Линейные списки: стеки, очереди, деки (динамические хранилища данных)

Лекция 3

Немного «истории»

При изучении указателей познакомить с простейшей динамической схемой - **динамическим массивом**. Который определяли примерно таким образом:

```
short* p = new short[n]
```

задавали указатель **`* p,`** далее с помощью оператора **`new`** выделяли **`n`** элементов этого массива.

Почему этот массив называется динамическим? Дело в том, что число его элементов зависит от значений от этой переменной **`n`**, а не задается просто какой-то константой, как в обычном массиве.

Сегодня поговорим о чуть более сложных способа организации данных. Точнее, частично, мы про них начали разговор на прошлой лекции – это были односвязные и двусвязные **списки**. Так вот, на базе этих динамических хранилищ данных, можно организовывать ещё более специфические хранилища.

Вспомним. «Стрелочка» в языке программирования C++ - это оператор косвенного обращения. Он используется для доступа к полям структуры или объекта через указатель на него. Например, если у вас есть указатель a и поле b, то вы можете обратиться к полю b с помощью стрелочки, написав a->b. Это то же самое, что написать (*a).b, но в более короткой и удобной форме.

Пример:

```
struct Point
{
int X, Y;
};
…
Point * point = new Point2;
point->X = 1; // В поле X структуры Point2 сейчас лежит 1
point->Y = 2; // В поле Y структуры Point2 сейчас лежит 2
...
(*point).X = 3; // В поле X структуры Point2 сейчас лежит 3
…
```

Итого

Оператор **Dot (.)** Используется для обычного доступа к членам структуры или объединения.

Оператор **Arrow (->)** существует для доступа к членам структуры или объединениям с помощью указателей.

В языке программирования C++ есть несколько операторов, которые используются для доступа к различным элементам классов и объектов:

- Двоеточие (:) используется для доступа к статическим переменным и функциям класса. Статические переменные и функции принадлежат классу в целом, а не отдельным объектам класса.
- Точка (.) используется для доступа к переменным членам и методам объекта класса. Это означает, что вы можете обратиться к переменным и функциям объекта, используя его имя и точку.
- Стрелочка (->) используется для доступа к переменным членам и методам объекта класса через указатель на объект. Это означает, что вы можете обратиться к переменным и функциям объекта, используя указатель на объект и

Вот пример использования этих операторов:

```
class MyClass {
public:
    int x;
    static int y;
    void print() {
        cout << x << " " << this->y <<
endl;
    }
};

int MyClass::y; // Объявление
//статической переменной

int main() {
    MyClass obj;
    obj.x = 5;
    obj.y = 10;
    obj.print();

    MyClass* ptr = &obj;
    ptr->x = 10;
    ptr->y = 20;
    ptr->print();

    obj.y = 30; //обращение к
//статической переменной
    obj.print();
    ptr->print();

    return 0;
}
```

```
class MyClass {
public:
    int x;
    static int y;

 void print() {
        cout << x << " " << this->y << endl;}
};

int MyClass::y;

int main() {
    MyClass obj;
      obj.x = 5;
     obj.y = 10;
     obj.print();

    MyClass* ptr = &obj;
     ptr->x = 10;
     ptr->y = 20;
     ptr->print();

     obj.y = 30;
     obj.print();
     ptr->print();

    return 0;
}
```

В этом примере мы создали класс **MyClass** с переменной **x** и статической переменной **y**.

Потом создали объект **obj** и указатель ***ptr** на этот объект.

Далее, использовали точку **(.)** для доступа к переменным и функциям объекта **obj**, а стрелочку **(->)** для доступа к переменным и функциям объекта через указатель **ptr**.

Мы также использовали двоеточие **(:)** для доступа к статической переменной **y** класса **MyClass**.

Результат работы программы:

```
5 10
10 20
10 30
10 30
```

Доступ к статическим переменным и функциям выполняется через двоеточие.
Доступ к переменным членам и методам : через точку объект класса
Доступ через -> указатель на объект класса

Дано:

```
class Sample
{
public:
int m_val; // переменная-член класса
static int s_val; // статическая переменная класса (не переменная-член класса!)

int func1(); // функция-член, или метод класса
static int func2(); // статическая функция, а не функция-член и не метод класса
};
```

//обращение к статическим переменным	// обращение к объекту класса	// Обращение через указатель на объект класса
Sample::s_val; **Sample**::func2();	**Sample** obj; obj.m_val; obj.func1();	**Sample** *pobj = &obj; pobj->m_val; pobj->func1();

Объединение (**union**) в языке программирования C++ - это специальный тип данных, который позволяет хранить **несколько различных типов данных в одном и том же блоке памяти**. Объединение используется для экономии памяти, когда несколько переменных используются вместе, но не одновременно.

Объединение определяется с помощью ключевого слова "**union**", за которым следует имя объединения и список членов объединения, разделенных запятыми. Каждый член объединения может быть разным типом данных.

Пример определения объединения:

```
union MyUnion {
    int x;
    double y;
    char z; };
```

В этом примере мы определили объединение **MyUnion** с тремя членами: **x** типа **int**, **y** типа **double** и **z** типа **char**.

Чтобы использовать объединение, нужно создать переменную этого типа и инициализировать один из членов объединения. **При инициализации одного члена, другие члены объединения автоматически сбрасываются в неопределенное состояние.**

Пример использования объединения:

```
MyUnion myUnion;
myUnion.x = 5;
cout << myUnion.x << endl; // выведет 5

myUnion.y = 10.5;
cout << myUnion.y << endl; // выведет 10.5

myUnion.z = 'A';
cout << myUnion.z << endl; // выведет A
```

В этом примере мы создали переменную **myUnion** типа **MyUnion** и инициализировали ее членами **x**, **y** и **z**.

Однако, если вы попытаетесь обратиться к другому члену объединения **после инициализации одного из них**, вы получите неопределенное значение, так как при инициализации одного члена, другие участники объединения **автоматически сбрасываются в неопределенное состояние**.

Также стоит отметить, что объединения не имеют конструкторов и деструкторов, поэтому они не могут содержать члены-объекты, которые требуют вызова конструктора или деструктора.

Не смотря на то, что объединение **(union)** и структура **(struct)** в языке программирования C++ схожи по обращению - это **два разных типа данных**, **которые используются для группировки нескольких переменных разных типов вместе**.

Ключевые отличия между объединением **(union)** и структурой **(struct)**:

1. Использование памяти: **Объединение использует один и тот же блок памяти для всех своих членов**, в то время как **структура** использует **отдельные блоки памяти для каждого члена**. Это означает, что объединение может быть эффективнее, когда нужно хранить несколько переменных в одном блоке памяти, но не одновременно.

2. Доступ к членам: В **структуре все члены доступны одновременно**, в то время как в **объединении только один член может быть доступен «одномоментно»**.

3. Конструкторы и деструкторы: **Структуры могут иметь конструкторы** и **деструкторы**, в то время как **объединения не могут**. Это означает, что структуры могут содержать члены-объекты, которые требуют вызова конструктора или деструктора, в то время как объединения не могут.

4. Наследование: **Структуры могут наследоваться** от других структур или классов, в то время как **объединения не могут**.

В целом, объединение используется для экономии памяти, когда несколько переменных используются вместе, но не одновременно, в то время как

Пример 1

```
// C программа для показа оператора
//Arrow используется в Структуре

#include <stdio.h>
#include <stdlib.h>

// Создание структуры
struct student {
    char name[80];
    int age;
    float percentage;
};
// Создание объекта структуры
struct student* emp = NULL;

int main()
{
    // Назначение памяти для структурной
переменной emp
    emp = (struct
student*)malloc(sizeof(struct student));

// Присваивание значения переменной возраста
// из emp используя оператор стрелки
    emp->age = 18;

// Печать присвоенного значения переменной
    printf("%d", emp->age);
    return 0; }
```

Вывод на экран: 18

Пример 2

```
// C программа для показа оператора Arrow
// используется в Объединении (Союзе)

#include <stdio.h>
#include <stdlib.h>

// Создание Объединения
union student {
    char name[80];
    int age;
    float percentage;
};

// Создание объекта объединения
union student* emp = NULL;

int main()
{
    // Назначение памяти для структурной
переменной emp
    emp = (union student*)malloc(sizeof(union
student));

    // Присваивание значения переменной возраста
    // из emp используя оператор стрелки
    emp->age = 18;

    // Воспроизведение назначенного значения
переменной
    printf("%d", emp->age); }
```

Вывод на экран: 18

Линейный список

Линейный список - это один из основных типов данных в программировании, который используется для хранения последовательности элементов. Линейный список представляет собой набор элементов, которые связаны между собой с помощью указателей.

