План лекции

1. Задача поиска. Абстракция поиска.
2. Последовательный поиск.
3. Распределяющий поиск. Поиск с использованием свойств ключа.
4. Поиск с сужением зоны.
5. Сравнительный анализ методов поиска.
6. Структура данных «список». Варианты представления.
7. Структура данных «дерево». Динамическая память. Деревья поиска. Обход деревьев.

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

$$a_1, a_2, \ldots, a_n$$

- 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	АВ	СА	ЯФ	АВ	АА	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 $P_i = \dfrac{1}{N}$

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

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	АВ	СА	ЯФ	АВ	АА	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

$$a_1 \leqslant a_2 \leqslant \cdots \leqslant a_n$$

- 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:

$$\{\underbrace{\overbrace{1,4,23,45,67}^{\text{левая часть}},\overbrace{68}^{\text{центр}},\overbrace{89,105,144,279,313,388}^{\text{правая часть}}}_{\text{12 элементов}}\}$$

$313 > 68 \rightarrow$ ключ справа

$$\{\underbrace{\overbrace{89,105}^{\text{левая часть}},\overbrace{144}^{\text{центр}},\overbrace{279,313,388}^{\text{правая часть}}}_{\text{6 элементов}}\}$$

$313 > 144 \rightarrow$ ключ справа

$$\{\underbrace{\overbrace{279}^{\text{левая часть}},\overbrace{313}^{\text{центр}},\overbrace{388}^{\text{правая часть}}}_{\text{3 элемента}}\}$$

$313 = 313 \rightarrow$ 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 $O(1) = O(N^0) \to d = 0$

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

Результат можно получить и интуитивно.

Поиск с сужением зоны

Оценка глубины рекурсии.
Поиск отсутствующего 10:

$$\underbrace{\{\overbrace{1, 4, 23, 45, 67}^{\text{левая часть}}, \overbrace{68}^{\text{центр}}, \overbrace{89, 105, 144, 279, 313, 388}^{\text{правая часть}}\}}_{\text{12 элементов}}$$

$$\underbrace{\{\overbrace{1, 4}^{\text{левая часть}}, \overbrace{23}^{\text{центр}}, \overbrace{45, 67}^{\text{правая часть}}\}}_{\text{5 элементов}}$$

$$\underbrace{\{\overbrace{\emptyset}^{\text{левая часть}}, \overbrace{1}^{\text{центр}}, \overbrace{4}^{\text{правая часть}}\}}_{\text{2 элемента}}$$

$$\underbrace{\{\overbrace{\emptyset}^{\text{левая часть}}, \overbrace{4}^{\text{центр}}, \overbrace{\emptyset}^{\text{правая часть}}\}}_{\text{1 элемент}}$$

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 < a[mid2]) {
      return ternarySearch(val, a, mid1+1, mid2-1);
   } else if (a == 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 $\dfrac{\log 3}{\log 2} = \log^2 3 \approx 1.58$ 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**.

$a = \{2, 7, 5, 3, 8, 6, 3, 9, 12\}$. $|D(a)| = 12 - 2 + 1 = 11$.
$a_{inv}[2..12] = \{-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1\}$
$a[0] = 2 \to a_{inv}[2] = 0$
$a[1] = 7 \to a_{inv}[7] = 1$
$a[2] = 5 \to a_{inv}[5] = 2$
$a_{inv}[2..12] = \{0, 6, -1, -1, 5, -1, 4, 7, -1, -1, 8\}$

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.

Структура данных «список». Варианты представления.

Списки

Список — структура данных, которая реализует абстракции:

- *insertToFront* — добавление элемента в начало списка.
- *insertToBack* — добавление элемента в конец списка
- *find* — поиск элемента
- *size* — определение количества элементов

Списки: реализация

Для реализации списков требуются явное использование указателей.

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

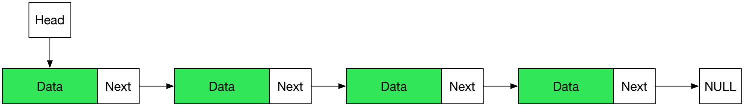
Внутренние операции создания элементов — через *malloc*, *calloc*, *new*.

