Алгоритмы и структуры данных

Александр Владимирович Иванов, к.э.н.

Лекция 1.
Общие представления о алгоритмах и структурах данных

Что такое алгоритм

- Вычислительный алгоритм - это строго детерминированная последовательность операций, преобразующих исходные данные в искомый результат.
- Вычислительный алгоритм - это точное предписание, задающее вычислительный процесс, обеспечивающий получение результатов, соответствующих определенным входным данным
- Практика выполнение расчета - при решении инженерных задач. Как правило, достаточно выполнить приближенные расчеты с заданной точностью

Формы записи вычислительных алгоритмов

- ***Символьная форма записи*** представляет собой набор математических выражений и текстовых пояснений, где с использованием традиционных обозначений (константы, имена переменных, знаки арифметических и логических операций, имена алгебраических и трансцендентных функций и т.д.) определяются необходимые преобразования исходных данных решаемой задачи.

Формы записи вычислительных алгоритмов

- *Графическая форма записи* - подразумевает представление алгоритма в виде
- *блок-схемы* - совокупности блоков (геометрических фигур, обозначающих отдельные операции алгоритма) и стрелок, указывающих последовательность выполнения вычислений
- *Программная запись алгоритма* представляет собой описание алгоритма на языке, «понятном» ЭВМ (языке программирования). Набор конструкций языка, оформленный по соответствующим правилам, называется *программой*

Структурная классификация алгоритмов

- *Алгоритмы с **линейной структурой** (или просто - **линейные алгоритмы**).*
- *Характерной особенностью алгоритмов этого типа является то, что все операции в них выполняются строго последовательно, без пропусков или повторений.*

Структурная классификация алгоритмов 2

- *Алгоритмы **разветвляющейся структуры**.*
- *Алгоритмическая структура называется разветвляющейся, если она содержит несколько ветвей вычислений и выбор конкретной ветви вычислений происходит в зависимости от выполнения (или не выполнения) заданных условий на переменные алгоритма*

Структурная классификация алгоритмов 3

- *Алгоритмы **циклической структуры**.*
- *Характерной особенностью циклических алгоритмов является наличие в них фрагмента вычислений, называемого телом цикла, который неоднократно выполняется при изменяющемся значении переменной - параметре цикла.*

Структурная классификация алгоритмов 4

- *Многие из реально существующих алгоритмов имеют смешанный характер, т.е. могут содержать линейные участки, разветвления, циклы с известным количеством повторений и итерационные циклы.*
- *В связи с этим составление алгоритмов сразу в законченной форме затруднено. Поэтому для составления сложных алгоритмов рекомендуется использовать нисходящее проектирование программ (метод пошаговой детализации, метод последовательных уточнений).*
- *Его суть: первоначально продумывается общая структура алгоритма, без детальной проработки его отдельных частей. Далее прорабатываются отдельные блоки, не детализированные на предыдущем шаге. Таким образом, на каждом шаге разработки уточняется реализация фрагмента алгоритма, т.е. решается более простая задача.*

Общая классификация алгоритмов 5

- *Существуют **фундаментальные алгоритмы** (сортировка, алгоритмы на графах, шифрование). Это алгоритмы дискретной математики.*
- ***Вычислительные алгоритмы** имеют отношение к непрерывной математике, искомым в которой являются непрерывные объекты или непрерывные процессы.*
- *Объекты – функции, вещественные числа. Процессы – пределы: производные, интегралы, отображения множеств*

Свойства алгоритмов

- *Конечность*

Алгоритм должен всегда заканчиваться после выполнения конечного числа шагов

Число шагов может очень большим, например миллионы или миллиарды, но обязательно конечным

- *Определенность*

Каждый шаг алгоритма должен быть точно определен.

Действия, которые нужно выполнить, должны быть строго и недвусмысленно определены для каждого возможного случая.

На практике алгоритмы могут описываться и на обычном языке, и на формализованных псевдоязыках так и на языках программирования. Метод вычислений, выраженный на языке программирования, называется программой.

- *Ввод*

Алгоритм имеет конечное возможно, равное нулю) число входных данных, которые задаются до начала его работы или определяются динамически во время его работы. Эти входные данные берутся из определенного набора объектов. Например, из множества целых положительных чисел.

Свойства алгоритмов 2

- *Вывод*

У алгоритма есть одно или несколько выходных данных,

т. е. величин, имеющих вполне определенную связь с входными данными.

У алгоритма Евклида имеется только одно выходное значение, а

именно – наибольший общий делитель двух входных значений.