Существует несколько типов линейных списков:

- Односвязный список: В односвязном списке каждый элемент содержит указатель на следующий элемент в списке.
- Двусвязный список: В двусвязном списке каждый элемент содержит указатели на предыдущий и следующий элементы в списке.
- Циклический список: В циклическом списке последний элемент ссылается на первый элемент, образуя замкнутую петлю.

Фактически линейный список - это множество, состоящее из n ($n \geq 0$) узлов (элементов) $X[1]$, $X[2]$, … , $X[n]$, структурные свойства которого ограничены линейным (одномерным) относительным положением узлов (элементов), т.е. следующими условиями:

- если $n > 0$, то $X[1]$ – первый узел;
- если $1 < k < n$,

  то k-му узлу $X[k]$ предшествует узел $X[k-1]$, а за узлом $X[k]$ следует узел $X[k+1]$;
- $X[n]$ – последний узел.

Операции над линейными списками

1. Получить доступ к k-му элементу списка, проанализировать и/или изменить значения его полей.
2. Включить новый узел перед k- м.
3. Исключить k-й узел.
4. Объединить два или более линейных списков в один.
5. Разбить линейный список на два или более линейных списков.
6. Сделать копию линейного списка.
7. Определить количество узлов.
8. Выполнить сортировку в возрастающем порядке по некоторым значениям полей в узлах.
9. Найти в списке узел с заданным значением в некотором поле.
10. … и т.д.

Не все операции нужны одновременно!

=>

Будем различать типы линейных списков по набору главных операций, которые над ними выполняются.

Что такое «очередь»

Давайте познакомимся с еще одним способом организации динамических данных – очередями. Само слово «очередь» сразу ассоциирует нас с очередью в магазин за каким-то дефицитным товаром. Тот кто первым пришел - тот первым и покидает очередь. По-английски это можно описать словами First In (первым пришел) - и First Out (первый вышел). Поэтому очередь такого типа называется сокращённо называют FIFO. В такой структуре данные всегда добавляются в конец очереди, а извлекается - из начала. Где может быть полезно такая организация данных – наиболее распространенный пример - это буфер приема или передачи какого-либо устройства (например – буфер клавиатуры). Или можно представить простейшую систему обработки заказов интернет-магазина. Здесь целесообразно организовать очередь и обрабатывать заказы в порядке их поступления и так далее.

Везде, где требуется работать с очередностью объектов, имеет смысл рассмотреть возможности использования очереди для хранения данных. Но это первый тип очереди

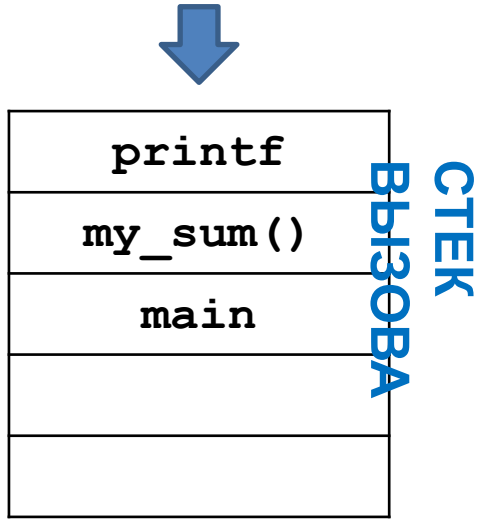
Что такое «очередь»

Есть еще один вид очереди, который сокращенно называется LIFO. То есть - последний пришел - первый вышел (Last In, First Out). Что это за очередь. Представим себе ящик в который складываются разные вещи, а вынимать эти вещи можно только сверху. В результате последняя положенная вещь будет извлечена первой, предпоследняя - второй и так далее – до первой. Классический пример использования этого типа очереди организации стека вызов функции. Например простая программа на языке C++ в которой последовательно вызывает три функции:

```
#include <iostream>
int my_sum(int a, int b) {
    int sum =;
    printf("Сумма чисел %d и %d равна %d\n", a, b, a + b);
    return 0;  }
int main() {
    int a = 5; int b = 10;
    my_sum(a, b);
    return 0;
}
```

Сначала в стек вызова помещается функция **main**, она вызывает функцию **my_sum()**, по этому опускается в стеке в низ. Функция **my_sum()** - вызывает функцию **printf()** и сама опускается на второе место. В результате у нас формируется стек вызова: функция **main**, затем **my_sum()** и, наконец, функции **printf** .

При работе программы, последняя вызванная функция, должна завершиться первый, поэтому сначала завершается функция **printf**, она удаляется из стека вызова, потом - функция **my_sum()**, она тоже удаляется из стека вызова, и, наконец, завершается функция **main**.



Что такое «очередь»

Это и есть принцип работы очередей. Они, как правило, добавляют и извлекают именно граничные (крайние) элементы не обращаясь к промежуточным. Хотя функционал очередей позволяет все-таки работать и с промежуточными элементами. Это используется крайне редко, но встречается, например Очередь с приоритетами. О ней поговорим позже.

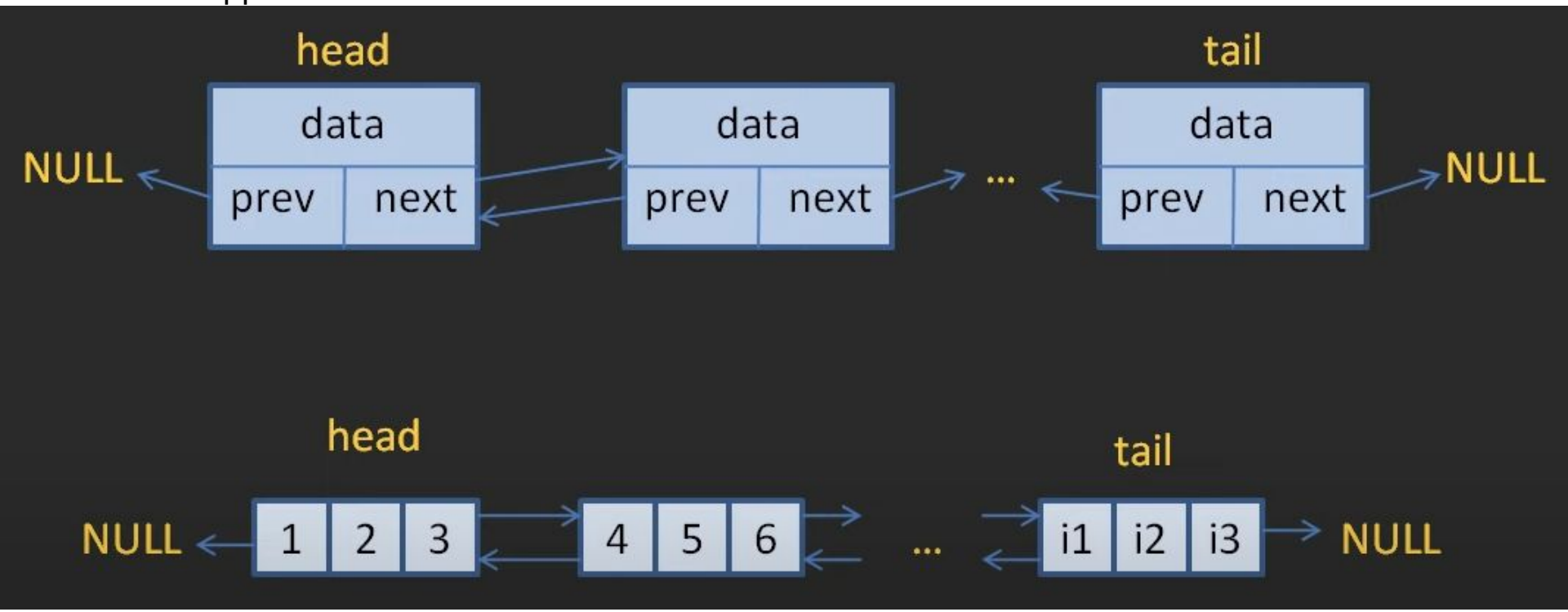
Для реализации очередей нам нужно выбрать динамическую структуру хранения данных, которая бы обладала высокой скоростью обработки крайних элементов. То есть имела объем вычислений O(1). Из прошлых лекций, можно вспомнить, что тут нам лучше всего подойдут односвязные и двухсвязные списки.

Чаще всего, для реализации очередей, применяют именно двухсвязные списки, потому что это более универсальная структура, но это не всегда. Здесь мы можем легко добавлять и удалять элементы либо с одного либо с другого конца списка. Сами же очереди на базе двусвязных списков, в которых можно добавлять и удалять элементы с обоих концов, сокращенно получили название Дек. Это сокращение от английской фразы double ended queue - двухсторонняя очередь.

