БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

СБОРНИК ЗАДАЧ

ПО ТЕОРИИ АЛГОРИТМОВ.

Структуры данных.
Часть. Абстрактные типы данных

МИНСК
2020

УДК 510.51(075.8)
ББК 22.12я73-1
С23

А в т о р ы :
С. А. Соболь, К. Ю. Вильчевский, В. М. Котов,
Е. П. Соболевская

Р е ц е н з е н т ы :
кафедра информатики и методики преподавания информатики
физико-математического факультета Белорусского государственного
педагогического университета им. М. Танка
(заведующий кафедрой, кандидат педагогических наук,
доцент *С. В. Вабищевич*);
профессор кафедры информационных технологий в культуре
Белорусского государственного университета культуры и искусства,
кандидат физико-математических наук, доцент *П. В. Гляков*

ISBN 978-985-518-412-4

Глава 1

Часть 2

АБСТРАКТНЫЕ ТИПЫ ДАННЫХ

Абстрактный тип данных (англ. *abstract data type*) — это математическая модель, где тип данных характеризуется поведением (семантикой) с точки зрения пользователя данных. Для абстрактного типа определяется *интерфейс* — набор операций, которые могут быть выполнены. Пользователь абстрактного типа, используя эти операции, может работать с данными, не вдаваясь во внутренние детали механизма хранения информации.

Реализациями абстрактных типов данных являются конкретные структуры данных. Реализация определяет, как именно представлены в памяти данные и как функционирует та или иная операция. На практике один абстрактный тип обычно может быть реализован несколькими альтернативными способами, которые различаются по времени работы и объёму используемой памяти.

Если алгоритм работает с данными исключительно через интерфейс, то он продолжит функционировать, если одну реализацию интерфейса заменить на другую. В этом и заключается суть абстракции: реализация скрыта за интерфейсом. Такой подход позволяет строить программное обеспечение из отдельных модулей и создавать универсальные алгоритмы.

2.1. СПИСОК

Список (англ. *list*) — это абстрактный тип данных, представляющий собой набор элементов, которые следуют в определённом порядке. Список является компьютерной реализацией математического понятия конечной последовательности.

Реализация абстрактного списка может предусматривать некоторые из следующих операций:

1) создание пустого списка;

2) проверку, является ли список пустым;

3) операцию по добавлению объекта в начало или конец списка;

4) операцию по получению ссылки на первый элемент («голову») или последний элемент («хвост») списка;

5) операцию перехода от одного элемента к следующему или предыдущему элементу;

6) операцию для доступа к элементу по заданному индексу.

Абстрактный тип данных «список» обычно реализуется на практике либо как массив (чаще всего динамический), либо как связный список (односвязный или двусвязный).

В стандартной библиотеке C++ нет специального контейнера, представляющего абстрактный список. Следует использовать либо `std::vector` (динамический массив), либо `std::list` (связный список). В языке Java существует интерфейс `List`, реализациями которого являются классы `ArrayList` (динамический массив) и `LinkedList` (связный список). В программах на Python широко используется тип данных `list`, внутри являющийся динамическим массивом.

2.2. СТЕК

Иногда при работе со списковой структурой возникает потребность включать и исключать элементы в определённом порядке. Если реализуется принцип «последним пришёл — первым вышел» (англ. LIFO — last in first out), то такую структуру данных называют *стеком* (англ. *stack*).

Основными операциями, которые выполняются над стеком, являются (рис. 2.1):

1) Init() — создание пустого стека;

2) IsEmpty() — проверка стека на пустоту; возвращается значение «истина», если стек пуст, и «ложь» в противном случае;

3) Push(x) — добавление элемента x; заданный элемент добавляется *на вершину* стека;

4) Pop() — удаление элемента из стека; выполняется при условии, что стек не пуст, поэтому сначала надо убедиться в этом, а затем — извлечь с вершины стека последний занесённый в него элемент.