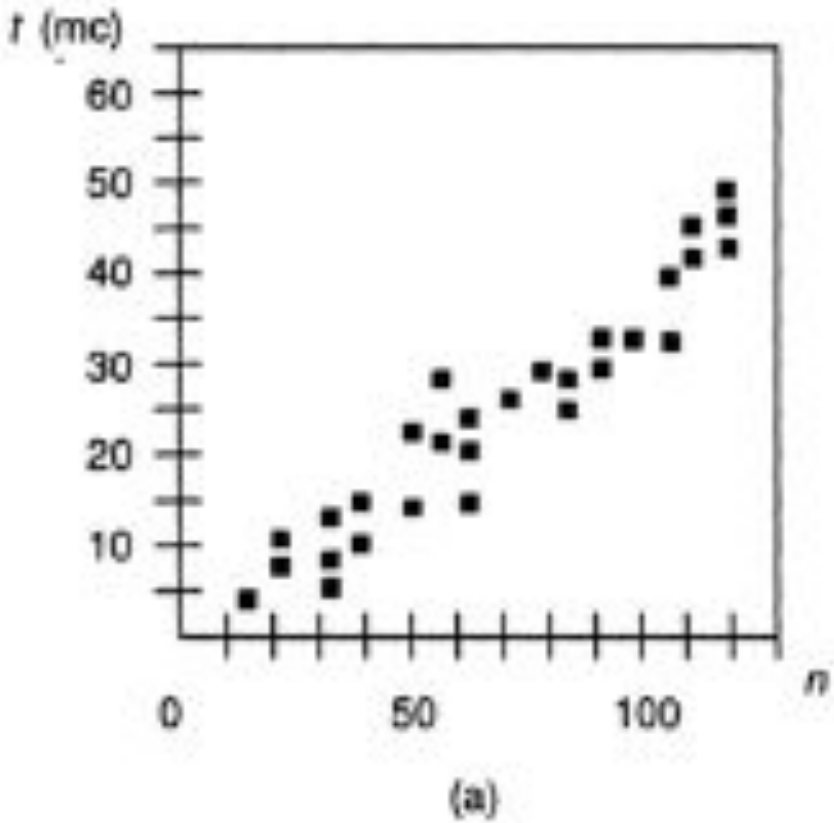
- *Эффективность*

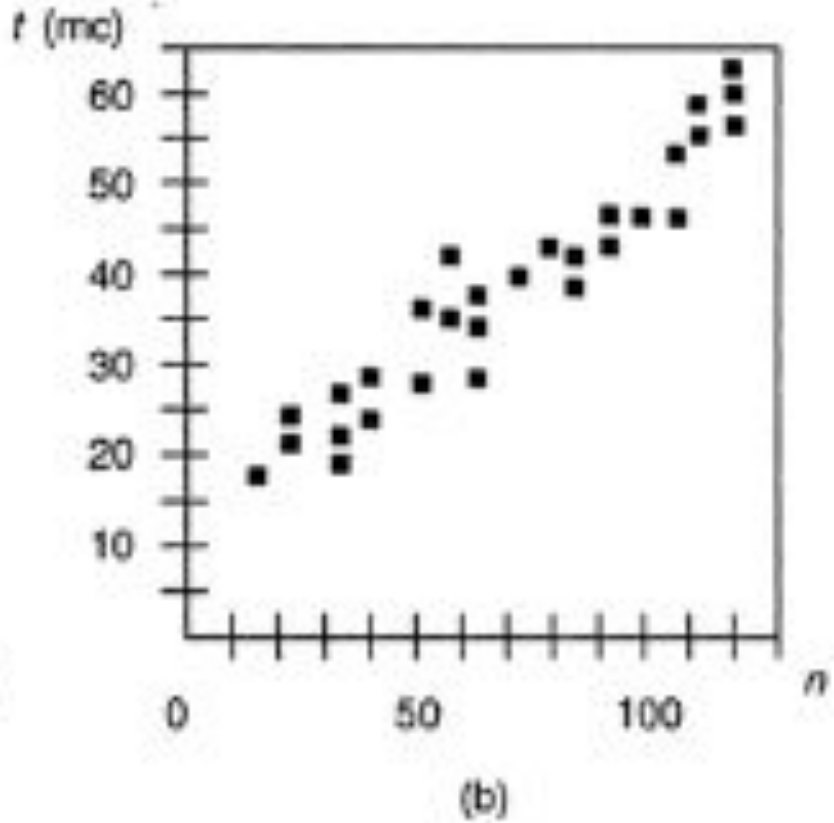
Алгоритм обычно считается эффективным, если все его операторы достаточно просты для того, чтобы их можно было точно выполнить в течение конечного промежутка времени с помощью карандаша и бумаги. В алгоритме Евклида используются только следующие операции: деление одного целого положительного числа на другое, сравнение с нулем и присвоение одной переменной значения другой. Эти операции являются эффективными, так как целые числа можно представить на бумаге с помощью конечного числа знаков и так как существует по меньшей мере один способ деления одного целого числа на другое ("алгоритм деления"). Но те же самые операции были бы неэффективными, если бы они выполнялись над действительными числами, представляющими собой бесконечные десятичные дроби, либо над величинами, выражающими длины физических отрезков прямой, которые нельзя измерить абсолютно точно.

Анализ алгоритмов 1

- *Мера эффективности - определение затрат времени, действительное время, затраченное на выполнение алгоритма в каждом отдельном случае запуска с различными исходными данными.*
- *Измерения должны проводиться с достаточной точностью с помощью системных вызовов, встроенных в язык или операционную систему, для которой написан данный алгоритм*
- *Для решения этой задачи можно провести ряд экспериментов, в которых будет использовано различное количество исходных данных. Далее полученные результаты наглядно представляются с помощью графика, где каждый случай выполнения алгоритма обозначается с помощью точки, координата x которой равна размеру исходных данных n, а координата y – времени выполнения алгоритма t.*

Анализ алгоритмов 2





Анализ алгоритмов 3

- *Три основных ограничения в экспериментальных исследованиях:*
- *• эксперименты могут проводиться лишь с использованием ограниченного набора исходных данных; результаты, полученные с использованием другого набора, не учитываются;*
- *• для сравнения эффективности двух алгоритмов необходимо, чтобы эксперименты по определению времени их выполнения проводились на одинаковом аппаратном и программном обеспечении;*
- *• для экспериментального изучения времени выполнения алгоритма необходимо провести его реализацию и выполнение.*

Анализ алгоритмов 4

- *Общая методология анализа времени выполнения алгоритмов*
- *• учитывает различные типы входных данных;*
- *• позволяет производить оценку относительной эффективности любых двух алгоритмов независимо от аппаратного и программного обеспечения;*
- *• может проводиться по описанию алгоритма без его – непосредственной реализации или экспериментов.*
- *Сущность такой методологии состоит в том, что каждому алгоритму соответствует функция, которая представляет время выполнения алгоритма как функцию размера исходных данных n. Наиболее распространенными являются функции n и n^2. Например, можно записать следующее утверждение: «Время выполнения алгоритма A пропорционально n».*

Псевдокод

- *Непрерывный объект – это объект теоретический. Можно доказать, что существует, но нельзя провести вычисления, так как невозможно записать бесконечную десятичную дробь.*
- *Вычислительный метод – получается при замене непрерывных процессов на дискретные (конечное число шагов каких-то вычислений) и непрерывных объектов на дискретные (функция – набор значений).*

Лекция 2.
Алгоритм и способ его описания

Этапы решения задачи

- *Постановка (формулировка) задачи;*
- *• построение модели, выбор метода решения задачи;*
- *• разработка алгоритма;*
- *• проверка правильности алгоритма;*
- *• реализация алгоритма;*
- *• анализ алгоритма и его сложности;*
- *• отладка программы, обнаружение, локализация и устранение*
- *возможных ошибок;*
- *• получение результата;*
- *• составление документации.*

Что такое алгоритм

- *Алгоритм – строгая и четкая система правил, определяющая последовательность действий над некоторыми объектами и после конечного числа шагов приводящая к достижению поставленной цели*
- *Реализация алгоритма – процесс корректного преобразования алгоритма в машинную программу. Требует также построения целой системы структур данных для представления модели.*
- *Задача анализа алгоритма и его сложности – получение оценок или границ для объема памяти или времени работы алгоритма. Полный анализ способен выявить узкие места в программах.*

Свойства алгоритма

- ***Определенность*** *(**детерминированность**) алгоритма предполагает такое составление предписания, которое не оставляет места для различных толкований или искажений результата, т.е. последовательность действий алгоритма строго и точно определена.*