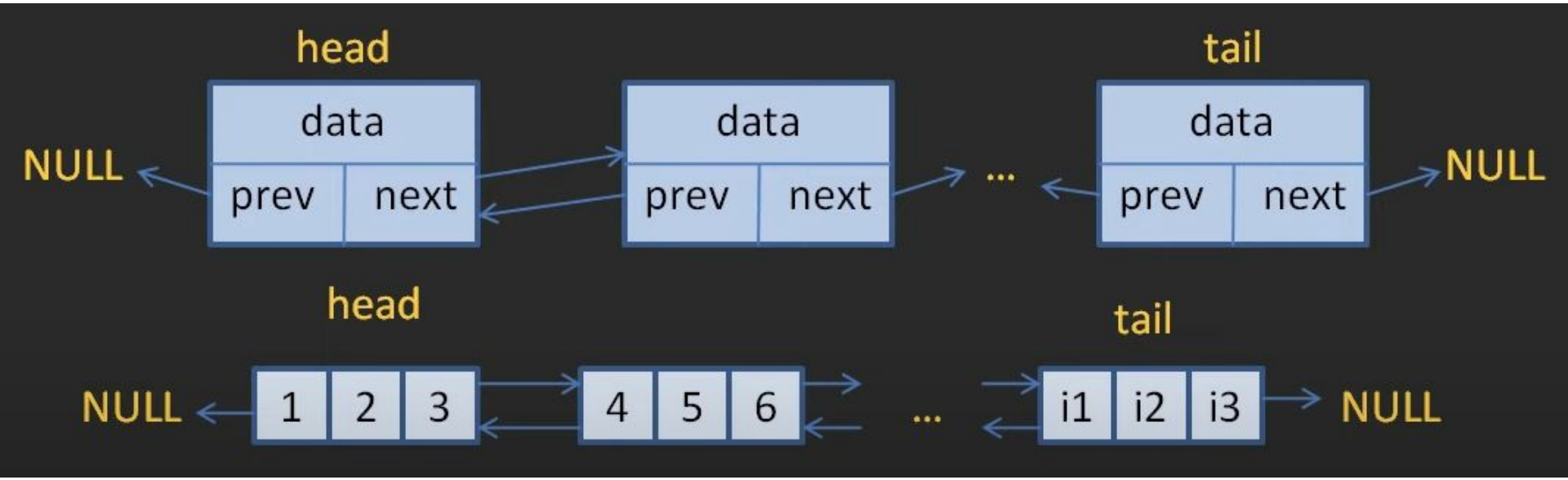
В результате, чтобы нам организовать очередь типа FIFO, достаточно реализовать и использовать два метода Push_front() - добавление в начало, и Pop_back() - извлечение с конца очереди. То есть, мы будем добавлять элементы с первого конца этой очереди, а извлекать с другого конца (фактически конца нашего двусвязного списка). У нас получается очередь типа FIFO. Для того чтобы реализовать другой тип очереди (LIFO) нужно использовать методы Push_front() -

Что такое «очередь»

Сразу отмечу, что очередь типа Дек реализуется не только на основе обычных двухсвязных списков, но и, например, на гибридной структуре двухсвязных списков и динамических массивов. То есть, в элементах двухсвязного списка хранятся динамический массивы определенной небольшой размерности. И в эти элементы уже динамического массива заносится данные нужной нам очереди. Это могут быть и числа и структуры и классы и все что угодно. Вот таким образом происходит организация данных, это может быть и обычный двухсвязный список и вот такая гибридная структура двухсвязнного списка из динамических массивов.



Что такое «очередь»



Благодаря такому подходу, доступ к отдельным промежуточным элементам очереди происходит несколько быстрее чем если у нас просто обычный двухсвязный список. Потому что каждый элемент двусвязного списка это просто какое-то одно значение, а с динамическим массивом мы сразу получаем доступ к группе элементов, и в пределах группы доступ к отдельному элементу осуществляется за время O(1), потому что в массивах доступ к произвольному элементу выполняется очень быстро. Поэтому, такая структура позволяет несколько быстрее обращаться к промежуточным элементам, и, в частности, она используется при реализации этой очереди Дек в стандартной библиотеке шаблонов STL на C++.

Но у такой структуры есть и недостаток - более медленный алгоритм вставки новых элементов в начало/конец этого модифицированного списка, потому что приходится сдвигать ранее записанные элементы в массиве, для того чтобы

Что такое «очередь»

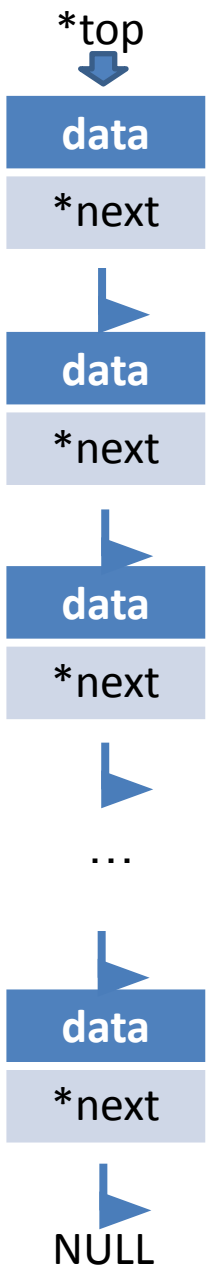
Вообще очереди можно реализовывать не только на базе связанных с списков, но, например, и динамических массивов. Тогда для очереди типа FIFO нам придется, для добавления элементов, использовать методы **`Push_front()`** – для добавления в начал массива и метод **`Pop()`** – для извлечения последних элементов. Однако, когда мы используем динамический массив, то метод **`Push_front()`** использует объем вычисления O(n), где n - это количество записанных туда элементов. Это гораздо более вычислительно ёмкая операция, чем в случае связанных списков, когда мы добавляем удаляем граничные элементы с объемом вычисления O(1). Согласитесь, что динамический массив в этом случае, для организации очередей, не самый эффективный подход, с точки зрения объёма вычислений.

Я привел этот пример с динамическим массивом, чтобы Вы ясно себе поняли, что очередь - это не какая-то конкретная структура данных, как связанные списки или массивы, а несколько абстрактная. Она лишь определяет порядок взаимодействия с элементами упорядоченной коллекции. А именно, как правило, добавление и удаление граничных элементов. С промежуточными элементами, тоже возможны взаимодействия, но производительность этих операций, как правило, значительно ниже и составляет O(n). Поэтому, в качестве основы для очередей, чаще всего все-таки используют двухсвязные списки. Точнее списки – вообще, а односвязные или двухсвязные – решает конкретный программист под конкретную задачу.

p.s. Для реализации очереди LIFO нам бы понадобились методы: **`append()`** и **`pop()`**. Почему – советую подумать самим. ☺

Стек

Стэк - это один из типов линейных списков, который использует принцип «последним вошел - первым вышел» (LIFO - Last In First Out). В стеке новые элементы добавляются и удаляются **только с одного конца списка**, который называется вершиной стека (на него указывает указатель *top). А указатель *next, нижнего элемента стека, всегда NULL



Стек

Основные операции, которые можно выполнять с стеком:

- Push: Добавление нового элемента в стек.
- Pop: Удаление элемента из стека.
- Top: Получение значения верхнего элемента стека.
- Size: Получение размера стека.
- Empty: Проверка, пуст ли стек.

Пример создания стека и операций над ним на языке программирования C++.

В этом примере мы создали **стек** типа **int** и добавили в него несколько элементов.

Затем мы вывели стек, удалив элементы из него.

Потом проверили, пуст ли стек, и получили размер стека.

```
#include <iostream>
#include <stack>
using namespace std;

int main() {
    stack<int> s;
    // Добавление элементов в стек
    s.push(1);
    s.push(2);
    s.push(3);
    s.push(4);
    // Вывод стека
    cout << "Stack: ";
    while (!s.empty()) {
        cout << s.top() << " ";
        s.pop();
    }
    cout << endl;
    // Проверка, пуст ли стек
    if (s.empty())
        cout << "Stack is empty" << endl;
    else
        cout << "Stack is not empty" << endl;
    // Получение размера стека
    cout << "Size of stack: " << s.size() << endl;
    return 0;
}
```

Стек

```cpp
#include <iostream>
#include <stack>
using namespace std;

int main() {
    stack<int> s;

    // Добавление элементов в стек
    s.push(1);
    s.push(2);
    s.push(3);
    s.push(4);

    // Вывод стека
    cout << "Stack: ";
    while (!s.empty()) {
        cout << s.top() << " ";
        s.pop();
    }
    cout << endl;

    // Проверка, пуст ли стек
    if (s.empty())
        cout << "Stack is empty" << endl;
    else
        cout << "Stack is not empty" << endl;

    // Получение размера стека
    cout << "Size of stack: " << s.size() << endl;

    return 0;
}
```

input

```
Stack: 4 3 2 1
Stack is empty
Size of stack: 0
```

Виды записи выражений

- Инфиксная или скобочная (**операция между операндами**) - это наиболее распространенный способ записи математических выражений. В инфиксной записи **операторы записываются между операндами**. Например, выражение (3 + 4) * 5 записывается как 3 + 4 * 5.