Существуют различные способы реализации интерфейса стека. Наиболее распространёнными являются моделирование стека на динамическом массиве и на связном списке. Если наибольшее число элементов,

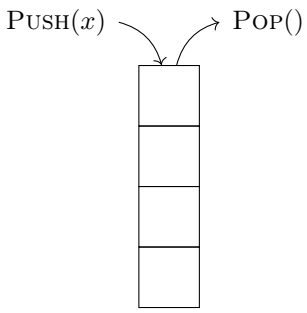

Рис. 2.1. Схема работы стека

которые будут одновременно находиться в стеке, заранее известно, то можно использовать и статический массив.

Все основные операции над стеком выполняются за время O(1).

В стандартной библиотеке C++ доступен контейнер-адаптер `std::stack`, реализующий интерфейс стека. По умолчанию `std::stack` функционирует на базе контейнера `std::deque` (см. раздел 2.4). В языке Java существует класс `Stack`. В языке Python специального класса для создания стека нет, предлагается использовать объект типа `list` (методы `append` и `pop`).

2.3. ОЧЕРЕДЬ

Если при работе со списковой структурой реализуется принцип «первым пришёл — первым вышел» (англ. FIFO — first in first out), то такую структуру данных называют *очередью* (англ. *queue*).

Основными операциям, которые выполняются над очередью, являются (рис. 2.2):

1) Init() — создание пустой очереди;

2) IsEmpty() — проверка очереди на пустоту;

3) Enqueue(x) — добавление элемента x; заданный элемент добавляется *в конец* очереди;

4) Dequeue() — удаление элемента из очереди; элемент удаляется *из начала* очереди; операция выполняется при условии, что очередь не пуста, поэтому сначала надо убедиться в этом, а затем — извлечь элемент.

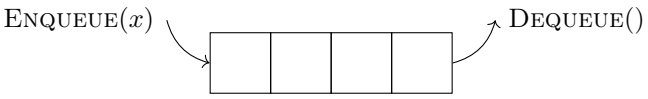

Рис. 2.2. Схема работы очереди

Наиболее простыми способами реализации интерфейса очереди являются моделирование очереди на обычном массиве и на связном списке.

Если очередь моделируется на статическом массиве, то очередь делают *кольцевой* (рис. 2.3). Кроме самого массива, достаточно хранить два индекса: индекс «головы» и индекс «хвоста». Максимальное число элементов очереди будет ограничено размером этого массива. При добавлении элемента сначала нужно проверить, есть ли в массиве свободное место для его размещения.

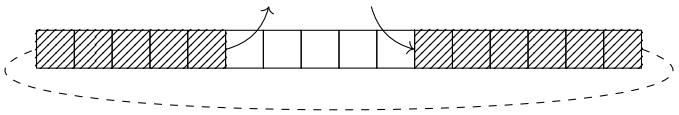

Рис. 2.3. Очередь на базе кольцевого буфера

Динамический массив не подходит для создания очереди, поскольку он, хоть и даёт возможность эффективно добавлять элементы в конец, не позволяет быстро удалять элементы из начала. Однако эту структуру данных можно доработать, предусмотрев наличие зарезервированных, но не занятых ячеек не только в конце, но и в начале блока памяти (рис. 2.4).

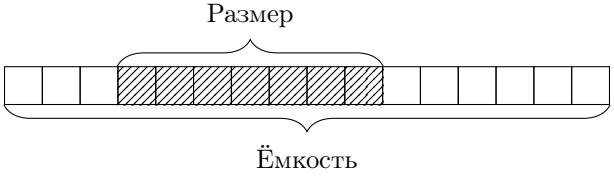

Рис. 2.4. Динамический массив, расширяющийся в обе стороны

Нетрудно понять, что для реализации интерфейса очереди можно также использовать односвязный список, элементы которого следуют в порядке от начала очереди к концу.

Все основные операции над очередью выполняются за время O(1) (при условии, что выбрана подходящая реализация).

