Алгоритмы и структуры данных

Лекция 7. Стек и очередь на массиве

Антон Штанюк (к.т.н, доцент)

17 апреля 2025 г.

Нижегородский государственный технический университет им. Р.Е. Алексеева
Институт радиоэлектроники информационных технологий
Кафедра ”Компьютерные технологии в проектировании и производстве”

Содержание

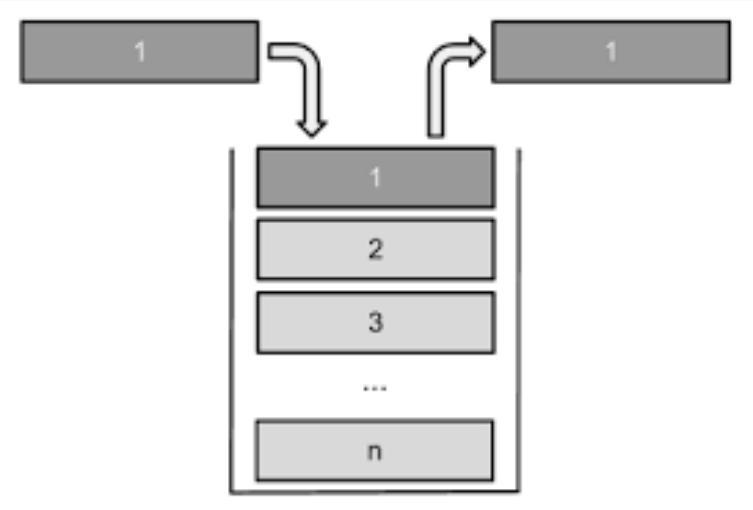
Стек на массиве

В этой и следующей темах мы остановимся на популярных абстрактых структурах данных - стеке и очереди. Абстрактность здесь обозначает то, что работа СД определяется не способом построения, а только принципом доступа к элементам. В дальнейшем мы рассмотрим примеры реализации на массивах и на списках.

Определение стека

Определение стека

Структура данных **Стек** является широко известной и очень распространенной. В ее основе лежит принцип доступа к элементам по названием **LIFO - последний пришел, первый вышел**.



Определение стека

Элементы помещаются и извлекаются с головы стека (**top**)

Реализация на массиве предполагает выбор типа массива:

- На массиве фиксированной длины (статическом массиве)
- На массиве переменной длины (динамическом массиве)

Добавление элемента и его удаление выполняются за время $O(1)$

Реализация стека на статическом массиве

Реализация на статическом массиве

Рассмотрим реализацию стека на статическом массиве. Реализацию сложных СД на объектно-ориентированных языках программирования лучше приводить в виде классов.

```
template<typename T, int size>
class TSStack {
   private:
     T arr[size]; // массив для хранения элементов
     int top;     // позиция вершины стека
   public:
     // конструктор класса
     TSStack():top(-1) { }
...
```

Реализация на статическом массиве

```
...
    // проверка на пустоту
    bool isEmpty() const {
       return top==-1;
    }
    // проверка на заполненность
    bool isFull() const {
       return top==size-1;
    }
...
```

