

Списки. Деревья. Поиск.

Лекция 4

План лекции

1. Структура данных «список».
2. Деревья. Обход деревьев.
3. Интерфейс абстракции *приоритетная очередь*
4. Бинарная куча
5. HeapSort
6. Задача поиска. Абстракция поиска.
7. Последовательный поиск.
8. Распределяющий поиск. Поиск с использованием свойств ключа.
9. Поиск с сужением зоны.
10. Сравнительный анализ методов поиска.

Структура данных «список».

Списки

Список — структура данных, которая реализует абстракции:

- ▶ `insertAfter` — добавление элемента за текущим.
- ▶ `insertBefore` — добавление элемента перед текущим.
- ▶ `insertToFront` — добавление элемента в начало списка.
- ▶ `insertToBack` — добавление элемента в конец списка
- ▶ `find` — поиск элемента
- ▶ `size` — определение количества элементов

Списки: реализация

Для реализации списков обычно требуется явное использование указателей.

```
struct linkedListNode { // Одна из реализаций
    someType data; // Таких полей - произвольное число
    linkedListNode *next; // Связь со следующим
};
```

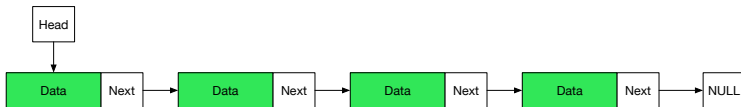
Внутренние операции создания элементов — через malloc, calloc, new.

```
...
linkedListNode *item = new linkedListNode();
item->data = myData;
...
```

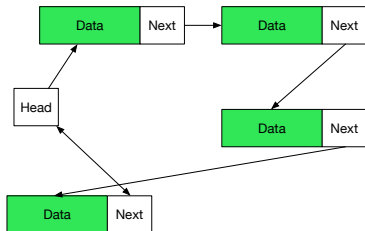
Списки: представления

Различные варианты представлений:

В линейном виде:



В виде кольца:



Списки: сложность

Стоимость операций:

Операция	Время	Память
<code>insertAfter</code>	$O(1)$	$O(1)$
<code>insertBefore</code>	$O(N)$	$O(1)$
<code>insertToFront</code>	$O(1)$	$O(1)$
<code>insertToEnd</code>	$O(N)$	$O(1)$
<code>find</code>	$O(N)$	$O(1)$
<code>size</code>	$O(N)$	$O(1)$

Списки: создание

```
typedef double myData;  
  
linkedListNode *list_createNode(myData data) {  
    linkedListNode *ret = new linkedListNode();  
    ret->data = data;  
    ret->next = NULL;  
    return ret;  
}
```

Создание списка из одного элемента:

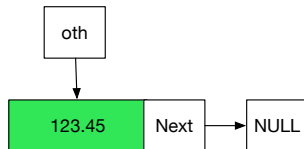
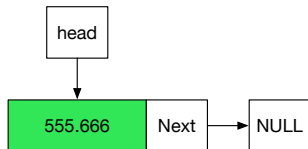
```
linkedListNode *head = list_createNode(555.666);
```



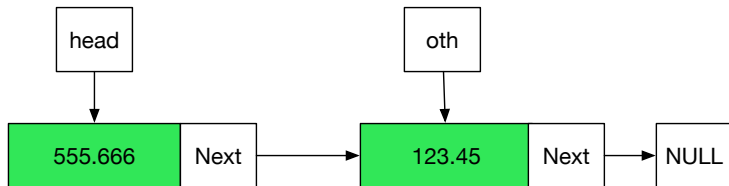
Списки: добавление

Добавление элемента в хвост списка:

```
linkedListNode *oth = list_createNode(123.45);
```

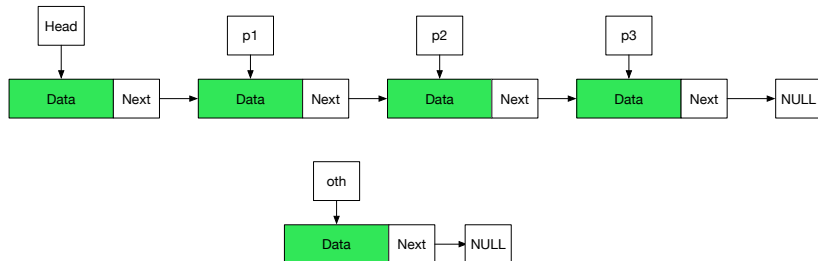


```
head->next = oth;
```



Списки: добавление

Добавление элемента в хвост списка, состоящего из нескольких элементов:

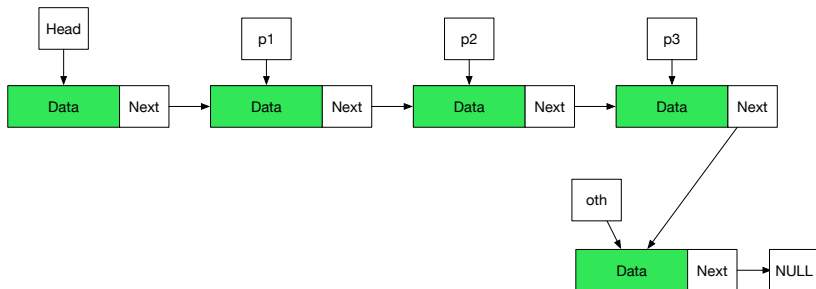


Проход по элементам до нужного (traversal, walk):

```
linkedListNode *ptr = head;
while (ptr->next != NULL) {
    ptr = ptr->next;
}
ptr->next = oth;
```

Списки: добавление

Заключительное состояние после вставки.

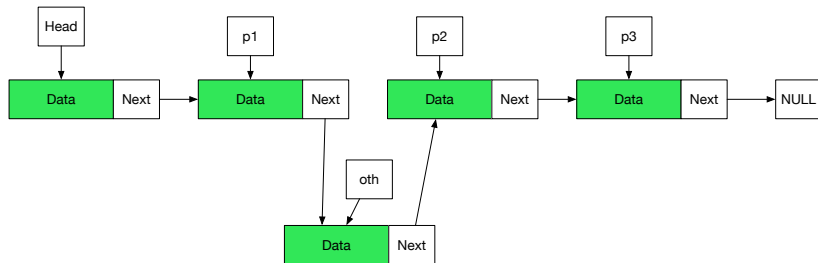


Сложность операции — $O(N)$

Списки: добавление

Вставка `insertAfter` за конкретным элементом `p1` примитивна.

```
// вставка ЗА элементом p1  
oth->next = p1->next;  
p1->next = oth;
```



Списки: добавление

Вставка ПЕРЕД известным элементом сложнее:

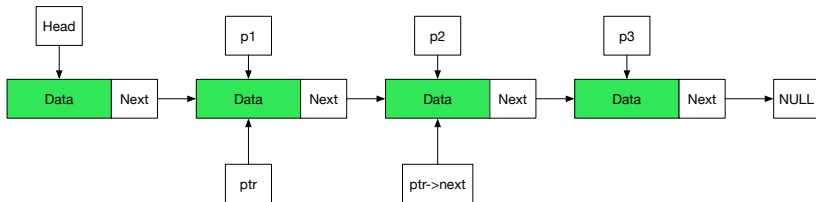
```
// вставка ПЕРЕД элементом p2
linkedListNode *ptr = head;
while (ptr->next != p2) {
    ptr = ptr->next;
}
oth->next = p2;
ptr->next = oth;
```

Списки: удаление

Удаление элемента p2 — непростая операция.