- Префиксная (**операция перед операндами**) - это способ записи математических выражений, в котором **операторы записываются перед операндами**. В префиксной записи выражение (3 + 4) * 5 записывается как * + 3 4 5.

- Постфиксная или обратная польская (**операция после операндов**) - это способ записи математических выражений, в котором **операторы записываются после операндов**. В постфиксной записи выражение (3 + 4) * 5 записывается как 3 4 + 5 *.

Примеры:

a + (f – b * c / (z – x) + y) / (a * r – k) - инфиксная
+a / + – f /*b c – z x y –*a r k - префиксная
a f b c * z x – / – y + a r * k – / + - постфиксная

Префиксная и постфиксная записи выражений часто используются в компьютерных программах для упрощения вычислений, так как они позволяют избежать использования скобок и упростить процесс парсинга выражений.

Парсинг - это процесс разбора текста на составные части и определения их структуры и значения. В компьютерной науке парсинг часто используется для разбора программных кодов, запросов к базам данных, математических выражений и других текстовых данных.

Парсер - это программа или алгоритм, который выполняет процесс парсинга. Парсер анализирует текст и разбивает его на составные части, такие как слова, фразы, выражения и т.д., а затем определяет их структуру и значение.

Парсеры могут быть реализованы на различных языках программирования и использовать различные алгоритмы и подходы для разбора текста. Например, парсеры для разбора программных кодов могут использовать синтаксический анализ, а парсеры для разбора математических выражений могут использовать алгоритмы, основанные на префиксной, инфиксной или постфиксной записи выражений.

Перевод из инфиксной формы в постфиксную

Вход: строка, содержащая арифметическое выражение, записанное в инфиксной форме

Выход: строка, содержащая то же выражение, записанное в постфиксной форме (обратной польской записи).

Обозначения:

числа, строки (идентификаторы) – операнды;

Знаки операций	Приоритеты операций
(	1
)	2
=	3
+, –	4
*, /	5

Алгоритм

Шаг 0:

Взять первый элемент из входной строки и поместить его в X.
Выходная строка и стек пусты.

Шаг 1:

- Если X – операнд, то дописать его в конец выходной строки.
- Если X = ‘(‘, то поместить его в стек.
- Если X = ‘)‘, то вытолкнуть из стека и поместить в конец выходной строки все элементы до первой встреченной открывающей скобки. Эту скобку вытолкнуть из стека.
- Если X – знак операции, отличный от скобок, то пока стек не пуст, и верхний элемент стека имеет приоритет, больший либо равный приоритету X, вытолкнуть его из стека и поместить в выходную строку. Затем поместить X в стек.

Шаг 2:

Если входная строка не исчерпана, то поместить в X очередной элемент входной строки и перейти на Шаг 1, иначе пока стек не пуст, вытолкнуть из стека содержимое в выходную строку.

Перевод из инфиксной формы в постфиксную.

Пример

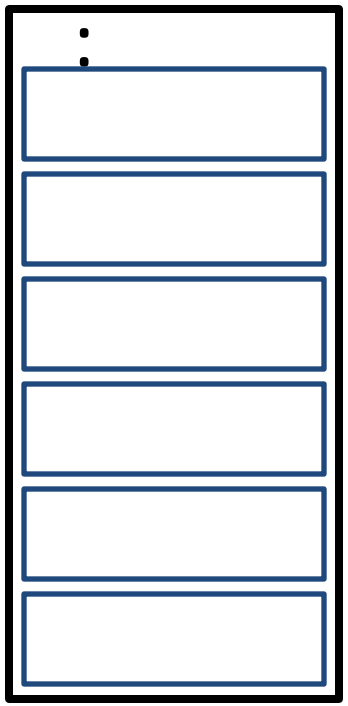
Входная строка:

a + (f – b * c / ((z – x) + y)) / ((a * r – k)

X =

Выходная строка:

Стек



Пример 2

(для самостоятельного изучения)

Входное выражение: (3 + 4) * 5

Шаги:

1. Создаем стек для хранения операторов.
2. Проходим по каждому символу входного выражения слева направо.
3. Символ "(" **- добавляем его в стек**.
4. Символ "3" - **добавляем его в результирующее выражение**.
5. Символ "+" - пока стек не пуст и верхний элемент стека имеет больший или равный приоритет, добавляем верхний элемент стека в результирующее выражение и удаляем его из стека. **Стек пуст, добавляем текущий оператор в стек.**
6. Символ "4" - **добавляем его в результирующее выражение**.
7. Символ ")" - пока верхний элемент стека не является открывающей скобкой, добавляем верхний элемент стека в результирующее выражение и удаляем его из стека. **Стек не пуст, верхний элемент стека - открывающая скобка, добавляем ее в результирующее выражение и удаляем ее из стека.**
8. Символ "*" - пока стек не пуст и верхний элемент стека имеет больший или равный приоритет, добавляем верхний элемент стека в результирующее выражение и удаляем его из стека. **Стек пуст, добавляем текущий оператор в стек.**
9. Символ "5" - **добавляем его в результирующее выражение**.
10. После обработки всех символов, добавляем все оставшиеся операторы из стека в результирующее выражение. **Стек пуст, ничего добавлять не нужно**.

Результирующее выражение: 3 4 + 5 *

Таким образом, мы получили постфиксную запись выражения (3 + 4) * 5.

Вычисления на стеке

Вход: строка, содержащая выражение, записанное в постфиксной форме.

Выход: число - значение заданного выражения.

Алгоритм:

Шаг 0:

Стек пуст.

Взять первый элемент из входной строки и поместить его в X.

Шаг 1:

Если X – операнд, то поместить его в стек.

Если X – знак операции, то вытолкнуть из стека два верхних элемента, применить к ним соответствующую операцию, результат положить в стек.

Шаг 2:

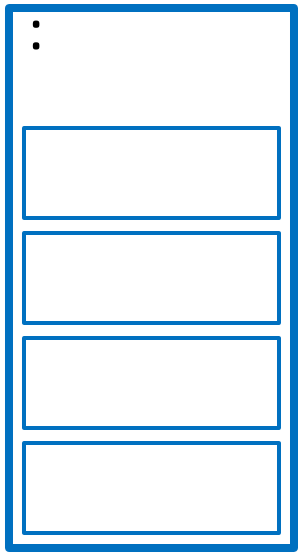
Если входная строка не исчерпана, то поместить в X очередной элемент входной строки и перейти на Шаг 1, иначе вытолкнуть из стека результат вычисления выражения.

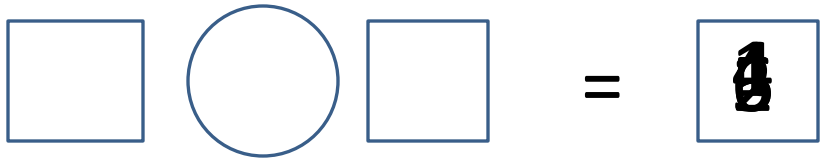
Вычисления на стеке. Пример

Входная строка:

5 2 3 * 4 2 / - 4 / + 1 -

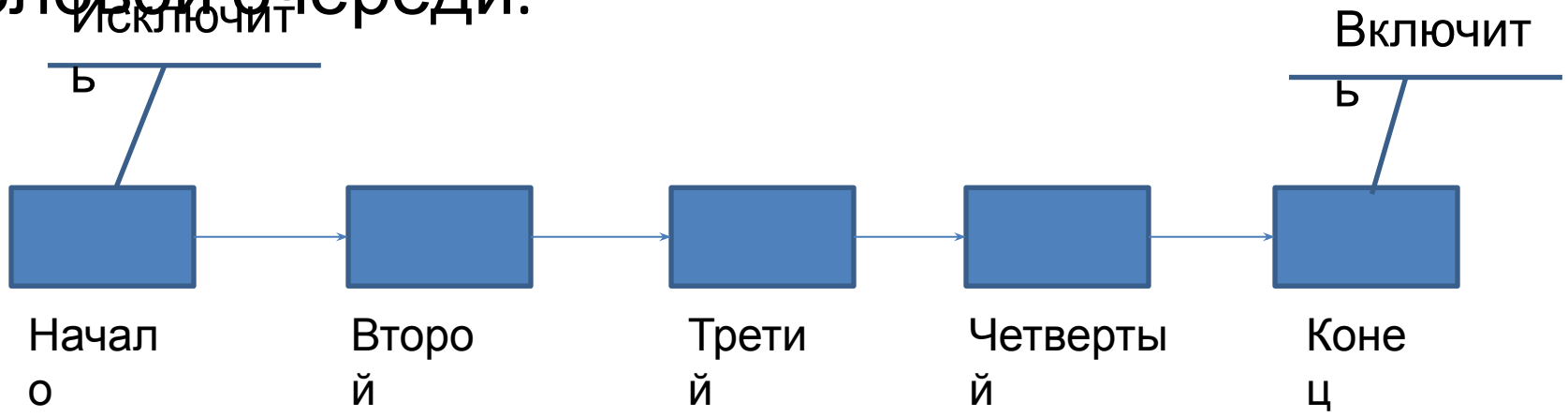
Стек





Очередь

Очередь - это другой тип линейных списков, который использует принцип "первым вошел - первым вышел" (FIFO - First In First Out). В очереди новые элементы **добавляются в один конец списка**, который называется хвостом очереди, а **удаляются из другого конца списка**, который называется головой очереди.



Очередь

Основные операции, которые можно выполнять с очередью:

- Enqueue: Добавление нового элемента в очередь.
- Dequeue: Удаление элемента из очереди.
- Front: Получение значения головного элемента очереди.
- Back: Получение значения хвостового элемента очереди.
- Size: Получение размера очереди.
- Empty: Проверка, пуста ли очередь.

Пример создания очереди и операций над ней на языке программирования C++.

В этом примере мы создали **очередь** типа **`int`** и добавили в нее несколько элементов.

Затем мы вывели очередь, удалив элементы из нее.

Потом проверили, пуста ли очередь, и получили размер очереди.

```
#include <iostream>
#include <queue>
using namespace std;

int main() {
    queue<int> q;
    // Добавление элементов в очередь
    q.push(1);
    q.push(2);
    q.push(3);
    q.push(4);

    // Вывод очереди
    cout << "Queue: ";
    while (!q.empty()) {
        cout << q.front() << " ";
        q.pop();
    }
    cout << endl;

    // Проверка, пуста ли очередь
    if (q.empty())
        cout << "Queue is empty" << endl;
    else
        cout << "Queue is not empty" << endl;
    // Получение размера очереди
    cout << "Size of queue: " << q.size() <<
endl;
    return 0;
}
```

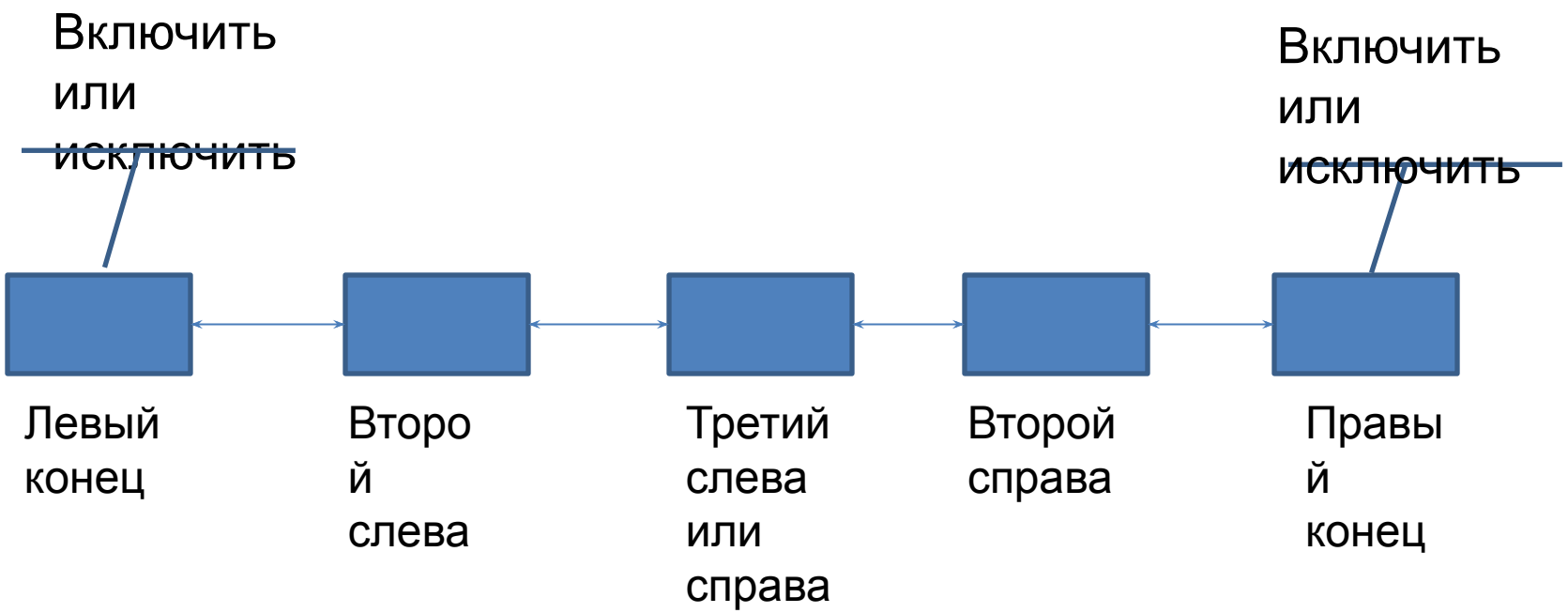
Очередь

```cpp
#include <iostream>
#include <queue>
using namespace std;

int main() {
    queue<int> q;

    // Добавление элементов в очередь
    q.push(1);
    q.push(2);
    q.push(3);
    q.push(4);

    // Вывод очереди
    cout << "Queue: ";
    while (!q.empty()) {
        cout << q.front() << " ";
        q.pop();
    }
    cout << endl;

    // Проверка, пуста ли очередь
    if (q.empty())
        cout << "Queue is empty" << endl;
    else
        cout << "Queue is not empty" << endl;

    // Получение размера очереди
    cout << "Size of queue: " << q.size() << endl;

    return 0;
}
```

```
Queue: 1 2 3 4
Queue is empty
Size of queue: 0
```

Дек (double-ended queue) очередь с двумя концами

Дек (от англ. double-ended queue) - это тип линейных списков, который является частным случаем очереди. В деке новые элементы могут **добавляться и удаляться с двух концов списка**, которые называются **головой** и **хвостом** дека.



Дек (double-ended queue) очередь с двумя концами

В программировании, ДЕК (двусторонняя очередь) применяется в следующих случаях:

- Когда требуется добавлять и удалять элементы из начала и конца очереди.
- Когда требуется обрабатывать данные в порядке FIFO (First In, First Out) и LIFO (Last In, First Out).
- Когда требуется использовать очередь для реализации стека и очереди.
- Когда требуется использовать очередь для реализации алгоритмов, таких как алгоритм Дейкстры и алгоритм Беллмана-Форда.
- Когда требуется использовать очередь для реализации алгоритмов обработки потоков данных, таких как алгоритмы обработки сетевых пакетов и алгоритмы обработки звука.
- Может быть использован в параллельных программах, где различные потоки выполняют операции ввода/вывода данных с очереди.
- Может быть применен в симуляционных моделях, где необходимо моделировать процессы, использующие очереди с

Дек (double-ended queue) очередь с двумя концами

Основные операции, которые можно выполнять с деком:

- Push_front: Добавление нового элемента в голову дека.
- Push_back: Добавление нового элемента в хвост дека.
- Pop_front: Удаление элемента из головы дека.
- Pop_back: Удаление элемента из хвоста дека.
- Front: Получение значения головного элемента дека.
- Back: Получение значения хвостового элемента дека.
- Size: Получение размера дека.
- Empty: Проверка, пуст ли дек.

Пример создания дека и операций над ним на языке программирования C++.

В этом примере, мы создали **дек** типа **`int`** и добавили в него несколько элементов с помощью функций **`push_back`** и **`push_front`**.

Затем мы вывели дек, удалив элементы из него с помощью функций **`pop_back`** и **`pop_front`**.

