GB20602 - Programming Challenges

Week 2 - Data Structures

Claus Aranha
caranha@cs.tsukuba.ac.jp

University of Tsukuba, Department of Computer Sciences

(last updated: April 24, 2022)

Version 2022

Introduction

CP4 – Chapter 2.1

Motivation

Choosing the right data structure is very important when writing a program:

- How easy is it to program?
- How fast are the operations?
- Does it have all the abilities necessary for my algorithm?

In this lecture, we review some data structures that are useful for programming challenges.

Most of these data structures are available in the standard library. A few you have to program by hand.

How to choose a data structure?

Main Operations of a Data Structure

Think about which operations you need for your program:

- Inserting new data once;
- Inserting new data after acessing;
- Accessing data in order;
- Accessing data out of order;
- Re-ordering data;
- Updating data;
- Deleting data;
- Finding data by position;
- Finding data by content;
- Summarizing data;

Different data structures will be better or worse at these operations.

You want to use **the simplest** data structure that can do what you need.

Data Structures and Standard Libraries

Most data structures that we use on programming contests are available in the **Standard Libraries**. It is important to know how to use the standard library of your language.

However, some specialist data structures are not available, so we have to code them by hand. Sometimes, we also have a modified version of a standard DS.

Personal Code Library

When you write many programs, you will discover that you write similar code many times. It is useful to store this code in a "Personal Library" file.

Topics we are studying today

Linear Data Structures

The STL array, sorting on arrays, searching, Deques and stacks

Non-Linear Data Structures

Priority Queues and Sets/Hashes

Hand Crafted Data Structures

Union-Find Disjoint Set

Topics we are NOT studying today

Data Structure Theory

In this lecture we are interested in *remembering, using and implementing* Data Structures that are useful in Programming Challenges. It is important to know them at a theoretical level, but please review the 2nd year DS lecture.

Big Number

In the past, we used to have a module on Big Number. Today, just use Python.

Part II – Linear Data Structures (Arrays)

CP4 – Section 2.2

Arrays are your friend

Arrays are the simplest data structure, but they are also the most used one.

- They are easy to use;
- They are very fast;
- They have many abilities;
- Complex data structures can be programmed as array + special functions;

It is important to know how to use arrays in your programming language.

Sometimes it is better to use a quick array than implement a complex DS!

1D arrays in C++: Basic Usage

```
# include <vector.h>

int arr[5] = {7,7,7};  // arr = {7,7,7,0,0} -- Fixed Size
vector<int> v(5, 5);   // v = {5,5,5,5,5}   -- size increases as necessary
int x = arr[2] + v[2]; // x = 12            -- Access them the same way.

arr[5] = 5;            // Runtime Error - Index Out of Bounds
cout << v[7];          // This returns 0! Be careful!

v.push_back(6);        // v = {5,5,5,5,5,6} - Increase the size (double)
```

Warning: "Index out of bounds" is a common source of Runtime Errors (RTE)

Understand your library: Many ways to do the same thing.

Example: How do you reseat an array?

```
#include <vector>
#include <string.h>
vector<int> v(10000,7)

memset(v, 0, 10000*__SIZEOF_INT__);       // Method 1
fill(v.begin(), v.end(), 0);              // Method 2
for (int i = 0; i < 10000; i++) v[i] = 0; // Method 3
v.assign(v.size(), 0);                    // Method 4
```

```
Method      |  executable size  |  Time Taken (in sec) |
            |  -O0    |  -O3    |  -O0      |  -O3     |
------------|---------|---------|-----------|----------|
1. memset   | 17 kB   | 8.6 kB  | 0.125     | 0.124    |
2. fill     | 19 kB   | 8.6 kB  | 13.4      | 0.124    |
3. manual   | 19 kB   | 8.6 kB  | 14.5      | 0.124    |
4. assign   | 24 kB   | 9.0 kB  | 1.9       | 0.591    |
```

Sorting in Arrays

Sorting an array is a frequent operation in programming challenges:

- Take the highest, lowest, median elements;
- Pre-computation step in many algorithms (binary search, greedy, etc);

How do we sort?

- Use the sort function from the standard library – $O(n \log n)$;

  Most cases.

- Make a simple sort algorithm by hand (bubble/selection sort) – $O(n^2)$

  When you need to sort with special conditions.

- Special sort for specific data (bucket/radix sort) – $O(n)$.

  Very large amount of data.