- ▶ Нужно найти удаляемый элемент и его предшественника:

```
// поиск элемента p2
linkedListNode *ptr = head;
while (ptr->next != p2) {
    ptr = ptr->next;
}
// ptr - предшественник p2
```

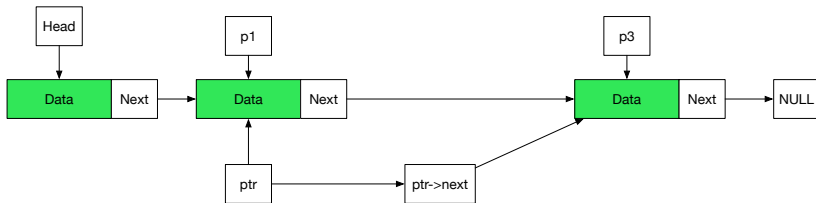


Списки: удаление

Удаление элемента из списка.

- ▶ Переместить указатели.

```
ptr->next = p2->next;  
delete p2;
```



Списки: размер

Операция size — две возможности:

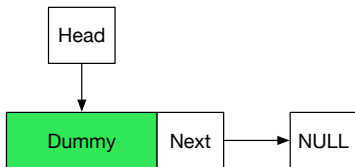
- ▶ Через операцию walk до NULL:

```
linkedListNode *ptr = head;
int size = 0;
while (ptr != NULL) {
    ptr = ptr->next;
    size++;
}
return size;
```

- ▶ Вести размер списка в структуре данных. Потребуется изменить все методы вставки/удаления.

Списки: альтернативное представление

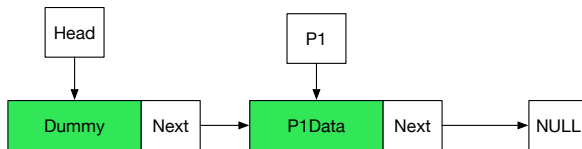
- ▶ При нашем представлении требуется всегда различать, работаем ли мы с головой списка или с другим элементом. При смене головы списка приходится заменять все указатели в программе.
- ▶ Существуют различные способы представления списков.
- ▶ Для абстрактного типа данных удобнее иметь список с неизменной головой.



- ▶ Это — пустой список, содержащий ноль элементов.

Списки: альтернативное представление

- ▶ Список, состоящий из одного элемента p1.



- ▶ Такое представление упрощает реализацию за счёт одного дополнительного элемента.

Списки: сложность

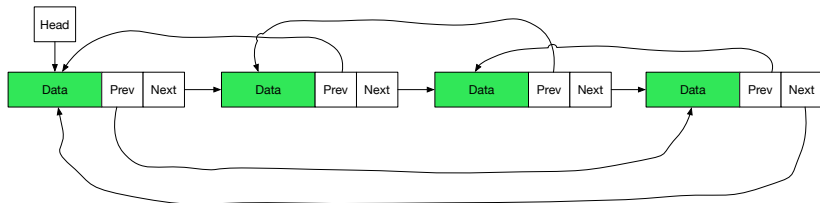
Ещё раз оценим сложность основных операций:

- ▶ Вставка элемента в голову списка — $O(1)$
- ▶ Вставка элемента в хвост списка — $O(N)$
- ▶ Поиск элемента — $O(N)$
- ▶ Удаление известного элемента — $O(N)$
- ▶ Вставка элемента ЗА известным — $O(1)$
- ▶ Вставка элемента ПЕРЕД известным — $O(N)$

Можно ли улучшить худшие случаи?

Списки: двусвязные списки

Худшие случаи можно улучшить, если заметить, что операция «слева-направо» более эффективно реализуется, чем «справа-налево» и восстановить симметрию.



Списки: двусвязные списки: сложность

Для двусвязного списка сложность такая:

- ▶ Вставка элемента в голову списка — $O(1)$
- ▶ Вставка элемента в хвост списка — $O(1)$
- ▶ Поиск элемента — $O(N)$
- ▶ Удаление известного элемента — $O(1)$
- ▶ Вставка элемента ЗА известным — $O(1)$
- ▶ Вставка элемента ПЕРЕД известным — $O(1)$

Списки: двусвязные списки: вставка

Операции вставки и удаления усложняются:

Для вставки элемента `oth` после элемента `p1`:

1. Подготавливаем вставляемый элемент.
2. Сохраняем указатель `s = p1->next`
3. `oth->prev = p1`
4. `oth->next = s`
5. `s->prev = oth`
6. `p1->next = oth`

Списки: двусвязные списки: удаление

Для удаления элемента $p1$ из списка:

1. Сохраняем указатель $s = p1 \rightarrow next$
2. $s \rightarrow prev = p1 \rightarrow prev$
3. $p1 \rightarrow prev \rightarrow next = s$
4. Освобождаем память элемента $p1$

Списки: использование

Когда используют списки? Для представления быстро изменяющегося множества объектов.

- ▶ Пример из математического моделирования: множество машин при моделировании автодороги. Они:
 - ▶ появляются на дороге (вставка в начало списка)
 - ▶ покидают дорогу (удаление из конца списка)
 - ▶ перестраиваются с полосы на полосу (удаление из одного списка и вставка в другой)
- ▶ Пример из системного программирования: представление множества исполняющихся процессов, претендующих на процессор. Представление множества запросов ввода/вывода. Важная особенность: лёгкий одновременный доступ от множества процессоров.

Списки: использование: менеджер памяти

Одна из реализаций выделения/освобождения динамической памяти (`calloc/new/free/delete`).

- ▶ Вначале свободная память описывается пустым списком.
- ▶ Память в операционной системе выделяется *страницами*.
- ▶ При заказе памяти:
 - ▶ если есть достаточный свободный блок памяти, то он разбивается на два подблока, один из которых помечается занятым и возвращается в программу;
 - ▶ если нет достаточной свободной памяти, запрашивается несколько страниц у системы и создаётся новый элемент в конце списка (или изменяется старый).

На практике применяется несколько списков, в зависимости от размера заявки.

Связные списки

- ▶ Сложность операций:
 - ▶ *find* — $O(N)$
 - ▶ *insert* — $O(1)$
 - ▶ *remove* — $O(1)$

Структура данных «дерево».

Деревья: особенности

Основная особенность деревьев — наличие нескольких наследников.

По максимальному числу наследников деревья делятся на

- ▶ двоичные (бинарные)
- ▶ троичные (тернарные)
- ▶ N-ричные

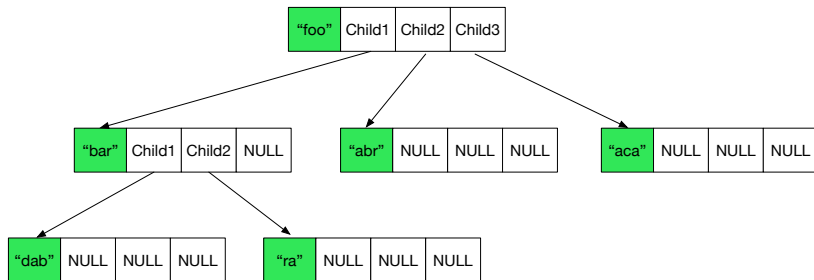
```
struct tree {  
    tree *children[3];  
    myType data;  
    ...  
};
```

Деревья: соглашения

- ▶ Любое N -ричное дерево может представлять деревья меньшего порядка.
- ▶ Соглашение: если наследника нет, соответствующий указатель равен `NULL`.
- ▶ Деревья 1-ричного порядка существуют (списки).

Деревья: троичное дерево

Пример дерева троичного дерева или дерева 3-порядка.



