
  void insertItem(int key);
  void deleteItem(int key);
  [[nodiscard]] int hashFunction(int x) const {
    return (x % BUCKET);
  }
  void displayHash();
};

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

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

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);
  }
}

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

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;
}
```

ただ、上の実装だとハッシュ関数によって決定された位置がシノニムとなってしまうので、線形走査法などでリハッシュする必要があると思われます。

感想

レポートで与えられた課題よりもHash searchを書いたりリハッシュの方法を思いつくほうが時間がかかった。