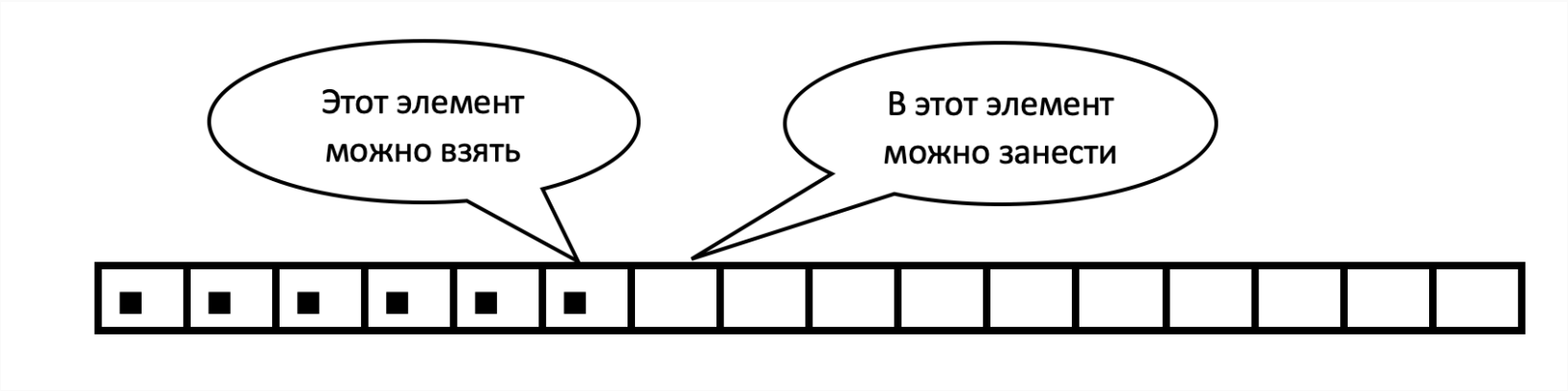
Реализация на статическом массиве

```
...
     T get() const {
        if(!isEmpty())
           return arr[top];
        else
           throw std::string(”empty”);
     }
     void pop() {
        if(!isEmpty())
          top--;
        else
           throw std::string(”empty”);
     }
     void push(T item) {
       if(!isFull())
         arr[++top]=item;
       else
         throw std::string(”full”);
     }
};
```

Детали реализации

- Данные хранятся в массиве **arr** в закрытой области класса (доступ напрямую в эту область запрещен).
- Для доступа к данным мы используем функции **get,push,pop**, которые позволяют контролировать доступ.
- Операция **get** возвращает элемент на вершине стека.
- Операция **pop** извлекает последний элемент.
- Операция **push** добавляет новый элемент в стек.
- Функции **isEmpty,isFull** проверяют, пустой ли стек или полный.
- Переменная класса **top** указывает на границу данных в массиве.
- При опустошении и переполнении стека ничего не происходит, операции игнорируются.
- Стек можно настроить для хранения данных любого типа, нужно отредактировать строку с **typedef**.

Детали реализации



Пример использования стека

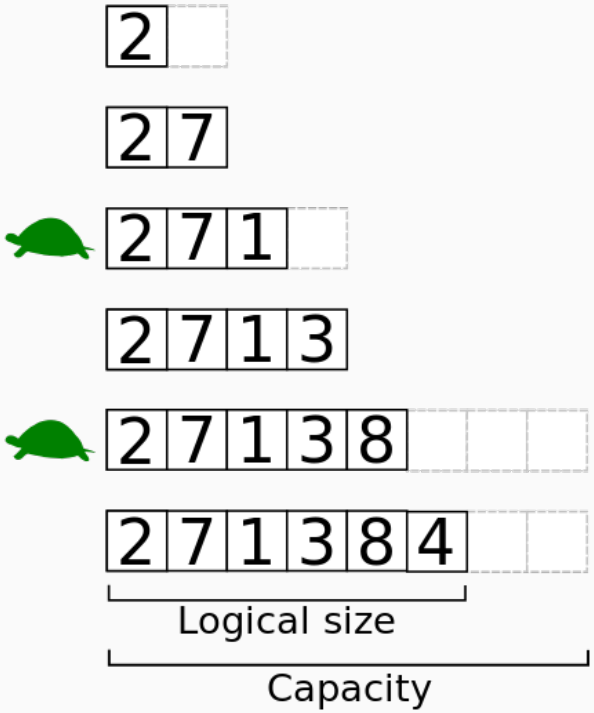
```
...
TSStack<int,10> stack; // создание нового стека
stack.push(5); // добавление первого элемента
stack.push(7); // добавление второго элемента
stack.push(11); // добавление третьего элемента

std::cout<<stack.get(); // вывод 11
stack.pop();
std::cout<<stack.get(); // вывод 7
stack.pop();
std::cout<<stack.get(); // вывод 5
stack.pop();
...
```

Реализация стека на динамическом массиве

Реализация на динамическом массиве

Реализация стека на динамическом массиве предполагает, что если исходный массив будет заполнен, стек автоматически увеличит свою память чтобы вместить новые элементы.

