Лекция 4

Раздел «Структуры данных»

СТЕК, ОЧЕРЕДЬ, ДЕК
АЛГОРИТМ СОРТИРОВОЧНОЙ СТАНЦИИ

ВВЕДЕНИЕ

На прошлом занятии были рассмотрены последовательные структуры данных *вектор* и *связные списки*, на основе которых строятся более сложные структуры. На данном занятии будут рассмотрены структуры данных *стек, очередь* и *дек*, а также способы их реализации через адаптацию интерфейсов базовых структур.

Данные структуры (стек, очередь) используются в классическом алгоритме разбора математических выражений — алгоритме сортировочной станции (Shunting Yard Algorithm).

1. Стек

Стек (Stack) — структура данных, представляющая собой набор данных, доступ к которым реализуется по принципу «последний пришел — первый ушел» (LIFO). До первого элемента в стеке можно добраться только после того, как будут последовательно сняты позже добавленные элементы.

Вершина стека — последний добавленный в стек элемент.

! Не стоит путать с областью памяти «стек»: она так называется из-за принципа хранения объектов в памяти (можно проверить простым выводом адресов автоматических объектов).

Примеры из реальной жизни: стопка книг/тарелок/etc.

Интерфейс структуры (возможные операции со структурой):

1) `push` — положить элемент на вершину стека;

2) `pop` — снять элемент с вершины;

3) `top` — посмотреть элемент на вершине.

Также могут реализовываться дополнительные методы: `size`, `isEmpty`.

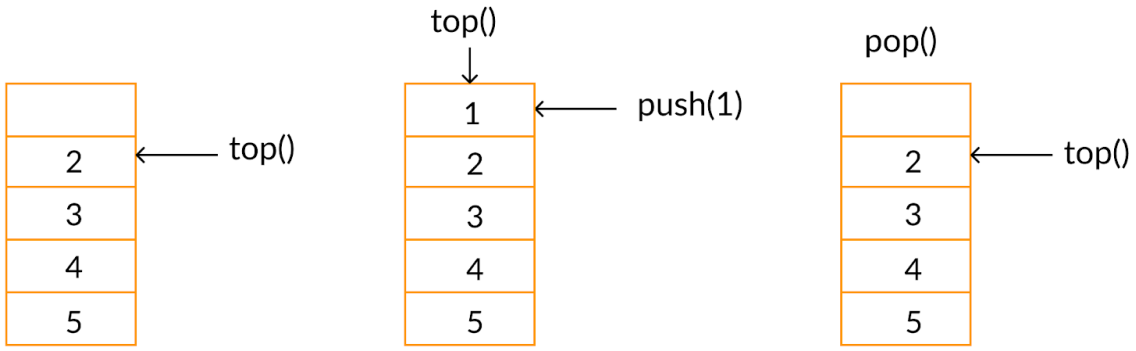

Рисунок 1 — Иллюстрация работы со стеком

Принцип LIFO часто используется в программировании и моделировании: системы массового обслуживания, алгоритмы на графах (поиск в ширину), алгоритмы разбора файлов, трансляторы и т.д..

Стек обычно строится на векторе, односвязном списке или деке (будет рассмотрен далее). По своей сути стек является *адаптером*: методы стека реализуются через методы базовой структуры, поведение ограничивается рассмотренными выше операциями. Сложность операций в таком случае совпадает со сложностями соответствующих операций базовой структуры. Адаптируемая структура выбирается в зависимости от задач, которые будут стоять перед вектором.

В таблице представлены операции вектора и односвязного списка, соответствующие операциям стека.

Stack	Vector	LinkedList
push	Добавление в конец [amort(O(1))]	Добавление в начало [O(1)]
pop	Удаление с конца [amort(O(1))]	Удаление из начала [O(1)]
top	Посмотреть нулевой элемент [O(1)]	Посмотреть значение head [O(1)]

! Если рассматривать асимптотические сложности, можно заметить, что в теории список всегда предпочтительнее вектора. Однако в реальности список оперирует объектами, расположенными в динамической памяти, что всегда медленнее, чем операции с объектами, расположенными на стеке. Потому очевидного предпочтения какой-либо структуре нет.

Вам самостоятельно необходимо будет реализовать класс стека с вариативными базовыми структурами. Здесь будет описан подход, которым необходимо воспользоваться при реализации.

Для реализации будет использован шаблон проектирования «Bridge» с помощью идиомы «Pimpl».

Суть паттерна «Bridge» — разделение классов на две отдельные иерархии — иерархию абстракции и иерархию реализации. Благодаря такому разделению, можно изменять иерархии независимо друг от друга. Шаблон используется, когда нужно иметь несколько версий одного класса в программе.

