1. Уровни описания структур данных. (203)

При описании объекта необходимо четко понимать его абстрактные свойства (и их реализацию в языках)
Для этого удобно иметь определение типа данных (класса всех объектов, обладающими этими свойствами).

Функциональная спецификация типа данных – внешнее формальное определение, не зависящее от языка программирования и ЭВМ. Дать формальное определение типа данных – это задать

множество значений этого типа
множество изображений этих значений с правилом интерпритаций
базовое множество атрибутов этого типа: изображения (некоторых) выделенных значений, операции/отношения и их свойства и функции создания, доступа и модификации объектов этого типа.

Пример**:** целый тип – последовательность цифр со знаком или без (множество изображений значений); с операциями [=, -, *, /, div, mod], их свойствами [ассоциативность, дистрибутивность, …] и отношениями [=, !=, <, <=, >=, >] (а также ряд функций аргументами/результатами которых являются значения этого типа); которую можно представить в полиномиальной форме в позиционной системе счисления с основанием 10

Логическое описание – отображение функциональной спецификации на средства выбранного языка программирования. Возможны 2 ситуации: в выбранном языке 1) есть подходящий тип данных или 2) нет (втаком случае – разбить объект на части, которые можно описать на выбранном языке).
Пример**:** x – переменная целого типа :intx;

Физическое представление – как тип представляется в памяти ЭВМ {описание всегда связаннос линеаризацией (2мерный массив в Фортране линеаризован “по столбцам”, а в С/Паскале - “по строкам”)}

2 вида физического представления объектов в памяти: сплошное и цепное
Сплошное – объект размещён в непрерывном куске памяти (для простых объектов)
Цепное – объект разбит на куски, расположенные в памяти “как попало” и в каждом есть указатель на следующий (для динамических объектов).

2. Статические и динамические объекты программ. (4.1 ст. 1/196)

Некоторые *связи между объектами* и *свойства объекта* остаются неизменными при любом использование их области действия (участка программы, где этот объект считается существующим). Такие свойства и связи называются *статическими.*Размер памяти не изменяется во время исполнения программы.

Другие свойства и связи называются *динамическими*. Во время исполнения программы выделяется память (значение переменной; связь формального параметра с конкретным фактическим в результате вызова процедуры). Например, конкретное значение переменной — динамическое свойство.

Часто статические и динамические характеристики называют соответственно характеристиками *периода компиляции* (трансляции) и *периода выполнения* (runtime). В период компиляции исходные данные программы неизвестны => динамические характеристики недоступны. Известна лишь информация, извлекаемая непосредственно из текста программы и тем самым относящаяся к любому ее исполнению (т. е. статическая информация).

Если память выделяется (распределяется) в процессе трансляции и ее объем не меняется от начала до конца выполнения программы, то такоѝ объект является *статическим.* Если же память выделяется во время выполнения программы и ее объем может меняться, то такой объект является *динамическим.*

Но такое деление можно назвать чёрно-белым, т.к. все машинные языки обладают неограниченным динамизмом (можно менять ВСЁ)
В тоже время есть статические языки (в которых почти всё статично).
+: более надёжно и быстро
-: язык сложнее, код объёмне

3. Ссылочный тип данных. (4.2 ст. 2/197)

В языках со строгой дисциплиной описания (Паскаль) принято не именовать, а обозначать динамический объекты через ссылку на него (сама ссылка – статический объект). Ссылка – указатель на место в памяти, с уточнением того, какого типа данные там находятся(разные ссылочные типы указывают на разные типы данных) Пустое ссылочное значение nil принадлежит любому из ссылочных типов. Оно указывает на отсутствие связи с объектом.
Перед использованием указанного участка памяти, следует его отчитстить от старых данных, для записи новых, т.к из-за неконтролируемости транслятором использования неопределенных значений, которые были определены для прошлой программы (1), но не для нынешней (2), т.е в памяти последовательность 1 и 0 имеет случайный порядок для (2), что может привести к непредсказуемым результатам.

Для “ссылок” определены операции присваивания (=), разыменования (*p) (обеспечивает доступ к значению обозначаемого ссылкоѝ объекта.) и равенства (==).
В С функция *malloc* возвращает указатель на выделенную память, функция *realloc* увеличивает место по данной ссылке а функция *free* – возвращает системе место по данной ссылке (эта ссылка становится не действительной). Память выделяется и возвращается в “кучу” - специально зарезервированную для этого память.