Свойства алгоритма 2

- ***Массовость*** *определяет возможность использования любых исходных данных из некоторого допустимого множества. Правило, сформулированное только для данного случая, не является алгоритмом (например, таблица умножения не является алгоритмом, а правило умножения «столбиком» есть алгоритм).*

Свойства алгоритма 3

- ***Результативность (конечность)*** *алгоритма означает, что при любом допустимом исходном наборе данных алгоритм закончит свою работу за конечное число шагов*

Классификация алгоритмов

- *Линейные алгоритмы описывают линейный вычислительный процесс, этапы которого выполняются однократно и последовательно один за другим.*
- *Разветвляющийся алгоритм описывает вычислительный процесс, реализация которого происходит по одному из нескольких заранее предусмотренных направлений. Направления, по которым может следовать вычислительный процесс, называются ветвями. Выбор конкретной ветви вычисления зависит от результатов проверки выполнения некоторого логического условия. Результатами проверки являются: «истина» (да), если условие выполняется, и «ложь» (нет), при невыполнении условия.*
- *Циклический алгоритм описывает вычислительный процесс, этапы которого повторяются многократно. Различают простые циклы, не содержащие внутри себя других циклов, и сложные (вложенные), содержащие несколько циклов. В зависимости от местоположения условия выполнения цикла различают циклы с предусловием и циклы с постусловием. В соответствии с видом условия выполнения циклы делятся на циклы с параметром и итерационные циклы.*

Классификация алгоритмов

- ***Линейные алгоритмы*** *описывают линейный вычислительный процесс, этапы которого выполняются однократно и последовательно один за другим..*

Классификация алгоритмов

- ***Разветвляющийся алгоритм*** *описывает вычислительный процесс, реализация которого происходит по одному из нескольких заранее предусмотренных направлений. Направления, по которым может следовать вычислительный процесс, называются ветвями. Выбор конкретной ветви вычисления зависит от результатов проверки выполнения некоторого логического условия. Результатами проверки являются: «истина» (да), если условие выполняется, и «ложь» (нет), при невыполнении условия.*

Классификация алгоритмов

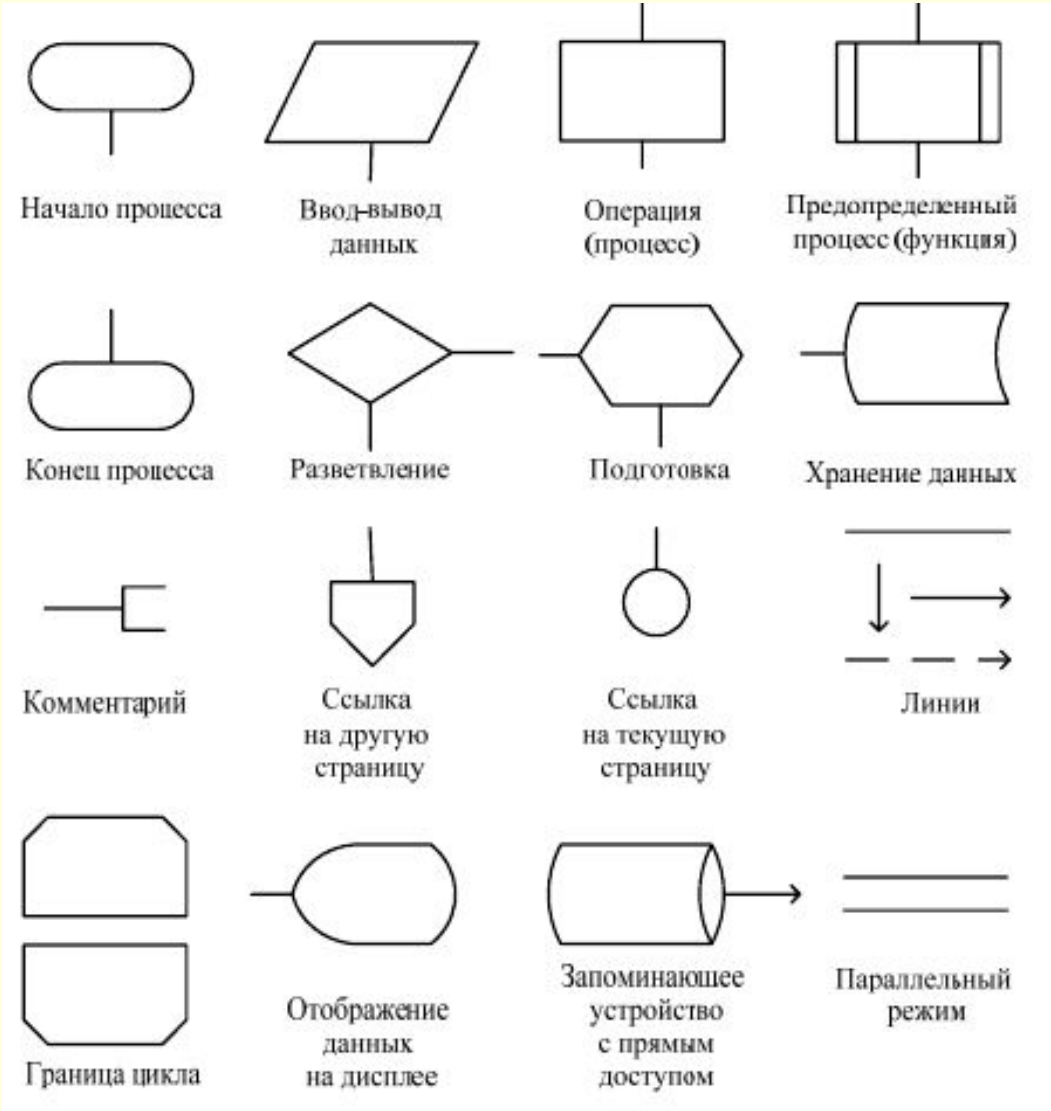
- ***Циклический алгоритм*** *описывает вычислительный процесс, этапы которого повторяются многократно. Различают простые циклы, не содержащие внутри себя других циклов, и сложные (вложенные), содержащие несколько циклов. В зависимости от местоположения условия выполнения цикла различают циклы с предусловием и циклы с постусловием. В соответствии с видом условия выполнения циклы делятся на циклы с параметром и итерационные циклы.*

