ЛЕКЦИЯ 06
СТРУКТУРЫ ДАННЫХ

АЛГОРИТМИЗАЦИЯ И
ПРОГРАММИРОВАНИЕ

ЛЕКТОР ФУРМАВНИН С.А.

СПИСОК КАК СТРУКТУРА ДАННЫХ

Представьте, что список вынесен в модуль как в encap-sll.h / encap-sll.cpp

Внешний интерфейс выглядит примерно так

```
struct ListNode; // объявление списка, детали спрятаны
ListNode *list_create(int d);
ListNode *list_push(ListNode *pre, int d);
ListNode *list_pop(ListNode *pre);
```

- Такой список в принципе не может случайно зациклиться, так как вся работа с ним осуществляется только функциями его **открытого интерфейса**
- И каждая из этих функций **поддерживает инварианты своего типа данных**

СТРУКТУРЫ ДАННЫХ

Список не единственная структура данных. Самые известные это:

- Динамические массивы
- Линейные и циклические списки
- Хеш-таблицы
- Поисковые деревья

Обсуждение: чем **отличаются** друг от друга структуры данных?

СТРУКТУРЫ ДАННЫХ

Список не единственная структура данных. Самые известные это:

- Динамические массивы
- Линейные и циклические списки
- Хеш-таблицы
- Поисковые деревья

Обсуждение: чем **отличаются** друг от друга структуры данных?

- Неправильный ответ: реализацией. Реализация у них спрятана и считается не слишком важной. Операции с каждым типом происходят только через его интерфейс
- Правильный ответ: алгоритмической сложностью функций открытого интерфейса

ПРИМЕР: ТЕЛЕФОННАЯ КНИГА

- У вас есть список друзей и список их номеров в международном формате (до 15 цифр)

Alice	44-7911-975-72-83
Bob	44-7911-486-92-83
Camilla	8-800-555-31-35
Daniel	8-800-765-91-35

- Вам нужно выбрать структуру данных, которая позволяла бы:
- Быстро добавить человека и его номер
- Найти имя человека по номеру телефона
- Просто хранить массив строк, индексированный номерами телефонов затратно

ОБСУЖДЕНИЕ

Можно выбрать сортированный по номеру массив

- Преимущества: быстрый поиск $O(lgN)$
- Недостатки: медленная вставка $O(N)$

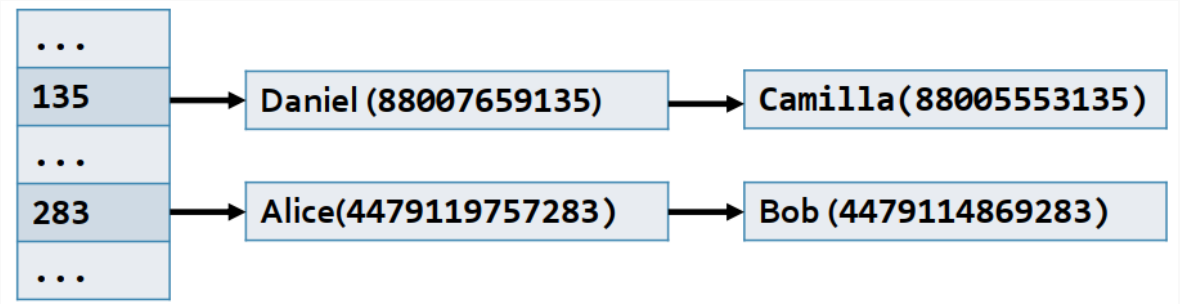
Можно выбрать список

- Преимущества: быстрая вставка $O(1)$
- Недостатки: медленный поиск $O(N)$

Конечно лучше всего было бы сделать прямую адресацию по номерам телефонов, тогда и поиск и вставка были бы $O(1)$
Можем ли мы отобразить все номера телефонов на разумный диапазон, скажем $0 \div 999$?