В стандартной библиотеке C++ доступен контейнер-адаптер `std::queue`, реализующий интерфейс очереди. По умолчанию он работает на основе контейнера `std::deque` (см. раздел 2.4). В языке Java существует интерфейс `Queue`. В языке Python предлагается использовать более общий контейнер типа `collections.deque`.

2.4. ДВУХСТОРОННЯЯ ОЧЕРЕДЬ

Абстрактный тип данных *двухсторонняя очередь* (англ. *double-ended queue*, или *deque*) — обобщение очереди, где добавление и удаление элементов возможно с обоих концов. Таким образом, интерфейсы стека и очереди являются частным случаем интерфейса двухсторонней очереди.

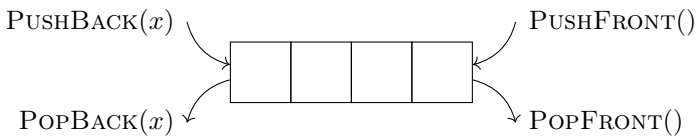

Рис. 2.5. Схема работы двухсторонней очереди

Роль двухсторонней очереди в стандартной библиотеке C++ играет контейнер `std::deque`. Отметим, что этот контейнер обеспечивает доступ к любому элементу по индексу за O(1), как и вектор, но не гарантирует, что все элементы будут лежать в памяти последовательно. В Java есть интерфейс `Deque`, и он реализуется, в частности, классами `ArrayDeque` и `LinkedList`. В языке Python в модуле `collections` предлагается контейнер `deque`.

2.5. ПРИОРИТЕТНАЯ ОЧЕРЕДЬ

Предположим, что для каждого элемента определён некоторый *приоритет*. В простейшем случае значение приоритета может совпадать со значением элемента. В общем случае соотношение элемента и приоритета может быть произвольным.

Приоритетной очередью (англ. *priority queue*) называется такая абстрактная структура данных, интерфейс которой включает в себя следующие операции:

1) PullHighestPriorityElement() — поиск и удаление элемента с самым высоким приоритетом;

2) InsertWithPriority$(x, \text{prior}(x))$ — добавление элемента x с указанным приоритетом.

Интерфейс структуры данных «приоритетная очередь» может быть реализован на основе различных структур данных. Хотя приоритетные очереди часто ассоциируются с кучами, они концептуально отличаются от куч. Приоритетная очередь — это абстрактное понятие. По аналогии с тем, как список может быть реализован с помощью связного списка или массива, приоритетная очередь может быть реализована с помощью кучи или другими способами.

Стек, очередь

Заметим, что стек и очередь могут рассматриваться как частный случай приоритетной очереди.

1. Предположим, что на вход алгоритма поступают заявки и самый высокий приоритет имеет та заявка, которая поступила раньше. В этом случае для моделирования процесса обслуживания заявок интерфейс приоритетной очереди можно реализовать, используя структуру данных очередь (FIFO).

2. Предположим, что на вход алгоритма поступают заявки и самый высокий приоритет имеет та заявка, которая поступила позже. В этом случае для моделирования процесса обслуживания заявок интерфейс приоритетной очереди можно реализовать, используя структуру данных стек (LIFO).

Кучи

Предположим, что на вход алгоритма поступают заявки, но теперь приоритет заявки никак не связан с моментом её поступления. В этом случае для моделирования процесса обслуживания заявок интерфейс приоритетной очереди можно реализовать, используя такую специальную структуру данных, как *куча* (англ. *heap*).

Существуют различные способы реализации структуры данных куча. В [3] подробно рассмотрены следующие реализации: с помощью полного бинарного дерева — *бинарная куча*, с помощью семейства биномиальных деревьев — *биномиальная куча*, с помощью семейства корневых деревьев — *куча Фибоначчи*. В каждой куче вершины деревьев упорядочены в соответствии со свойством неубывающей пирамиды: приоритет предка не ниже приоритета его потомков. На практике в силу своей простоты наиболее часто используется бинарная куча, поэтому на этой специальной структуре данных мы остановимся подробнее в разделе **??**.