Способы описания алгоритмов

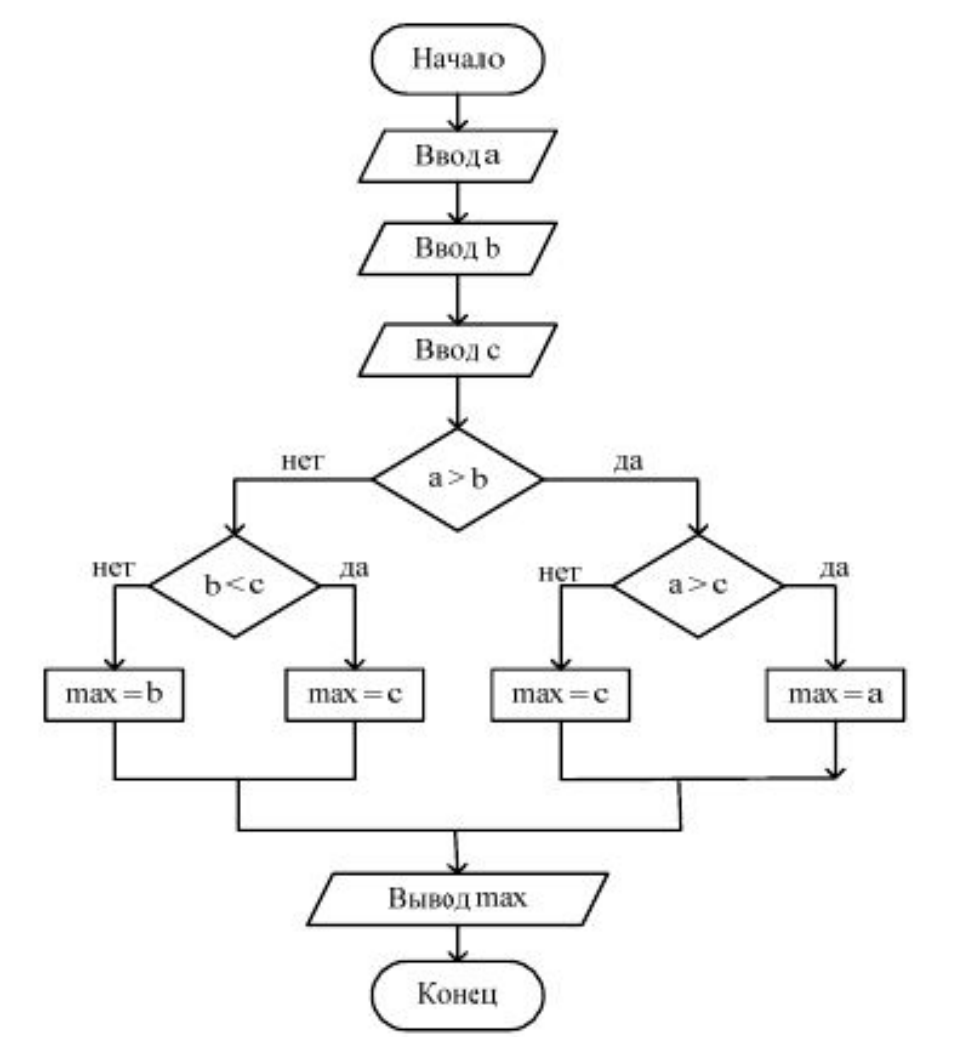
- *Существуют следующие способы описания алгоритмов:*
- *1) запись на естественном языке (словесное описание);*
- *2) изображение в виде схемы (графическое описание);*
- *3) запись на алгоритмическом языке (составление программы).*
- *Способы словесного описания алгоритмов отличаются применяемыми метаязыками (языки, предназначенные для описания языка программирования).*
- *Например, словесное описание алгоритма решения квадратного уравнения* $a \cdot x^2 + b \cdot x + c = 0$ *будет выглядеть следующим образом:*
  1) $D := b^2 - 4\ a\ c;$
  2) если $D < 0$*, идти к 4;*
  3) $x_1 := (-b + (D)^{1/2}) / (2\ a);$
  $x_2 := (-b - D^{1/2}) / (2\ a);$
- *4) Останов*

