МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«Национальный исследовательский университет ИТМО»

ФАКУЛЬТЕТ ПРОГРАММНОЙ ИНЖЕНЕРИИ И КОМПЬЮТЕРНОЙ ТЕХНИКИ

БАЗОВЫЕ ЗАДАЧИ 9–12

по дисциплине

«АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ»

Выполнил:
Студент группы P3218
Рамеев Тимур
Ильгизович
Преподаватель:
Косяков М. С.
Тараканов Д. С.

Санкт-Петербург 2024

Содержание

Машинки

Исходный код

```cpp
#include <iostream>
#include <unordered_map>
#include <set>
#include <list>
#include <vector>

using namespace std;

list<int> main_array[100001]{};
int all_toys[500001]{};

int main() {

	int n, k, p;
	cin >> n >> k >> p;
	int main_counter = 0;

	for (int i = 0; i < p; i++) {
		int current_toy;
		cin >> current_toy;
		main_array[current_toy].push_back(i);
		all_toys[i] = current_toy;
	}

	auto cmp = [](int a, int b) {
		if (main_array[b].size() == 0) {
			return false;
		}
		if (main_array[a].size() == 0) {
			return true;
		}
		return main_array[a].front() > main_array[b].front();
	};

	set<int, decltype(cmp)> current_toys(cmp);
	for (int i = 0; i < p; i++) {
		if (current_toys.contains(all_toys[i])) {
			current_toys.erase(all_toys[i]);
			main_array[all_toys[i]].pop_front();
			current_toys.insert(all_toys[i]);
		}
		else {
			if (current_toys.size() == k) {
				current_toys.erase(current_toys.begin());
			}
			main_array[all_toys[i]].pop_front();
			current_toys.insert(all_toys[i]);
			main_counter++;
		}
	}

	cout << main_counter << endl;
	return 0;
}
```

Пояснение к примененному алгоритму

Входные данные

- N – различных видов игрушек
- K – сколько игрушек может находиться на полу
- P – строк, игрушки, в которые Петя будет играть по очереди

Алгоритм решения

Для начала считываем все элементы в массив all_toys. Параллельно с этим заводим массив листов main_array, и также заполняем его. Также заводим set current_toys, который будет хранить текущие игрушки, лежащие на полу.

Переопределяем компоратор. Таким образом после этого элементы внутри множества current_toys будут иметь следующий порядок: В начале будут элементы, имеющие больший индекс в начале своего list.

Затем, все что остается сделать – это пройтись P раз в цикле, и на каждой итерации проверять наличие текущей игрушки внутри current_toys. Если она есть, то удаляем и добавляем заново текущую игрушку. Это нужно, чтобы обновить порядок сортировки внутри множества.

Если же такого элемента нет, то добавляем его в current_toys, не забывая при этом перед этим, обновить его передний элемент внутри списка в all_toys. Инкрементируем счетчик, который потом будем выводить. Если размер current_toys совпадает с максимальным допустимым количеством игрушек на полу, то удаляем первый элемент в множестве current_toys.

Последним шагом выводим счетчик на экран.

Сложность

$$O(P \cdot \log K)$$

Гоблины и очереди

Исходный код

```cpp
#include <iostream>
#include <string>
#include <list>
#include <iterator>

using namespace std;

int main() {

	int n;
	cin >> n;
	list<int> main_list;
	list<int> result_list;
	list<int>::iterator iter;
	int current_pos;
	string current_sign, current_number;

	for (int i = 0; i < n; i++) {
		if (main_list.size() <= 1) {
			iter = main_list.begin();
			current_pos = 0;
		}
		else {
			int current_center = (main_list.size() + 1) / 2;
			advance(iter, current_center - current_pos);
			current_pos = current_center;
		}
		cin >> current_sign;
		if (current_sign == "*") {
			cin >> current_number;
			if (main_list.size() == 1) {
				main_list.push_back(stoi(current_number));
			}
			else {
				main_list.emplace(iter, stoi(current_number));
				current_pos += 1;
			}

		}
		if (current_sign == "+") {
			cin >> current_number;
			main_list.push_back(stoi(current_number));
		}
		if (current_sign == "-") {
			result_list.push_back(main_list.front());
			main_list.pop_front();
			current_pos--;
		}
	}

	for (int x : result_list) {
		cout << x << endl;
	}

	return 0;
}
```

Пояснение к примененному алгоритму

Входные данные

- N – количество запросов
- N – запросов, каждый из которых состоит из знака «+», «-», «*» и номер текущего гоблина.