Деревья: классификация

- ▶ Условно все элементы дерева делят на две группы:
 - ▶ **Вершины**, не содержащие связей с потомками.
 - ▶ **Узлы**, содержащие связи с потомками.
- ▶ Второй вариант — все элементы дерева называют **узлами**, а **вершина** — частный случай **узла**, **терминальный узел**.
- ▶ Ещё термины:
 - ▶ **Родитель (parent)**
 - ▶ **Дети (children)**
 - ▶ **Братья (sibs)**
 - ▶ **Глубина (depth)**

$$D_{node} = D_{parent} + 1$$

Деревья: создание узла

- Добавим метод создания элемента (узла) дерева:

```
struct tree {  
    string data;  
    tree *child[3];  
    tree(string init) { // Конструктор  
        child[0] = child[1] = child[2] = NULL;  
        data = init;  
    }  
    ...  
};
```


Деревья: пример построения

- ▶ Дерево на примере строится, например, так:

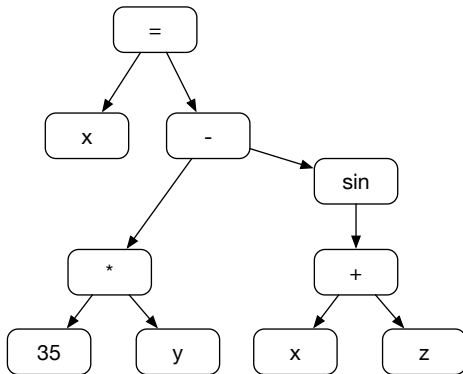
```
tree *root = new tree("foo");  
root->child[0] = new tree("bar");  
root->child[1] = new tree("abr");  
root->child[2] = new tree("aca");  
root->child[0]->child[0] = new tree("dab");  
root->child[0]->child[1] = new tree("ra");
```

Деревья: пример использования

Использование деревьев:

- Для представления выражений в языках программирования.

$x = 35y - \sin(x+z);$



Деревья: вариант представления

- ▶ Вариант хранения N -дерева: массив.
 - ▶ Все узлы нумеруются, начиная с 0.
 - ▶ Для узла с номером K номера детей

$$K \cdot N + 1 \dots K \cdot N + N$$

- ▶ Для 2-дерева корневой узел = 0, дети 1-го уровня ($Depth = 2$) 1 и 2 ...

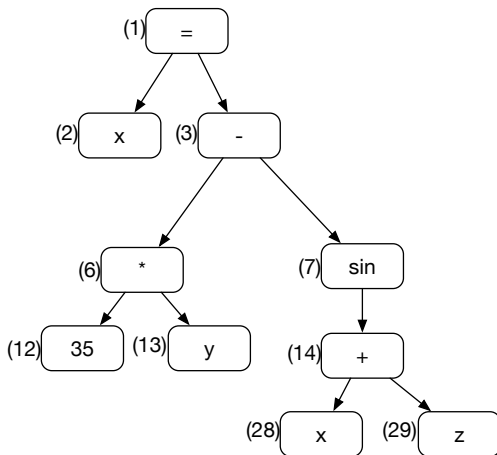
Часто удобнее нумеровать с 1. Тогда дети нумеруются

$$[K \cdot N \dots (K + 1) \cdot N)$$

Деревья: нумерация

Для дерева выражений нумерация будет такой:

$$x = 35y - \sin(x+z);$$



Деревья: альтернативное представление

- ▶ Представление в виде массива (фрагмент):

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
=	x	-			*	sin					35	y	+			

- ▶ Количество памяти = $O(2^{D_{max}})$
- ▶ Невыгодно при разреженном дереве

Деревья: обход

- ▶ Алгоритмы работы с деревьями часто рекурсивны.
- ▶ Всего существует $6=3!$ способов обхода бинарного дерева.
- ▶ На практике применяют четыре основных варианта рекурсивного обхода:
 - ▶ Прямой
 - ▶ Симметричный
 - ▶ Обратный
 - ▶ Обратно симметричный

Деревья: обход

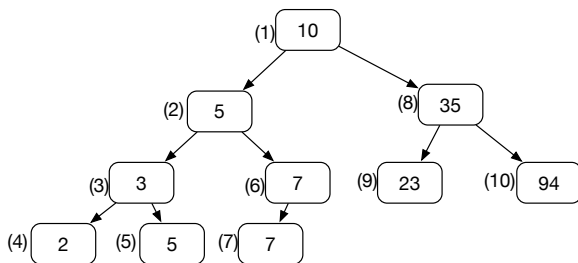
Бинарное дерево.

```
struct tree {  
    string data;  
    tree *left;  
    tree *right;  
    tree(string init) { // Конструктор  
        left = right = NULL;  
        data = init;  
    }  
};
```

Деревья: обход

Прямой способ обхода.

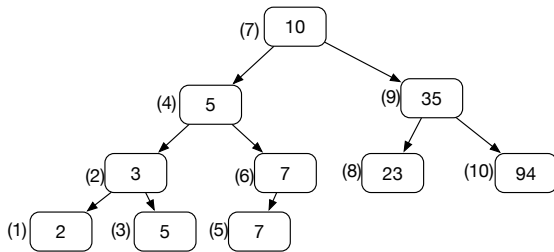
```
void walk(tree *t) {  
    work(t);  
    if (t->left != NULL) walk(t->left);  
    if (t->right != NULL) walk(t->right);  
}
```



Деревья: обход

Симметричный способ обхода.

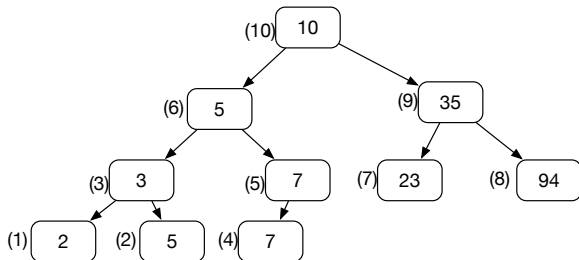
```
void walk(tree *t) {  
    if (t->left != NULL) walk(t->left);  
    work(t);  
    if (t->right != NULL) walk(t->right);  
}
```



Деревья: обход

Обратный способ обхода.

```
void walk(tree *t) {  
    if (t->left != NULL) walk(t->left);  
    if (t->right != NULL) walk(t->right);  
    work(t);  
}
```



Деревья: обход

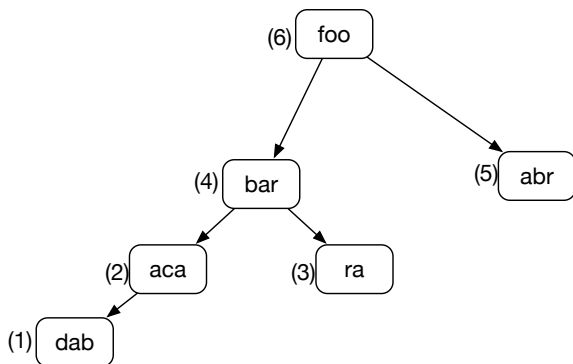
Функция обработки может быть параметром.

```
using walkFunction = void (*)(tree *);
void walk(tree *t, walkFunction wf) {
    if (t->left != NULL) walk(t->left, wf);
    if (t->right != NULL) walk(t->right, wf);
    wf(t);
}

void printData(tree *t) {
    printf("t[%p]='%s'\n", t, t->data.c_str());
}

int main() {
    tree *root = new tree("foo");
    root->left = new tree("bar");
    root->right = new tree("abr");
    root->left->left = new tree("aca");
    root->left->left->left = new tree("dab");
    root->left->right = new tree("ra");
    walk(root, printData);
}
```

Деревья: обход



```
t[0x7ff0e1c03290]='dab'  
t[0x7ff0e1c03260]='aca'  
t[0x7ff0e1c032d0]='ra'  
t[0x7ff0e1c03200]='bar'  
t[0x7ff0e1c03230]='abr'  
t[0x7ff0e1c031d0]='foo'
```