- *Недостаток описания на естественном языке – плохая наглядность*

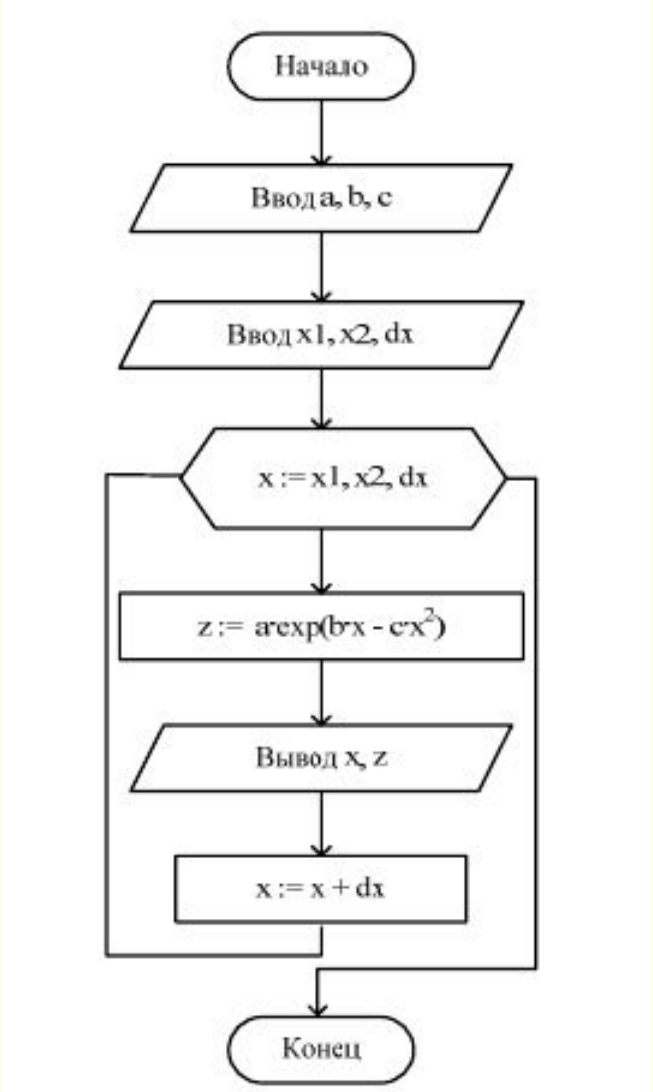
Графическое описание алгоритма



Блок-схема алгоритма нахождения максимального из трех заданных чисел

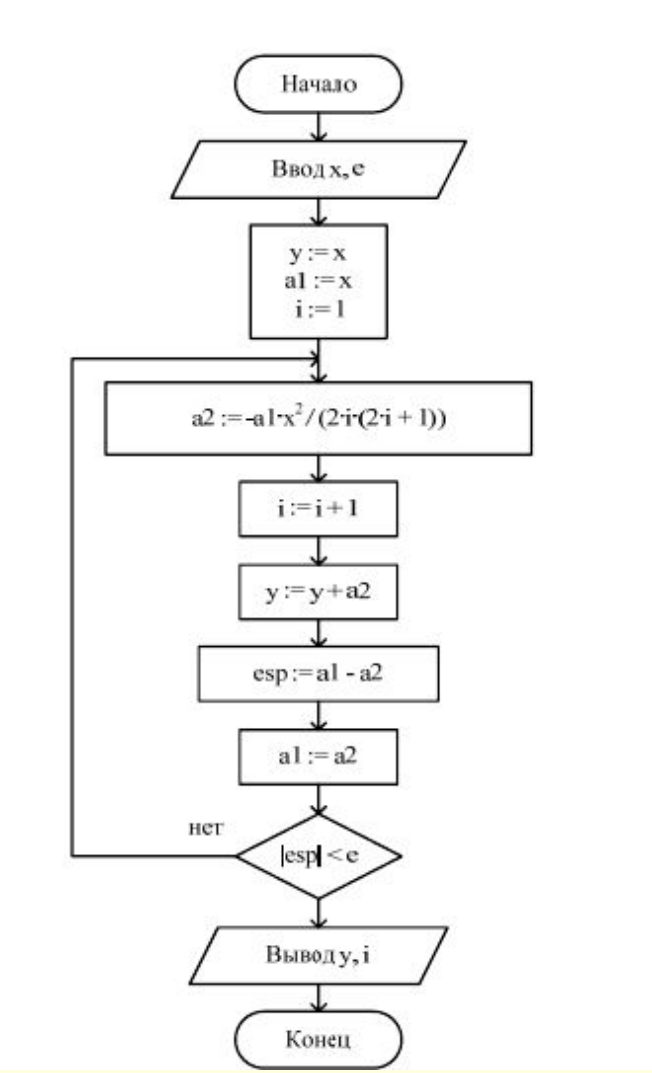


Блок-схема алгоритма вычисления графика функции в заданном интервале



Вычисление значения функции с переменным количеством шагов

- $Y = \sin x = x - x^3 / 3! + x^5 / 5! - x^7 / 7! + \ldots .$



Основные понятия

- *Абстрактный тип данных АТД (формально введен в 1974 г. Б.Лисков.)*
  Но фактически использовалось Кнутом начиная с 1968.
- *АТД – класс абстрактных объектов, которые полностью характеризуется операциями, которые можно производить с этим объектами.*
- *Полезность для программиста: не использует ничего кроме операций с объектами*

Основные понятия

- *Линейно связанный список (linked list) – набор элементов, последовательно связанных друг с другом так, как будто они находятся на одной линии.*
- *Линейная связь - для каждого элемента списка, кроме первого и последнего можно определить свойство следует или предшествует, причем предшественник или последующий элемент может быть один и только один.*
- *Массив – частный случай списка (порядок через индекс)*

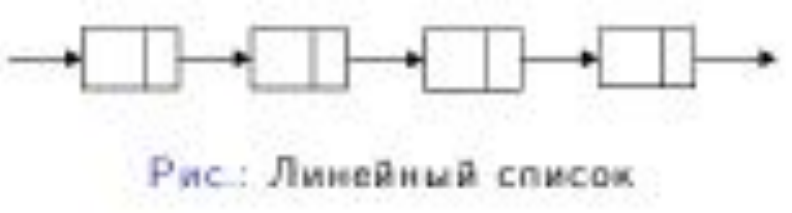


Рис.: Линейный список

Операции со списком

- *1) Доступ к элементам:*

n = first(L)

n = last(L)

n = next(L,n)

n = prev(L,n)

- *2) Изменение списка:*

L = insert_first(L,n) - вставка в начало

insert_last(L,n) - вставка в конец

insert_after(L,p,n) - вставка до p

insert_before(L,p,n) - вставка после p

n = remove(L,n) - удаление n из списка

- *3) Поиск элемента:*

n = search(L,k) - поиск по ключу k

Операции со списком

- *Что нового по сравнению с массивами?*

У массива есть размер и произвольный доступ.

Операция

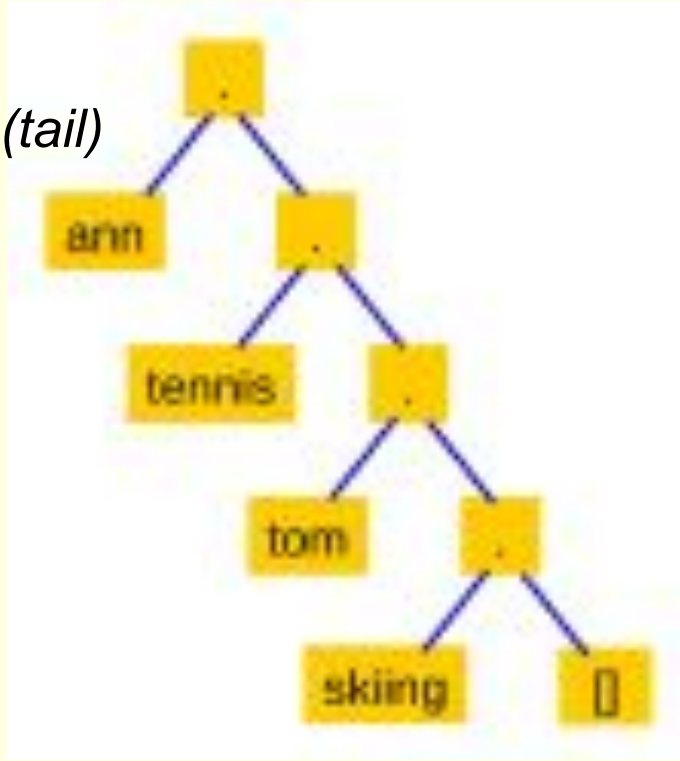
*insert_*

время выполнения O(N) для массива.

У списков есть реализация, в которой операции вставки имеют время выполнения O(1).

