академия больших данных

mail.ru group

Базовые структуры данных

Мацкевич Степан

Алгоритмы и структуры данных

Программа

1. Базовые структуры данных
2. Сортировки и порядковые статистики
3. Деревья поиска
4. Хеш-таблицы
5. RSA, блокчейн, сжатие данных
6. Обходы графов. Топсорт. Кратчайшие пути
7. Потоки. Паросочетания. Венгерский алгоритм
8. Поиск строк. Trie
9. Вычислительная геометрия
10. Парные игры

План лекции

- Массив, динамический массив
- Списки
- Стек, очередь, дек
- Куча

Массив

Массив

Массив – набор однотипных элементов, расположенных в памяти непосредственно друг за другом, доступ к которым осуществляется по индексу.

Традиционно индексирование элементов массива начинают с 0.

Линейный поиск

Задача. Найти заданный элемент в массиве.

Решение. Последовательно проверяем все элементы, пока не найдем заданный.

Время работы. O(n) в худшем случае. O(1) в лучшем.

Реализации в stl. std::find, std::find_if – https://en.cppreference.com/w/cpp/algorithm/find

```
template< class InputIt, class T >
InputIt find( InputIt first, InputIt last, const T& value );
```

Бинарный поиск

Задача. Найти заданный элемент в упорядоченном массиве. Вернуть позицию его первого вхождения, если он есть.

Решение. Шаг. Сравниваем элемент в середине массива (медиану) с заданным элементом. Выбираем нужную половинку массива в зависимости результата сравнения.
Повторяем этот шаг до тех пор, пока размер массива не уменьшится до 1.

Время работы. O(log n).

Реализации в stl:

- std::lower_bound – https://en.cppreference.com/w/cpp/algorithm/lower_bound,
- std::upper_bound

Бинарный поиск

```
// Возвращает позицию вставки элемента на отрезке [first, last).
// Равные элементы располагаются после.
int LowerBound(const double* arr, int count, double element)
{
    int first = 0;
    int last = count; // Элемент в last не учитывается.
    while (first < last) {
        int mid = (first + last) / 2;
        if (arr[mid] < element)
            first = mid + 1;
        else // В случае равенства arr[mid] останется справа.
            last = mid;
    }
    return first;
}
```

Динамический массив

Интерфейс динамического массива:

- operator[]. Доступ по индексу.
- push_back. Добавление в конец.

Реализация в stl: std::vector

https://en.cppreference.com/w/cpp/container/vector

```
template<class T, class Allocator = std::allocator<T>> class vector;
```

Динамический массив. Реализация.

- Содержит внутренний массив фиксированной длины – **буфер**,
- помнит текущее количество добавленных элементов.

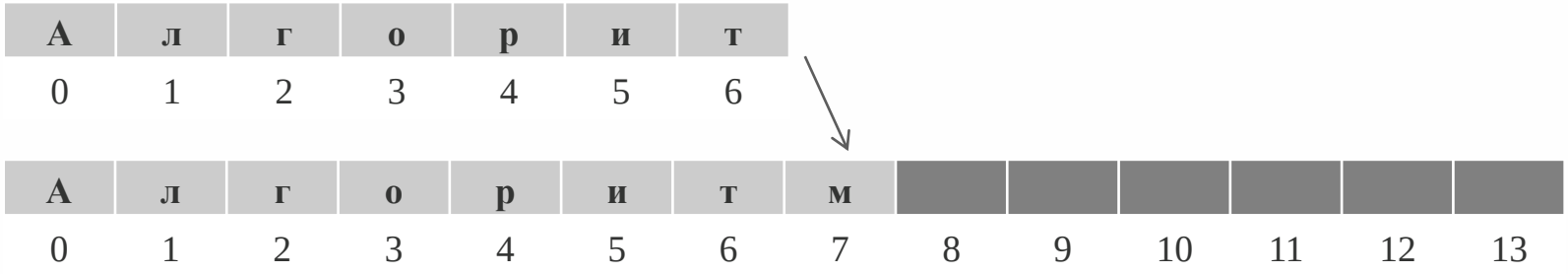
Буфер имеет некоторый запас для возможности добавления новых элементов.