Деревья: обход

Вывод генеалогического дерева (обратно симметричный обход):

```
typedef void (*walkFunction)(tree *, int lev);

void walk(tree *t, walkFunction wf, int lev) {
    if (t->right != NULL) walk(t->right, wf, lev+1);
    wf(t, lev);
    if (t->left != NULL) walk(t->left, wf, lev+1);
}

void printData(tree *t, int lev) {
    for (int i = 0; i < lev; i++) {
        printf("  ");
    }
    printf("%s\n", t->data.c_str());
}

int main() {
    ...
    walk(root, printData, 0);
}
```

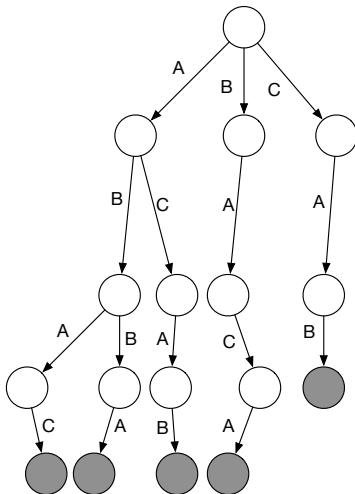
Деревья: обход

Вывод программы:

```
abr
foo
  ra
bar
  aca
  dab
```

Деревья: обход

- ▶ При использовании динамических структур данных для некоторых операций важно выбрать верный обход дерева.
- ▶ Вспомним префиксное дерево из второй лекции:



Деревья: обход

- ▶ Заказ памяти под поддеревья происходил динамически.
- ▶ Имелся узел, от которого шло построение дерева.
- ▶ Так как в данном дереве не хранится информация о том, кто является предком узла, корневой узел — центр всего построения.
- ▶ При операции освобождения памяти узла искажутся значения подузлов.

Деревья: динамическая память

- ▶ Напомним порядок выделения и уничтожения памяти в конструкторе и деструкторе:

```
struct node {  
    node *children[3];  
    bool  is_leaf;  
    node();  
    ~node();  
};
```

Деревья: динамическая память

► Конструктор:

```
node::node() {  
    children[0] = children[1] = children[2] = NULL;  
    is_leaf = false;  
}
```

1. Система выделяет память из «кучи», достаточную для хранения всех полей структуры.
2. После этого выполняется инициализация полей (написанный нами код).

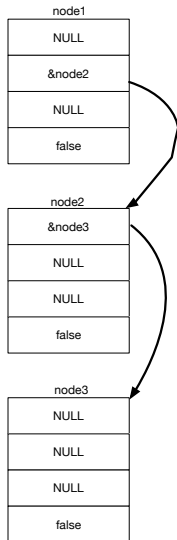
Деревья: динамическая память

► Деструктор:

```
node::~~node() {  
    printf("node destructor is called\n");  
}
```

1. Исполняется написанный нами код.
2. Система освобождает занятую память.
3. Обращение к освобождённой памяти приводит к ошибкам.

Деревья: динамическая память



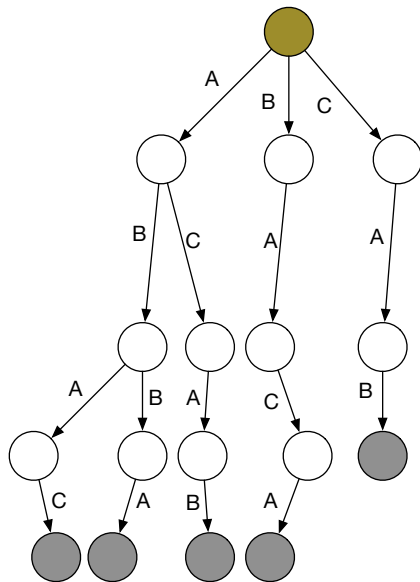
Деревья: динамическая память

- ▶ Удаление корневого узла приводит к тому, что остальные узлы останутся недоступны.
- ▶ Такие недоступные узлы называются *висячими ссылками* (*dangling pointers*)
- ▶ Ситуация, возникшая в программе называется *утечкой памяти* (*memory leak*)
- ▶ Чтобы не было утечки памяти, удаление узлов нужно производить с самого нижнего.

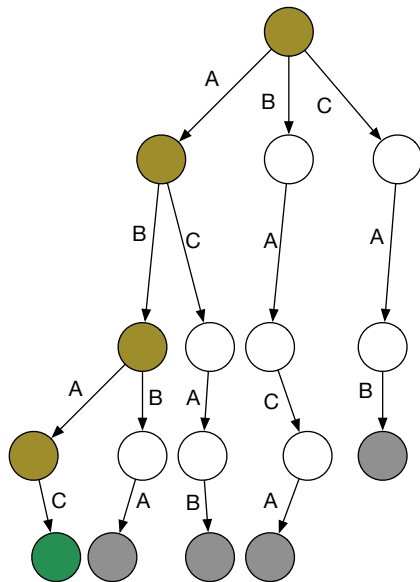
Деревья: освобождение памяти

```
void destroy(node *n) {  
    for (int i = 0; i < 3; i++) {  
        if (n->children[i] != NULL) {  
            destroy(n->children[i]);  
        }  
    }  
    delete n;  
}
```

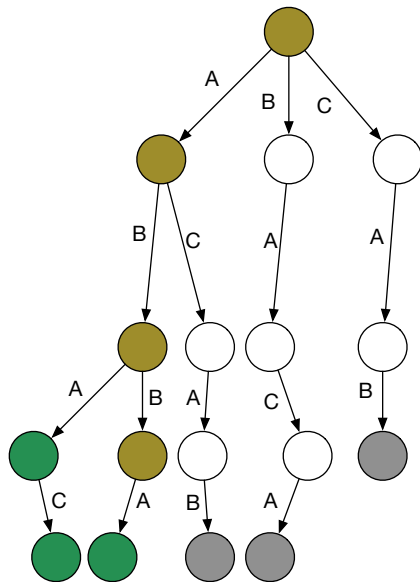
Деревья: освобождение памяти



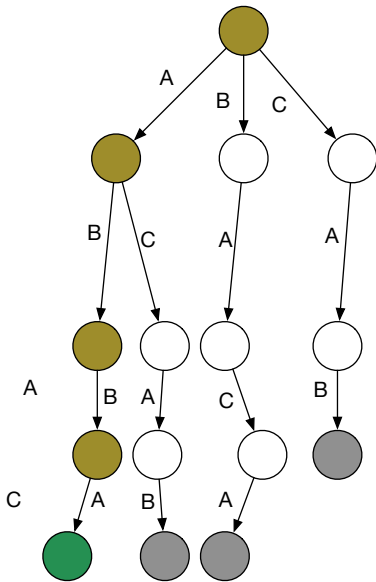
Деревья: освобождение памяти



Деревья: освобождение памяти



Деревья: освобождение памяти



Деревья: свойства

Свойства деревьев:

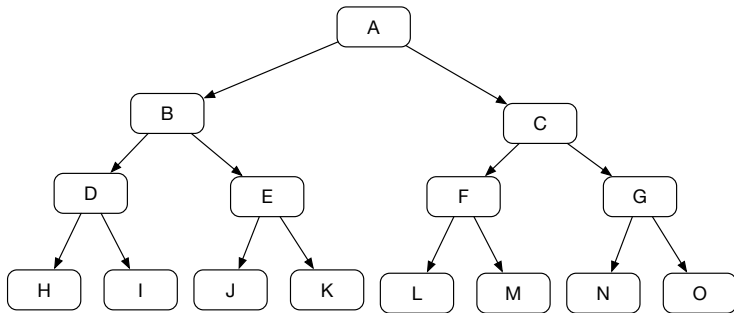
- ▶ Позволяют использовать быстро изменяющиеся структуры данных.
- ▶ Есть надежда, что операции вставки и удаления окажутся быстрыми (быстрее $O(N)$).
- ▶ Есть надежда, что операции поиска окажутся быстрыми (быстрее $O(N)$).