Ну и проверили, пуст ли дек, и получили размер нашего дека.

```
#include <iostream>
#include <deque>
using namespace std;

int main() {
    deque<int> d;
    // Добавление элементов в дек
    d.push_back(1);
    d.push_back(2);
    d.push_front(3);
    d.push_front(4);
    // Вывод дека
    cout << "Deque: ";
    for (int i = 0; i < d.size(); i++)
        cout << d[i] << " ";
    cout << endl;
    // Удаление элементов из дека
    d.pop_back();
    d.pop_front();
    // Вывод дека после удаления элементов
    cout << "Deque after pop: ";
    for (int i = 0; i < d.size(); i++)
        cout << d[i] << " ";
    cout << endl;
    // Проверка, пуст ли дек
    if (d.empty())
        cout << "Deque is empty" << endl;
    else
        cout << "Deque is not empty" << endl;
    // Получение размера дека
    cout << "Size of deque: " << d.size() << endl;

    return 0;
}
```

Дек (double-ended queue) очередь с двумя концами

```cpp
#include <deque>
using namespace std;

int main() {
    deque<int> d;

    // Добавление элементов в дек
    d.push_back(1);
    d.push_back(2);
    d.push_front(3);
    d.push_front(4);

    // Вывод дека
    cout << "Deque: ";
    for (int i = 0; i < d.size(); i++)
        cout << d[i] << " ";
    cout << endl;

    // Удаление элементов из дека
    d.pop_back();
    d.pop_front();

    // Вывод дека после удаления элементов
    cout << "Deque after pop: ";
    for (int i = 0; i < d.size(); i++)
        cout << d[i] << " ";
    cout << endl;

    // Проверка, пуст ли дек
    if (d.empty())
        cout << "Deque is empty" << endl;
    else
        cout << "Deque is not empty" << endl;

    // Получение размера дека
    cout << "Size of deque: " << d.size() << endl;

    return 0;
}
```

```
Deque: 4 3 1 2
Deque after pop: 3 1
Deque is not empty
Size of deque: 2
```

Стеки

- push-down список
- реверсивная память
- гнездовая память
- магазин
- LIFO (last-in-first-out)
- список йо-йо

Операции работы со стеками

1. makenull (S) – делает стек S пустым
2. create() – создает стек
3. top (S) – выдает значение верхнего элемента стека, не удаляя его
4. pop(S) – выдает значение верхнего элемента стека и удаляет его из стека
5. push(x, S) – помещает в стек S новый элемент со значением x
6. empty (S) - если стек пуст, то функция возвращает 1 (истина), иначе – 0 (ложь).

Реализация стека на си

В данных примерах кода будут представлены функции для работы со стеком, реализованным на основе связанного списка.

1. Структура **`list`** представляет собой узел списка, содержащий целочисленное значение и указатель на следующий узел.

```
struct list {
   int data;
    struct list * next;
};
```

2. Структура **`stack`** представляет собой стек, содержащий указатель на вершину стека.

```
typedef struct stack { struct list *top; } Stack;
```

3. Функция **`makenull()`** очищает стек, освобождая память, выделенную под все узлы списка.

```
void makenull (Stack  *S)
{  struct list *p;
   while (S->top) {
      p = S->top;
       S->top = p->next;
      free(p); }
}
```

Реализация стека на си - продолжение

4. Функция `create()` создает новый стек и возвращает указатель на него.

```
Stack * create ( )
{
   Stack * S;
   if ((S = (Stack *) malloc (sizeof (Stack))) == NULL)
   {
    printf("Ошибка выделения памяти\n");
    exit(1);
   }
   S->top = NULL;
   return S;
}
```

Реализация стека на си - продолжение

5. Функция **top()** возвращает значение вершины стека или 0, если стек пуст.

```
int top (Stack *S)
{
     if (S->top)
          return (S->top->data);
     else
          return 0;
       //здесь может быть реакция на
//ошибку – обращение к пустому стеку
}
```

```
Или (лучше):
int top (Stack *S)
{
     if (S->top)
          return (S->top->data);
     else {
          printf("Ошибка: обращение к
пустому стеку\n");
          exit(1);
     }
}
```

В п. 4 и 5 использовался оператор **exit()**. В языке C++ он **завершает выполнение программы и возвращает указанное значение, которое обычно используется для оценки успешности выполнения программы.**

Если аргументом функции **exit()** не указано никакое значение, то по умолчанию возвращается 0, что обычно интерпретируется как успешный завершение программы.
Оператор **exit()** также может быть использован для принудительного завершения программы в случае возникновения критических ошибок, которые невозможно обработать внутри программы.

Однако, использование оператора **exit()** не рекомендуется в многопоточных программах, так как он может привести к непредсказуемым последствиям, таким как несостоятельность потоков или утечка памяти. Вместо этого рекомендуется использовать более безопасные методы завершения программы, такие как вызов функции **abort()** или вызов функции **exit()** с указанием кода возврата, который может быть обработан внешними программами.

Реализация стека на си - продолжение

Оператор **`return()`** в языке C++ используется **для завершения выполнения функции и возврата управления вызывающей функции**. Он может использоваться в любой функции, которая возвращает значение, и может быть использован для возврата значения, которое может быть использовано вызывающей функцией.

Оператор **`exit()`**, напротив, **используется для принудительного завершения выполнения программы**. Он может быть использован в любом месте программы, но обычно используется для завершения программы в случае возникновения критических ошибок, которые невозможно обработать внутри программы.

Таким образом, основное отличие между оператором **`return()`** и оператором **`exit()`** заключается в том, что первый используется для завершения выполнения **функции** и возврата управления вызывающей функции, а второй используется для принудительного завершения выполнения **программы**.

Реализация стека на си - продолжение

6. Функция `pop()` удаляет вершину стека и возвращает ее значение.

```
int pop(Stack *S)  {
    int a;
    struct list *p;
    p = S->top;
    a = p->data;
    S-> top = p->next;
    free(p);
    return a;  }
```

Однако, эта функция, в таком виде, содержит несколько программных недоработок:

- Нет проверки на пустой стек. Если стек пуст, тогда попытка получить значение из S->top приведет к неопределенному поведению.
- Нет проверки на ошибки выделения памяти. Если free(p) не удалось освободить память, тогда функция не будет знать об этом.
- Нет проверки на ошибки при выделении памяти. Если malloc() не смог выделить память для p, тогда функция не будет знать об этом.
- Нет проверки на ошибки при присваивании значения S->top. Если S->top не является указателем на структуру list, тогда присваивание S->top = p->next; приведет к неопределенному поведению.
- Нет проверки на ошибки при присваивании значения p->data. Если p->data не является указателем на целое число, тогда присваивание a = p->data; приведет к неопределенному поведению.
- Чтобы исправить эти недоработки, следует добавить соответствующие проверки и обработку ошибок.

Реализация стека на си - продолжение

6. **Исправленная(!)** функция **pop()** удаляет вершину стека и возвращает ее значение.

```
int pop(Stack *S) {
    // Проверка на пустой стек
    if (S->top == NULL) {
        std::cerr << "Error: stack is empty" << std::endl;
        exit(1);     }
 struct list *p = S->top; // Создание указателя на структуру list
    int a = p->data; // Получение значения из стека
    S->top = p->next; // Перемещение указателя на вершину стека
    free(p); // Освобождение памяти
    return a; // Возвращение значения из стека
}
```

В этой программе использован оператор **cerr** используется для вывода сообщений об ошибках на стандартный поток вывода ошибок. Он является объектом класса **ostream**, который предоставляет функции для вывода данных на стандартный поток вывода ошибок.

Оператор **cerr** не блокирует выполнение программы, поэтому его можно использовать для вывода сообщений об ошибках, которые не должны останавливать выполнение программы. Однако, если вы хотите остановить выполнение программы после вывода сообщения об ошибке, можно использовать функцию **exit()**.

Оператор **cerr** также не добавляет автоматически перевод строки после вывода сообщения, поэтому если вы хотите добавить перевод строки, нужно использовать оператор **<<** для вывода символа новой строки **'\n'**.

Реализация стека на си - продолжение

7. Функция **push()** добавляет новое значение в вершину стека.

```
void push(int a, Stack *S)
{
   struct list *p;
   p = (struct list *) malloc ( sizeof (struct list));
   p->data = a;
   p->next = S-> top;
   S->top = p ;
}
```

Она может быть использована в таком виде, однако, она тоже содержит несколько недоработок:

- Нет проверки на ошибки выделения памяти. Если malloc() не смог выделить память для p, тогда функция не будет знать об этом.
- Нет проверки на ошибки при присваивании значения p->data. Если p->data не является указателем на целое число, тогда присваивание p->data = a; приведет к неопределенному поведению.
- Нет проверки на ошибки при присваивании значения p->next. Если S->top не является указателем на структуру list, тогда присваивание p->next = S->top; приведет к неопределенному поведению.
- Нет проверки на ошибки при присваивании значения S->top. Если S->top не является указателем на структуру list, тогда присваивание S->top = p; приведет к неопределенному поведению.