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

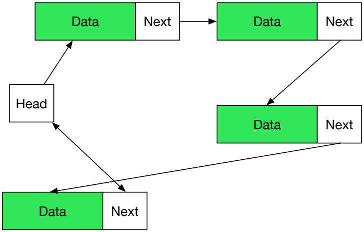
Списки: представления

Различные варианты представлений:

В линейном виде:



В виде кольца:



Списки: сложность

Стоимость операций:

Операция	Время	Память
insertToFront	O(1)	O(1)
insertToEnd	O(N)	O(1)
find	O(N)	O(1)
size	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;
}
```

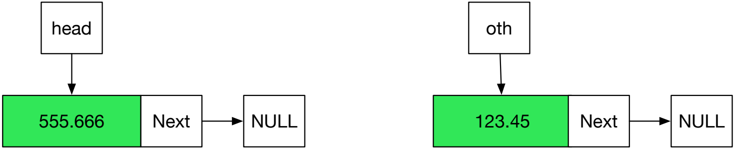
Создание списка из одного элемента:

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

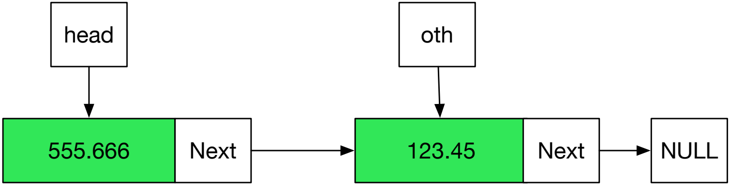
Списки: добавление

Добавление элемента в хвост списка:

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

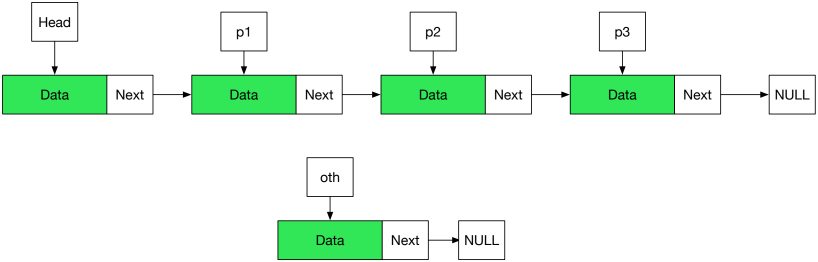



```
head->next = oth;
```



Списки: добавление

Добавление элемента в хвост списка, состоящего из нескольких элементов:

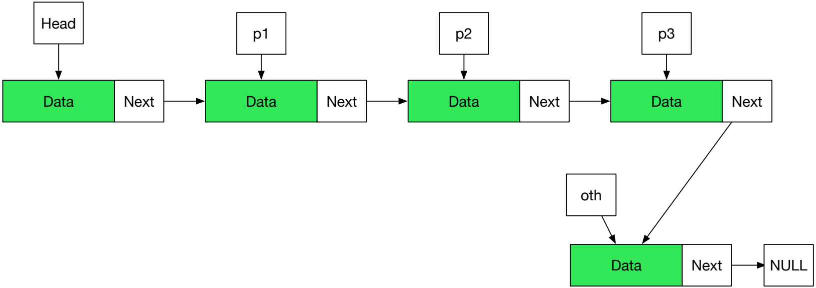


Проход по элементам до нужного (traversal, walk):

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

Списки: добавление

Заключительное состояние после вставки.

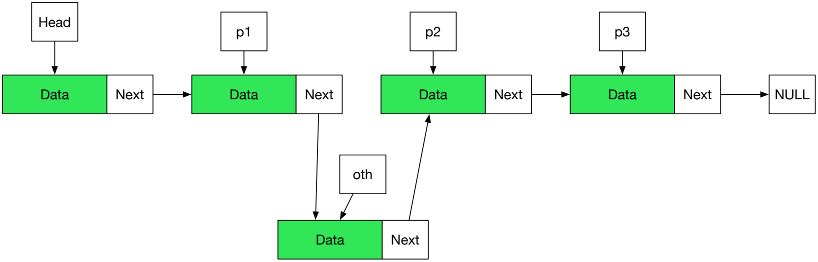


Сложность операции — $O(N)$

Списки: добавление

Вставка за конкретным элементом $p1$ примитивна.

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



Списки: добавление

Вставка ПЕРЕД известным элементом сложнее:

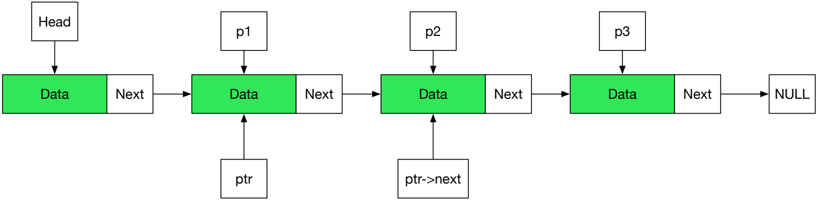
```
// вставка ПЕРЕД элементом p2
linkedListNode *ptr = head;
while (ptr->next != p2) {
    ptr = ptr->next;
}
oth->next = ptr;
ptr->next = oth;
```

Списки: удаление

Удаление элемента *p*2 — 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
```