Sorting using the standard library

```
#include<iostream>
#include<algorithm>
#include<vector>

using namespace std;
int main() {
    int n, t; vector<int> values;
    cin >> n;

    for (int i=0; i<n; i++) { cin >> t; values.push_back(t); }

    sort(values.begin(), values.end())
    cout << values[n/2] << endl;       // find the median value
}
```

Sorting with a specific function

You can define a comparison function to sort in special cases (multiple variables, etc);

```
struct team{ string name; int point; int penal;
             team(string _n, int _po, int _pe) :
               name(_n), point(_p), penalty(_g){} };

bool cmp(team a, team b) {      % Sorting Function
  if (a.point != b.point)     return a.point > b.point;
  if (a.penalty != b.penalty) return a.penalty < b.penalty;
  return strcmp(a.name, b.name);
}

vector<team> v;
sort(v.begin(), v.end(), cmp); // sort using cmp
```

Binary Search in Sorted Arrays – $O(\log n)$

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main () {
  int n, t, search; vector<int> v;
  cin >> n >> search;  // Find "search" in an array with "n" elements

  for (int i=0; i<n; i++) { cin >> t; v.push_back(t); }
  sort (v.begin(), v.end()); // Need to sort before binary search!

  vector<int>::iterator low, up;
  low = lower_bound (v.begin(), v.end(), search); // lowest index
  up  = upper_bound (v.begin(), v.end(), search); // highest index
  cout << "Between" << (low-v.begin()) << " and " << (up-v.begin());
}
```

Special $O(n+k)$ sorting: Counting Sort

If we have to sort a large (n) quantity of numbers, but there is a small variation (k) of values, we can use **counting sort** (or bucket sort).

1. Let A be the input array, with possible values $K = \{k_0, k_1, \ldots, k_K\}$
2. Let F be a *cumulative frequency array*. $F[0]$ is the number of times k_0 happens in A, $F[1] = F[0]+$ number of times k_1 happens in A, and so on...
3. Proccess A element by element, starting from the back.
4. If $A[j] == k_i$, place $A[j]$ in the $F[i]$ position of the new array, and decrease $F[i]$ by 1.

Counting sort only works when k is not very big.

Stacks, Queues and Deques

These are special variants on vectors. They are highly optimized for inserting and removing from the start and the end of the array.

- **stack**: *push()* and *pop()*: Add and remove from the top of stack; *top()*: Check top of stack; *empty()*: Check if stack is empty.
- **queue**: *push()* and *pop()*: Add to the back, remove from the front; *front()*, *back()*: check front and back; *empty()*
- **deque**: *pop_front(), push_front(), pop_back(), push_back()*; *front(), back(), empty()*

We will use these variants for many algorithms in the future (graphs, geometry). Check your 2nd year Data Structures material for details!

Example of Using Stack

Input: A string containing "(" and ")". Example: "(())()((()()())"
Output: "balanced" or "unbalanced"

```
#include <stack>
stack<char> s;
char c;

while(cin >> c) {
  if (c == ’(’) s.push(c);
  else {
    if (s.size() == 0) { s.push(’⋆’); break; }
    s.pop();
  }
}
cout << (s.size() == 0 ? "balanced" : "unbalanced");
```

More Code Examples

The Github repository of the textbook has more code examples:

Basic Arrays

`https://github.com/stevenhalim/cpbook-code/blob/master/ch2/lineards/resizeable_array.cpp`

Sort, Binary Search and Permutation

`https://github.com/stevenhalim/cpbook-code/blob/master/ch2/lineards/array_algorithms.cpp`

Stack, Queue, Deque

`https://github.com/stevenhalim/cpbook-code/blob/master/ch2/lineards/list.cpp`

Part III – Non Linear Data Structures
CP4 – Section 2.3

Non-Linear Data Structures

Non-Linear Data Structures refer to Data Structures that are implemented as trees (balanced trees, red-black trees, etc).

They often have some useful property, such as being really fast for ordered insertion, deletion or update.

These Data Structures are usually available as part of the **standard libraries**, so today we will just remember them and check how to use them.

It is still useful to understand how they work. Review your 2nd year DS material, and check the link at the end of this video.

Priority Queue

Access the maximum element in $O(1)$, and insert/delete in $O(\log n)$.

```
#include <utility>     // pair
#include <queue>       // priority_queue
using namespace std;
typedef pair<int, string> is;

priority_queue<is> pq;
pq.push(make_pair(100, "john")); // insertion in O(log n)
pq.push({10, "billy"});          // alternative way with {}
pq.push({20, "andy"});
pq.push({2000, "grace"});

is first = pq.top(); pq.pop();  // first = (2000, grace)
is second = pq.top(); pq.pop(); // second = (100, john)
```

Priority Queue

Priority Queue is useful for a large variety of problems.

We will use PQ in the future for:

- Dijkstra Algorithm
- Minimum Spanning Tree (Prim's, Krushkal's Algorithms)
- etc...

Sets and Maps: Accessing Very Large sets

Imagine that we have two very large sets:

- Set size: 10^7
- Element size: 10^{12} (**UNIQUE)**

If we want to identify the common elements of the two sets, we would need a data structure that can access large amounts of unique data quickly.

C++ Data Structures

- **unordered_set**: Store unique keys. Search/Insert/Delete in $O(1)$, no order.
- **unordered_map**: Store key/data, Search/Insert/Delete in $O(1)$, no order.
- **set**: Store unique keys. Search/Insert/Delete in $O(1)$.
- **map**: Store key/data, Search/Insert/Delete in $O(1)$.

Unordered Map/Set Example