Две очереди

Следующий пример демонстрирует реализацию интерфейса приоритетной очереди на двух очередях (FIFO).

На практике хорошо известен алгоритм Э. Дейкстры, который находит во взвешенном графе кратчайшие маршруты между вершиной s и всеми вершинами, достижимыми из неё. Алгоритм работает только

для графов без рёбер отрицательного веса. На каждом шаге алгоритма Дейкстры в качестве текущей вершины выбирается та вершина, которая ещё не просмотрена и расстояние до которой минимально. Значит, для хранения набора вершин можно применить приоритетную очередь, при этом та вершина более приоритетна, расстояние до которой меньше, т. е. в качестве значения приоритета можно использовать минус расстояние до вершины. Подчеркнём, что использование противоположного знака связано с тем, что интерфейс приоритетной очереди выдаёт элемент с максимальным приоритетом, а для алгоритма требуется вершина с минимальным расстоянием.

В зависимости от того, как именно реализована приоритетная очередь (на базе обычного массива, на базе бинарной кучи, на базе кучи Фибоначчи), время работы алгоритма Дейкстры для взвешенного (n, m)-графа будет различным [4]. Оказывается, для графов специального вида можно применить особую реализацию и достичь оптимального времени работы.

Так, если веса рёбер графа $c_{u,v}$ принимают одно их двух значений a или b, то в алгоритме Дейкстры интерфейс приоритетной очереди можно реализовать на двух обычных очередях Q_a и Q_b, в которых хранятся пары (вершина, её приоритет). На каждом шаге будет поддерживаться такое свойство: элементы каждой очереди упорядочены по убыванию приоритетов (в начале очереди стоит самая приоритетная вершина). Алгоритм работает следующим образом.

На начальном этапе стартовую вершину s добавляем в любую из очередей (Q_a или Q_b) с приоритетом 0.

На текущей итерации алгоритма Дейкстры для выбора текущей вершины v возьмём первые элементы каждой из двух очередей и выберем из них тот, приоритет которого больше. Предположим, что это вершина u с приоритетом $-d_u$ (данная вершина среди всех вершин обеих очередей обладает самым высоким приоритетом). Значит, длина кратчайшего маршрута из стартовой вершины s до вершины u равна d_u. Удаляем вершину u из очереди, полагаем её просмотренной и выполняем релаксацию, т. е. добавляем все смежные с ней непросмотренные вершины v в одну из двух очередей (рис. 2.6):

- если $c_{u,v} = a$, то добавляем вершину v с приоритетом $-(d_u + a)$ в очередь Q_a;
- если $c_{u,v} = b$, то добавляем вершину v с приоритетом $-(d_u + b)$ в очередь Q_b.

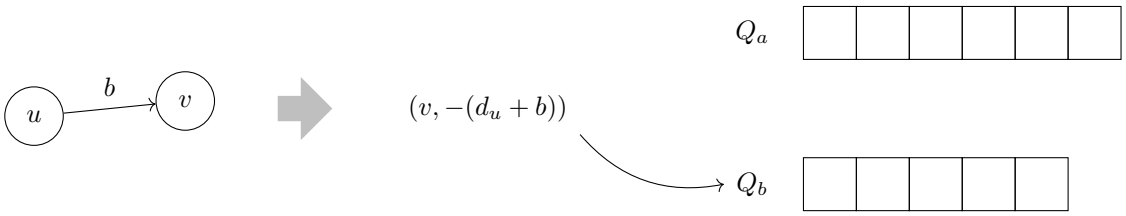

Рис. 2.6. Вершина v добавляется в очередь в зависимости от веса ребра $c_{u,v}$

Алгоритм продолжает свою работу до тех пор, пока обе очереди не станут пустыми. Приоритет, с которым вершина u удаляется из очереди, равен длине кратчайшего маршрута из стартовой вершины s в вершину u, взятой с противоположным знаком.