А	л	г	о	р	и	т	м	ы	!				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Динамический массив. Реализация.

Если буфер закончился, то при добавлении нового элемента:

- выделим новый буфер, больший исходного;
- скопируем содержимое старого буфера в новый;
- добавим новый элемент.



Динамический массив. Оценка.

Время работы Add?

- В лучшем случае = $O(1)$
- В худшем случае = $O(n)$
- В среднем?

Рассмотрим несколько операций добавления и оценим среднее время в контексте последовательности операций.

Амортизационная (учетная) оценка

Пусть S(n) – время выполнения последовательности всех n операций в наихудшем случае.
Амортизированным (учетным) временем AC(n) называется среднее время, приходящееся на одну операцию

$$AC(n) = {}^{S(n)}/_{n}$$

Амортизационная (учетная) оценка

Пусть S(n) – время выполнения последовательности всех n операций в наихудшем случае.
Амортизированным (учетным) временем AC(n) называется среднее время, приходящееся на одну операцию

$$AC(n) = {}^{S(n)}/_{n}$$

Учетная оценка времени добавления в дин. массив

Утверждение. Пусть в реализации динамического массива буфер удваивается.

Тогда амортизированное время работы функции Add составляет $O(1)$.

(б/д)

Пример неэффективного удаления

Неэффективное удаление: уменьшаем размер буфера в два раза, если количество элементов меньше половины.

Амортизированное время операций более не будет O(1).

Пример. Массив из 16 элементов (полный буфер).

Последовательность:

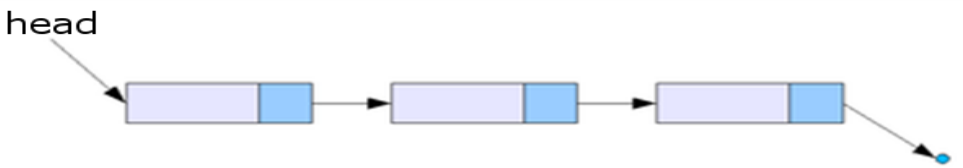
Add, Delete, Add, Delete, Add, Delete,...

Каждая выполняется за O(n).

Списки

Односвязный список

```
Узел = {
  Данные,
  Указатель на следующий узел,
}
Список = { Head }
```

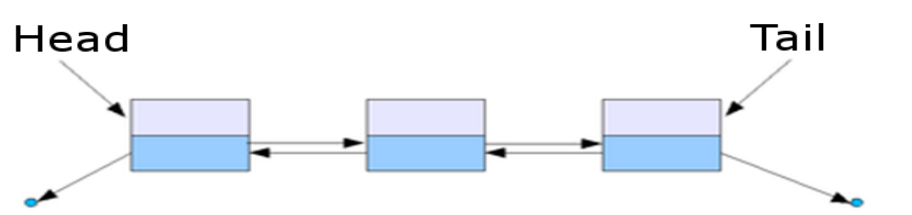


В stl: **std::forward_list**

https://en.cppreference.com/w/cpp/container/forward_list

Двусвязный список

```
Узел = {
  Данные,
  Указатель на следующий узел,
  Указатель на предыдущий узел,
}
Список = { Head, Tail }
```



В stl: **std::list**

https://en.cppreference.com/w/cpp/container/list

Сравнение с массивами

Недостатки списков:

- Нет доступа по индексу.
- Расходуется доп. память.
- Узлы могут располагаться в памяти разреженно.

Преимущества списков:

- Быстрая вставка узла.
- Быстрое удаление узла.

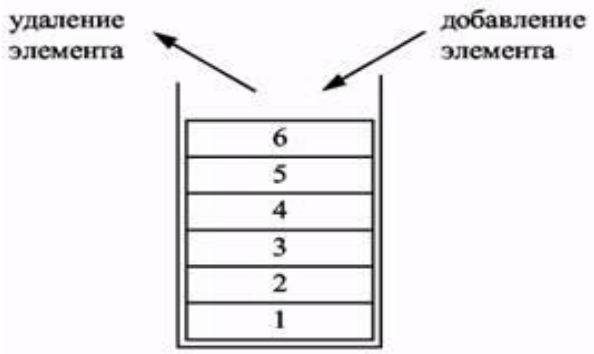
Стек, очередь, дек

Стек

Стек – список элементов, организованный по принципу LIFO = Last In First Out.

