2.1) Создание мультисписка на С++

#include <vector>

#include <any>

#include <iostream>

class Multilist {

private:

std::vector<std::vector<std::any>> sublists;

public:

void add\_sublist(const std::vector<std::any>& items) {

sublists.push\_back(items);

}

std::vector<std::any> flatten() {

std::vector<std::any> result;

for (const auto& sublist : sublists) {

for (const auto& item : sublist) {

result.push\_back(item);

}

}

return result;

}

std::vector<std::any> get\_sublist(int index) {

return sublists[index];

}

};

// Использование

Multilist ml;

ml.add\_sublist({1, 2, 3});

ml.add\_sublist({"a", "b"});

2.2) Создание очереди на С++

#include <queue>

#include <stdexcept>

template<typename T>

class Queue {

private:

std::queue<T> queue;

public:

void enqueue(const T& item) {

queue.push(item);

}

T dequeue() {

if (is\_empty()) {

throw std::runtime\_error("Queue is empty");

}

T item = queue.front();

queue.pop();

return item;

}

bool is\_empty() const {

return queue.empty();

}

size\_t size() const {

return queue.size();

}

};

2.3) создание Дек на С++

from collections import deque

class Deque:

def \_\_init\_\_(self):

self.\_deque = deque()

def add\_front(self, item):

self.\_deque.appendleft(item)

def add\_rear(self, item):

self.\_deque.append(item)

def remove\_front(self):

if self.is\_empty():

raise IndexError("Deque is empty")

return self.\_deque.popleft()

def remove\_rear(self):

if self.is\_empty():

raise IndexError("Deque is empty")

return self.\_deque.pop()

def is\_empty(self):

return len(self.\_deque) == 0

def size(self):

return len(self.\_deque)

2.4 создание приоритетной очереди на С++

#include <queue>

#include <vector>

#include <stdexcept>

template<typename T>

class PriorityQueue {

private:

struct Item {

T value;

int priority;

int index;

bool operator>(const Item& other) const {

if (priority == other.priority) {

return index > other.index;

}

return priority > other.priority;

}

};

std::priority\_queue<Item, std::vector<Item>, std::greater<Item>> heap;

int index\_counter = 0;

public:

void push(const T& item, int priority) {

heap.push({item, priority, index\_counter++});

}

T pop() {

if (is\_empty()) {

throw std::runtime\_error("Priority queue is empty");

}

T item = heap.top().value;

heap.pop();

return item;

}

bool is\_empty() const {

return heap.empty();

}

size\_t size() const {

return heap.size();

}

};