Алгоритм решения

Решение пришлось немного закостылить. Используется основной list main_list, а также result_list для вывода результата. Основной идеей является использование итератора на main_list если на каждой итерации определять его позицию, то скорость решения будет высокой. Итак, первым делом отсекаем случай, когда разрмер листа меньше двух. В таком случае указываем текущее положение итератора 0. Иначе рассчитываем его как половина размера main_list.

Далее разделяем обработку запроса на три части, в зависимости от знака, который передается вместе с запросом. Если знак +, то добавляем в конец списка. Если знак *, то добавляем на место, куда указывает итератор, именно для этого он нам и нужен. Ну и если знак -, то просто снимаем первый элемент из main_list. Также не стоит забывать о корректировке положения итератора в зависимости от текущего запроса. Я имею в виду прибавление и отнимание единицы, если это необходимо.

Сложность

$$O(N)$$

Менеджер памяти – 1

Исходный код

```
#include <iostream>
#include <set>
#include <map>
#include <vector>
#include <utility>

using namespace std;

struct block {
	bool status;
	int size;
	int index;
	int number;
	struct block* previous;
	struct block* next;
};

int main() {

	auto cmp = [](pair<int, int> a, pair<int, int> b) {
		if (a.first != b.first) {
			return a.first > b.first;
		}
		return a.second < b.second;
		};

	map<pair<int, int>, struct block*, decltype(cmp)> free_blocks; // key - value : pair(size,
index) - link
	map<int, struct block> all_blocks; // key - value : index - link
	map<int, struct block*> busy_blocks; /// key - value : number - link

	vector<int> result;

	int n, m;
	cin >> n >> m;

	struct block init_block = { true, n, 1, 0, NULL, NULL };
	all_blocks[1] = init_block;
	free_blocks[make_pair(n, 1)] = &all_blocks[1];

	for (int i = 1; i <= m; i++) {
		int current_request;
		cin >> current_request;
		if (current_request > 0) {
			if (free_blocks.size() == 0) {
				result.push_back(-1);
				continue;
			}
			struct block* max_free_block = free_blocks.begin()->second;
			if (max_free_block->size == current_request) {
				max_free_block->status = false;
				max_free_block->number = i;
				busy_blocks[i] = max_free_block;
				free_blocks.erase(free_blocks.begin());
				result.push_back(max_free_block->index);
				continue;
			}

			if (max_free_block->size > current_request) {
				int index = max_free_block->index + current_request;
				all_blocks[index] = { true, max_free_block->size - current_request, index,
0, max_free_block, max_free_block->next };
```

```
                        max_free_block->status = false;
                        max_free_block->size = current_request;
                        max_free_block->number = i;
                        max_free_block->next = &all_blocks[index];
                        busy_blocks[i] = max_free_block;
                        free_blocks.erase(free_blocks.begin());
                        free_blocks[make_pair(all_blocks[index].size, all_blocks[index].index)] =
&all_blocks[index];
                        result.push_back(max_free_block->index);
                        continue;
                }
                result.push_back(-1);
        }
        else {
                current_request *= -1;
                if (busy_blocks[current_request] == NULL) {
                        continue;
                }
                struct block need_free = *busy_blocks[current_request];
                /*struct block* need_free = &(temp);*/

                need_free.status = true;
                busy_blocks.erase(need_free.number);

                if (need_free.next != NULL && need_free.next->status == true) {
                        int next_size = need_free.next->size;
                        int next_index = need_free.next->index;
                        free_blocks.erase(make_pair(next_size, next_index));
                        need_free = { true, need_free.size + need_free.next->size, need_free.index,
0, need_free.previous, need_free.next->next };
                        all_blocks.erase(next_index);
                }
                if (need_free.previous != NULL && need_free.previous->status == true) {
                        int previous_size = need_free.previous->size;
                        int previous_index = need_free.previous->index;
                        free_blocks.erase(make_pair(previous_size, previous_index));
                        need_free = { true, need_free.size + need_free.previous->size,
need_free.previous->index, 0, need_free.previous->previous, need_free.next };
                }

                all_blocks.erase(need_free.index);
                all_blocks[need_free.index] = need_free;
                if (need_free.previous != NULL)
                        need_free.previous->next = &all_blocks[need_free.index];
                if (need_free.next != NULL)
                        need_free.next->previous = &all_blocks[need_free.index];

                free_blocks[make_pair(need_free.size, need_free.index)] =
&all_blocks[need_free.index];
        }
    }

    for (int x : result) {
        cout << x << endl;
    }

    return 0;
}
```