Методы:

- push
- pop – извлечение элемента, добавленного последним.
- top – ссылка на элемент, добавленный последним.



Стек в stl

Стек в stl является адаптером над контейнером

https://en.cppreference.com/w/cpp/container/stack

От контейнера требуется наличие методов

- back()
- push_back()
- pop_back()

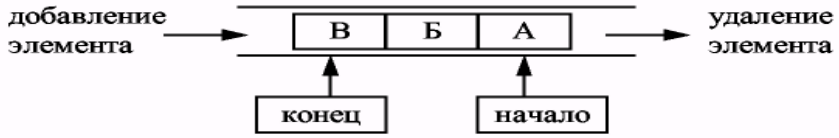
```
template<class T, class Container = std::deque<T>> class stack;
```

Очередь

Очередь – список элементов, организованный по принципу FIFO = First In First Out.

Методы:

- push – вставка в конец.
- pop – извлечение из начала.
- front – доступ к первому элементу.
- back – доступ к последнему элементу.



Очередь в stl

Очередь в stl также является адаптером над контейнером

https://en.cppreference.com/w/cpp/container/queue

От контейнера требуется наличие методов

- back()
- front()
- push_back()
- pop_front()

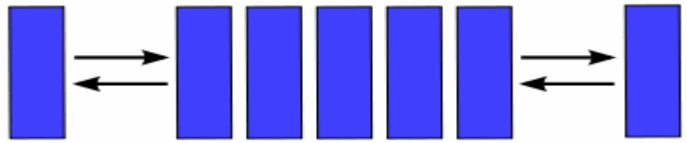
```
template<class T, class Container = std::deque<T>> class queue;
```

Дек

Дек (двусвязная очередь) – список элементов,
в котором элементы можно добавлять и удалять как в начало, так и в конец.

Методы:

- front,
- back,
- push_back,
- pop_back,
- push_front,
- pop_front.



Дек в stl

Дек в stl является полноценным контейнером с доступом по индексу

https://en.cppreference.com/w/cpp/container/deque

Плюсы в сравнении с vector:

- Константная вставка в начало и в конец всегда.
- Ссылки и итераторы на еще не удаленные элементы в деке не инвалидируются.
- Автоматически расширяется и освобождает память при добавлении/удалении элементов.

Минусы:

- Использует довольно много памяти, когда элементов в деке еще мало.

```
template<class T, class Allocator = std::allocator<T>> class deque;
```

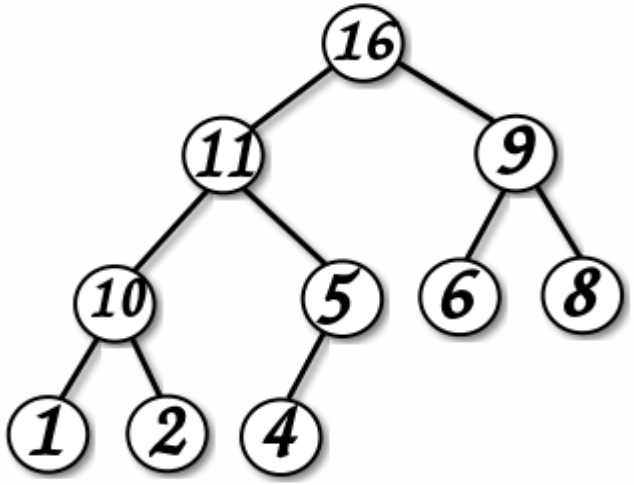
Куча

Куча

Куча (пирамида) – почти полное двоичное дерево, для которого выполнены три условия:

- Значение в любой вершине не меньше, чем значения её потомков (max-heap).
- Глубина листьев (расстояние до корня) отличается не более чем на один.
- Последний слой заполняется слева направо.

Глубина кучи = $O(\log n)$, где n – количество элементов.