Динамический массив



Увеличение размера массива происходит быстро пока он меньше объёма массива. Когда нужно увеличить размер массива, а свободного места в нём нет, создаётся ещё один массив большего объёма, все элементы старого объёма копируются в новый массив, ссылка на старый массив удаляется.

Реализация на динамическом массиве

```
template<typename T>
class TDStack {
   private:
     type *arr;
     int size; // размер массива
     int top;
     // метод увеличения размера массива
     void resize(int nsize) {
        T *temp=new type[nsize];
        for(int i=0;i<size;i++)
          temp[i]=arr[i];
        delete[]arr;
        arr=temp;
        size=nsize;
     }
   public:
     DStack(int size):top(-1) {
       this->size=size;
       arr=new T[size];
     }

...
```

Реализация на динамическом массиве

```
...
    ~DSize() {
      delete[]arr;
    }
    bool isEmpty() const {
       return top==-1;
    }
    T get() const {
       if(!isEmpty())
          return arr[top];
       else
          throw std::string(”empty”);
    }
    void pop() {
       if(!isEmpty())
         top--;
       else
          throw std::string(”empty”);
    }
    void push(T item) {
      if(isFull())
        resize(2*size);
      arr[++top]=item;
    }
}
```

Замечания по реализации

- При создании стека нужно указать первоначальный размер массива.
- В классе **TDStack** определяется закрытый метод **resize** для увеличения размера памяти.
- Поскольку при использовании **TDStack** выделяется динамическая память, то в классе определяется специальный метод - *деструктор TDStack*, который освобождает выделенную в конструкторе динамическую память.
- Используемая память может только увеличиваться, освобождение происходит при разрушении стека.

Использование стека

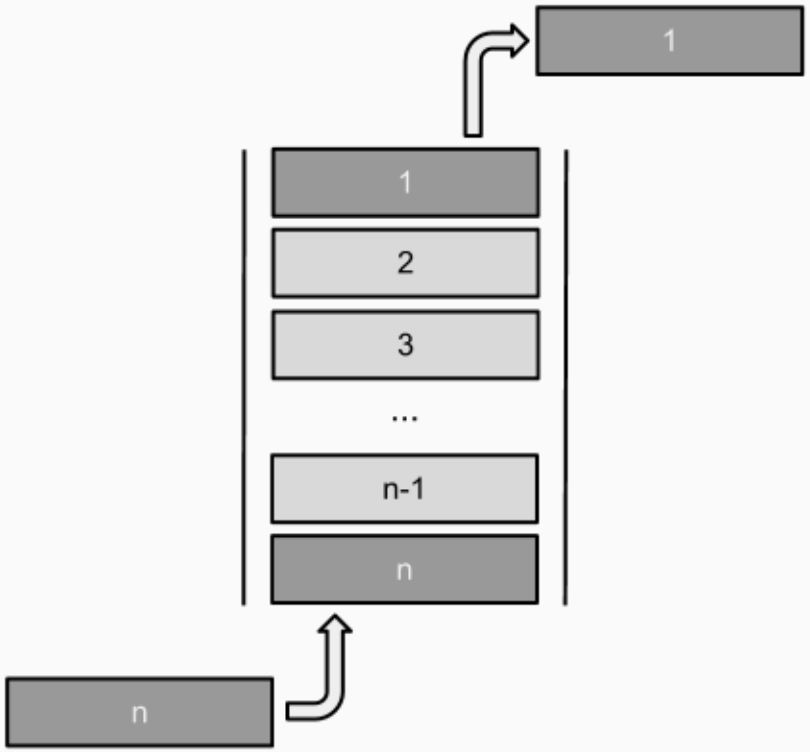
Рассмотрим теперь, как можно воспользоваться таким стеком:

```
#include ”tstack.h”
#include <iostream>

int main() {
  TDStack<int> istack;
  TDStack<char> cstack;

  for(int i=1;i<=10;i++)
    istack.push(i);

  while(istack.isEmpty()==false) {
    std::cout<<istack.get()<<” ”;
    istack.pop();
  }
  std::cout<<std::endl;
  return 0;
}
```