Полные бинарные деревья

Определение:

- ▶ **Полное бинарное дерево** T_H высоты H есть бинарное дерево, у которого путь от корня до любой вершины содержит ровно H рёбер, при этом у всех узлов дерева, не являющимися листьями, есть и правый, и левый потомок.

Полное бинарное дерево высоты 3.



Полные бинарные деревья

Рекурсивное определение:

- ▶ **Полное бинарное дерево** T_H высоты H есть бинарное дерево, у которого к корню прикреплены левое и правое поддеревья T_{H-1} высоты $H - 1$.
- ▶ По этому определению число узлов в дереве T_H есть $N = 2^{H+1} - 1$

$$H = \log_2(N + 1)$$

Абстракция *приоритетная* *очередь*

Приоритетная очередь

- ▶ Приоритетная очередь (priority queue) — очередь, элементы которой имеют приоритет.
- ▶ После вставки элемента очередь остаётся в упорядоченном по приоритету состоянии
- ▶ Первым извлекается наиболее приоритетный элемент (максимум или минимум).

Интерфейс абстракции:

- ▶ `insert` — добавление элемента из очереди
- ▶ `extractMin(Max)` — извлекает самый приоритетный элемент
- ▶ `fetchMin(Max)` — получает самый приоритетный элемент
- ▶ `increasePriority` — изменяет приоритет элемента
- ▶ `merge` — сливает очереди

Приоритетная очередь

Пример приоритетной очереди:

Значение (value)	Приоритет (priority)
Москва	12000000
Казань	1500000
Урюпинск	10000
Малиновка	200

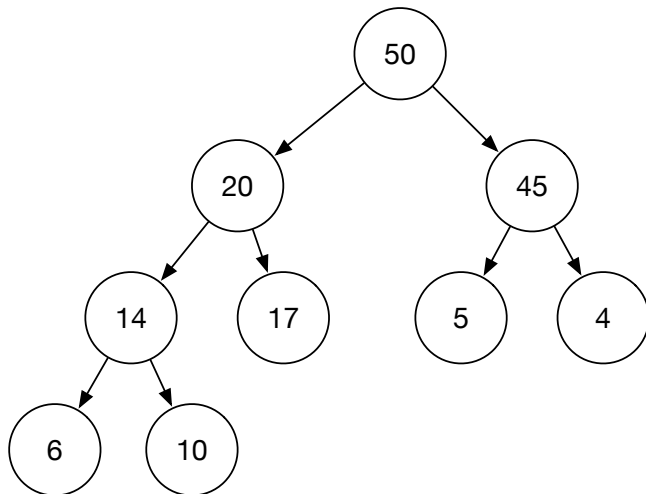
Приоритетная очередь:представление

- ▶ *Бинарная куча* — бинарное дерево, удовлетворяющее условиям:
 - ▶ Для любой вершины её приоритет не меньше (больше) приоритета потомков.
 - ▶ Дерево является правильным подмножеством полного бинарного.

Другое название — пирамида (heap)

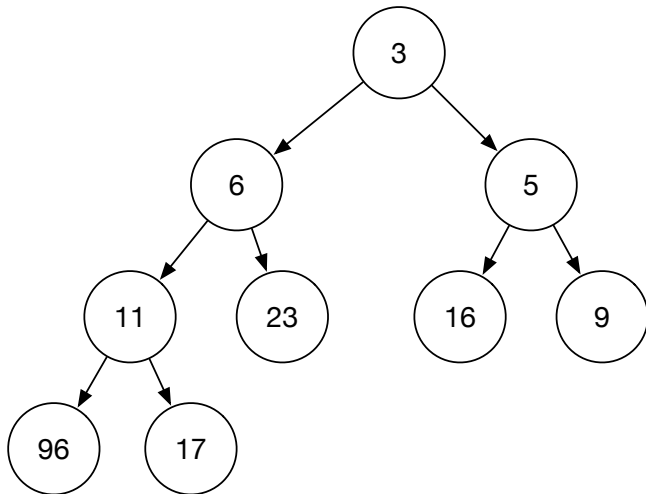
Приоритетная очередь

► Невозрастающая пирамида

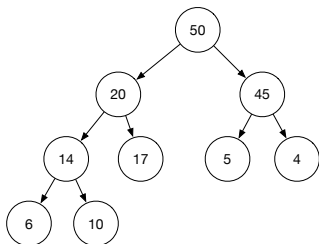


Приоритетная очередь

► Неубывающая пирамида



Приоритетная очередь:реализация



Хранение в виде массива с индексами от 1 до N :

50	20	45	14	17	5	4	6	10
----	----	----	----	----	---	---	---	----

- ▶ Индекс корня дерева всегда равен 1 — максимальный (минимальный) элемент
- ▶ Индекс родителя узла i равен $\lfloor \frac{i}{2} \rfloor$
- ▶ Индекс левого потомка узла i равен $2i$
- ▶ Индекс правого потомка узла i равен $2i + 1$

Приоритетная очередь

- Реализация на базе массива.

```
struct bhnode { // Узел
    string data;
    int priority;
};

struct binary_heap {
    bhnode *body;
    int      bodysize;
    int      numnodes;
    binary_heap(int maxsize);
    ...
};
```

Бинарная куча

► Создание бинарной кучи

```
binary_heap::binary_heap(int maxsize) {  
    body = new bhnode[maxsize+1];  
    bodysize = maxsize;  
    numnodes = 0;  
}
```

```
~binary_heap::binary_heap() {  
    delete body;  
}
```

```
void binary_heap::swap(int a, int b) {  
    bhnode temp = body[a];  
    body[a] = body[b];  
    body[b] = temp;  
}
```

$$T_{create} = O(N)$$

Бинарная куча: поиск минимума(максимума)

► Поиск в min-heap:

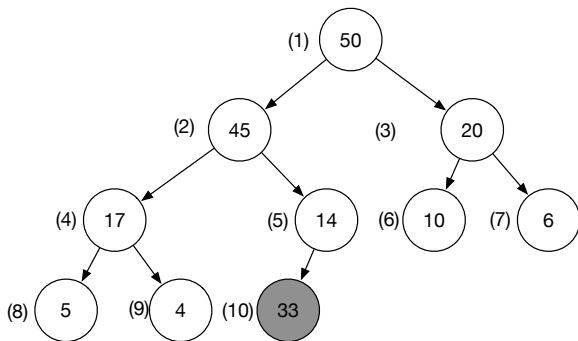
```
bhnode *binary_heap::fetchMin() {  
    return numnodes == 0? NULL : body + 1;  
}
```

$$T_{fetchMin} = O(1)$$

Бинарная куча: вставка элемента

- Этап 1. Вставка в конец кучи.

Вставка элемента 33



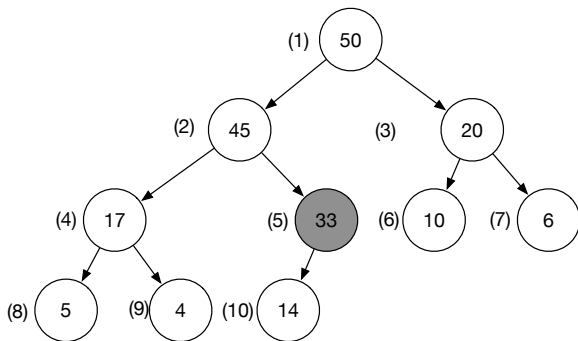
Отлично! Структура кучи не испортилась!

50	45	20	17	14	10	6	5	4	33
----	----	----	----	----	----	---	---	---	----

Бинарная куча: вставка элемента

- Этап 2. Корректировка значений.

