7.Опишите идею непрерывной реализации стека на базе вектора и приведите код этой реализации в виде класса Stack на языке Ruby. Оцените временную сложность построенной реализации.

Идея:

Идея реализации состоит в том, чтобы представить стек в виде вектора. Вершиной стека будет последний элемент вектора. Для правильной работы стека нам понадобятся всего лишь 4 метода: 1)конструктор; 2)метод, помещающий элемент в стек (push); 3)метод, берущий элемент из стека (pop) и метод, показывающий вершину стека (top)

Реализация класса Stack на языке Ruby:

```
class Stack
    def initialize
        @array = Array.new
    end
    def push(c)
        @array.push(c)
    end
    def pop
        @array.pop
    end
    def top
        @array.last
    end
end
```

Пример использования:

```
stack = Stack.new
stack.push(1) # [1]
stack.push(2) # [1,2]
stack.push(3) # [1,2,3]
```

```
stack.pop      # [1,2]
```

Независимо от размера стека, в каждом методе выполняется всего лишь одна операция. Следовательно, временная сложность реализации стека равняется O(1).

10.С помощью теоремы о рекуррентных оценках оцените максимальную сложность следующих алгоритмов: двоичный поиск элемента в упорядоченном массиве, сортировка слиянием, быстрая сортировка.

Теорема о рекуррентных оценках:

Пусть $a \geqslant 1$, $b > 1$—константы, $f(n)$—асимптотически положительная функция, а для неизвестной нам функции $T(n)$ выполнено при неотрицательных n соотношение

$$T(n) = aT(n/b) + f(n),$$

где под n/b понимается одно из двух ближайших целых. Тогда:

- 1) если $f(n) = O(n^{\log_b a - \varepsilon})$ для некоторого $\varepsilon > 0$, то $T(n) = \Theta(n^{\log_b a})$;
- 2) если $f(n) = \Theta(n^{\log_b a})$, то $T(n) = \Theta(n^{\log_b a} \log n)$;
- 3) если $f(n) = \Omega(n^{\log_b a + \varepsilon})$ для некоторого $\varepsilon > 0$ и если $af(n/b) \leqslant cf(n)$ для некоторой константы $c < 1$ и достаточно больших n, то $T(n) = \Theta(f(n))$.

Суть этой теоремы в следующем. Сравнивается скорость роста функций $f(n)$ и $n^{\log_b a}$. Если одна из них растёт *значительно* быстрее другой, то она и определяет порядок роста функции $T(n)$. В случае $f(n) = \Theta(n^{\log_b a})$ появляется *дополнительный множитель* $\log n$. Функции $f(n)$ и $n^{\log_b a}$ могут оказаться и такими, что данную теорему применить будет *нельзя*. Для этого они должны быть не Θ-эквивалентными, а скорости их роста—достаточно близкими.

1) Двоичный поиск элемента в упорядоченном массиве

В анализе может помочь и дерево решений для процесса поиска. В узлах дерева решений стоят элементы, которые проверяются на соответствующем проходе. Элементы, проверка которых будет осуществляться в том случае, если целевое значение меньше текущего, располагаются в левом поддереве от текущего элемента, а сравниваемые в случае, если целевое значение больше текущего — в правом поддереве. Дерево решений для списка из семи элементов изображено на рис. 2.1. В общем случае дерево относительно сбалансировано, поскольку мы всегда выбираем середину различных частей списка.

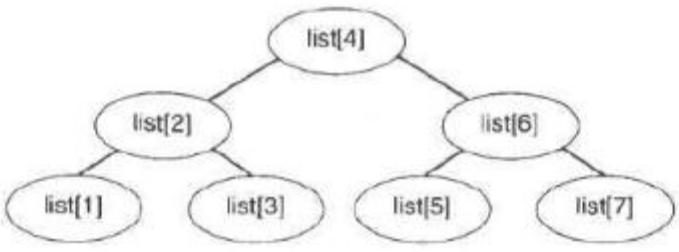


Рис. 2.1. Дерево решений для поиска в списке из семи элементов

Так как высота двоичного дерева равна logn, сложность алгоритма в наихудшем случае равняется Θ(logn)

2) Сортировка слиянием

```
MERGE-SORT(A, p, r)
1  if p < r
2     then q ← ⌊(p + r)/2⌋
3          MERGE-SORT(A, p, q)
4          MERGE-SORT(A, q + 1, r)
5          MERGE(A, p, q, r)
```

Схема работы процедуры Merge-sort(A[1 . . n]):

1. Если n = 1 – готово, выдать A

2. Рекурсивно отсортировать массивы A[1 . . ⌈n/2⌉] и A[⌊n/2⌋ + 1 . . n]

3. Объединить их (merge)

Теперь запишем все шаги алгоритма и их оценки:

Θ(1) 1. Если n = 1 – готово

2T(n/2) 2. Отсортировать A[1 . . ⌈n/2⌉] и A[⌊n/2⌋ + 1 . . n]

Θ(n) 3. Объединить их (merge)

Обычно мы можем пренебречь случаем, когда T(n) = Θ(1) для достаточно малых n, т.к. он не играет роли при асимптотической оценке, но следует быть внимательными.