В нашем случае абстракцией будет являться класс стека, а реализациями — конкретные реализации на базовых структурах. Это позволит нам в будущем удобно добавлять новые способы реализации стека без изменения класса самого стека. Клиентский код будет видеть только абстракцию (стек), реализация же будет развиваться в иерархии реализации.

Реализовывается данный паттерн с помощью композиции объекта иерархии реализации в классе абстракции.

Идиома Pimpl (Pointer to Implementation) реализует композицию так, чтобы реализация была делегирована объекту-части, причем таким образом, чтобы в хедере «целого» не было включений файлов с конкретными реализациями: это позволяет увеличить скорость компиляции, а также скрыть от клиента особенности реализации. Отрицательное качество данной идиомы — ухудшение производительности за счет обращения к объекту на куче.

Иерархия абстракции будет представлена всего одним классом — стеком (`Stack`). Иерархия реализации будет содержать интерфейс `StackImplementation`, содержащий чистые виртуальные функции стека, которые будут реализовывать классы `StackVector` и `StackList`. `Stack` в приватной секции будет содержать указатель на `StackImplementation`, а реализация стека будет просто дергать методы `StackImplementation`.

При таком подходе, для добавления новой реализации нужно будет всего лишь добавить новое условие в конструктор класса `Stack` и для нового контейнера реализовать интерфейс `StackImplementation`.

Структура классов представлена в репозитории https://github.com/MaximGolubev/DataStructures.

2. Очередь

Очередь (Queue) — структура данных, представляющая собой набор данных, доступ к которым реализуется по принципу «первый пришел — первый ушел» (FIFO). До последнего элемента в очереди можно добраться только после того, как будут последовательно сняты ранее добавленные элементы.

Голова очереди (head, front) — первый элемент очереди.

Хвост очереди (tail, back) — последний элемент очереди.

Пример из жизни: очередь в магазине.

Интерфейс структуры (возможные операции со структурой):

1) `enqueue` — поставить элемент в очередь;
2) `dequeue` — достать первый элемент из очереди;
3) `front` — посмотреть элемент в голове очереди.

Также могут реализовываться дополнительные методы: `size`, `isEmpty`.

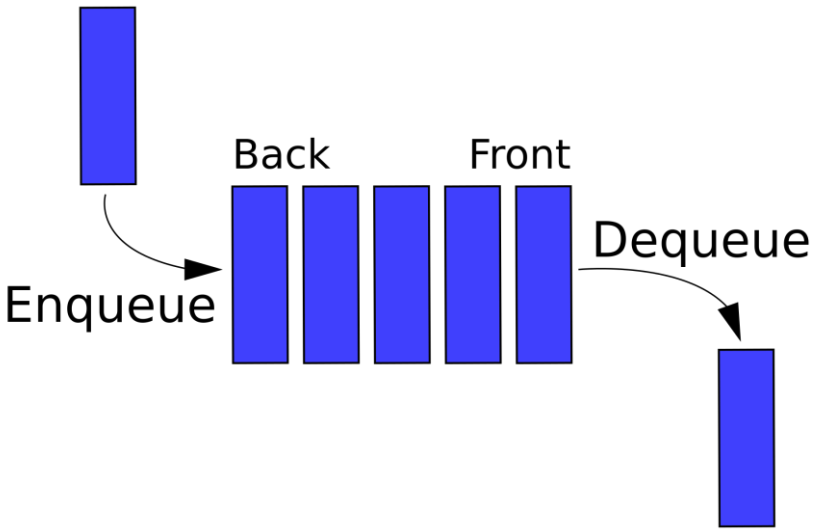

Рисунок 2 — Иллюстрация операций очереди

Очереди так же часто используется в программировании и моделировании: системы массового обслуживания, алгоритмы на графах (поиск в глубину), задачи радиолокации (поступление сигналов от локатора, обработка лучей радиолокационного изображения, обмен сообщениями между программными комплексами…) и т.д..

Очередь можно реализовать множеством способов: на векторе, на массиве (для очереди ограниченного размера), односвязном списке, двусвязном списке, деке, двух стеках (используется, когда нужно быстро получать динамические максимумы и минимумы в очереди). По своей сути очередь так же является *адаптером*.

В таблице представлены операции вектора, односвязного списка и двусвязного списка, соответствующие операциям очереди.