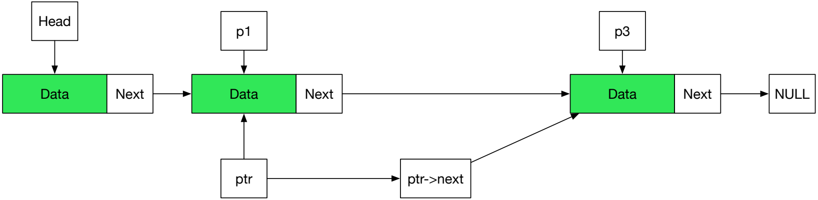


Списки: удаление

Удаление элемента из списка.

- Переместить указатели.

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



Списки: размер

Операция *size* — две возможности:

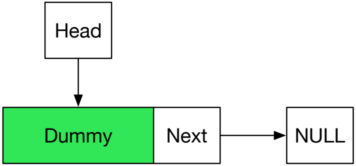
- Через операцию *walk* до NULL:

```
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

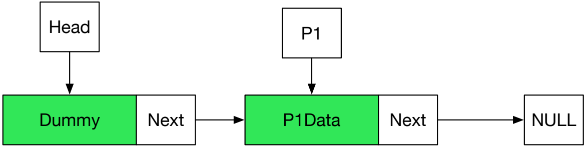
- При нашем представлении требуется всегда различать, работаем ли мы с головой списка или с другим элементом. При смене головы списка приходится заменять все указатели в программе.
- Существуют различные способы представления списков.
- Для абстрактного типа данных удобнее иметь список с неизменной головой.



- 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

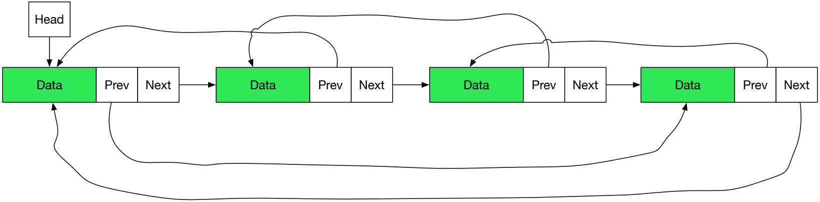
Ещё раз оценим сложность основных операций:

- 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 $s = p1 \to next$
3. $oth \to prev = p1$
4. $oth \to next = s$
5. $s \to prev = oth$
6. $p1 \to next = oth$

Списки: двусвязные списки: удаление

Для удаления элемента $p1$ из списка:

1. Сохраняем указатель $s = p1 \rightarrow next$
2. $s \rightarrow prev = p1$
3. $p1 \rightarrow next = s$
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 которые 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 поиска. Обход деревьев.