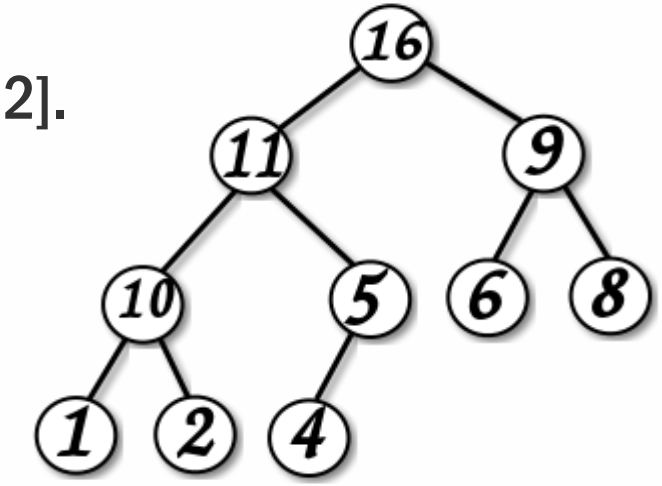
Хранение в массиве

Последовательно храним все элементы по слоям.

Индексация:

- A[0] – элемент в корне,
- потомки элемента A[i] – элементы A[2i + 1] и A[2i + 2].
- предок элемента A[i] – элемент A[(i – 1)/2].

16	11	9	10	5	6	8	1	2	4



Восстановление свойств

Если в куче изменяется один из элементов, то она может перестать удовлетворять свойству упорядоченности.

Для восстановления этого свойства служат две процедуры SiftUp и SiftDown. Обе работают за $O(\log n)$.

SiftDown спускает элемент, который меньше дочерних.
Если i-й элемент больше, чем его сыновья, всё поддерево уже является кучей, и делать ничего не надо. В противном случае меняем местами i-й элемент с наибольшим из его сыновей, после чего выполняем SiftDown для этого сына.

SiftUp поднимает элемент, который больше родительского.
Если элемент больше отца, меняет местами его с отцом. Если после этого отец больше деда, меняет местами отца с дедом, и так далее.

Построение кучи (BuildHeap)

Задача. Дан массив. Передвинуть в нем элементы так, чтобы массив стал кучей.

Решение. Если выполнить Sift Down для всех элементов массива A, начиная с последнего и кончая первым, он станет кучей.

Можно листовые пропустить.

Время работы O(n). (б/д)

```
// Построение кучи.
void BuildHeap(std::vector<int>& arr, int i) {
    for (int i = arr.Size() / 2 - 1; i >= 0; --i) {
        SiftDown(arr, i);
    }
}
```

Добавление элемента

1. Добавим элемент в конец кучи.
2. Восстановим свойство упорядоченности, проталкивая элемент наверх с помощью SiftUp.

Время работы – $O(\log n)$, если буфер для кучи позволяет добавить элемент без реаллокации.

```
// Добавление элемента.
void Add(std::vector<int>& arr, int element) {
    arr.push_back(element);
    SiftUp(arr, arr.Size() - 1);
}
```

Извлечение максимума

Максимальный элемент располагается в корне. Для его извлечения:

1. Сохраним значение корневого элемента для возврата.
2. Скопируем последний элемент в корень, удалим последний элемент.
3. Вызовем SiftDown для корня.
4. Возвратим сохраненный корневой элемент.

Время работы – $O(\log n)$.

Очередь с приоритетом

Интерфейс очереди с приоритетом:

- push(T&& e) – добавление элемента
- pop() – извлечение максимального элемента
- const T& top() – доступ к максимальному элементу

Реализуется кучей.

Очередь с приоритетом в stl

Функции в stl для работы с кучей:

- std::is_heap
- std::pop_heap
- std::make_heap
- std::push_heap
- std::sort_heap

```
template<class RandomIt>
void make_heap(RandomIt first, RandomIt last);
```

@stepa_ma (телеграмм)

smatsk@yandex.ru

https://data.mail.ru/profile/s.matskevich

Степан Мацкевич

академия больших данных

mail.ru group

made

Вопросы???