Пояснение к примененному алгоритму

Входные данные

- N – количество ячеек памяти
- M – количество запросов
- M строк, каждая из которых – запрос на выделение или освобождение памяти

Алгоритм решения

Первым делом создал структуру, олицетворяющую блок памяти. Она содержит поля status – свободна / несвободна; поле size – размер данного блока; index – индекс, с которого начинается данный блок; number – порядковый номер запроса, который аллоцировал данный блок памяти; а также ссылки на соседние блоки previous и next.

Все блоки памяти хранятся в трех структурах данных map – free_blocks, all_blocks, busy blocks. По их названию, думаю понятно, что в них хранится. А также используется вектор result для аккумулирования данных и дальнейшего вывода на экран.

Вручную аллоцируем блок размером N, который будет считаться свободным. Далее в цикле M раз считываем очередной запрос. Обрабатывается запрос следующим образом. Для начала определяется тип запроса – на аллоцирование или на освобождение памяти. Это определяется простым сравнением запроса с нулем.

Если это запрос на аллоцирование данных, то мы действуем следующим образом: обращаемся к map free_blocks, который отсортирован специальным образом с помощью компоратора cmp. В начале будут находиться самые большие свободные блоки, а если размер блоков одинаков, то первыми будут блоки с меньшим индексом. Поэтому достаточно обратиться к первому элементу в free_blocks. Если его размер меньше запрашиваемого, смело выводим -1. Иначе аллоцируем часть свободного блока под наш запрос. Аллоцирование подразделяется на 2 случая: когда размер равен свободному блоку, и когда размер меньше размера свободного блока. Во втором случае свободный блок разделяется на два: занятый и остаточный свободный. Также мы добавляем эти два новых блока в глобальные map и удаляем свободный блок до аллоцирования.

Если это запрос на освобождение данных, то первым делом мы проверяем, есть ли такой элемент в числе занятых, так как могут быть ситуации, когда такого элемента нет. Если элемента нет, то ничего не делаем. Далее освобождаем необходимый блок, не забывая посмотреть соседние блоки на возможность объединить свободные блоки друг с другом. Если найден последующий свободный элемент, добавляем его размер к текущему, не меняя начальный индекс. Если найден предыдущий элемент, меняем и начальный индекс и размер блока. Также не забываем удалять старые блоки, до слияния, и добавлять новый блок после слияния. И в конце переводим ссылки соседних элементов на новый – после слияния.

В конце выводим содержимое вектора result. Так как по ходу программы мы, когда нужно, добавляли в него числа, с этим проблем не будет.

Сложность

$$O(M \cdot \log N)$$

Минимум на отрезке

Исходный код

```
#include <iostream>
#include <set>
#include <list>
using namespace std;
int main() {
	multiset<int> main_set;
	list<int> prev_numbers;
	int n, k;
	cin >> n >> k;
	for (int i = 0; i < k; i++) {
		int current_number;
		cin >> current_number;
		main_set.insert(current_number);
		prev_numbers.push_back(current_number);
	}
	cout << *main_set.begin() << " ";
	for (int i = 0; i < n - k; i++) {
		int current_number;
		cin >> current_number;
		main_set.extract(prev_numbers.front());
		prev_numbers.pop_front();
		main_set.insert(current_number);
		prev_numbers.push_back(current_number);
		cout << *main_set.begin() << " ";
	}
	return 0;
}
```

Пояснение к примененному алгоритму

Входные параметры

- N – длина входящей последовательности
- K – длина «окна»
- N элементов – входящая последовательность

Алгоритм решения

Для решения использовался multiset main_set, в который на каждой итерации добавляется входящий элемент. Также используется list prev_numbers, который по сути олицетворяет «окно». Благодаря тому, что в list можно легко и быстро добавлять и удалять в начало и в конец, он идеально подходит для роли «окна». На каждой итерации мы удаляем из main_set элемент prev_numbers.front() потому, что именно этот элемент, последний раз находится внутри «окна». А после удаления добавляем элемент – current_nubmer, который мы только что получили в качестве входных данных. На каждой итерации выводим головной элемент main_set. Так как никаких компораторов нет, именно в начале main_set будет находиться минимум для всего «окна».

Сложность

$$O((N - K) \cdot \log K)$$