

Информатика (основы программирования)

Лекция 6.

Алгоритмы обработки массивов

Автор: Бабалова И.Ф.

Доцент, каф.12

Массивы

- **Массив** – это набор элементов одного типа, доступ к которым обеспечивается по номеру (индексу).
- **Массивы** - это абстрактная (логическая) структура данных, в каждом элементе которого могут быть как числа разных форматов, так и структуры, состоящие из множества полей данных. Строки – это тоже массивы, но символы записываются в памяти одним или двумя байтами, в зависимости от их физического представления в компьютере. Каждый элемент массива описывается своим именем и индексом, то есть своим номером в последовательности значений.

Основные действия с массивами данных

1. Объявление массива в программе
2. Выделение памяти для хранения массива
3. Поиск заданного элемента в массиве
4. Удаление некоторого элемента из массива
5. Добавление некоторого элемента в массив
6. Сортировка элементов массива по заданному признаку

Стандартного типа массив в языке С нет


Каждый элемент массива должен занимать в памяти компьютера одинаковое количество байтов.

Удаление элемента из массива (1)

- Алгоритмическое решение этой задачи начинается с поиска удаляемого значения. Удаляем a_4 .

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	x
9	1	12	42	17	8	4	15	17

- Удаление найденного элемента требует сдвига влево элементов массива и уменьшения размерности массива.



a_0	a_1	a_2	a_3	a_4	a_5	a_6
9	1	12	42	8	4	15

- В массиве на месте удалённого элемента должен находиться следующий элемент массива.
- Размерность массива должна быть уменьшена.

Удаление элемента из массива (2)




Добавление нового элемента массива на заданное место (1)

- Алгоритм добавления элемента в массив должен учитывать свойства массива – его упорядоченность или неупорядоченность.
- Рассмотрим вставку элемента в упорядоченный массив. Поиск места, куда требуется вставить новое значение – это обычный поиск, в котором результатом будет не значение, а номер места, на которое необходимо вставить новое значение.

Добавление нового элемента массива на заданное место (2)

- Например, вставить значение $x=14$ в упорядоченный массив. Обычным последовательным поиском находим, что новый элемент должен быть в массиве между a_4 и a_5 . Следовательно, все элементы, начиная с a_5 , должны быть сдвинуты вправо. Чтобы физически реализовать увеличение размерности массива, надо увеличить размер выделенной памяти.



a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	x
1	4	8	9	12	15	17	42	14

- Оценка временных характеристик вставки и удаления элемента массива определяется временем поиска и временем сдвига. В наихудшем случае $t \sim 2N$.

Алгоритмы поиска в массивах данных любого типа (1)

- Для нахождения информации в неупорядоченном массиве требуется **последовательный поиск**, начинающийся с первого элемента и заканчивающийся при обнаружении подходящих данных, либо при достижении конца массива. Оценка времени поиска в этом варианте поиска всегда порядка $\sim O(N)$, так как вероятность найти требуемое значение $P_i = \frac{1}{N}$.
- Для упорядоченного массива применим, как последовательный поиск данных, так и другие алгоритмы, позволяющие существенно ускорить поиск.

Алгоритмы поиска в массивах данных любого типа (2)

- Хорошим алгоритмом поиска для упорядоченных массивов считается **алгоритм бинарного поиска**. Он даёт оценку скорости поиска $\sim O(\log_2 N)$.
- В набор стандартной библиотеки языка C входит функция поиска **bsearch()**.
- Как и в случае сортировки, функции общего назначения иногда совсем не эффективны при использовании в критических ситуациях из-за накладных расходов, связанных с их обобщением. Кроме того, функцию **bsearch()** невозможно применить к неупорядоченным данным.

Последовательный поиск заданного значения в массиве (1)

- Рассмотрим массив, который не был упорядочен. Произвольные значения записывались в массив по мере их поступления. Для нахождения информации в неупорядоченном массиве самым простым алгоритмом является последовательный поиск.
- Просматриваем последовательность с первого элемента.
- Вероятность нахождения требуемого значения $P_i = \frac{1}{N}$. Соответственно, математическое ожидание $M_x = \frac{n}{2}$. Количество сравнений в худшем случае равно $2n$.
- Оценка времени поиска в этом варианте поиска получается порядка $\sim O(n)$.

a_0	a_1	a_2	...	a_{n-2}	a_{n-1}	x
3	9	5	...	8	7	8

- Перебираются последовательно все n элементов до получения истинного значения результата сравнения $a_i = x$.
- Количество сравнений $\sim n$

Последовательный поиск заданного значения в массиве (2)

- Можно уменьшить количество сравнений, дополнив исходный массив искомым элементом. Мы увеличиваем требуемую память, но количество сравнений уменьшаем в два раза. В просмотре элементов последовательности исключаем сравнение индексов элементов массива.