Ссылочный тип – внутримашинный, хотя в С он согласован с целым → можно пользоваться printf и scanf (%p)

4. Файл. Функциональная спецификация. (5.2 ст.12/207)

Понятие *файл* используется как абстракция данных, хранящихся в памяти, что позволяет единообразно формулировать общие характеристики файлов и определять операции над ними.
Файл – динамический объект (=>файл так же может быть пустым), т.к. его размер может быть изменён.
Но, в отличии от *динамических объектов в основной памяти*, у файла должно быть *имя* для нахождения, что является следствием большего времяни “жизни” (обычно, файл живёт дольше программы)

При этом, операции, которые можно проводить над файлами заложены в конструкцию хранящей памяти (на жёстком диске можно читать, изменять и удалять; на CD-ROM – только читать; принтер – только писать)

По отношению к программе файл может быть внутренним и внешним.
Внутренний – файл, созданный программой для временного хранения → память, выделенная под файл, должна быть возвращена системе после завершения программы.

Внешний – файл, сгенерированный до начала выполнения программы или в процессе её выполнения и оставленный для “длительного” хранения.

ВС файлами также являются стандартные потоки (stdin, stdout, stderr).
{также, в С любой файл может быть открыт и использован как текстовый или нетекстовый (двоичный)}

Функциональная спецификация: Обозначим F_T – файловый тип с компонентами типа T. Значения Ft – конечные последовательности T. Такое бесконечн множество можно определить формально через операцию конкатениции: f_1 = {x_1, x_2, …, x_m} f_2 = {y_1, y_2, …, y_n} → f1 || f2 = {x_1, …, x_m, y_1, …, y_n}. Где || - знак конкатениции. Тогда множество значений файлового типа строго определяется только следующими порождающими правилами:
{} есть файл типа F_T (пустая последовательность/файл)
если f – файл типа F_T и t – объект типа T, то f || {t} – файл типа F_T.

Операции:
конкатенации; её свойство – несимметричность,
присваивание – покомпонентное копирование с сохранением порядка
[отношение] равенства – 2 файла равны ←→ длины файлов и их соответствующие компоненты равны
функции:
- создание (порождение пустого файла)
- доступ (последовательно к каждой компоненте)
- модификация (дозапись новой(ых) компоненты в конец файла)
- уничтожение (стирание)

5. Файл. Логическое описание. Физическое представление.

Логическое описание:
Для описания файла в Си необходимо объявить переменную предопределённого типа FILE*. Созданный компилятором виртуальный файловый дескриптор может быть *динамически* связан с конкретным файлом, потоком или устройством с помощью функций стандартной библиотеки языка Си.
FILE* <имя объекта—файла>;//переменная-дискриптер файла (ссылка на файл)

Физическое представление: При обработке файла программа выделяет память под буфер-файла, достаточная для размещения значения одной компоненты этого файла.

6. Вектор. Функциональная спецификация. Логическое описание и физическое представление. (213)

Вектор или динамический массив – одномерный массив переменной длины.
Из-за спецификации статических массивов (фиксированная размерность) они не всегда полезни в применении, тогда используют динамические массивы или вектора с переменной длиной (в идеале ограниченность размерности зависит только от доступной памяти) =>непроизводительные задержки отсутствуют. Недостатками таких массивов является большее среднее время доступа к элементу.

Функциональная спецификация (+ логическое): Вектор – последовательность однотипных элементов переменной длины и время доступа к элементам постоянна и не зависит от длины последовательности. Индексация элементов производится с 0. Длина важна, т.к.

операции (+ , скалярное умножение, сравнение) можно проводить только с векторами равной длинны.
Операции:
создание

```
typedef struct {
T* data;
int size ; } Vector;

void Create(Vector* v, int sz) {
  v—>size = sz;
  v—>data = malloc(sizeof(T) * v—>size); }
```

проверка на пустоту

```
bool Empty(Vector* v) {
  return v—>size == 0; }
```

получение длинны

```
int Size (Vector* v) {
return v—>size; }
```

получение i-го элемента

```
T Load(Vector* v, int i) {
if((i >= 0) &&(i < v—>size))
  return v—>data[i]; }
```

дозапись (сохранение нового элемента)

```
void Save(Vector* v, int i, T t) {
if((i >= 0) &&(i < v—>size))
  v—>data[i] = t; }
```

изменение длинны (урезание / выделение дополнительного места для дозаписи)

```
void Resize(Vector* v, int sz)
v—>size = sz;
v—>data = realloc(v—>data, sizeof(T)* v—>size); }
```

равенство векторов

```
bool Equal(Vector* 1, Vector* r)
if (1 — >size != r—>size)
  return false;
for(int i = 0; i < 1—>size; i++)
  if(1 —>data[i] != r—>data[i])
  return false;
return true; }
```

уничтожение

```
void Destroy (Vector* v) {
v—>size = 0;
free (v—>data); }
```

Физическое представление: Набор пронумерованных однотипных элементов, записанных друг за другом. Если место кончилось, то выделяем ещё место рядом (расширяем место, занятое последовательностью). Если рядом свё занято – выделяем место там, где можно и копируем туда наш набор

7. Очередь. Функциональная спецификация. (5.4 ст. 216)

Очередь – упорядоченное множество переменных (возможно нулевым), числом элементов, на котором определены следующие операции: а) постановка в очередь нового элемента
проверка на пустоту