Queue	Vector	LinkedList	DoubleLinkedList
enqueue	Добавление в конец [amort(O(1))]	Добавление в конец, до которого нужно пробежать из самого начала [O(N)]	Добавление в конец [O(1)]
dequeue	Фиктивное удаление с начала: создание промежуточного индекса «головы», при удалении инкрементируем счетчик. В какой-то момент возможно реаллоцировать вектор, чтобы не жрать много памяти [amort(O(1))]. При реальном удалении с начала сложность O(N)	Удаление первого узла [O(1)]	Удаление первого узла [O(1)]
front	Посмотреть нулевой элемент [O(1)]	Посмотреть значение head [O(1)]	Посмотреть значение head [O(1)]

Самостоятельно необходимо будет реализовать очередь по аналогии со стеком.

3. Дек

Дек (Deque, двухсторонняя очередь) — обобщение очереди и стека.

Интерфейс структуры (возможные операции со структурой):

1) `pushFront` — добавить элемент в начало;

2) `popFront` — достать первый элемент;

3) `pushBack` — добавить элемент в конец;

4) `popBack` — достать последний элемент;

5) `back` – посмотреть элемент в хвосте;

6) `front` — посмотреть элемент в голове.

Также могут реализовываться дополнительные методы: `size`, `isEmpty`, итераторы, доступ к элементам по индексу (в некоторых реализациях).

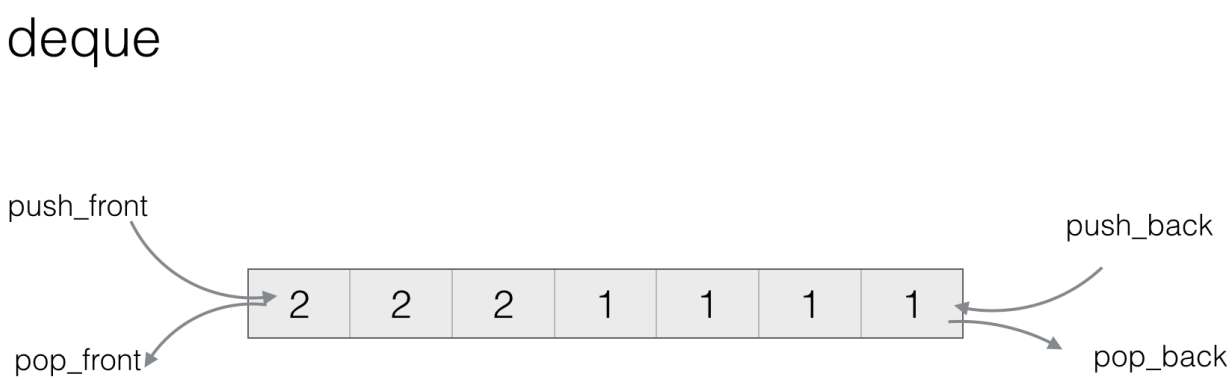

Рисунок 3 — Операции над двунаправленной очередью

Популярные реализации деки достаточно удобно и часто применяются, так как структура получается вполне себе универсальной.

Классическая реализация — двусвязные списки, реализация в STL — Chunked Vector (список массивов фиксированного размера).

Заголовочный файл в стандартной библиотеке — deque.

Вы можете самостоятельно реализовать дек, отдельно время посвящать этому мы не будем: реализация либо типична и тривиальна, либо вы реализуете Chunked Vector — это чуть более интересно, но долго.

4. Алгоритм сортировочной станции

Алгоритм сортировочной станции — алгоритм разбора математических выражений, представленных в инфиксной записи. Выход — очередь токенов, из которой можно получить: выражение в постфиксной нотации [обратной польской нотации (RPN)], абстрактное синтаксическое дерево выражения (AST).

Инфиксная запись — как привыкли люди. «2 * (1 + 3) / 4»

Постфиксная запись — как привыкли машины. «2 1 3 + * 4 /»

Очевидный плюс — отсутствие скобок.

Абстрактное синтаксическое дерево — представление выражение в виде дерева (это на следующей неделе).

Токен — логически неделимая последовательность символов в выражении.

Токенизация (лексический анализ) — получение из входной последовательности токенов.

В выражении «2.5 * (1 + 3) / 4 + sin(10)»:

- 2.5, 1, 3, 4, 10 — токены, тип «число»;
- (,), (,) — токены, тип «скобка»;
- *, +, /, + — токены, тип «операция»;
- sin — токен, тип «функция».

С более точными и подробными понятиями лексического анализа вы ознакомитесь в курсе «Проектирование трансляторов».

В ходе работы алгоритм сортировочной станции использует следующие конструкции:

- стек операторов;
- результирующая очередь (очередь вывода).