- Например, надо в массиве найти число 17.

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	x
9	1	12	42	17	8	4	15	17

- Сравниваем только значения элементов массива с последним элементом. Скорость выполнения поиска практически удваивается. Неупорядоченные данные обрабатывать достаточно трудно. Если массив упорядочен, то алгоритмические решения позволяют сократить время поиска

Алгоритмы поиска данных в массиве

- Алгоритм бинарного поиска

a_0	a_1	a_2	a_3	a_4	a_5	a_6
-1	2	5	7	8	10	11

left = 1 right = 7

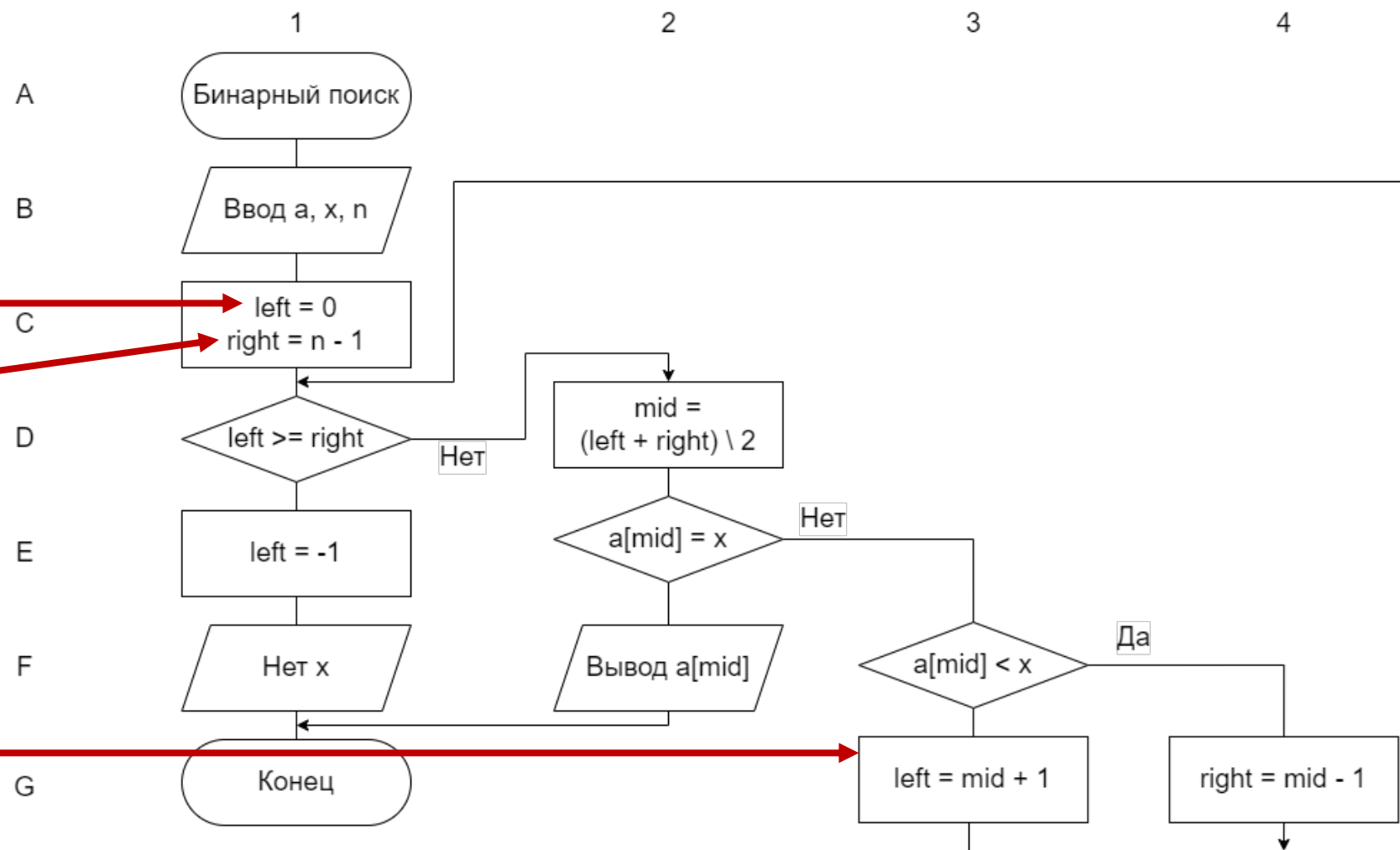
$$\text{middle} = (\text{left} + \text{right}) \text{ div } 2 = (1 + 7) \text{ div } 2 = 4$$

$$i = \text{middle}$$

- Сравниваем искомое значение x со средним элементом. Если $x <$ среднего, тогда right делаем равным middle , иначе $\text{left} = \text{middle}$
- Необходима проверка величины индекса чтобы левый индекс не стал больше правого индекса.
- **Количество сравнений $\sim \log_2(n)$**
- **Для любого варианта поиска обязателен ответ об отсутствии искомого объекта поиска.**