Так как добавление и удаление элемента из очереди выполняется за время $\mathrm{O}(1)$, а количество элементов этих очередей ограничено m, то время работы алгоритма Дейкстры для заданного графа есть $\mathrm{O}(n+m)$.

Двухсторонняя очередь

Если в графе веса рёбер графа принимают одно их двух значений 0 или 1, то в алгоритме Дейкстры интерфейс приоритетной очереди можно реализовать на структуре данных *двухсторонняя очередь* (англ. *deque* — double ended queue). Для структуры данных двухстороняя очередь добавление и удаление элементов допустимо как с начала очереди, так и с конца. Рассуждая аналогично предыдущему случаю, при релаксации по ребру веса 0 добавляем элемент в начало двухсторонней очереди, а при релаксации по ребру веса 1 — в конец. В качестве текущего элемента на итерациях алгоритма Дейкстры всегда выбирается первый элемент очереди, а алгоритм продолжает свою работу, пока двухсторонняя очередь не станет пустой. Так как добавление и удаление элемента из двухсторонней очереди выполняется за $\mathrm{O}(1)$, а количество элементов очереди ограничено m, то время работы алгоритма Дейкстры для даного класса графов равно $\mathrm{O}(n+m)$.

В стандартной библиотеке C++ предоставляется контейнер-адаптер под названием `std::priority_queue`, представляющий приоритетную очередь, основанную на бинарной куче. В языке Java существует класс `PriorityQueue`, также содержащий внутри бинарную кучу. В языке Python нет абстрактного интерфейса приоритетной очереди, есть лишь модуль `heapq`, в котором реализована бинарная куча.

2.6. МНОЖЕСТВО

Множество (англ. *set*) — абстрактная структура данных, которая хранит набор попарно различных объектов без определённого порядка. Эта структура данных является компьютерной реализацией математического понятия конечного множества. Подчеркнём важные отличия множества от списка:

- в списке одинаковые элементы могут храниться несколько раз, а в множестве все элементы уникальны;
- в списке порядок следования элементов сохраняется, а в множестве — нет.

Интерфейс множества включает три основные операции:

1) INSERT(x) — добавить в множество ключ x;
2) CONTAINS(x) — проверить, содержится ли в множестве ключ x;
3) REMOVE(x) — удалить ключ x из множества.

Для реализации интерфейса множества обычно используются такие структуры данных:

- сбалансированные поисковые деревья: например, красно-чёрные деревья [1], AVL-деревья [3], 2-3-деревья [3], декартовы деревья (см. раздел **??**);
- хеш-таблицы (см. часть **??**).

В стандартной библиотеке C++ есть контейнер `std::set`, который реализует множество на основе сбалансированного дерева (обычно красно-чёрного), и контейнер `std::unordered_set`, построенный на базе хеш-таблицы. В языке Java определён интерфейс `Set`, у которого есть несколько реализаций, среди которых классы `TreeSet` (работает на основе красно-чёрного дерева) и `HashSet` (на основе хеш-таблицы). В языке Python есть только встроенный тип `set`, использующий хеширование, но нет готового класса множества, построенного на сбалансированных деревьях.

2.7. АССОЦИАТИВНЫЙ МАССИВ

Ассоциативный массив (англ. *associative array*), или *отображение* (англ. *map*), или *словарь* (англ. *dictionary*), — абстрактная структура данных, которая хранит пары вида (ключ, значение), при этом каждый ключ встречается не более одного раза.

Название «ассоциативный» происходит от того, что значения ассоциируются с ключами. Такой массив удобно рассматривать как обычный массив, в котором в качестве индексов можно использовать не только целые числа, но и значения других типов (например строки).

Интерфейс ассоциативного массива включает операции:

1) INSERT(k, v) — добавить пару, состоящую из ключа k и значения v;

2) FIND(k) — найти значение, ассоциированное с ключом k, или сообщить, что значения, связанного с заданным ключом, нет;

3) REMOVE(k) — удалить пару, ключ в которой равен k.