Для использования алгоритма нужно определить некоторые свойства операторов:

1) Приоритет оператора

Свойство, которое определяет порядок применения оператора. Чем выше приоритет, тем раньше применяется оператор (к примеру, сложение, вычитание — 1, умножение, деление — 2, возведение в степень — 3).

2) Ассоциативность оператора

Свойство, определяющее правила применения оператора к операндам (вот и линал пригодился).

Ассоциативность: 2 + 3 + 4 = (2 + 3) + 4 = 2 + (3 + 4)

Левая ассоциативность: 2 – 4 – 6 = (2 – 4) – 6 != 2 – (4 – 6)

Правая ассоциативность: 2 ^ 3 ^ 4 = 2 ^ (3 ^ 4) != (2 ^ 3) ^ 4

3) Количество операндов

Сколько операндов нужно оператору (также относится и к функциям: количество аргументов функции).

Бинарные операторы: 2 – 3, 2 + 3, 2 * 3 …

Унарные операторы: -1, ~5, …

Эти свойства заранее задаются операторам еще до начала разбора в зависимости от вашей аксиоматики.

Непосредственно алгоритм разбора:

Алгоритм принимает на вход последовательность токенов.

```
1) Последовательный проход по токенам:
   1.1) ЕСЛИ токен — число, то добавить его в очередь вывода;
   1.2) ЕСЛИ токен — функция, то добавить его в стек операторов;
   1.3) ЕСЛИ токен — оператор, то:
        а) ПОКА (на вершине стека операторов функция
                ИЛИ оператор с большим приоритетом, чем у токена
                ИЛИ (оператор с тем же приоритетом, что у токена,
                     И токен — левоассоциативный оператор)
                И оператор на вершине стека операторов не '(':
                    Достать оператор из стека и переложить в
```

```
                очередь;
        б) положить токен на стек операторов;
    1.4) ЕСЛИ токен — '(', то добавить его в стек операторов;
    1.5) ЕСЛИ токен — ')', то:
        ПОКА оператор на вершине стека не '(':
            Достать оператор из стека и переложить в очередь;
        ЕСЛИ стек операторов пуст, ТО:
            Скобки не сбалансированы,
            конец разбора с ошибкой.
        Достать '(' с вершины стека операторов;
2) ПОКА стек операторов не пуст:
    ЕСЛИ на вершине стека скобка, ТО:
        Скобки не сбалансированы,
        конец разбора с ошибкой.
    Достать оператор из стека и переложить в очередь.
```

Иллюстрация работы алгоритма представлена на рис. 3.

Результат разбора — очередь токенов. Если последовательно вывести значения токенов, то получается постфиксная запись выражения. Также, используя очередь, можно посчитать значение исходного выражения.

