Estratégias Algorítmicas 2022/23

Week 1 - Intro to Mooshak and Data Structures

1 2 9 0

UNIVERSIDADE D
COIMBRA

Mooshak

Introduction

Mooshak is a web application to run and manage programming competitions, and to support programming classes (like ours).

Main features (for our purposes):

- Compile, execute and grade submitted code against predefined test cases

Registering

To register on Mooshak you should:

1. Go to `https://mooshak.dei.uc.pt/~ea/`
2. Select the `EA2023_PL00` contest (if it is not the only one)
3. Click on Register
4. Use your student number as your name (e.g. 2023999999)
5. Fill out the rest of the details and click Submit
6. You should receive your password in your email's inbox

Submitting code

To submit code to a problem:

1. Login to the contest at `https://mooshak.dei.uc.pt/~ea/`
2. Select the problem
3. Click on Browse to select your source code file
4. Click on Submit
5. Check the outcome of your submission

Possible outcomes

The most likely outcomes are:

- Accepted
- Compile Time Error
- Runtime Error
- Time Limit Exceeded
- Wrong Answer
- Presentation Error

Supported languages

The following languages are supported

- C11 (GCC 8.3.1)
  - `gcc -std=c11 -O2 "$file" -lm`
  - `./a.out`
- C++17 (GCC 8.3.1)
  - `g++ -std=c++17 -O2 "$file" -lm`
  - `./a.out`
- Java 11 (OpenJDK 11.0.9.1)
  - `javac "$file"`
  - `java -Xss8m -classpath .  "$name"`
- Python 3.7.9 (PyPy 7.3.3)
  - `pypy3 -m py_compile "$file"`
  - `pypy3 "$file"`

Input & Output

Input is read from `stdin` and the output is written to `stdout`. To simulate this on your computer you can, for example, run the following command in the command line:

```
./a.out < inputFile.txt > outputFile.txt
```

Java and Python beware

The standard methods to deal with I/O in Java and Python are slow and can result in a verdict of *Time Limit Exceeded*.

However, there are other options.

Fast Java I/O

Snippet at https://git.dei.uc.pt/snippets/23

```
import java.util.*;
import java.io.*;

class Main {

  public static void main(String[] args) {
    InputReader in = new InputReader(System.in);
    PrintWriter out = new PrintWriter(System.out);

    int n = in.nextInt();
    out.println(n);
    out.close(); // Very important
  }

  static class InputReader {
    // Implemented on the next slide
  }
}
```

Fast Java I/O

```
static class InputReader {
  public BufferedReader reader;
  public StringTokenizer tokenizer;
  public InputReader(InputStream stream) {
    reader = new BufferedReader(new InputStreamReader(stream));
    tokenizer = null;
  }
  public String next() {
    while (tokenizer == null || !tokenizer.hasMoreTokens()) {
      try {
        tokenizer = new StringTokenizer(reader.readLine());
      } catch (IOException e) {
        throw new RuntimeException(e);
      }
    }
    return tokenizer.nextToken();
  }
  public int nextInt() {
    return Integer.parseInt(next());
  }
}
```

Extra notes for Java

- The code should be implemented inside a single class
- The java file should have the same name as that class, e.g. `class Main` should be inside a `Main.java` file
- The java source code cannot have a `package abc;` instruction

Fast Python I/O

Snippet at https://git.dei.uc.pt/snippets/24

```
from sys import stdin,stdout

def readln():
    return stdin.readline().rstrip()

def outln(n):
    stdout.write(str(n))
    stdout.write("\n")

n = int(readln())
outln(n)
```

Fast(er) C++ I/O

Snippet at https://git.dei.uc.pt/snippets/25

```cpp
#include <iostream>

int main() {
  // We probably do not need this but it is faster
  std::ios_base::sync_with_stdio(0);
  std::cin.tie(0);

  int n;
  std::cin >> n;
  std::cout << n << "\n";

  return 0;
}
```