Определение очереди

Определение очереди

Структура данных **Очередь**, также как и стек, является широко распространенной и популярной. В ее основе лежит принцип доступа к элементам под названием **FIFO - первый пришел, первый вышел**.

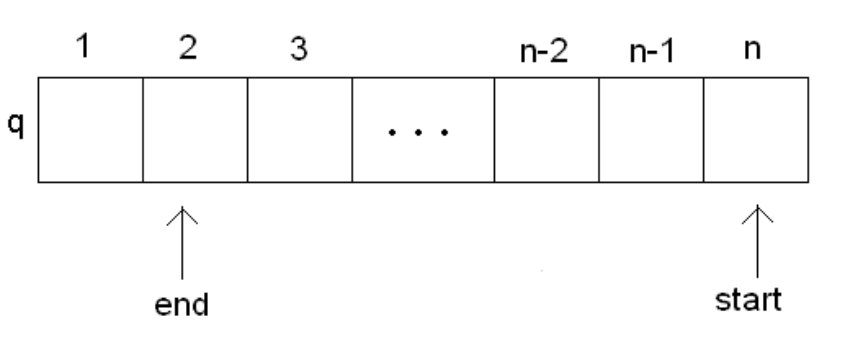


Определение очереди

Реализация очереди на массиве предполагает тоже два подхода:

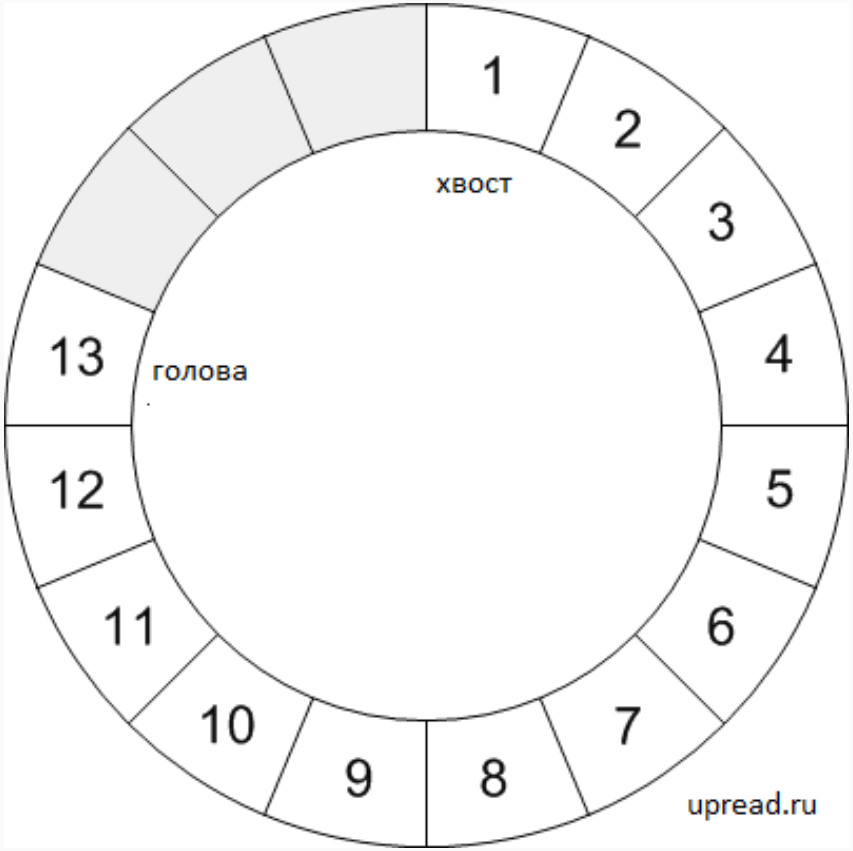
- на основе массива фиксированной длины (статического)
- на основе связанного списка.

