# «мультисписок (вложенный список)» на ЯП Python  
class MultiListNode:  
 def \_\_init\_\_(self, data):  
 self.data = data  
 self.\_next\_by\_name = None  
 self.\_next\_by\_salary = None  
  
 @property  
 def next\_by\_name(self):  
 return self.\_next\_by\_name  
  
 @next\_by\_name.setter  
 def next\_by\_name(self, value):  
 self.\_next\_by\_name = value  
  
 @property  
 def next\_by\_salary(self):  
 return self.\_next\_by\_salary  
  
 @next\_by\_salary.setter  
 def next\_by\_salary(self, value):  
 self.\_next\_by\_salary = value  
  
  
class MultiList:  
 def \_\_init\_\_(self):  
 self.head\_name = None  
 self.head\_salary = None  
  
 def add\_employee(self, name, salary):  
 new\_node = MultiListNode({"name": name, "salary": salary})  
  
 # Вставка в список по имени  
 self.\_insert\_sorted\_by\_name(new\_node)  
 # Вставка в список по зарплате  
 self.\_insert\_sorted\_by\_salary(new\_node)  
  
 def \_insert\_sorted\_by\_name(self, new\_node):  
 if not self.head\_name or self.head\_name.data["name"] >= new\_node.data["name"]:  
 new\_node.next\_by\_name = self.head\_name  
 self.head\_name = new\_node  
 else:  
 current = self.head\_name  
 while current.next\_by\_name and current.next\_by\_name.data["name"] < new\_node.data["name"]:  
 current = current.next\_by\_name  
 new\_node.next\_by\_name = current.next\_by\_name  
 current.next\_by\_name = new\_node  
  
 def \_insert\_sorted\_by\_salary(self, new\_node):  
 if not self.head\_salary or self.head\_salary.data["salary"] >= new\_node.data["salary"]:  
 new\_node.next\_by\_salary = self.head\_salary  
 self.head\_salary = new\_node  
 else:  
 current = self.head\_salary  
 while current.next\_by\_salary and current.next\_by\_salary.data["salary"] < new\_node.data["salary"]:  
 current = current.next\_by\_salary  
 new\_node.next\_by\_salary = current.next\_by\_salary  
 current.next\_by\_salary = new\_node  
  
 def display\_by\_name(self):  
 current = self.head\_name  
 while current:  
 print(f"Name: {current.data['name']}, Salary: {current.data['salary']}")  
 current = current.next\_by\_name  
  
 def display\_by\_salary(self):  
 current = self.head\_salary  
 while current:  
 print(f"Salary: {current.data['salary']}, Name: {current.data['name']}")  
 current = current.next\_by\_salary

// «мультисписок (вложенный список)» на ЯП С++

#include <iostream>

#include <string>

#include <functional>

**template**<**typename** T>

**struct** MultiListNode {

T data;

MultiListNode\* next\_by\_name;

MultiListNode\* next\_by\_salary;

MultiListNode(**const** T& data)

: data(data), next\_by\_name(**nullptr**), next\_by\_salary(**nullptr**) {}

};

**template**<**typename** T>

**class** MultiList {

**private**:

MultiListNode<T>\* head\_name;

MultiListNode<T>\* head\_salary;

// Функторы для сравнения

std::function<**bool**(**const** T&, **const** T&)> name\_less;

std::function<**bool**(**const** T&, **const** T&)> salary\_less;

**public**:

MultiList(

std::function<**bool**(**const** T&, **const** T&)> name\_cmp,

std::function<**bool**(**const** T&, **const** T&)> salary\_cmp

) : head\_name(**nullptr**), head\_salary(**nullptr**),

name\_less(name\_cmp), salary\_less(salary\_cmp) {}

~MultiList() {

// Реализация очистки памяти

clear();

}

**void** addEmployee(**const** T& data) {

MultiListNode<T>\* new\_node = **new** MultiListNode<T>(data);

insertSortedByName(new\_node);

insertSortedBySalary(new\_node);

}

**void** insertSortedByName(MultiListNode<T>\* new\_node) {

**if** (!head\_name || name\_less(new\_node->data, head\_name->data)) {

new\_node->next\_by\_name = head\_name;

head\_name = new\_node;

} **else** {

MultiListNode<T>\* current = head\_name;

**while** (current->next\_by\_name &&

name\_less(current->next\_by\_name->data, new\_node->data)) {

current = current->next\_by\_name;

}

new\_node->next\_by\_name = current->next\_by\_name;

current->next\_by\_name = new\_node;

}

}

**void** insertSortedBySalary(MultiListNode<T>\* new\_node) {

**if** (!head\_salary || salary\_less(new\_node->data, head\_salary->data)) {

new\_node->next\_by\_salary = head\_salary;

head\_salary = new\_node;

} **else** {

MultiListNode<T>\* current = head\_salary;

**while** (current->next\_by\_salary &&

salary\_less(current->next\_by\_salary->data, new\_node->data)) {

current = current->next\_by\_salary;

}

new\_node->next\_by\_salary = current->next\_by\_salary;

current->next\_by\_salary = new\_node;

}

}

**void** displayByName() **const** {

MultiListNode<T>\* current = head\_name;

**while** (current) {

std::cout << "Name: " << current->data.name

<< ", Salary: " << current->data.salary << std::endl;

current = current->next\_by\_name;