Вставка элемента 33



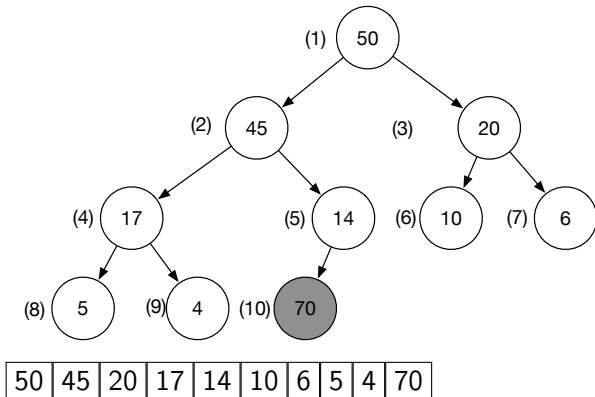
Куча удовлетворяет всем условиям.

50	45	20	17	33	10	6	5	4	14
----	----	----	----	----	----	---	---	---	----

Бинарная куча: вставка элемента

- Попробуем вставить максимальный элемент.

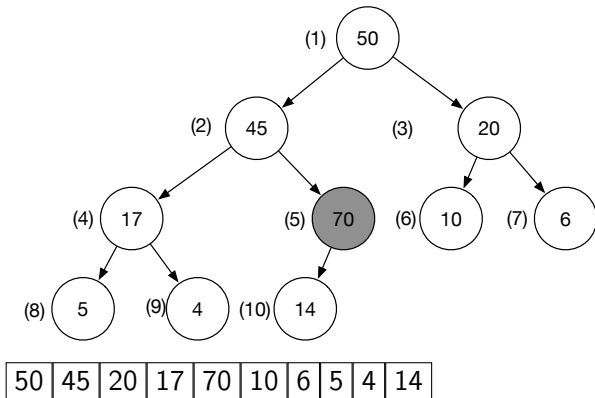
Вставка элемента 70



Бинарная куча: вставка элемента

- Максимальный элемент ползёт вверх по дереву.

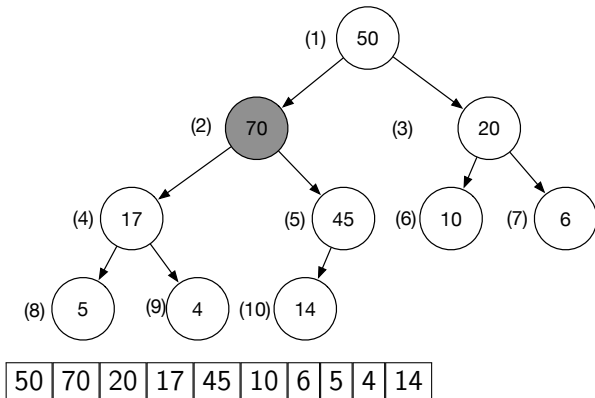
Вставка элемента 70



Бинарная куча: вставка элемента

- Максимальный элемент ползёт вверх по дереву.

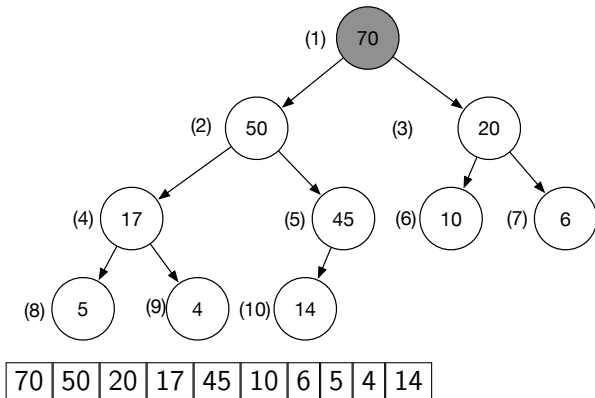
Вставка элемента 70



Бинарная куча: вставка элемента

- Максимальный элемент ползёт вверх по дереву.

Вставка элемента 70



Бинарная куча: вставка элемента

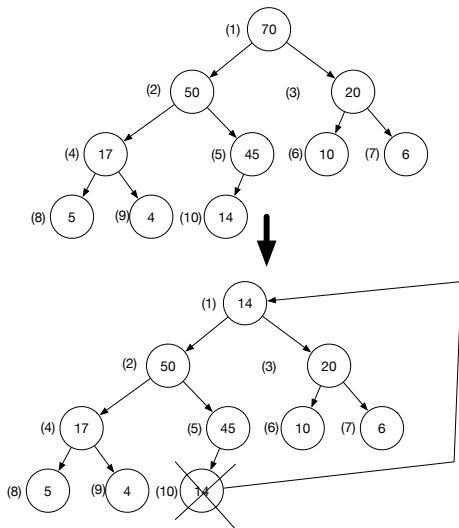
► Реализация.

```
int binary_heap::insert(bhnode node) {  
    if (numnodes > bodysize) {  
        return -1; // или расширяем.  
    }  
    body[++numnodes] = node;  
    for (int i = numnodes; i > 1 &&  
        body[i].priority > body[i/2].priority;  
        i /= 2) {  
        swap(i, i/2);  
    }  
}
```

$$T_{Insert} = O(\log N)$$

Бинарная куча: удаление максимального (минимального)

Удаление максимального



Свойства кучи нарушены. Требуется восстановление.

Бинарная куча: восстановление свойств

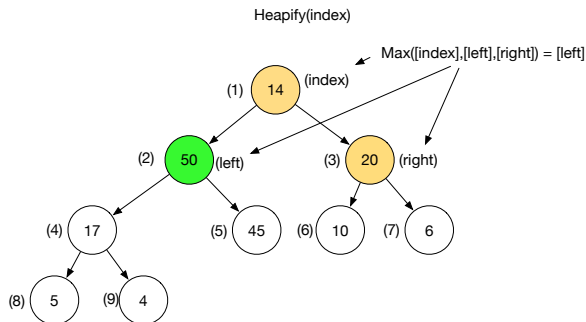
- Для восстановления свойств применяем функцию *heapify*.

```
void binary_heap::heapify(int index) {  
    for (;;) {  
        int left = index + index; int right = left + 1;  
        // Кто больше, [index], [left], [right]?  
        int largest = index;  
        if (left <= numnodes &&  
            body[left].priority > body[index].priority)  
            largest = left;  
        if (right <= numnodes &&  
            body[right].priority > body[largest].priority)  
            largest = right;  
        if (largest == index) break;  
        swap(index, largest);  
        index = largest;  
    }  
}
```

$$T_{\text{heapify}} = O(\log N)$$

Бинарная куча: восстановление свойств

► Восстановление индекса (1)

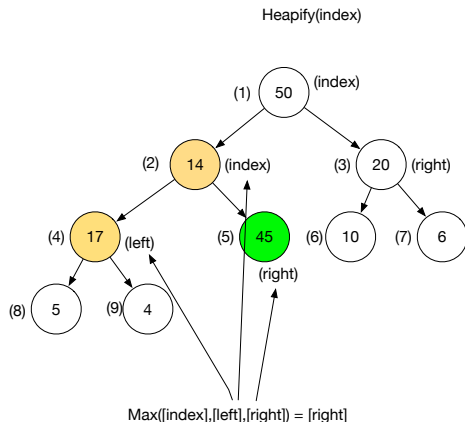


14	50	20	17	45	10	6	5	4
----	----	----	----	----	----	---	---	---

Новый индекс для восстановления (2)

Бинарная куча: восстановление свойств

► Восстановление индекса (2)

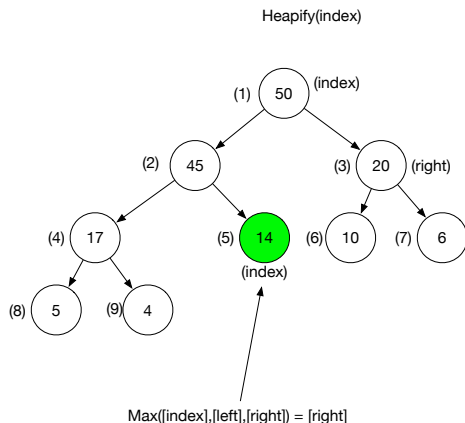


50	14	20	17	45	10	6	5	4
----	----	----	----	----	----	---	---	---

Новый индекс для восстановления (5)

Бинарная куча: восстановление свойств

► Восстановление индекса (5)



50	45	20	17	14	10	6	5	4
----	----	----	----	----	----	---	---	---

Восстановление закончено.