Реализация на массиве предполагает использование двух указателей: на начало очереди и на ее конец. В процессе работы оба указателя движутся в одном направлении и неизбежно упираются в правую границу массива.



Определение очереди

Данное обстоятельство приводит к использованию массива фиксированной длины, но с возможностью перехода на начальные элементы массива. Возникает эффект кольца, поэтому такую реализацию часто назвывают **очередью на кольцевом буфере**.



Программная реализация очереди

Программная реализация

Реализуем очередь в виде шаблонного класса с использованием массива фиксированного размера. Приведем две популярные реализации кольцевого буфера: с использованием остатка от деления и без него.

```
template<typename T, int size>
class TSQueue {
    private:
       T arr[size];
       int begin, int end, int count;
    public:
       TSQueue():begin(0), end(0), count(0) {}
    ...
}
```

Программная реализация

```
...
bool isEmpty() const {
    return count == 0;
}
bool isFull() const {
    return count == size;
}
T get() const {
    if(isEmpty())
       throw std::string(”empty”);
    return arr[begin % size];
}
...
```

Программная реализация

```
...
 void push(const T& value) const {
    if(isFull())
       throw std::string(”full”);
    arr[end++ % size] = value;
    count++;
 }
 T pop() const {
    if(isEmpty())
       throw std::string(”empty”);
    T value = arr[begin++ % size];
    count++;
    return value;
 }
};
```

Программная реализация

Идея данной реализации основана на свойствах операции взятия остатка от деления. Любые значения **begin** и **end** не будут выходить за границы допустимых индексов массива [0-(size-1)]

Недостатком данной реализации является постоянный рост значений **begin** и **end**, который может привести к их переполнению.

Программная реализация

Реализуем очередь в виде шаблонного класса с использованием массива фиксированного размера. Описание шаблонного класса мы разобъем на две части: сам шаблонный класс и описание его методов.

```
#include <cassert>

template<typename T>
class TQueue {
private:
    T *arr;           // массив с данными
    int size;         // максимальное количество элементов в очереди
     размер( массива)
    int begin,        // начало очереди
        end;          // конец очереди
    int count;        // счетчик элементов
public:
    TQueue(int =100);          // конструктор по умолчанию
    ~TQueue();                 // деструктор
...
```

Программная реализация

```
...
    void push(const T &); // добавить элемент в очередь
    T pop();              // удалить элемент из очереди
    T get() const;        // прочитать первый элемент
    bool isEmpty() const;      // пустая ли очередь?
    bool isFull() const;       // заполнен ли массив?
};
```

Детали реализации

Далее, рассмотрим описание конструктора, который будет автоматически вызываться при создании очереди. На вход конструктора подается размер очереди в элементах.

```
// конструктор по умолчанию
template<typename T>
TQueue<T>::TQueue(int sizeQueue) :
    size(sizeQueue),
    begin(0), end(0), count(0) {
    // дополнительный элемент поможет нам различать конец и начало
     очереди
    arr = new T[size + 1];
}
```

Детали реализации

Деструктор будет автоматически вызываться при уничтожении очереди и его главная задача - освободить память, выделоенную под массив.

```
// деструктор класса Queue
template<typename T>
TQueue<T>::~TQueue() {
    delete [] arr;
}
```

Детали реализации

Функция добавления элемента в очередь с проверкой необходимости перехода в начало массива.

```
template<typename T>
void TQueue<T>::push(const T & item) {
    // проверяем, ести ли свободное место в очереди
    assert( count < size );

    arr[end++] = item;
    count++;

    // проверка кругового заполнения очереди
    if (end > size)
        end -= size + 1; // возвращаем end на начало очереди
}
```