ХЭШИРОВАНИЕ

- Самый простой способ это взять три последние цифры номера
- Запишем это как функцию $h\ n = n\ \%\ m$ (где m это мощность отображения, в данном случае $m = 1000$). Мощность определяет количество «бакетов».
- Тогда имеем массив из 1000 списков



- Уже сейчас видно, что если повезёт и коллизий не будет, то поиск $O(1)$ и вставка $O(1)$. Увы сейчас хеш-функция $h\ n$ довольно плоха

УНИВЕРСАЛЬНЫЕ СЕМЕЙСТВА ФУНКЦИЙ

Случайно выбранная функция из такого семейства гарантированно будет в среднем не хуже, чем любая другая для этого класса

Везде далее m это мощность хеша, p это простое число, большее m

Для целых чисел это семейство

$$h_{a,b}(n) = \big((ax + b)\ \%\ p\big)\ \%\ m, \text{при этом } a \neq 0$$

Для строк это семейство

$h_r(c_1 \dots c_l) = h_{int}\left(\left(\sum_{i=1}^{l} c_i r^{l-i}\right)\ \%\ p\right)$, где h_{int} это произвольно выбранная $h_{a,b}$

- Иногда параметры действительно выбирают случайно и меняют с каждым запуском программы

РЕАЛИЗАЦИЯ ХЕШ-ФУНКЦИЙ

- Если что-то считается миллионы раз за выполнение программы, его реализации лучше быть как можно более оптимальной
- Пусть w это количество бит в машинном слове (обычно 16 или 32) и мощность $m = 2^M, M < w$
- Тогда неплохая хеш-функция для целых реализуется как

```
unsigned hashint(unsigned a, unsigned b, unsigned x) {
  return (a*x + b) >> (w - M); }
```

- Заметьте, здесь `(a*x + b)` уже вычисляется по модулю 2^{32} без явного деления
- Аналогично выкручиваются со строками

ЗАДАЧА. ПОДСЧЕТ КОЛЛИЗИЙ

- Ваша задача: имея произвольную функцию хеширования строки и произвольную последовательность строк, подсчитать количество коллизий

```
typedef int (*get_hash_t)(const char *s);
int ncollisions(char **strs, get_hash_t f) {
  // TODO: your code here }
```

- Эта функция впоследствии может быть использована для оценки качества хеш-функций над строками

ХЕШ-ТАБЛИЦА КАК СТРУКТУРА ДАННЫХ

- Как структура данных, хеш-таблица тоже инкапсулируется в отдельном модуле
- Интерфейс может выглядеть как:

```
struct hashmap_t; // объявление таблицы, детали спрятаны
hashmap_t *hashmap_create(unsigned m);
int hashmap_add(hashmap_t *h, unsigned key,
                const char *value);
const char *hashmap_find(struct hashmap_t *h, unsigned key);
void hashmap_destroy(struct hashmap_t *h);
```

- Разумеется реальный интерфейс может быть гораздо богаче и интереснее

О ПОИСКЕ ПОДСТРОКИ В СТРОКЕ

- Можем ли мы использовать хеширование для эффективного поиска подстроки?
- Вычислим хеш $h_{target} = h_{string}(ABACAB)$

A	B	A	B	A	C	A	A	B	A	C	A	B	C	A	C	B	C	A	A	A
A	B	A	C	A	B															

- Здесь функция h_{string} - это упрощённая универсальная функция для строк

$h_r(c_1 \dots c_l) = \left(\sum_{i=1}^{l} c_i r^{l-i}\right)\%p$

- Первые шесть символов строки в которой мы ищем дают $h_{string}(ABABAC)$
- Можем ли мы легко перейти от него к $h_{string}(BABACA)$?

ЦИКЛИЧЕСКИЕ СВОЙСТВА ХЕШ-ФУНКЦИИ

- Ещё раз посмотрим на формулу $h_r(c_1 \dots c_l) = \left(\sum_{i=1}^{l} c_i r^{l-i}\right) \% p$
- $h_r(c_2 \dots c_{l+1}) = \left(\left(h_r(c_1 \dots c_l) - \boldsymbol{c_1 r^{l-1}}\right) \cdot r + \boldsymbol{c_{l+1}}\right) \% p$
- Интуитивно: мы убираем **первый** символ, сдвигаем строку умножением и добавляем **последний**
- Можно предварительно подсчитать $n = r^{l-1} \% p$
- Сигнатура для функции обновления

```
unsigned update_hash(unsigned hash, unsigned n,
                     char cf, char cl);
```