Бинарная куча: увеличение (уменьшение) приоритета элемента

- Для max-heap при увеличении приоритета элемент может ползти только вверх.

```
int binary_heap::increasePriority(int index, int priority) {  
    if (body[index].priority >= priority)  
        return -1;  
    for (body[index].priority = priority;  
         index > 1  
         && body[index].priority > body[index/2].priority;  
         index /= 2) {  
        swap(index, index/2);  
    }  
    return index;  
}
```

$$T_{increasePriority} = O(\log N)$$

HeapSort

HeapSort

- ▶ На основе бинарной кучи можно реализовать алгоритм сортировки со сложностью $O(N \log N)$ в худшем случае.
- ▶ Как?
- ▶ Является ли отсортированным массив, являющийся представлением бинарной кучи?

HeapSort

- ▶ На основе бинарной кучи можно реализовать алгоритм сортировки со сложностью $O(N \log N)$ в худшем случае.
- ▶ Как?
- ▶ Является ли отсортированным массив, являющийся представлением бинарной кучи?
- ▶ Нет.
- ▶ Кто виноват? Что делать?

HeapSort

- ▶ Нужно скомбинировать методы бинарной кучи.
 1. Создать бинарную кучу.
 2. Вставить в неё элементы массива
 3. Извлекать из неё максимальный (минимальный) элемент с удалением.

HeapSort

► Примерный код сортировки HeapSort

```
struct bhnode { // Узел
    int priority;
};

void heapsort(int v[], int vsize) {
    binary_heap *h = new binary_heap(vsize);
    for (int i = 0; i < vsize; i++) {
        bhnode b; b.priority = v[i];
        h->insert(b);
    }
    for (int i = 0; i < vsize; i++) {
        v[i] = h->extractMin()->priority;
    }
    delete h;
}
```

HeapSort

► Сложность алгоритма:

1. Создание бинарной кучи — $T_1 = O(1)$.
2. Вставка N элементов — $T_2 = O(N \log N)$.
3. Извлечение удалением N элементов — $T_3 = O(N \log N)$.

$$T_{heapsort} = T_1 + T_2 + T_3 = O(1) + O(N \log N) + O(N \log N) = O(N \log N)$$

HeapSort

Сложность операций.

- ▶ **insert** — $O(\log N)$
- ▶ **extractMin(Max)** — $O(\log N)$
- ▶ **fetchMin(Max)** — $O(1)$
- ▶ **increasePriority** — $O(\log N)$
- ▶ **merge** — $O(N \log N)$

Задача поиска. Абстракция поиска.

Задача поиска. Абстракция поиска

Информация нужна для того, чтобы ей пользоваться.

Расширенная задача поиска:

1. Накопление информации (сбор)
2. Организация информации (переупорядочивание, сортировка)
3. Извлечение информации (собственно поиск)

Расширенная задача поиска

- ▶ Задача: построение эффективного хранилища данных.
- ▶ Требования:
 - ▶ Поддержка больших объёмов информации.
 - ▶ Возможность быстро находить данные.
 - ▶ Возможность быстро модифицировать данные.
- ▶ Реализация абстракций:
 - ▶ *insert*
 - ▶ *remove*
 - ▶ *find*

Задача поиска. Абстракция поиска

- ▶ Имеется множество ключей

$$a_1, a_2, \dots, a_n$$

- ▶ Требуется определить индекс ключа, совпадающего с заданным значением *key*.

```
bunch a;  
index = a.find(key);
```

bunch — абстрактное хранилище элементов, содержащих ключи (массив, множество, дерево, список...)

Хорошая организация хранилища входит в расширенную задачу поиска.

Последовательный поиск.

Последовательный поиск

Ситуация: к поиску не готовились, ключи не упорядочены.

Индекс	0	1	2	3	4	5	6	7	8	9
Ключ	132	612	232	890	161	222	123	861	120	330
Данные	AB	CA	ЯФ	AB	AA	НД	ОР	ОС	ЗЛ	УГ

`find(a, 222) = 5`

`find(a, 999) = 10` (элемент за границей поиска).

Последовательный поиск

```
int dummysearch(int a[], int N, int key) {  
    for (int i = 0; i < N; i++) {  
        if (a[i] == key) {  
            return i;  
        }  
    }  
    return N;  
}
```

Вероятность найти ключ в i -м элементе $P_i = \frac{1}{N}$

Матожидание числа поисков $E = \frac{N}{2}$

Число операций сравнения $2N$ в худшем случае.

$$T(N) = 2 \cdot N = O(N)$$

Последовательный поиск

Небольшая подготовка:

Индекс	0	1	2	3	4	5	6	7	8	9	10
Ключ	132	612	232	890	161	222	123	861	120	330	999
Данные	AB	CA	ЯФ	AB	AA	НД	ОР	ОС	ЗЛ	УГ	??

Результаты не изменились.

`find(a, 222) = 5`

`find(a, 999) = 10` (элемент за границей поиска).

Последовательный поиск

```
int cleversearch(int a[], int N, int key) {  
    a[n] = key;  
    int i;  
    for (i = 0; a[i] != key; i++)  
        ;  
    return i;  
}
```

Число операций сравнения N в худшем случае.

$$T(N) = N = O(N)$$

Поиск ускорен в два раза!

Без подготовки лучших результатов не добиться.

Неупорядоченный массив

- ▶ Сложность операций:
 - ▶ *find* — $O(N)$
 - ▶ *insert* — $O(1)$
 - ▶ *remove* — $O(N)$

Поиск с сужением зоны.

Поиск с сужением зоны

Если в зоне поиска имеется упорядочивание — всё становится значительно лучше.

Возможное действие: упорядочить по отношению.

- ▶ Имеется множество ключей

$$a_1 \leq a_2 \leq \dots \leq a_n$$

- ▶ Требуется определить индекс ключа, совпадающего с заданным значением *key*.

Поиск с сужением зоны

Принцип «разделяй и властвуй».

1. Искомый элемент равен центральному? Да — нашли.
2. Искомый элемент меньше центрального? Да — рекурсивный поиск в левой половине.
3. Искомый элемент больше центрального? Да — рекурсивный поиск в правой половине.

Поиск с сужением зоны

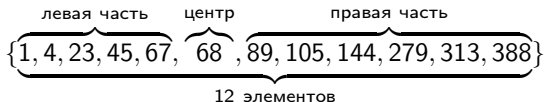
- ▶ Вход алгоритма: упорядоченный по возрастанию массив, левая граница поиска, правая граница поиска.
- ▶ Выход алгоритма: номер найденного элемента или -1.

```
int binarySearch(int val, int a[], int left, int right) {  
    if (left >= right) return a[left] == val? left : -1;  
    int mid = (left+right)/2;  
    if (a[mid] == val) return mid;  
    if (a[mid] < val) {  
        return binarySearch(val, a, left, mid-1);  
    } else {  
        return binarySearch(val, a, mid+1, right);  
    }  
}
```

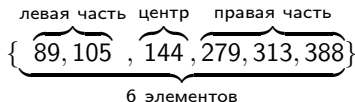
Поиск с сужением зоны

Оценка глубины рекурсии.