Для простоты будем предполагать, что размер массива (n) есть степень двойки. Тогда на каждом шаге сортируемый участок делится на две равные половины. Разбиение на части требует времени O(1), а слияние – времени O(n). Получаем рекуррентное соотношение:

$$T(n) = \begin{cases} \Theta(1), & \text{если } n = 1, \\ 2T(n/2) + \Theta(n/2), & \text{если } n > 1. \end{cases}$$

Рассматривая более общий случай дерева, как показано на рис. 10.7, можно заметить, что из определения «времени, затрачиваемого на один узел» следует, что общее время выполнения алгоритма сортировки слиянием равно общему времени, затраченному на все узлы дерева T. Заметьте, что T имеет ровно 2^i узлов на глубине i. Из этого замечания вытекает, что общее время, затраченное на все узлы дерева T на глубине i, составляет $O(2^i \cdot n/2^i)$, что есть $O(n)$. Согласно утверждению 10.1, высота дерева T составляет $\lceil \log n \rceil$. Таким образом, поскольку время, затраченное на каждый уровень $\lceil \log n \rceil + 1$ дерева T, равно $O(n)$, получим следующий результат.

Утверждение 10.3. Алгоритм сортировки методом слияния сортирует последовательность размером n элементов в худшем случае за $O(n \log n)$ времени.

В худшем случае размер последовательности не равен степени 2

3) Быстрая сортировка

```
QUICKSORT(A, p, r)
1  if p < r
2     then q ← PARTITION(A, p, r)
3          QUICKSORT(A, p, q − 1)
4          QUICKSORT(A, q + 1, r)

PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for j ← p to r − 1
4     do if A[j] ⩽ x
5           then i ← i + 1
6                Обменять A[i] ↔ A[j]
7  Обменять A[i + 1] ↔ A[r]
8  return i + 1
```

Наихудшее поведение алгоритма быстрой сортировки имеет место в том случае, когда подпрограмма, выполняющая разбиение, порождает одну подзадачу с n – 1 элементов, а вторую с 0 элементов. Для выполнения разбиения требуется время O(n). Поскольку рекурс. вызов процедуры разбиения, на вход которой подается массив размера 0, приводит к возврату из этой процедуры, T(0) = O(1). Рекуррентное отношение процедуры разбиения: T(n) = T(n-1) + T(0) + O(n)= T(n-1) + O(n). При суммировании промежутков времени, затрачиваемых на каждый уровень рекурсии получается арифметическая прогрессия, что дает в результате максимальное время работы алгоритма $O(n^2)$. Т.о если на каждом уровне рекурсии разбиение максимально несбалансированное, то время работы равно $O(n^2)$.

13.Напишите на языке ассемблер для архитектуры i386 реализацию функции

int strncmp(const char *s1, const char *s2, size_t n) из стандартной библиотеки языка Си, сравнивающей не более n символов двух заданных строк текста.