Реализация стека на си - продолжение

7. **Исправленная** функция `push()` добавляет новое значение в вершину стека.

```
void push(int a, Stack *S) {

    struct list *p = (struct list *) malloc ( sizeof (struct list)); // Выделение памяти
        if (p == NULL) {            // Проверка на ошибки выделения памяти
        std::cerr << "Error: memory allocation failed" << std::endl;
        exit(1);     }

       p->data = a; // Присвоение значения

        if (p->data != a) {    // Проверка на ошибки при присваивании значения
        std::cerr << "Error: assignment failed" << std::endl;
        exit(1);     }

        p->next = S->top; // Присвоение значения next

       if (p->next != S->top) {    // Проверка на ошибки при присваивании значения
        std::cerr << "Error: assignment failed" << std::endl;
        exit(1);      }

         S->top = p; // Присвоение значения top

        if (S->top != p) {    // Проверка на ошибки при присваивании значения
        std::cerr << "Error: assignment failed" << std::endl;
        exit(1);      }
}
```

Реализация стека на си - продолжение

8. Функция **empty()** предназначена для проверки стека на пустоту. Она принимает указатель на вершину стека и проверяет, пустой ли стек. Если вершина стека равна NULL, то стек пуст, и функция возвращает **true**. Если вершина не равна NULL, то стек не пуст, и функция возвращает **false**.

```
int empty (Stack *S)
{
    return (S->top == NULL);
}
```

Однако эта функция может быть не очень быстра, поскольку для проверки пустоты стека ей нужно проверить указатель на вершину. Чтобы сделать эту функцию более быстрой, можно добавить флаг «пустой» **в структуру стека**. Флаг «пустой» (логическая переменная **is_empty**) будет указывать, пустой ли стек или нет.

Вот приведен пример улучшенной реализации:

```
struct Stack {
    Node* top;
    bool is_empty;
};

bool empty(Stack* S) {
    return S->is_empty;
}
```

Очереди

Очередь в программировании - это самая популярная динамическая структура данных, которая использует принцип "первый пришел - первый вышел" (First-In-First-Out, FIFO).

Очередь состоит из двух концов: голова (head) и хвост (tail). Новые элементы добавляются в хвост очереди, а извлекаются из головы.

Очередь необходима для решения различных задач, связанных с последовательным выполнением операций или обработкой данных. Например:

- Очередь задач: в многозадачных операционных системах задачи добавляются в очередь, и операционная система выполняет их по очереди.

- Очередь запросов: в сетевых программах или серверах запросы клиентов обрабатываются по очереди.

- Очередь сообщений: в системах рассылки сообщений или очередях сообщений (например, RabbitMQ или Apache Kafka) сообщения добавляются в очередь и обрабатываются по очереди получателями.

- Очередь вызовов: в многопоточных программах или библиотеках, таких как asyncio в Python, очередь вызовов используется для планирования выполнения асинхронных операций.

Операции работы с очередями

1. makenull (Q) – делает очередь Q пустой
2. create( ) – создает очередь
3. first (Q) – выдает значение первого элемента очереди, не удаляя его
4. dequeue(Q) – выдает значение первого элемента очереди и удаляет его из очереди
5. enqueue(x, Q) – помещает в конец очереди Q новый элемент со значением x
6. empty (Q) - если очередь пуста, то функция возвращает 1 (истина), иначе – 0 (ложь).

Реализация очереди на циклическом массиве

(нет проверки на пустоту и переполнение!)

Очередь на циклическом массиве - это структура данных, которая использует массив для хранения элементов, но при этом обрабатывает его как если бы он был замкнутым контуром. Это позволяет использовать массив более эффективно, так как нет необходимости постоянно расширять или уменьшать его размер.

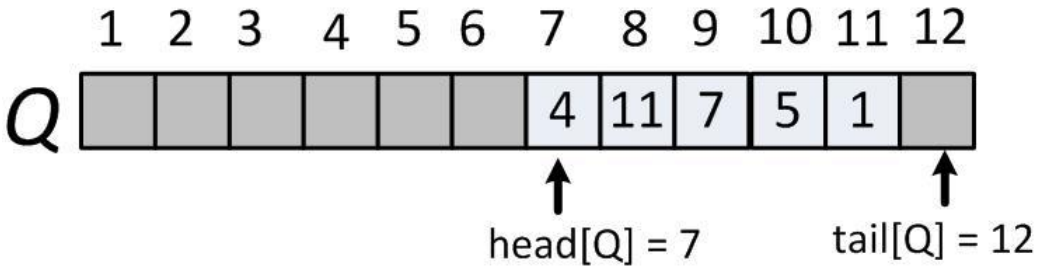
Для объяснения работы очереди на циклическом массиве, рассмотрим две процедуры: **`Enqueue`** (добавление элемента в очередь) и **`Dequeue`** (удаление элемента из очереди).

Реализация очереди на циклическом массиве

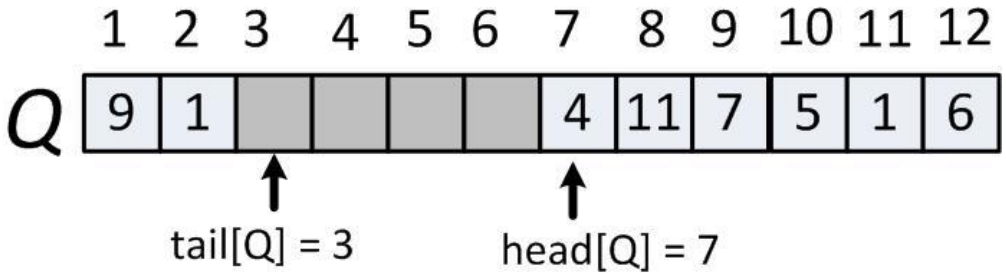
(нет проверки на пустоту и переполнение!)

```
Enqueue (Q, x)
{
   Q[tail[Q]] ← x
   if tail[Q] = length[Q]
   then tail[Q] ← 1
   else tail[Q] ← tail[Q] + 1
}
```

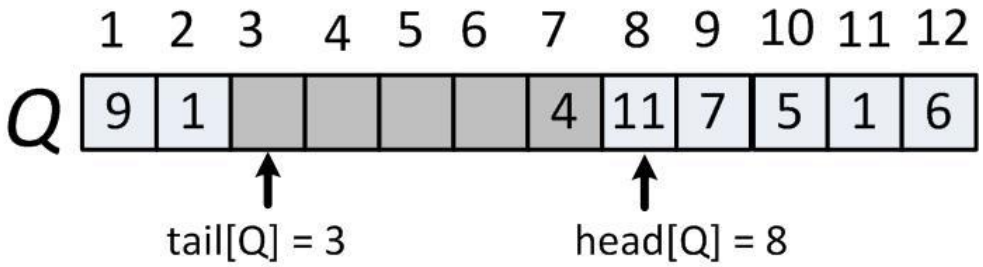
```
Dequeue (Q)
{
   x ← Q[head[Q]]
   if head[Q]= length[Q]
   then head[Q] ←  1
   else head[Q] ← head[Q]+1
   return x
}
```



Enqueue(Q, 6); Enqueue(Q, 9); Enqueue(Q, 1);



Dequeue(Q);



Реализация очереди на циклическом массиве

(нет проверки на пустоту и переполнение!)

```
Enqueue (Q, x)
{
   Q[tail[Q]] ← x    // Вставляем элемент x в массив Q по индексу tail[Q]
   if tail[Q] = length[Q] // Если индекс tail[Q] равен длине массива
   then tail[Q] ← 1   // тогда устанавливаем tail[Q] в 1 (это означает, что
          //очередь зацикливается)
   else tail[Q] ← tail[Q] + 1   // иначе увеличиваем tail[Q] на 1
}

Dequeue (Q)
{
   x ← Q[head[Q]] // Извлекаем элемент из массива Q по индексу head[Q] и
       //сохраняем его в x
   if head[Q]= length[Q] // Если индекс head[Q] равен длине массива
   then head[Q] ←  1    // тогда устанавливаем head[Q] в 1 (это означает, что
             //очередь зацикливается)
   else head[Q] ← head[Q]+1 // иначе увеличиваем head[Q] на 1
   return x  // возвращаем извлеченный элемент x
}
```

Реализация очереди с помощью двух стеков

Очередь можно реализовть с помощью двух стеков. В этом случае используют два стека: S1 и S2 для хранения элементов. Стек S1 используется для добавления элементов в очередь (enqueue), а стек S2 используется для удаления элементов из очереди (dequeue).