Деревья: особенности

Основная особенность деревьев — наличие нескольких наследников.
По максимальному числу наследников деревья делятся на

- двоичные
- троичные
- $N-$ричные

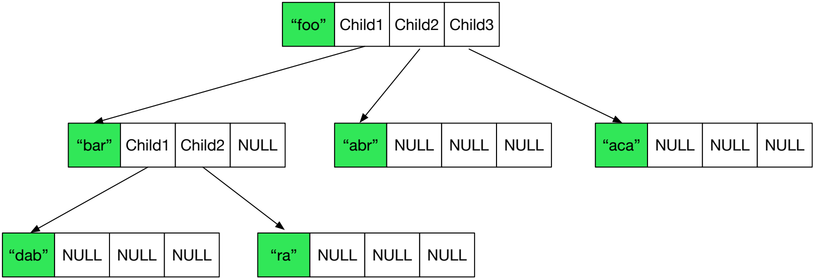
```
struct tree {
    tree *children[3];
    myType data;
    ...
};
```

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:

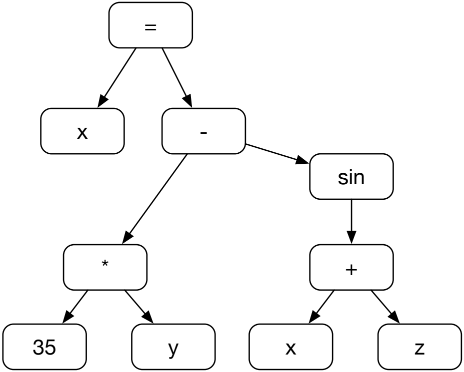
```
tree *root = new tree("foo");
root->child[0] = new tree("bar");
root->child[1] = new tree("abr");
root->child[2] = new tree("aca");
root->child[0]->child[0] = new tree("dab");
root->child[0]->child[1] = new tree("ra");
```

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 1.
  - Для узла с номером K номера детей

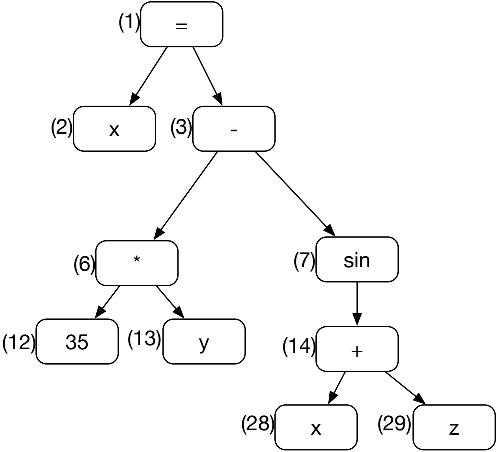
$$K \cdot N \ldots K \cdot N + N - 1$$

  - Для 2-дерева корневой узел $= 1$, дети 1-го уровня ($Depth = 2$) 2 и 3...

Деревья: нумерация

Для дерево 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

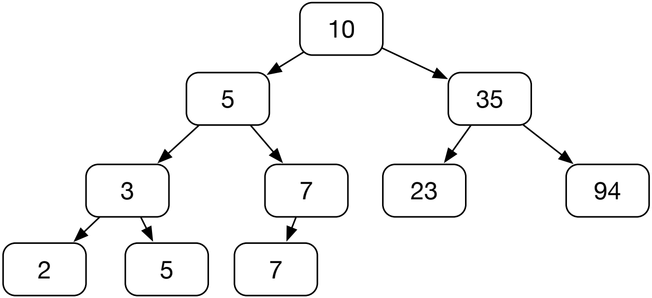
Деревья: поиск

Использование деревьев для поиска.
Задача:

- Вход: последовательность чисел.
- Выход: 2-дерево, в котором все узлы справа от родителя больше родителя, а слева — не больше.

Деревья: поиск

{10, 5, 35, 7, 3, 23, 94, 2, 5, 7}