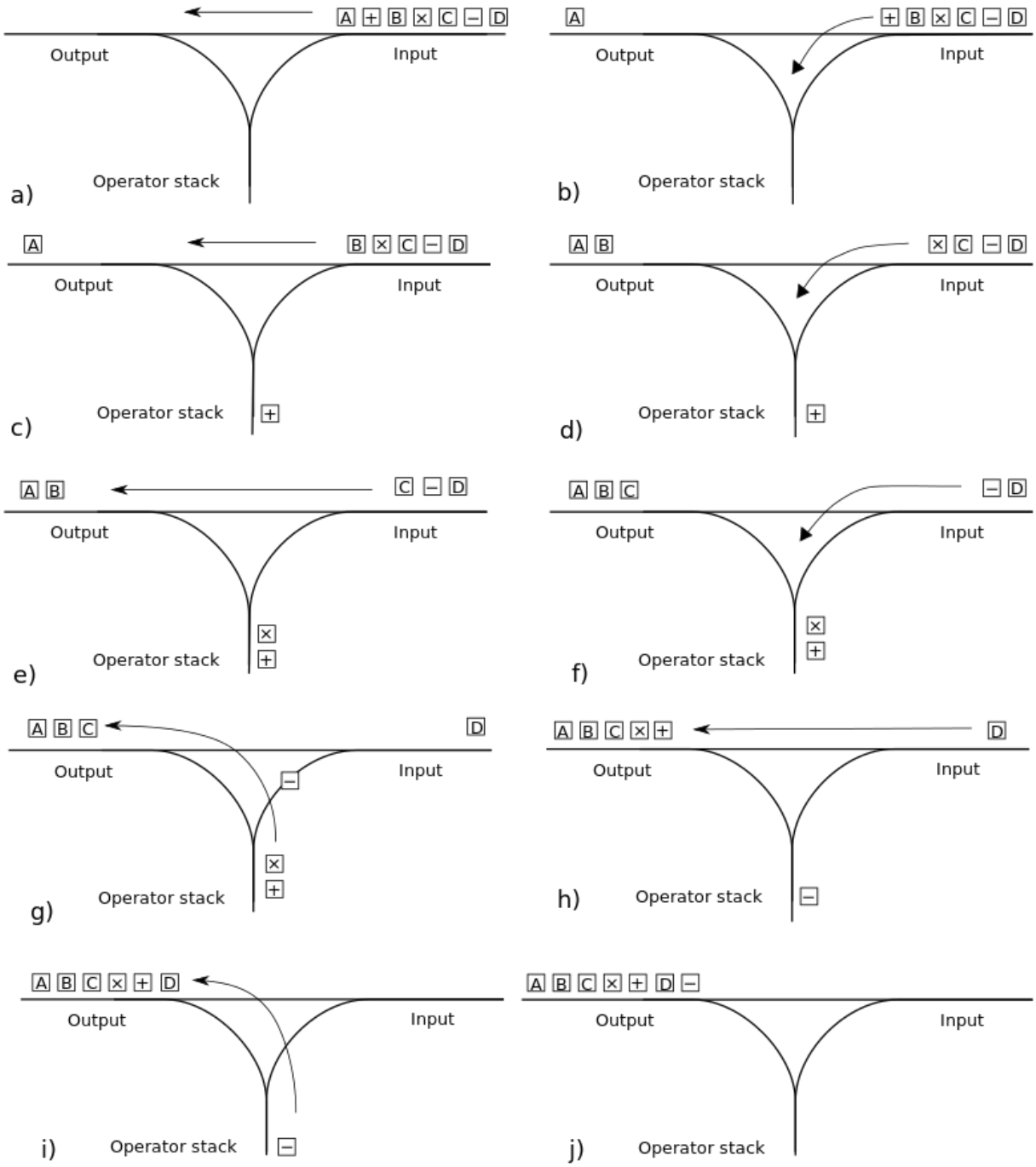

Рисунок 4 — Алгоритм сортировочной станции

Алгоритм подсчета значения выражения:

Вход алгоритма — очередь токенов, полученная после разбора выражения.

Алгоритм использует стек значений как вспомогательную конструкцию.

```
1) ПОКА очередь не пуста:
       Снять токен с очереди
       ЕСЛИ токен — число, ТО положить его на стек
       ЕСЛИ токен — оператор, ТО:
           Снять со стека нужное оператору кол-во операндов
           Применить оператор к операндам в порядке снятия
2) Единственный элемент в стеке — результат.
```

Обобщенный алгоритм получения значения выражения:

```
1)  tokens := tokenize(expression)
2)  outputQueue := shuntingYard(tokens)
3)  result := calc(outputQueue)
```

Что делать с унарным минусом?

Если минус стоит у числа, то на этапе токенизации можно сразу считать отрицательное число.

Если минус стоит у функции, то можно преобразовать подстроку *–f(x)* к виду *–1 * f(x)*.

Для реализации данных алгоритмов на C++ можно использовать различные подходы. Я предлагаю создавать иерархию наследования токенов: создать класс абстрактного токена, конкретизировать его токенами для числа, оператора, скобки, функции. Используя полиморфизм подтипов, возможно будет работать с токенами разных типов в контейнерах, хранящих указатели на абстрактный токен.

На выходе токенизации — вектор указателей на абстрактный токен.

На выходе разбора — очередь указателей на абстрактный токен.

На выходе калькулятора — число типа double.

Самостоятельно необходимо будет реализовать класс, реализующий перечисленные выше алгоритмы для выражений, содержащих вещественные числа и операции «+, -, *, /, ^», а также скобки «(,)».

ИТОГИ

1) Понимание устройства таких классических структур данных, как стек, очередь, дек позволят вам решать более сложные и интересные задачи;
2) Понимание устройства вектора и списков позволят вам понимать устройство стека, очереди и деки;

На этом мы заканчиваем рассмотрение последовательных структур данных.

Самостоятельно необходимо реализовать стек и очередь, как это описано в п. 1, а также алгоритм сортировочной станции по заданию из п. 4.

Дедлайн по стеку и очереди: 19.04.2020, 20:00.

Дедлайн по сортировочной станции: 21.04.2020, 20:00.

По всем вопросам:

TG: MaximGolubev

Vk: https://vk.com/club186630936

ИСТОЧНИКИ

1. Идиома Pimpl https://habr.com/ru/post/111602/
2. Паттерн «Мост» https://refactoring.guru/ru/design-patterns/bridge
3. Алгоритм сортировочной станции https://en.wikipedia.org/wiki/Shunting-yard_algorithm