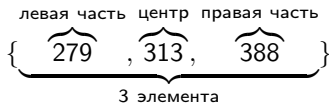
Поиск ключа 313:



$313 > 68 \rightarrow$ ключ справа



$313 > 144 \rightarrow$ ключ справа



$313 = 313 \rightarrow$ ключ найден

Поиск с сужением зоны

Попрактикуемся в основной теореме о рекурсии.

- ▶ Количество подзадач $a = 1$.
- ▶ Каждая подзадача уменьшается в $b = 2$ раза.
- ▶ Сложность консолидации $O(1) = O(N^0) \rightarrow d = 0$

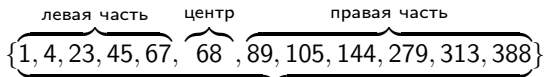
$$d = \log_b a \rightarrow T(N) = \log N$$

Результат можно получить и интуитивно.

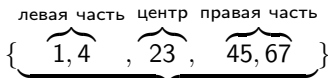
Поиск с сужением зоны

Оценка глубины рекурсии.

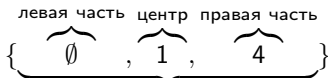
Поиск отсутствующего 10:



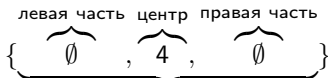
12 элементов



5 элементов



2 элемента



1 элемент

Поиск с сужением зоны

Переход от рекурсии к итерации.

```
int binarySearch(int val, int a[], int left, int right) {  
    while (left < right) {  
        int mid = (left + right)/2;  
        if (a[mid] == val) return mid;  
        if (a[mid] < val) {  
            right = mid - 1;  
        } else {  
            left = mid + 1;  
        }  
    }  
    return a[left] == val? left : -1;  
}
```

Поиск с сужением зоны

- ▶ Можно ли быстрее? Хотим уменьшить коэффициент C в формуле $T(N) = C \cdot O(\log N)$.
- ▶ Варианты поиска: N –ричный поиск. Попытка 1: троичный поиск.

Поиск с сужением зоны

► Троичный поиск.

```
int ternarySearch(int val, int a[], int left, int right) {  
    if (left >= right) return a[left] == val? left : -1;  
    int mid1 = (left*2+right)/3;  
    int mid2 = (left+right*2)/3;  
    if (val < a[mid1]) {  
        return ternarySearch(val, a, left, mid1-1);  
    } else if (val == a[mid1]) {  
        return mid1;  
    } else if (a[mid1] < a[mid2]) {  
        return ternarySearch(val, a, mid1+1, mid2-1);  
    } else if (a[mid1] == a[mid2]) {  
        return mid2;  
    } else {  
        return ternarySearch(val, a, mid2+1, right);  
    }  
}
```


Поиск с сужением зоны

Добились ли мы выигрыша?

По числу рекурсивных вызовов — выигрыш в

$$\frac{\log 3}{\log 2} = \log_2 3 \approx 1.58 \text{ раз.}$$

Количество сравнений увеличилось с 3 до 5, проигрыш в ≈ 1.67 раз.

Упорядоченный массив

- ▶ Сложность операций:
 - ▶ *find* — $O(\log N)$
 - ▶ *insert* — $O(N)$
 - ▶ *remove* — $O(N)$

Распределяющий поиск.
Поиск с использованием
свойств ключа.

Распределяющий поиск

- ▶ Можно ли найти ключ в неотсортированном массиве быстрее, чем за $O(N)$?

Распределяющий поиск

- ▶ Можно ли найти ключ в неотсортированном массиве быстрее, чем за $O(N)$?
- ▶ Без вспомогательных данных — нет.

Распределяющий поиск

- ▶ Можно ли найти ключ в неотсортированном массиве быстрее, чем за $O(N)$?
- ▶ Без вспомогательных данных — нет.
- ▶ Какова сложность нахождения $M \approx N$ значений в неотсортированном массиве?

Распределяющий поиск

- ▶ Можно ли найти ключ в неотсортированном массиве быстрее, чем за $O(N)$?
- ▶ Без вспомогательных данных — нет.
- ▶ Какова сложность нахождения $M \approx N$ значений в неотсортированном массиве?
- ▶ Вариант ответа: если $M > \log N$, то предварительной сортировкой. Сложность составит $O(N \log N) + M \cdot O(\log N) = O(N \log N)$.

Распределяющий поиск

- ▶ Можно ли найти ключ в неотсортированном массиве быстрее, чем за $O(N)$?
- ▶ Без вспомогательных данных — нет.
- ▶ Какова сложность нахождения $M \approx N$ значений в неотсортированном массиве?
- ▶ Вариант ответа: если $M > \log N$, то предварительной сортировкой. Сложность составит $O(N \log N) + M \cdot O(\log N) = O(N \log N)$.
- ▶ А быстрее можно?

Распределяющий поиск

- ▶ Можно ли найти ключ в неотсортированном массиве быстрее, чем за $O(N)$?
- ▶ Без вспомогательных данных — нет.
- ▶ Какова сложность нахождения $M \approx N$ значений в неотсортированном массиве?
- ▶ Вариант ответа: если $M > \log N$, то предварительной сортировкой. Сложность составит $O(N \log N) + M \cdot O(\log N) = O(N \log N)$.
- ▶ А быстрее можно?
- ▶ В некоторых случаях — да.

Распределяющий поиск

- ▶ Если $|D(Key)|$ невелико, то имеется способ, похожий на сортировку подсчётом.
- ▶ Создаётся **инвертированный массив**.

$$a = \{2, 7, 5, 3, 8, 6, 3, 9, 12\}. |D(a)| = 12 - 2 + 1 = 11.$$

$$a_{inv}[2..12] = \{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1\}$$

$$a[0] = 2 \rightarrow a_{inv}[2] = 0$$

$$a[1] = 7 \rightarrow a_{inv}[7] = 1$$

$$a[2] = 5 \rightarrow a_{inv}[5] = 2$$

$$a_{inv}[2..12] = \{0, 6, -1, -1, 5, -1, 4, 7, -1, -1, 8\}$$

Распределяющий поиск

index	0	1	2	3	4	5	6	7	8
key	2	7	5	3	8	6	3	9	12

key	2	7	5	3	8	6	3	9	12
index	0	1	2	3	4	5	6	7	8

Распределяющий поиск

Два этапа. Первый этап — инвертирование.

```
int * prepare(int a[], int N, int *min, int *max) {
    *min = *max = a[0];
    for (int i = 1; i < N; i++) {
        if (a[i] > *max) *max = a[i];
        if (a[i] < *min) *min = a[i];
    }
    if (*max - *min > THRESHOLD) return NULL;
    int *ret = new int[*max - *min + 1];
    for (int i = *min; i <= *max; i++) {
        ret[i] = -1;
    }
    for (int i = 0; i < N; i++) {
        ret[a[i] - *min] = i;
    }
    return ret;
}
```

Распределяющий поиск

Второй этап: поиск.

```
// Подготовка
int min, max;
int *ainv = prepare(a, N, &min, &max);
// Поиск ключа key
result = -1;
if (key >= min && key <= max) result = ainv[key - min];
```

- ▶ $O(N)$ на подготовку.
- ▶ $O(M)$ на поиск M элементов.
- ▶ $T(N, M) = O(N) + O(M) = O(N)$

Распределяющее хранение

- ▶ Сложность операций:
 - ▶ *find* — $O(1)$
 - ▶ *insert* — $O(1)$
 - ▶ *remove* — $O(1)$
- ▶ Жёсткие ограничения на множество ключей.
- ▶ При наличии $f(key)$ сводится к хеш-поиску.

Спасибо за внимание.

Следующая лекция —
деревья: сбалансированные и
специальные.