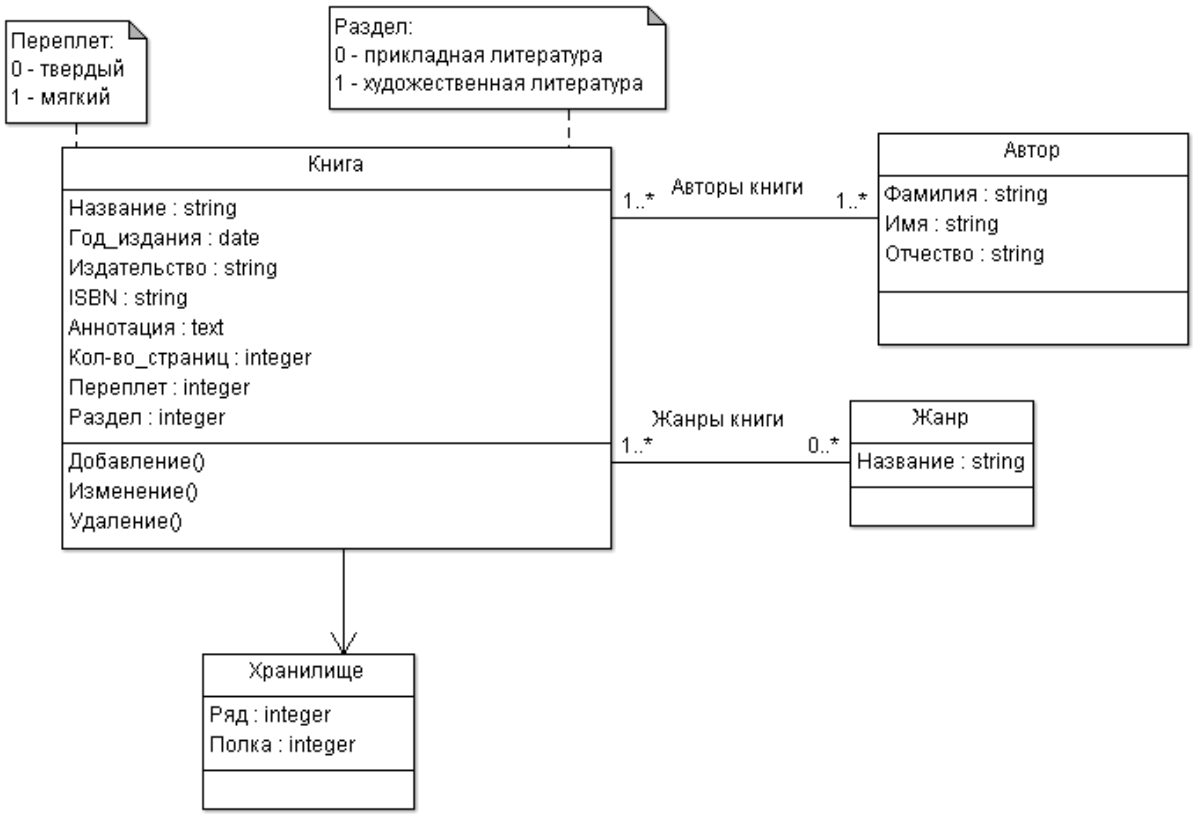
```
.intel_syntax noprefix // синтаксис ассемблера для intel
.globl mystrncmp // глобальное определение mystrncmp
.type mystrncmp, @function // определение mystrncmp как функции
mystrncmp:
        mov edx, [esp+4] // положить в edx первый аргумент (указатель на 1
строку)
        mov ebx, [esp+8] // положить в ebx первый аргумент (указатель на 2
строку)
        mov ecx, [esp+12] // положить в ecx первый аргумент (число n)

// главный цикл
cycle:
        mov byte ptr al, [ebx] // положить в регистр al значение регистра ebx
(кладем в al символ строки 1)
        cmp ecx, 0 // сравниваем ecx с нулем (n == 0)
        je end2 // если ecx равен нулю переходим на метку end2 (n достигло
нуля, выходим)
        cmp byte ptr al, [edx] // иначе сравниваем символ первой строки с
символом второй строки (значение edx)
        jnz end // если разница между ними не равна нулю, переходим в end
(символы не равны)
        cmp byte ptr al, 0 // здесь смотрим не дошли ли мы до конца первой
строки
        jz end2 // если дошли, переходим в end
        inc ebx // увеличиваем ebx на единицу (передвигаем указатель первой
строки к следующему символу)
        inc edx // увеличиваем ebx на единицу (передвигаем указатель второй
строки к следующему символу)
        dec ecx // уменьшаем ecx на единицу ( n = n - 1)
        jmp cycle // прыгаем на cycle (продолжаем цикл)

// завершение алгоритма

// 1 случай, когда дошли до конца строки либо символы не совпали
end:
        xor ecx, ecx
        mov cl, [edx]

        sub byte ptr cl, byte ptr al
        mov eax, ecx
        cbw
        cwde
        ret

// 2 случай, когда n равно нулю и строки равны
end2:
        mov eax, 0
        ret
```

17. Имеется отношение со следующими атрибутами: фамилия преподавателя, кафедра, предмет, название учебника, группа. Преподаватели могут работать на нескольких кафедрах и вести различные предметы, при этом разные преподаватели могут вести предметы с одинаковым названием только у различных групп. Для каждого предмета всегда используется один и тот же набор учебников.
Выясните, находится ли заданное отношение в нормальных формах 1НФ, 2НФ, 3НФ, НФБК, 4НФ, 5НФ, и, если оно не находиться в НФБК, то приведите его к этому виду. Докажите все сформулированные утверждения.

Фамилия преподавателя	Кафедра	Предмет	Название учебника	Группа
Иванов	31	Математика	Высшая математика	1331
Иванов	31	Физика	Учебник по физике	1361
Сидоров	31	Математика	Высшая математика2	1362
Петров	32	Физика	Учебник по физике 2	1371

Первичный ключ: {Фамилия преподавателя, Кафедра, Предмет, Группа(не уверен)}
1 нф по определению
Отношение не состоит в 2 нф, т.к нет полной функциональной зависимости каждого неключевого атрибута от части ключа.
Приводим к 2 нф:
Декомпозируем на 4 отношения:
1)Предмет, Учебник – все атрибуты в ПК, след. Находится во всех нф;
2)Ф. препод., Кафедра – все атрибуты в ПК, след. Находится во всех нф;
3)Ф. препод., Предмет – все атрибуты в ПК, след. Находится во всех нф;
4)Предмет, Группа – все атрибуты в ПК, след. Находится во всех нф.

20. С помощью графической системы нотации UML спроектируйте классы объектов с определёнными атрибутами и операциями для каталога библиотеки.


Переплет:
0 - твердый
1 - мягкий
Раздел:
0 - прикладная литература
1 - художественная литература
Книга
Название : string
Год_издания : date
Издательство : string
ISBN : string
Аннотация : text
Кол-во_страниц : integer
Переплет : integer
Раздел : integer
Добавление()
Изменение()
Удаление()
1..*
Авторы книги
1..*
Автор
Фамилия : string
Имя : string
Отчество : string
Жанры книги
1..*
0..*
Жанр
Название : string
Хранилище
Ряд : integer
Полка : integer