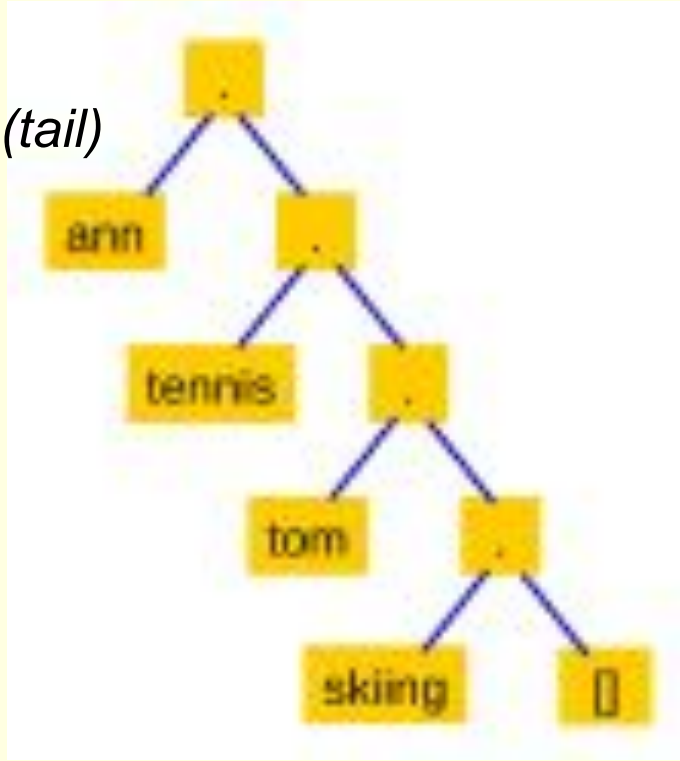
Операции со списком

- *Пример*
- *[ann, tennis, tom, skiing]*
- *[ ]*
- *Элемент ann – голова списка (head)*
- *Список – [tennis, tom, skiing] – хвост (tail)*
- *Хвост – всегда список.*



Операции со списком

- *Пример*
- *[ann, tennis, tom, skiing]*
- *[ ]*
- *Элемент ann – голова списка (head)*
- *Список – [tennis, tom, skiing] – хвост (tail)*
- *Хвост – всегда список.*

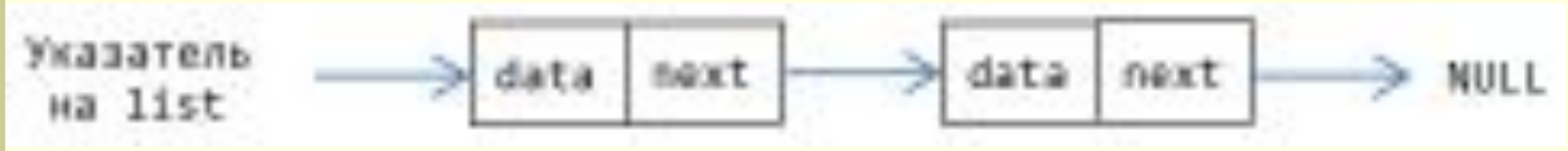


Реализация списка

- *На С*

```
typedef struct list_tag {
int data; // здесь может быть, что угодно
struct list_tag *next;
} list;
```

Для последнего элемента next == NULL.

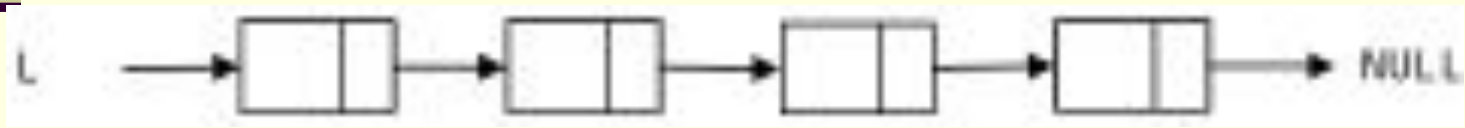


Инициализация списка

```
list * L = 0;
```

Реализация списка

- *1) Операции first и next*
- *list * first(list*L) { return L; }*
- *list * next(list*L) { return L->next; }*
- *2) Операция last*
- *list * last(list*L)*
- *{*
- *while(L->next) L = L->next;*
- *return L;*
- *}*



- *Алгоритм прохода по списку. Операция prev требует того же.*

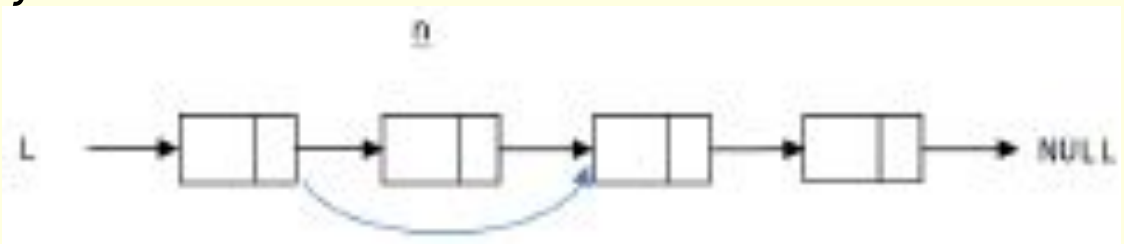
Алгоритмы и структуры

- *3) Операция вставка n после p*
- *void insert_after(list*L,list*p,list*n)*
- *{*
- *n->next = p->next;*
- *p->next = n;*
- *}*
- *4) Операция вставка n до p*
- *void insert_before(list*L, list*p, list*n)*
- *{*
- *insert_after(L,p,n);*
- *int tmp = p->data; p->data = n->data;*
- *n->data = tmp;*
- *}*
- *Исключение: если n->next != NULL. Или n == L.*

Реализация списка

5) Операция remove

```
list* remove(list*L, list*n)
{
list * r = n->next;
// меняем данные со следующим
int data = r->data; r->data = n->data; n->data = data;
// удаляем после n
n->next = r->next; r->next = NULL;
return r;
}
```



- *Исключение: что будет, если n == L. Или n – последний.*

Реализация списка. Важные выводы

Все рассмотренные выше операции имеют сложность O(1). Кроме особых случаев. При этом считаем, что обмен стоит меньше, чем O(N). Иначе нужна реализация класса O (N), чтобы не было обмена. Операция добавления в конец тоже класса O(N)

Реализация списка. Продолжение 6

6) Операция search

```
list* search(list*L, int key)
{
do {
if( L->data == key ) return L;
L = L->next;
} while( L->next );
return 0;
}
```

Сложность операции O(N).