C I/O

Snippet at https://git.dei.uc.pt/snippets/26

```c
#include <stdio.h>

int main() {
  int n;
  scanf("%d", &n);
  printf("%d\n", n);
  return 0;
}
```

Exercise A: Product of two numbers

Exercise A on the `EA2023_PL00` contest is a very simple example to get you acquainted with Mooshak

You need to read two integers from `stdin` and output the result of their product to `stdout`

Basic Data Structures

Arrays

- $O(1)$ for lookup and in general $O(n)$ for insertions and deletions
- However, insertions and deletions at the end can have $O(1)$
- Can usually be stored either in the stack or the heap
    - Stack is preferable when possible (e.g. you know the size of the array *a priori* and it fits on the stack)
- Contiguous memory is cache friendly

Arrays - C

```c
// Static size array on the stack
int arr[3] = {0,0,0};
arr[i];  // O(1)

// Struct to keep a dynamic size array on the heap
typedef struct vector {
  int* data;       // malloc(n*sizeof(int));
  size_t size;     // 0
  size_t capacity; // n
} vector;

// We would need to implement these
vector vec_new();
void   vec_free(vector*);
void   vec_get(vector*, size_t);       // O(1)
void   vec_set(vector*, size_t, int);  // O(1)
void   vec_insert_back(vector*, int);  // Can be O(1) amortized
void   vec_remove_back(vector*, int);  // Can be O(1)
```

Note: For simplicity only the most relevant operations are shown.

Arrays - C++

```cpp
// Static size array saved on the stack
#include <array>
std::array<int, 3> arr;
arr[i]; // O(1)

// Dynamic size array saved on the heap
#include <vector>
std::vector<int> vec();
vec[i];              // O(1)
vec[i] = ...;        // O(1)
vec.push_back(int); // O(1) amortized
vec.pop_back();      // O(1)
```

Arrays - Java

```java
// Static size array saved on the heap
var arr = new int[3];
arr[i]; // O(1)

// Dynamic size array saved on the heap
import java.util.ArrayList;
var vec = new ArrayList<Integer>();
vec.get(int);               // O(1)
vec.set(int, Integer);      // O(1)
vec.add(Integer);           // O(1) amortized
vec.remove(vec.size()-1);   // O(1)
```

There is also `java.util.Vector<T>` which is thread safe, but generally less efficient.

Arrays - Python

```python
# Dynamic size array saved on the heap
v = list()
v[i]              # O(1)
v[i] = ...        # O(1)
v.append(value) # O(1) amortized
v.pop()           # O(1)
```

Arrays - Exercises

B: Read a list of numbers and output them in reverse order (max size is known in advance) - use a static size array

C: Read a list of numbers and sort them (the number of elements is not known in advance, read until EOF) - use a dynamic size array and the language sorting function:

- C: `qsort` from `<stdlib.h>`
- C++: `sort` from `<algorithm>`
- Java: `Collections.sort` from `java.util`
- Python: `list.sort()` or `sorted`

Code to read list of numbers until EOF:

- C: `https://git.dei.uc.pt/snippets/28`
- C++: `https://git.dei.uc.pt/snippets/27`
- Java: `https://git.dei.uc.pt/snippets/29`
- Python: `https://git.dei.uc.pt/snippets/30`

Deque

- An array of arrays that can have $O(1)$ insertion and deletion at the beginning as well, with a small cost in terms of random access and insertion/deletion at the end when compared to dynamically sized arrays
- Implementations
  - `std::deque` in C++
  - `java.util.ArrayDeque` in Java

Linked Lists

- $O(1)$ for insertions and deletions but $O(n)$ for lookup
- Useful whenever there are many insertions or deletions in the middle or at the beginning