Удаление элемента

Функция удаления элемента из очереди с проверкой необходимости перехода в начало массива.

```
// функция удаления элемента из очереди
template<typename T>
T TQueue<T>::pop() {
    // проверяем, есть ли в очереди элементы
    assert( count > 0 );

    T item = arr[begin++];
    count--;

    // проверка кругового заполнения очереди
    if (begin > size)
        begin -= size + 1; // возвращаем begin на начало очереди

    return item;
}
```

Чтение элемента из очереди

Функция чтения элемента из начала очереди, не меняя состояние очереди.

```
// функция чтения элемента на первой позиции
template<typename T>
T TQueue<T>::get() const  {
    // проверяем, есть ли в очереди элементы
    assert( count > 0 );
    return arr[begin];
}
```

Проверка на пустоту

```
// функция проверки очереди на пустоту
template<typename T>
bool TQueue<T>::isEmpty() const {
  return count==0;
}
```

Проверка на заполненность

```
// функция проверки очереди на заполненность
template<typename T>
bool TQueue<T>::isFull() const {
  return count==size;
}
```

Замечания по реализации

Рассмотрим детали реализации очереди:

- Переменная **size** хранит максимальную емкость кольцевого буфера.
- Переменная **count** хранит реальное количество элементов в очереди.
- Конструкция **assert** проверяет результат выражения и если он **false** аварийно завершает программу.
- При достижении указателями **begin, end** правой границы массива, они переводятся на левую границу.

Пример работы

В следующей программе мы создаем очередь из целых чисел с максимальным размером 50 элементов, заполняем ее числами от 1 до 45, а потом выводим значения на экран.

```
#include ”tqueue.h”
#include <iostream>

int main() {
    TQueue<int> que(50);
    for(int i=1;i<=45;i++)
       que.push(i);
    while(que.isEmpty()==false)
      std::cout<<que.pop()<<std::endl;
    return 0;
}
```

Список литературы

Список литературы i

- Кормен Т.,Лейзерсон Ч., Ривест Р.
  Алгоритмы: построение и анализ
  МЦНМО, Москва, 2000
- Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ.
  2-е изд. — М.: «Вильямс», 2006
- Википедия
  Алгоритм
  http://ru.wikipedia.org/wiki/Алгоритм
- Википедия
  Список алгоритмов
  http://ru.wikipedia.org/wiki/Список_алгоритмов
- Традиция
  Задача коммивояжёра
  http://traditio.ru/wiki/Задача

Список литературы ii

- Википедия
  NP-полная задача
  http://ru.wikipedia.org/wiki/NP-полная
- Серджвик Р.
  Фундаментальные алгоритмы на C++. Части 1-4
  Diasoft,2001
- Седжвик Р.
  Фундаментальные алгоритмы на C. Анализ/Структуры данных/Сортировка/Поиск
  СПб.: ДиаСофтЮП, 2003
- Седжвик Р.
  Фундаментальные алгоритмы на C. Алгоритмы на графах
  СПб.: ДиаСофтЮП, 2003
- Ахо А., Хопкрофт Д., Ульман Д. Структуры данных и алгоритмы.
  Издательский дом «Вильямс», 2000

Список литературы iii

- Кнут Д.
  Искусство программирования, том 1. Основные алгоритмы
  3-е изд. — М.: «Вильямс», 2006
- Кнут Д.
  Искусство программирования, том 2. Получисленные методы
  3-е изд. — М.: «Вильямс», 2007
- Кнут Д.
  Искусство программирования, том 3. Сортировка и поиск
  2-е изд. — М.: «Вильямс», 2007
- Кнут Д.
  Искусство программирования, том 4, выпуск 3. Генерация всех сочетаний и разбиений
  М.: «Вильямс», 2007

Список литературы iv

Кнут Д.
Искусство программирования, том 4, выпуск 4. Генерация всех деревьев. История комбинаторной генерации
М.: «Вильямс», 2007