Реализация списка. Продолжение 7 и 8

Вспомогательные операции – создание и уничтожение списков

Конструкторы и деструкторы

7) Операция создания элемента (= одноэлементного списка)

```
list* create(int data)
{
list *n = malloc(sizeof(list));
n->next = 0;
n->data = data;
return n;
}
```

8) Операция удаления списка

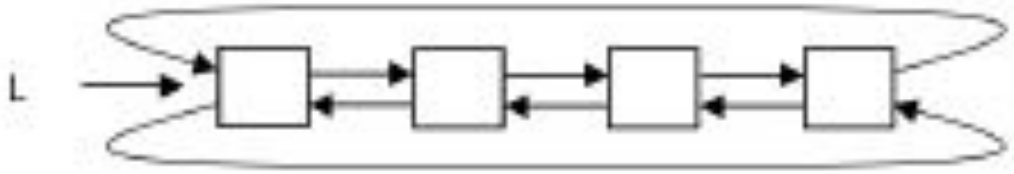
```
list* destroy(list*L)
{
while( L != NULL ) {
list * n = L->next;
free(n);
L = n;
}
return 0;
}
```

Сложность операции O(N).

Реализация двусвязного списка

Если использовать тот же подход:

Но в этом случае операции вставки с $O(N)$ такими же и останутся.
Надо иметь быстрый доступ как к началу, так и к концу списка.



Реализация двусвязного списка

Определение списка

```
typedef struct list_tag {
int data; // здесь может быть, что угодно
struct list_tag *next,*prev;
} list;
```

Инициализация списка (создание списка)

```
list* create(void)
{
list *head = malloc(sizeof(list));
head->next=head->prev=head;
return head;
}
```

Реализация двусвязного списка

1) Операции first и next

```
int isempty(list*L)
{
return L->next == L ? 1 : 0;
}
list * first(list*L)
{
return isempty(L) ? 0 : head->next;
}
list * next(list*L) { }
list * prev(list*L) { }
```

Исключения: если next для последнего или prev для первого

2) Операция last

```
list * last(list*L)
{
return isempty(L) ? 0 : head->prev;
}
```

Не нужен алгоритм прохода по списку.

Реализация двусвязного списка

3) Операция вставка n после p

```
void insert_after(list*L,list*p,list*n)
{
n->next = p->next; n->prev = p;
p->next = n->next->prev = n;
}
```

4) Операция вставка n до p

```
void insert_before(list*L, list*p, list*n)
{
n->next = p; n->prev = p->prev;
n->prev->next = p->prev = n;
}
```

Реализация двусвязного списка

5) Операция remove

```
void remove(list*L, list*n)
{
n->prev->next = n->next;
n->next->prev = n->prev;
n->prev = n->next = 0;
}
```

Исключение: что будет, если n == L. Или n == NULL.

Реализация двусвязного списка

Вспомогательные операции – создание и уничтожение списков

Конструкторы и деструкторы

7) Операция создания элемента (не списка)

```
list* create_elem(int data)
{
list *n = malloc(sizeof(list));
n->next = n->prev = 0;
n->data = data;
return n;
}
```

8) Операция удаления списка

```
void destroy(list*L)
{
while( !isempty(L)) {
list * n = first(L);
remove(n);
free(n);
}
free(L);
}
```

Сложность операции O(N).

.

Реализация двусвязного списка

Пример

```
int main(void)
{
list *head = list_create(0),*n;
int i,N = 6;
for(i=0;i<N;i++) {
n = list_create(rand()%N);
printf("%d\n",n->data);
list_insert_after(head,n);
}
for(n = head; n; n = n->next) {
printf("%d\n",n->data);
}
}.
```

Стеки

Определение стека

Стек – это специальный вид списка, для которого все операции выполняются исключительно с первым элементом списка. Вершина стека (top).

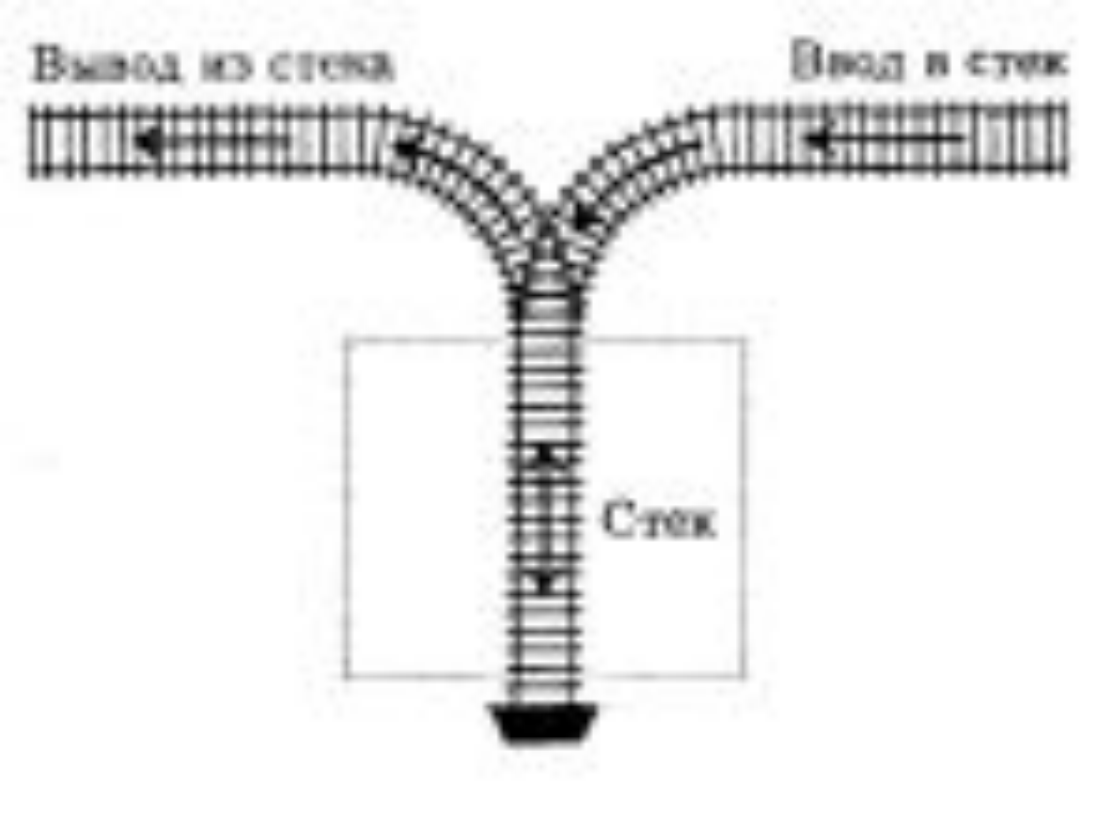
Стек также может называться как LIFO (Last Input First Output).

Абстракция стека имеет большое значение в информатике.