Linked Lists - C

```c
typedef struct node {
  int value;
  struct node *next;
  struct node *prev; // if we need to go back
} node;

typedef struct list {
  node* head;
  node* tail;
} list;

// We can implement these
list ll_new();
void ll_free(list*);
void ll_insert_back(list*, int);   // O(1)
void ll_remove_back(list*);        // O(1)
void ll_insert_front(list*, int);  // O(1)
void ll_remove_front(list*);       // O(1)
void ll_insert_after(node*, int);  // O(1)
void ll_insert_before(node*, int); // O(1)
void ll_remove(node*);             // O(1)
```

Linked Lists - C++

```cpp
#include <list>

std::list<int> l();
l.push_back(int);          // O(1)
l.pop_back();              // O(1)
l.push_front(int);         // O(1)
l.pop_front();             // O(1)
l.insert(iterator, int);   // O(1)
l.erase(iterator);         // O(1)
```

There is also `std::forward_list<T>` that can only move forward (i.e. no prev pointer)

Linked Lists - Java

```java
import java.util.LinkedList;

var l = new LinkedList<Integer>();
l.addLast(Integer);              // O(1)
l.removeLast();                  // O(1)
l.addFirst(Integer);             // O(1)
l.removeFirst();                 // O(1)
list_iterator.insert(Integer); // O(1)
list_iterator.remove();          // O(1)
```

Linked Lists - Python

```
class Node:
  def __init__(self):
    self.value = None
    self.next = None
    self.prev = None

  # To implement
  def insert_before(self, value)
  def insert_after(self, value)

class LinkedList:
  def __init__(self):
    self.head = None
    self.tail = None

  # To implement
  def insert_back(self, value)
  def insert_front(self, value)
  def pop_back(self)
  def pop_front(self)
```

Linked Lists - Exercises

D - Building a list: follow a set of instructions to build an integer sequence

Stacks and Queues

- Stacks are used to implement last-in first-out (LIFO) mechanisms, which can be useful, for example, to simulate recursion
- Queues are used to implement first-in first-out (FIFO) mechanisms
- They can be easily implemented with arrays and linked lists, or using a language's data structures
  - `std::stack` and `std::queue` in C++
  - `java.util.Stack` and `java.util.Queue` in Java

Stack and Queues - Exercises

E - Postfix calculator (stack)

F - Queue processing time (queue)

Self-balancing binary trees

- $O(\log n)$ for insertion, deletion and lookup
- AVL-trees and red-black trees are the most efficient but hard to implement

Self-balancing binary trees - C

```c
// Red-black tree example
typedef struct node{
  int value;
  bool color;
  struct node *left, *right, *parent;
} node;

typedef struct tree{
  node* root;
} tree;

// We can implement these
tree  rb_new();
void  rb_free(tree*);
void  rb_insert(tree*, int);      // O(log n)
node* rb_find(tree*, int);        // O(log n)
void  rb_erase(tree*, int);       // O(log n)
```

Self-balancing binary trees - C++

```cpp
#include <set>

std::set<int> s();
s.insert(int);       // O(log n)
s.find(int);         // O(log n)
s.lower_bound(int); // O(log n) smallest element >= input
s.upper_bound(int); // O(log n) smallest element >  input
s.erase(int);        // O(log n)
```

There is also `std::map<K, V>` for different keys and values. Also, you have `std::multiset<V>` and `std::multimap<K, V>` if you need to support repeated values in the tree.

Self-balancing binary trees - Java

```
import java.util.TreeSet;

var s = new TreeSet<Integer>();
s.add(Integer);       // O(log n)
s.contains(Integer);  // O(log n)
s.floor(Integer);     // O(log n) greatest element <= input
s.lower(Integer);     // O(log n) greatest element < input
s.ceiling(Integer);   // O(log n) smallest element >= input
s.higher(Integer);    // O(log n) smallest element >  input
s.remove(Integer);    // O(log n)
```

There is also `java.util.TreeMap<K, V>` for different keys and values. There is no version that allows repeated elements, but for a set with repeated elements you can use the `TreeMap` if you are careful to remove the elements once their count reaches 0.