Данный интерфейс реализуется на практике теми же способами, что и интерфейс множества. Реализация ассоциативного массива технически немного сложнее, чем множества, но использует те же идеи.

Для языка программирования C++ в стандартной библиотеке доступен контейнер `std::map`, работающий на основе сбалансированного дерева (обычно красно-чёрного), и контейнер `std::unordered_map`, работающий на основе хеш-таблицы. В языке Java определён интерфейс `Map`, который реализуется несколькими классами, в частности классом `TreeMap` (базируется на красно-чёрном дереве) и `HashMap` (базируется на хеш-таблице). В языке Python очень широко используется встроенный тип `dict`. Этот словарь использует внутри хеширование.

БИБЛИОГРАФИЧЕСКИЕ ССЫЛКИ

1. Алгоритмы: построение и анализ / Т. Кормен [и др.]. — М. : Вильямс, 2005. — 1296 с.

2. *Котов В. М.*, *Мельников О. И.* Информатика. Методы алгоритмизации : учеб. пособие для 10 – 11 кл. общеобразоват. шк. с углубл. изучением информатики. — Минск : Нар. асвета, 2000. — 221 с.

3. *Котов В. М.*, *Соболевская Е. П.*, *Толстиков А. А.* Алгоритмы и структуры данных : учеб. пособие. — Минск : БГУ, 2011. — 267 с. — (Классическое университетское издание).

4. Сборник задач по теории алгоритмов : учеб.-метод. пособие/ В. М. Котов [и др.]. — Минск : БГУ, 2017. — 183 с.

5. Теория алгоритмов : учеб. пособие / П. А. Иржавский [и др.]. — Минск : БГУ, 2013. — 159 с.

6. *Соболь С. А.*, *Котов В. М.*, *Соболевская Е. П.* Опыт использования образовательной платформы Insight Runner на факультете прикладной математики и информатики Белорусского государственного университета // Роль университетского образования и науки в современном обществе : материалы междунар. науч. конф., Минск, 26–27 февр. 2019 г. / Белорус. гос. ун-т ; редкол.: А. Д. Король (пред.) [и др.]. — Минск : БГУ, 2019. — С. 263-267.

7. *Соболь С. А.*, *Котов В. М.*, *Соболевская Е. П.* Методика преподавания дисциплин по теории алгоритмов с использованием образовательной платформы iRunner // Судьбы классического университета: национальный контекст и мировые тренды [Электронный ресурс] : материалы XIII Респ. междисциплинар. науч.-теорет. семинара «Инновационные стратегии в современной социальной философии» и междисциплинар. летней школы молодых ученых «Экология культуры», Минск, 9 апр. 2019 г. / Белорус. гос. ун-т ; сост.: В. В. Анохина, В. С. Сайганова ; редкол.: А. И. Зеленков (отв. ред.) [и др.] — С. 346–355.

СОДЕРЖАНИЕ

Учебное издание

Соболь Сергей Александрович
Вильчевский Константин Юрьевич
Котов Владимир Михайлович и др.

СБОРНИК ЗАДАЧ ПО ТЕОРИИ АЛГОРИТМОВ. СТРУКТУРЫ ДАННЫХ

Учебно-методическое пособие

Редактор *Х. Х. XXXXXXX*
Художник обложки *С. А. Соболь*
Технический редактор *Х. Х. XXXXXXX*
Компьютерная вёрстка *С. А. Соболя*
Корректор *Х. Х. XXXXXX*

Подписано в печать 29.02.2020. Формат 60×84/16. Бумага офсетная.
Печать офсетная. Усл. печ. л. 10,69. Уч.-изд. л. 9,6.
Тираж 150 экз. Заказ

Белорусский государственный университет.
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/270 от 03.04.2014.
Пр. Независимости, 4, 220030, Минск.

Республиканское унитарное предприятие
«Издательский центр Белорусского государственного университета».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №2/63 от 19.03.2014.
Ул. Красноармейская, 6, 220030, Минск.