Стековая память программы, стековая виртуальная машина, стековая архитектры ЭВМ, вычисление арифметиченских выражений.

Достоинство стека - это простота операций и скорость их выполнений.

Стек подобен железнодорожному депо



Операции со стеком

Операции со стеком

Доступ к элементам и изменение стека:

n = pop(S) -- извлечение элемента n из стека S

push(S, n) -- вставка элемента n в стек S

Остальные операции вспомогательные и зависят от реализации

Реализация стека

Реализация стека

Можно просто использовать односвязный список. Сделаем через свою реализацию.

```
typedef struct stack_tag stack_t;
struct stack_tag {
void* data;
stack_t * next;
};
```

Пример использования (вывод чисел в обратном порядке)

```
stack_t *s = 0;
for(i=0;i<21;i++) {
a = i*i;
PUSH(s,&a,sizeof(int));
}
while( !EMPTY(s) ) {
POP(s,&a,sizeof(int));
printf(" a = %d\n",a);
}
```

Реализация стека 2

Реализация стека

```
void PUSH(stack_t*S,void*data,int size)
{
stack_t*n=malloc(sizeof(stack_t)+size);
n->data = n + sizeof(stack_t);
memcpy(n->data, data, size);
n->next = S; S = n;
}
void POP(stack_t*S,void*data,int size)
{
stack_t *n = S;
S = S->next;
memcpy(data, n->data,size);
free(n);
}
int EMPTY(stack_t*s) { return !s; }
```

В первой функции две ошибки – с распределением памяти и возвратом. Во второй – ошибка с возвратом.

Реализация стека 3

Реализация стека

1) Память под данные должна распределяться так

```
n->data = n + 1;
```

а не так

```
n->data = n + sizeof(stack_t);
```

2) Изменение головы стека требует его возврата. Итого правильный вариант

```
stack_t *PUSH(stack_t*S,void*data,int size)
{
stack_t*n=malloc(sizeof(stack_t)+size);
n->data = n + 1;
memcpy(n->data, data, size);
n->next = S;
return n;
}
s = PUSH(s,&a,sizeof(int));
```

то же с POP

Реализация стека 4

Реализация стека

Используем макросы

```
#define PUSH(S,d,size) do{ \
STACK*n=malloc(sizeof(STACK)+size); \
n->data = n + 1; \
memcpy(n->data, (d), size); \
n->next = S; S = n; \
} while(0)
#define POP(S,d,size) do{ \
STACK *n = S; \
S = S->next; \
memcpy((d), n->data,size); \
free(n); \
} while(0)
#define EMPTY(S) (S==0 ? 1 : 0)
```

Очереди

Определение очереди

Очередь – это специальный вид списка, для которого добавление элемента осуществляется в конец списка, а извлечение – с начала списка.

Очередь также может называться как FIFO (First Input First Output).

Абстракция очереди имеет большое значение в информатике.

Очередь заданий (процессов) в операционных системах, очередь сообщений в программах или устройствах (буфер клавиатуры)

Реализация очереди

Реализация очереди

Можно просто использовать односвязный список. Но операция добавления в конец будет O(N).

Можно через двусвязный список с фиктивным первым узлом (как выше). Но проход из конца в начало не используется в очереди.

Указатель prev – лишний и лишние операции.

Реализация очереди 2

Реализация очереди

Сделаем через свою реализацию.

```
typedef struct queue_link_tag queue_link;
struct queue_link {
void* data;
queue_link * next;
};
typedef struct queue_tag queue_t;
struct queue_t {
int size;
queue_link * head,*tail;
};
```

Пример использования (вывод чисел в том же порядке)

```
queue_t *q = create_queue(sizeof(int));
for(i=0;i<21;i++) {
a = i*i;
ADD(q,&a);
}
while( !EMPTY(q) ) {
REMOVE(q,&a);
printf(" a = %d\n",a);
}
```

Реализация очереди 3

Реализация очереди

```
void ADD(queue_t*q,void*data)
{
queue_link*n=malloc(sizeof(queue_link)+q->size);
n->data = n + 1;
memcpy(n->data, data, q->size);
q->tail->next = n;
n->next = 0;
q->tail = n;
}
void REMOVE(queue_t*q,void*data)
{
stack_link *n = q->head;
q->head = q->head->next;
memcpy(data, n->data,q->size);
free(n);
}
```

Деки

Определение дека

Дек (deque - double ended queue) – это специальный вид списка, для которого как добавление элемента, так и извлечение могут осуществляться и с конца, и с начала списка. (Двусторонняя очередь)

Четыре операции - аналогичные стеку и очереди

Задание. Деки

1) Реализовать все операции для списка и двусвзяного списка в данных выше реализациях

2) Реализовать абстрактный тип дек.

Метод пузырька

```
#include <stdio.h>
int main() {
   int n, i, j;
    scanf_s("%d", &n);
   int a[n];
   // считываем количество чисел n

   // формируем массив n чисел
   for(i = 0 ; i < n; i++) {
       scanf_s("%d", &a[i]);
   }
   for(i = 0 ; i < n - 1; i++) {
      // сравниваем два соседних элемента.
      for(j = 0 ; j < n - i - 1 ; j++) {
          if(a[j] > a[j+1]) {
             // если они идут в неправильном порядке, то
             //  меняем их местами.
             int tmp = a[j];
             a[j] = a[j+1] ;
             a[j+1] = tmp;
          }
       }
   }
}
```

Понятно, что после первого «пробега» самый большой элемент массива окажется на последнем месте. После второго пробега мы будем уверены, что второй по величине элемент находится на предпоследнем месте.

Метод вставки

```
#include <stdio.h>
int main() {
   int n, i, j;
    scanf_s("%d", &n);
   int a[n];
   // считываем количество чисел n

   // формируем массив n чисел
   for(i = 0 ; i < n; i++) {
       scanf_s("%d", &a[i]);
   }
```

Метод Шелла

```
#include <stdio.h>
int main() {
   int n, i, j;
    scanf_s("%d", &n);
   int a[n];
   // считываем количество чисел n

   // формируем массив n чисел
   for(i = 0 ; i < n; i++) {
       scanf_s("%d", &a[i]);
   }
```

Усовершенствованные методы

```
#include <stdio.h>
int main() {
   int n, i, j;
    scanf_s("%d", &n);
   int a[n];
   // считываем количество чисел n

   // формируем массив n чисел
   for(i = 0 ; i < n; i++) {
       scanf_s("%d", &a[i]);
   }
.
```

Спасибо

за

внимание!

Иванов Александр Владимирович

alexanderivanov52@yandex.ru