Алгоритм бинарного поиска

- Начало последовательности
- Конец последовательности
- Новый конец последовательности справа или слева



Пример записи на языке функции для бинарного поиска

// Алгоритм бинарного поиска

```
int binSearch(int *a, int n, int x) {
    int l = 0;
    int r = n - 1;
    while (l <= r) {
        int m = (l + r) / 2;
        if (a[m] == x) {
            return m;
        }
        if (a[m] < x) {
            l = m + 1;
        }
        else {
            r = m - 1;
        }
    }
    return -1;
}
```

// Создание массива из случайных чисел

```
int *ptr = malloc(n * sizeof(int))
for (int i = 0; i < n; ++i) {
    ptr[i] = rand();
}
return ptr;
```

- **Примечание.** Для инициализации датчика случайных чисел используется стандартная функция:

```
srand(time(NULL));
```

```
int res = binSearch(array, smas, rand());
```

// Случайное число выбираем для поиска

Функции управления динамической памятью

- Функции управления динамической памятью дают возможность легко управлять памятью. Они являются функциями библиотеки общего назначения.
- Объявления этих функций находятся в заголовочном файле *stdlib.h*, поэтому в программах, использующих эти функции, необходимо включать следующую директиву препроцессора:
`#include <stdlib.h>`

Выделение и освобождение памяти

- В библиотеке содержится 4 функции для работы с динамической памятью.

```
void * malloc(size_t size);
```

```
void * realloc(void *memblock, size_t size);
```

```
void * calloc(size_t number, size_t size);
```

```
void free(void *memblock); // Освобождение памяти
```


malloc

```
void * malloc(size_t size);
```

Функция выделяет память размером `size` байт и возвращает адрес начала области памяти. При присвоении указателю происходит явное преобразование типа. Выравнивание границы происходит автоматически, но выделенная память не инициализируется нулями.

realloc

```
void * realloc(void *memblock, size_t size);
```

Функция делает размер блока по указателю `memblock` равным `size` байтов и возвращает указатель на блок, возможно перемещенный. В случае перемещения блока данные из исходного блока копируются в новый блок, освобождая исходный блок.

calloc

```
void * calloc(size_t number, size_t size);
```

Функция выделяет блок, который вмещает **number** элементов размером **size** байтов каждый, возвращает адрес начала области памяти. Выделенная память инициализируется нулями.

Выделение памяти

- У каждого блока памяти есть байты, где хранятся длина блока и признак того, занят он или свободен.
- Когда функциям **malloc**, **calloc** и **realloc** нужно изменить границу выделяемой памяти, они обычно делают это с запасом. Если будущим запросам хватает выделенного пространства, они не используют системный вызов. Так библиотечные функции минимизируют число выполнения системных вызовов.
- Если вызов функции закончился неуспешно, то она возвращает значение **NULL**

free

```
void free(void *memblock);
```

- Функция освобождает блок памяти по указателю *memblock и делает его доступным для последующих выделений памяти.
- Указатель должен указывать на блок, полученный ранее вызовом malloc, calloc или realloc
- Функция free() никогда не возвращает память назад ядру. Вместо этого она устанавливает признак того, что блок свободен. Будущие запросы выделения памяти ищут среди всех свободных блоков блок подходящего размера, прежде чем использовать системный вызов увеличения памяти. Смежные свободные блоки сливаются во время поиска для их более эффективного нового варианта использования.

Ввод элементов массива

// 1. Ввод длины массива

```
int arrLen = 0;
```

```
printf("Enter arrLen = ");
```

```
scanf("%d", &arrLen); // вводим конкретную длину массива
```

// 2. Выделение памяти для размещения элементов массива:

```
int *ptr = malloc(arrLen * sizeof(int)); // может быть calloc
```

// 3. Заполнение массива

```
int *array = genArray(arrLen);
```

// 4. Печать массива

```
void printArray(int *array, int size) {
```

```
    for (int i = 0; i < size; ++i) {
```

```
        printf("%d ", array[i]);
```

```
    }
```

```
    printf("\n");
```

```
}
```

Thank you