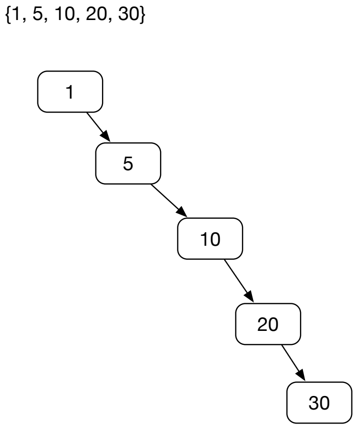
Деревья: поиск

Поиск по дереву после получения элемента с ключом X:

1. Делаем текущий узел корневым
2. Переходим в текущий узел C.
3. Если $X = C.Key$ то алгоритм завершён.
4. Если $X > C.Key$ и C имеет потомка справа, то делаем текущим узлом потомка справа. Переходим к п. 2.
5. Если $X < C.Key$ и C имеет потомка слева, то делаем текущим узлом потомка слева. Переходим к п. 2.
6. Ключ не найден. Конец алгоритма.

Деревья: поиск

Наивный вариант построения дерева может привести к странным результатам:



Подробнее о бинарных деревьях поиска — в следующей лекции.

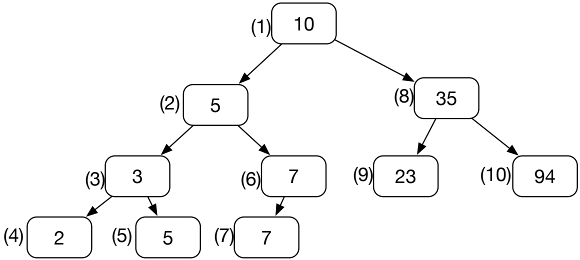
Деревья: поиск

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

Деревья: обход

Прямой способ обхода.

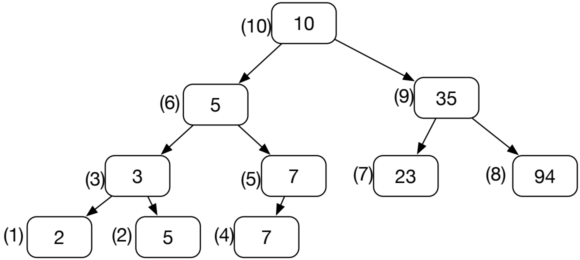
```
void walk(tree *t) {
   work(t);
   if (t->left != NULL) walk(t->left);
   if (t->right != NULL) walk(t->right);
}
```



Деревья: обход

Симметричный способ обхода.

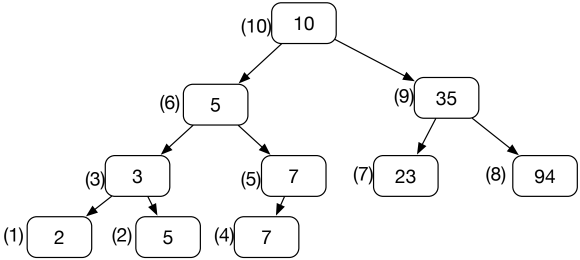
```
void walk(tree *t) {
   if (t->left != NULL) walk(t->left);
   work(t);
   if (t->right != NULL) walk(t->right);
}
```



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.

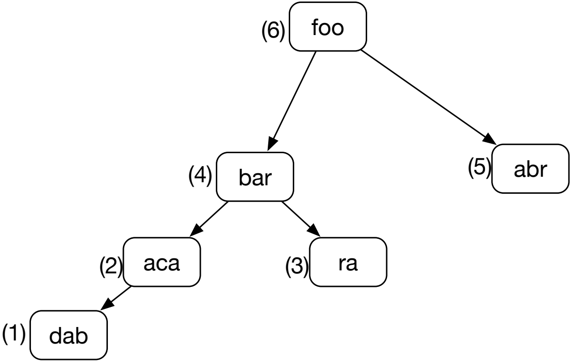
```
typedef void (*walkFunction)(tree *);