```
#include <unordered_map>
using namespace std;
unordered_map<string, int> mapper;

mapper["john"]   = 78; mapper["billy"]  = 69; mapper["andy"]   = 80;
mapper["steven"] = 77; mapper["felix"]  = 82; mapper["grace"]  = 75;

for (auto &[key, value] : mapper) // Results are in mixed order
  printf("%s %d\n", key.c_str(), value);

printf("steven's score is %d\n", mapper["steven"]);
if (mapper.find("andy") == mapper.end())
  printf("not found\n");

mapper.clear();
```

Map/Set Example

```
#include <set>
using namespace std;
set<int> uv;

uv.insert(78); uv.insert(69); uv.insert(80);
uv.insert(77); uv.insert(82); uv.insert(75);
printf("%d\n", *uv.find(77));  // O(log n) search

for (auto it = uv.begin(); it != uv.lower_bound(77); it++)
  printf("%d,", *it); // returns [69, 75] (before 77)
for (auto it = uv.lower_bound(77); it != uv.end(); it++)
  printf("%d,", *it); // returns [77, 78, 80, 81, 82]

used_values.clear();
```

Some Extra Code

See more examples in the textbook Github.

Priority Queue

https://github.com/stevenhalim/cpbook-code/blob/master/ch2/nonlineards/priority_queue.cpp

Maps and Sets

https://github.com/stevenhalim/cpbook-code/blob/master/ch2/nonlineards/unordered_map_unordered_set.cpp
https://github.com/stevenhalim/cpbook-code/blob/master/ch2/nonlineards/map_set.cpp

Visualization of Data Structures: https://visualgo.net/

Hand-made Data Structures: The Union-Find Disjoint Set (UFDS) (CP4 2.4.2)

Union-Find Disjoint Set (UFDS)

Motivating Problem

Network Connections – UVA793

We define a network with *n* computers. Using the commands "c" and "q", we **set** and **test** the connection between the computers.

Input: The number of computers *n*, and a sequence of commands:

- c i j – Make computer *i* and *j* connected.
- q i j – Ask if computer *i* is connected to computer *j*. (yes/no)

Output: The number of queries (q) with answer "yes", and the number of queries with answer "no".

Union-Find Disjoint Set (UFDS)

Motivating Problem – Naive answer

Neighborhood Graph

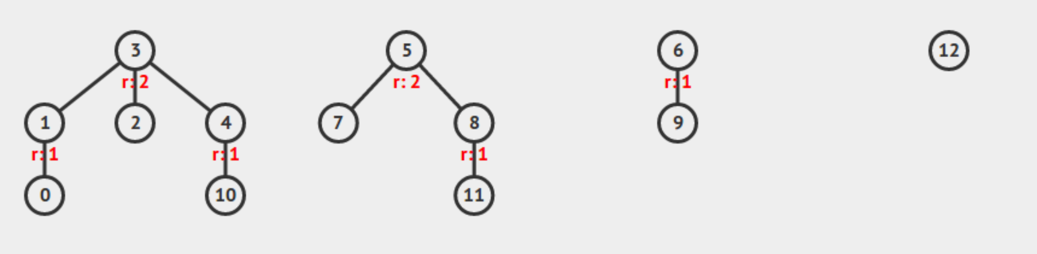
- Initialize an $n \times n$ matrix with zeros.
- For every “c i j” input, $N_{i,j}$ and $N_{j,i}$ becomes 1.
- For every “q i j”, we perform a breadth first search on the graph.

How good is this solution?

- Cost to insert a new connection: $O(1)$
- Cost to check if “q i j”: $O(V + E)$

We can do better!

Union-Find Disjoint Set



- The UFDS keeps sets of items, each is represented by a parent;
- When you join two sets You join their parents;
- When you test the parent of an item You flatten the tree;
- Test_item and Join_item are both O(1); *(amortized)*
- Visualization: `https://visualgo.net/ja/ufds`;

UFDS Implementation using Arrays

```
int p[MAX], r[MAX];
                             # which groups x belong to?
int find(int x) { return x == p[x] ? x : p[x]=find(p[x]); }

int join(int x, int y) {     # x and y are the same group
    x = find(x), y = find(y);
    if(x != y) {
        if(r[x] < r[y])     { p[x] = y; r[y] += r[x]; }
        else                { p[y] = x; r[x] += r[y]; }
        return 1;
    }
    return 0;
}
void init() { # Initialize each element as separate group
    for(int i = 0; i < MAX; i++)  { p[i] = i; r[i] = 1; }
}
```

About these Slides

Individual images in some slides might have been made by other authors. Please see the following pages for details.

Image Credits I