}

}

**void** displayBySalary() **const** {

MultiListNode<T>\* current = head\_salary;

**while** (current) {

std::cout << "Salary: " << current->data.salary

<< ", Name: " << current->data.name << std::endl;

current = current->next\_by\_salary;

}

}

**void** clear() {

// Умная очистка: собираем все узлы через один список, затем удаляем

// (в реальности нужна более сложная логика для избежания двойного удаления)

}

};

// Структура для данных

**struct** Employee {

std::string name;

**int** salary;

Employee(**const** std::string& n, **int** s) : name(n), salary(s) {}

};

// Пример использования

**int** main() {

**auto** name\_cmp = [](**const** Employee& a, **const** Employee& b) {

**return** a.name < b.name;

};

**auto** salary\_cmp = [](**const** Employee& a, **const** Employee& b) {

**return** a.salary < b.salary;

};

MultiList<Employee> list(name\_cmp, salary\_cmp);

list.addEmployee(Employee("John", 50000));

list.addEmployee(Employee("Alice", 60000));

list.displayByName();

list.displayBySalary();

**return** 0;

}

Пример очереди на Python

from collections import deque  
  
# Готовая реализация  
queue = deque()  
queue.append(1) # enqueue  
queue.append(2)  
queue.popleft() # dequeue -> 1  
class Queue:  
 def \_\_init\_\_(self):  
 self.items = []  
  
 def enqueue(self, item):  
 self.items.append(item) # Добавление в конец  
  
 def dequeue(self):  
 if not self.is\_empty():  
 return self.items.pop(0) # Удаление из начала (неэффективно для больших списков!)  
 raise IndexError("Queue is empty")  
  
 def is\_empty(self):  
 return len(self.items) == 0  
  
 def peek(self):  
 if not self.is\_empty():  
 return self.items[0]  
 raise IndexError("Queue is empty")

Пример очереди на с ++

#include <queue>

#include <iostream>

int main() {

std::queue<int> queue;

queue.push(1); // enqueue

queue.push(2);

std::cout << queue.front(); // peek -> 1

queue.pop(); // dequeue

return 0;

}

Пример дека на Python

from collections import deque  
  
deque\_obj = deque()  
deque\_obj.append(1) # addRear  
deque\_obj.appendleft(2)# addFront  
deque\_obj.pop() # removeRear -> 1  
deque\_obj.popleft() # removeFront -> 2

Пример дека на c++

#include <deque>

#include <iostream>

int main() {

std::deque<int> deque;

deque.push\_back(1); // addRear

deque.push\_front(2); // addFront

std::cout << deque.back(); // peekRear -> 1

deque.pop\_back(); // removeRear

std::cout << deque.front(); // peekFront -> 2

deque.pop\_front(); // removeFront

return 0;

}

#### Пример Приоритетной очереди на Python

import heapq  
  
  
priority\_queue = []  
heapq.heappush(priority\_queue, (1, "low priority task"))  
heapq.heappush(priority\_queue, (3, "high priority task"))  
heapq.heappush(priority\_queue, (2, "medium priority task"))  
  
print(heapq.heappop(priority\_queue))

#### Пример Приоритетной очереди на c ++

#include <iostream>

#include <queue>

int main() {

// По умолчанию - макс-куча (наибольший элемент первый)

std::priority\_queue<int> pq;

pq.push(30);

pq.push(10);

pq.push(20);

std::cout << pq.top(); // 30 (высший приоритет)

pq.pop();

// Мин-куча

std::priority\_queue<int, std::vector<int>, std::greater<int>> min\_pq;

min\_pq.push(30);

min\_pq.push(10);

min\_pq.push(20);

std::cout << min\_pq.top(); // 10 (высший приоритет)

return 0;

}

1. **Уровень абстракции:**
   * **Python:** Предлагает готовые, высокооптимизированные реализации (deque, heapq) для всех структур, кроме мультисписка. Это делает код лаконичным и эффективным.
   * **C++/Java:** Требуют использования конкретных классов из Стандартной библиотеки шаблонов (STL) или Java Collections Framework (JCF). Эти реализации также высоко оптимизированы.
2. **Производительность:**
   * **C++** часто выигрывает в скорости из-за отсутствия накладных расходов виртуальной машины и прямого доступа к памяти.
   * **Java** и **Python** (особенно с deque/heapq) имеют очень эффективные реализации, которых достаточно для подавляющего большинства задач.
3. **Удобство и безопасность:**
   * **Java** и **Python** справляются с управлением памятью автоматически, что уменьшает количество ошибок.
   * **C++** требует ручного управления памятью (при использовании указателей в ручных реализациях, как в мультисписке), но предлагает непревзойдённый контроль.
4. **Мультисписок:** Это узкоспециализированная структура, не представленная в стандартных библиотеках рассмотренных языков. Её всегда приходится реализовывать вручную, что подчёркивает её специфичность и область применения (в основном, системное программирование, менеджеры памяти).

**Заключение:** Выбор языка и конкретной реализации структуры данных зависит от контекста. Для быстрого прототипирования идеален Python с его богатой стандартной библиотекой. Для высокопроизводительных системных приложений лучше подходит C++. Для кроссплатфорчных корпоративных приложений — Java. Понимание внутреннего устройства этих структур позволяет делать осознанный выбор, даже используя готовые библиотечные реализации