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);
}
```

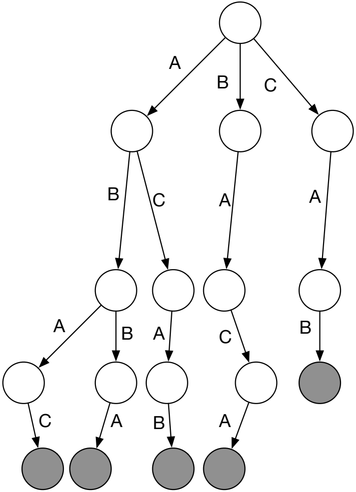
Деревья: обход

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 исказятся 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 исполняется инициализация полей (написанный нами код).

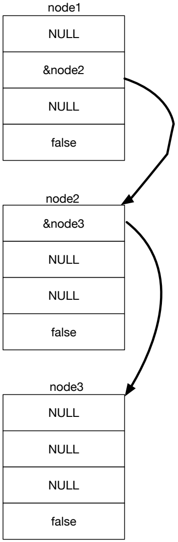
Деревья: динамическая память

- 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



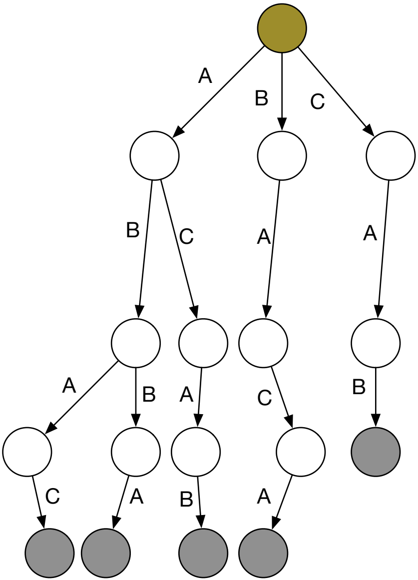
Деревья: динамическая память

- 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.

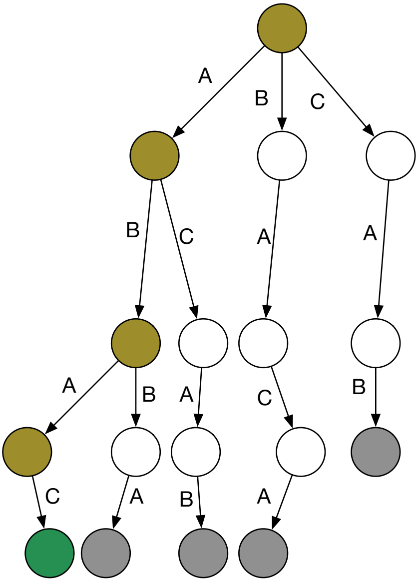
Деревья: освобождение памяти

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

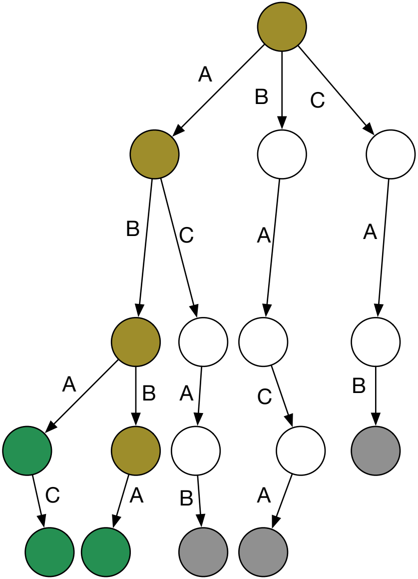
Деревья: освобождение памяти



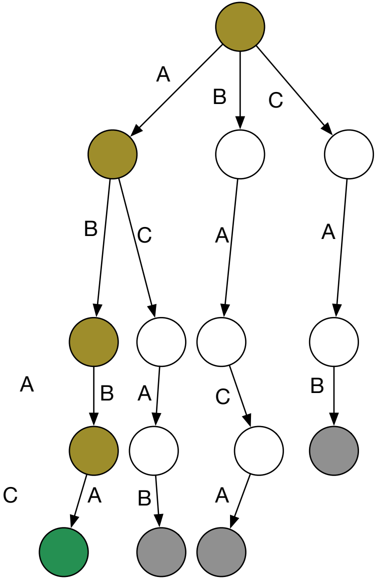
Деревья: освобождение памяти



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 $O(N)$).

Сравнение основных методов поиска

Структура хранилища	вставка	удаление	поиск
Неупорядоченный массив	$O(1)$	$O(N)$	$O(N)$
Упорядоченный массив	$O(N)$	$O(N)$	$O(\log N)$
Неупорядоченный список	$O(1)$	$O(N)$	$O(N)$
Двоичное дерево поиска	$O(N)$	$O(N)$	$O(N)$
Сбалансированное дерево поиска	$O(\log N)$	$O(\log N)$	$O(\log N)$