Self-balancing binary trees - Python

Needs to be implemented by hand

```
class Node:
  def __init__(self):
    self.value = None
    self.color = None
    self.left = None
    self.right = None
    self.parent = None

class RBTree:
  def __init__(self):
    self.root = None

  # Some possible methods
  def rb_insert(self, int) # O(log n)
  def rb_find(self, int)   # O(log n)
  def rb_erase(self, int)  # O(log n)
```

Self-balancing binary trees - Exercises

G - Bowling shoes: operations on a sorted list of integers

Heap Priority Queue

- The minimum (or maximum) element can be accessed or removed in $O(1)$, insertions are $O(\log n)$
- Implementations
  - `std::priority_queue` in C++
  - `java.util.PriorityQueue` in Java
  - `heapq` standard library in Python

Hash Tables

- A hash table takes $O(1)$ for insertion, deletion and lookup, when there are no hash collisions
- But it may take $O(n)$ if there are many hash collisions
- As such a good hashing function is very important

Hash Tables - C

```c
typedef struct {
  int **data;
  size_t size;
} hash_table;

// Hash index for value (0 <= 0 < hash_table.size)
size_t ht_hash_index(hash_table*, int);

void ht_insert(hash_table*, int);     // O(1) best, O(n) worst
bool ht_contains(hash_table*, int);   // O(1) best, O(n) worst
void ht_erase(hash_table*, int);      // O(1) best, O(n) worst
```

Different values may have the same hash index (hash collision), in which case, we need to compare every value in the table with the same hash to the value we are currently inserting/remove/finding. Also note that this is not the only valid design.

Hash Tables - C++

```cpp
#include <unordered_set>

std::unordered_set<int> s;
s.insert(int);  // O(1) best, O(n) worst
s.find(int);    // O(1) best, O(n) worst
s.erase(int);   // O(1) best, O(n) worst
```

There is also `std::unordered_map<K, V>` for different keys and values.

Hash Tables - Java

```java
import java.util.HashSet;

var s = new HashSet<int>();
s.add(int);       // O(1) best, O(n) worst
s.contains(int);  // O(1) best, O(n) worst
s.remove(int);    // O(1) best, O(n) worst
```

There is also `java.util.HashMap<K, V>` for different keys and values, as well `java.util.LinkedHashSet<V>` and `java.util.LinkedHashMap<K, V>`, for different implementations.

Hash Tables - Python

```
s = set() # hash set
s.add(value)            # O(1) best, O(n) worst
s.__contains__(value)   # O(1) best, O(n) worst
s.remove(value)         # O(1) best, O(n) worst
```

There is also `d = dict()` for keeping distinct keys and values

Hash Tables - Exercise

H - Inventory Management: operations on an unsorted key-value store

Summary - Usage guidelines

Use static size arrays when:

- You want a fixed number of items
- You want a stack-allocated array (when supported by the programming language)

Use dynamic size arrays when:

- You want a re-sizable array
- You want an heap-allocated array
- You will mostly be inserting and removing from the end (or near the end)

Summary - Usage guidelines

Use deques when:

- You want a dynamic size array, but
- You want to insert/remove from both ends of the array (or near them)

Use linked lists when:

- You need to constantly insert and remove from the middle of array in known locations
- You don't need random access by index

Summary - Usage guidelines

Use priority queues when:

- You only ever need the "largest" or "smallest" element of the collection

Use self-balancing binary trees when:

- You want to keep elements sorted
- You need to find if elements exists or not in a list
- You need to find the elements that are closest to another

Use hash tables when:

- You need to see if elements exist or not in a list and you do not care about order

Conclusion

- The right choice for a data structure depends on how many insertions, deletions an lookup have to be performed and in which way
- There may be several theoretically good choices for the problem at hand, so testing may be needed to filter out the worse options
- There are many more data structures we did not discuss here but these cover many cases
- `https://en.wikipedia.org/wiki/List_of_data_structures`