- Напишите эту функцию

ОБСУЖДЕНИЕ

- Циклические свойства хеш-функций наталкивают на идею «циклического хеша», который обновляется с каждым новым продвижением подстроки

A	B	A	B	A	C	A	A	B	A	C	A	B	C	A	C	B	C	A	A	A
	A	B	A	C	A	B														

- Вместо $O(Nm)$ имеем $O(N)$, потому что обновление хеша происходит за константное время
- Это называется **алгоритмом Рабина-Карпа**

АЛГОРИТМ РАБИНА-КАРПА

- Проверяет наличие подстроки needle в строке haystack

```
// assume strlen(needle) much lesser then strlen(haystack)
int rabin_karp(const char *needle, const char *haystack) {
  unsigned n, target, cur, count = 0, left = 0,
    right = strlen(needle);
  target = get_hash(needle, needle + right);
  cur = get_hash(haystack, haystack + right);
  n = pow_mod(R, right - 1, Q); // алгоритм POWM
  while(target != cur && haystack[right] != 0) {
    cur = update_hash(cur, n, haystack[left], haystack[right]);
    left += 1; right += 1;
  }
  return (target == cur) ? left : 0;
}
```

ЗАДАЧА. КОЛЛИЗИИ РАБИНА-КАРПА

- В алгоритме RK никак не обработан случай коллизии хеш-функции, когда хеши совпали, а строка найдена неверно
- Ваша задача доработать с учётом коллизий функцию

```
int rabin_karp(const char *needle, const char *haystack);
```

- Выберите в качестве хеш-функции нечто не слишком совершенное, например

$$h(c_1 \dots c_l) = \left(\sum_{i=1}^{l} c_i 10^{l-i} \right) \%31$$

- Проверьте как работает поиск с коллизиями

ОБСУЖДЕНИЕ: АЛГОРИТМ РАБИНА-КАРПА

- Несколько проигрывает КМП для поиска точного совпадения
- Зависит от выбора хеш-функции
- Но находит своё применение если нужно не слишком точное совпадение
- Как бы вы модифицировали алгоритм RK чтобы он искал подстроку без учёта регистра и при этом игнорировал запятые?

ОБСУЖДЕНИЕ

- Главный недостаток хеш-таблиц (и шире хеш-функций как идеи) это стирание информации о естественном порядке объектов
- Например если нужно индексировать города расстоянием от Москвы, то сложить их в хеш-таблицу по этому расстоянию легко. Также легко получить город на расстоянии 60 километров. Но увы, вынуть из хеш-таблицы все города от 50 до 100 километров невозможно
- Говорят, что хеш-отображения не позволяют делать range-queries

ОБСУЖДЕНИЕ

- Главный недостаток хеш-таблиц (и шире хеш-функций как идеи) это стирание информации о естественном порядке объектов
- Например если нужно индексировать города расстоянием от Москвы, то сложить их в хеш-таблицу по этому расстоянию легко. Также легко получить город на расстоянии 60 километров. Но увы, вынуть из хеш-таблицы все города от 50 до 100 километров невозможно
- Говорят, что хеш-отображения не позволяют делать range-queries
- И это естественным образом приводит нас к поисковым деревьям

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Бьерн Страуструп, Язык программирования C++/ ред. А. Боборыкин. – 4-е изд. - Москва: Издательство БИНОМ, 2023. – 1213 с.
2. Дональд Кнут Искусство программирования, том 1. Основные алгоритмы = The Art of Computer Programming, vol.1. Fundamental Algorithms. — 3-е изд. — М .: «Вильямс» , 2006. – 720 с.
3. Дональд Кнут Искусство программирования, том 2. Получисленные алгоритмы = The Art of Computer Programming, vol.2. Seminumerical Algorithms. — 3-е изд. — М .: «Вильямс» , 2007. — 832 с.
4. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. М.: МЦМНО, 1999. – 960 с.
5. Скотт Мейерс, Эффективное использование C++. 55 верных способов улучшить структуру и код ваших программ / ред. Д.А. Мовчан – 3-е изд. – Москва: ДМК Пресс, 2017. – 300 с.
6. Robert Sedgewick Algorithms, 4th edition, 2011