Для объяснения работы очереди с помощью двух стеков, рассмотрим две процедуры: Enqueue (добавление элемента в очередь) и Dequeue (удаление элемента из очереди).

```
Enqueue (Q, x)
{
    Push (S2, x) // Вставляем элемент x в стек S2
}

Dequeue (Q)
{
    if Stack_Empty (S1) // Если стек S1 пуст
    then Переложить все содержимое из S2 в S1 // тогда
    //перекладываем все элементы из стека S2 в стек S1
    Pop(S1) // Извлекаем элемент из стека S1 и удаляем его
}
```

Реализация очереди с помощью двух стеков

```
Enqueue (Q, x)
{
    Push (S2, x) // Вставляем элемент x в стек S2
}

Dequeue (Q)
{
    if Stack_Empty (S1) // Если стек S1 пуст
    then Переложить все содержимое из S2 в S1 // тогда
    //перекладываем все элементы из стека S2 в стек S1
    Pop(S1) // Извлекаем элемент из стека S1 и удаляем его
}
```

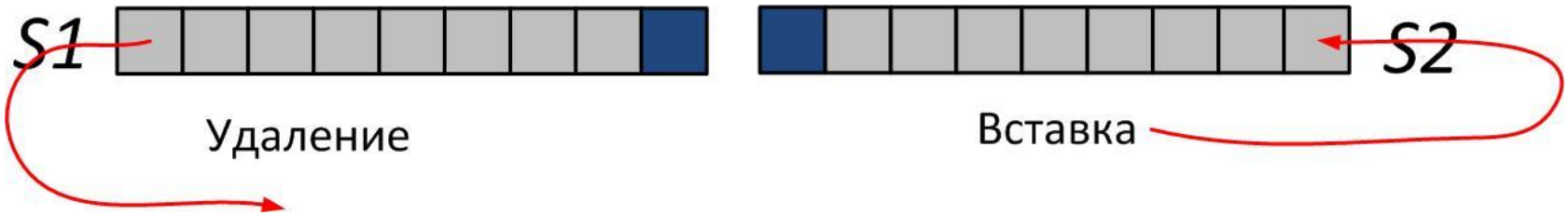
Теперь давайте объясним, как работают эти процедуры:

Enqueue (добавление элемента в очередь): когда мы добавляем элемент в очередь, мы просто помещаем его в стек S2.

Dequeue (удаление элемента из очереди): когда мы хотим извлечь элемент из очереди, мы сначала проверяем, пуст ли стек S1. Если стек S1 пуст, мы перекладываем все элементы из стека S2 в стек S1. Затем мы извлекаем элемент из стека S1 и удаляем его.

Важно отметить, что при реализации очереди с помощью двух стеков, время выполнения операции **Dequeue** может быть неконстантным, так как может потребоваться время для перекладывания всех элементов из стека S2 в стек S1.

Реализация очереди с помощью двух стеков




```
Enqueue (Q, x)
{
    Push (S2, x) // Вставляем элемент x в стек S2
}

Dequeue (Q)
{
    if Stack_Empty (S1) // Если стек S1 пуст
    then Переложить все содержимое из S2 в S1 // тогда
        //перекладываем все элементы из стека S2 в
//стек S1
    Pop(S1) // Извлекаем элемент из стека S1 и удаляем
его
}
```

Реализация очереди на динамических списках

```
struct list
{
    int data;
     struct list * next;
};

typedef struct queue
{
    struct list *first;
    struct list *end;
} Queue;
```

Реализация очереди на массиве

```
typedef struct _Queue
{
    int size; // размер массива
    int first; // номер первого элемента очереди
    int leng; // длина очереди
    int * arr; // указатель на начало массива
} Queue;
```

Реализация очереди на массиве (продолжение)

```
Queue * create()
{
	Queue *q = (Queue *) malloc ( sizeof( Queue ) );
	q -> first = 0;
	q -> leng = 0;
	q -> size = 1;
	q -> arr = (int *)malloc(sizeof (int) * q -> size);
	return q;
}

int empty( Queue * q )
{
	return (q -> leng == 0);
}
```

Реализация очереди на массиве(продолжение)

```
void enqueue(Queue * q, int a)
{
    if ( q->leng == q->size )
    {
        q->arr = (int *)realloc(q->arr,  sizeof(int)* q->size * 2);
        if (q -> first > 0)
            memcpy( q->arr + q->size, q->arr,  (q->first) * sizeof(int) );
        q->size *= 2;
    }
    q->arr [ (q->first + q->leng++) % q->size ] = a;
}

int dequeue(Queue * q)
{
    int a = q->arr [q->first++];
    q->first %= q->size;
    q->leng --;
    return a;
}
```

Пятиминутка

1. Перевести данное арифметическое выражение в обратную польскую запись: (10 + (6 - 9) * 4 + 3 / 6) * (2 - 1)

2. Пусть функция push добавляет число в стек, а функция pop извлекает число из стека и печатает его. Что будет напечатано при выполнении последовательности вызовов функций:

push(3); push(4); pop(); push(1); pop(); push(7); pop(); pop();

3. Пусть функция put добавляет число в очередь, а функция get извлекает число из очереди и печатает его. Что будет напечатано при выполнении последовательности вызовов функций

put(1); get(); put(3); put(5); get();get(); put(2); get();

Пятиминутка

Задача:

Перевести данное арифметическое выражение в обратную польскую запись: (10 + (6 - 9) * 4 + 3 / 6) * (2 - 1)

Решение:

Чтобы перевести данное арифметическое выражение в обратную польскую запись, нужно выполнить следующие шаги:

1) Определить приоритеты операций:
   - Условные скобки: самый высокий приоритет
   - Умножение и деление: выше сложения и вычитания
   - Сложение и вычитание: самый низкий приоритет
2) Использовать стек для хранения операторов и чисел.
3) Пройтись по каждому символу в выражении слева направо.
   - Если символ - число, добавить его в выходное выражение.
   - Если символ - оператор, то **пока стек не пуст и операторы на вершине стека имеют больший или равный приоритет**, добавлять операторы из стека в выходное выражение и удалять их из стека. После этого добавить текущий оператор в стек.
   - Если символ - открывающая скобка, добавить ее в стек.
   - Если символ - закрывающая скобка, то пока символ на вершине стека не является открывающей скобкой, добавлять операторы из стека в выходное выражение и удалять их из стека. После этого удалить открывающую скобку из стека.
4) После обработки всех символов, если стек не пуст, добавить операторы из стека в выходное выражение и удалить их из стека.

Пятиминутка

ШАГ	ВЫРАЖЕНИЕ	СТЕК	ВЫХОД
1		(	
2	10	(	10
3	+	(+	10
4	(	(+ (	10
5	6	(+ (	10 6
6	-	(+ (-	10 6
7	9	(+ (-	10 6 9
8	)	(+	10 6 9-
9	*	(+ *	10 6 9-
10	4	(+ *	10 6 9- 4
11	+	(+	10 6 9- 4 *
12		(+	10 6 9- 4 *+
13	3	(+	10 6 9- 4 *+ 3
14	/	(+ /	10 6 9- 4 *+ 3
15	6	(+ /	10 6 9- 4 *+ 3 6
16	)	(	10 6 9- 4 *+ 3 6 /+
17			10 6 9- 4 *+ 3 6 /+
18	*	(*	10 6 9- 4 *+ 3 6 /+
19	(	(* (	10 6 9- 4 *+ 3 6 /+
20	2	(* (	10 6 9- 4 *+ 3 6 /+ 2
21	-	(* (-	10 6 9- 4 *+ 3 6 /+ 2
22	1	(* (-	10 6 9- 4 *+ 3 6 /+ 2 1
23	)	(*	10 6 9- 4 *+ 3 6 /+ 2 1-
24			10 6 9- 4 *+ 3 6 /+ 2 1-*

Пятиминутка

Проверка:

1) (10 + (6 - 9) * 4 + 3 / 6) * (2 - 1) = (10 + (-3)*4 + 0.5) * 1 = (10 – 12 + 0.5) = -1.5

2) 10 6 9 - 4 * + 3 6 / + 2 1 - * =>

ШАГ	ВЫРАЖЕНИЕ	СТЕК		
1	10	10		
2	6	10	6	
3	9	10	6	9
4	-	10	-3	
5	4	10	-3	4
6	*	10	-12	
7	+	-2		
8	3	-2	3	
9	6	-2	3	6
10	/	-2	0.5	
11	+	-1.5		
12	2	-1.5	2	
13	1	-1.5	2	1
14	-	-1.5	1	
15	*	-1.5		
	ОТВЕТ:	**-1.5**		