просмотр первого (самого древнего) элемента, если он есть
извлечение (удаление из очереди) первого элемента, если он есть

Очереди применяются в информатике для 2х классов задач:
моделирование реальных очередей (электронные системы связи + предсказание появления человеческих очередей
решение собственных задач информатики (в области ОС – запуск/завершение процессов + допуск к регистрам + …)

Более редкая форма очереди – дек (DouleEndedQueue) – запись и чтение возможны с обоих концов (если ограничить чтение 1м концом – модель очереди с блатными) {железнодорожные разъезды, трамвайные депо, ...}

Функциональная спецификация: Обозначим Qt – тип очереди компонент типа T, который можно характеризовать следующими операциями:
создать (всегда создаётся пустая очередь)
проверка на пустоту
чтение первого элемента
постановка в очередь нового элемента
удаление из очереди первого элемента
подсчёт длины очереди
уничтожение очереди

8. Очередь. Логическое описание и физическое представление (файл) (219)

Логическое описание: {пустой очереди соответствует пустой файл (внешний или внутренний)}
fopen + fread + fwrite + удаление из очереди можно сделать 2я способами – переписать файл или пропускать n первых элементов (сделать цикл, который n раз прогонит чтение “в пустую”)

Физическое представление: Сейвим данные в файл; [до]запись по умолчанию идет в конец, удалять элементы с начала – либо копируем файл без первого элемента во временный файл, а потом назат, либо просто пропускаем первые n элементов

9. Очередь. Логическое описание и физическое представление (массив).(ст. 222)

3 стратегии:
“трудоголик” (голова зафиксирована, хвост подвижен, при каждом извлечение первого все элементы сдвигаются вперёд, не эффективная)
“лентяй” (как “трудоголик”, но сдвижение НЕ происходит до тех пор, пока это возможно)
кольцевой буфер (голова и хвост – один элемент (подвижный), заффиксирован только размер, хранятся (в структуре) начальный элемент и длина очереди, когда очередь подходит к концу выделенной памяти – новые элементы записываются в начало, на освободившиеся извлечением первых элементов места)

```
const int POOL SIZE = 100;
typedef struct {
int first;
int size ;
T data[POOL_SIZE];
} queue;

void Create (queue* q) {
q—>first = 0;
q—>size = 0; }
```

```
bool Empty(const queue* q) {
return q—>size == 0;}

int Size (const queue* q) {
return q—>size;}
```

Запись новой компоненты в массив кольцевого буфера производится в свободный элемент, индекс которого вычисляется по формуле *(first + size)* **mod** *POOL SIZE,* происходит так, что выхода за границу массива не возникает.

```
bool Push(queue* q, const T t) {
if (q—>size    ==POOLSIZE) return false;
// Операторвзятияостатка % обеспечиваетзакольцованностьбуфера
q—>data[(q—>first + q—>size++) % POOLSIZE] = t;
return true;}

bool Pop(queue* q){
if (! q—>size) return  false;
q—>first++;
q—>first %= POOL SIZE;
q—>size--;
return true;}

T Top(const queue* q) {_
if (q—>size) return q— >data[q— >first];
```

10. Очередь. Логическое описание и физическое представление (динамические объекты) (227).

Держим 2 структуры:

сама очередь – указател на первый и следующий после последнего (терминатор – упрощает добавление нового элемента в очередь) элементы (можно также хранить размер очереди)

элемент очереди – собственное значение элемента и указатель на следующий.

```
struct Item {
T data;
struct Item* next;

typedef struct {
struct Item* first;
struct Item* last;
int size ;
} queue;

void Create (queue* q){
q—>first = q—>last = malloc(sizeof( struct Item));
q—>size = 0;

bool Empty(const queue* q){
return q—>first == q—>last;

int Size (const queue* q){
return q—>size;

bool Push(queue* q, const T t){
if (!(q—>last—>next = malloc( sizeof(struct Item))))
  return false;
q—>last —>data = t;
q—>last = q—> last—> next;
q—>size++;
return true;

bool Pop(queue* q){
if(q—>first == q—>last)
```

```
  return false;
struct Item* pi = q—>first;
q—>first = q—>first—>next;
q—>size;
free (pi);
return true;

T Top(const queue* q){
if(q—>first != q—>last)
  return q— >first—>data;

void Destroy (queue* q){
while (! Empty (q)){
  struct Item* pi = q—>first;
  q—>first = q—>first—>next;
  free (pi);}
free(q—>first);
q—>first = q—>last = 0:
q—>size = 0;

void Reverse(queue* q){
if(!Empty(q){
  T t = Top(q);
  Pop(q);
  Reverse (q);
  Push(q, t);}}
```

11. Стек. Функциональная спецификация.

Стек – структура, с единственной головкой (чтение + запись), последовательным доступом и неразрушимой записью.

Функциональная спецификация: Обозначим St – стеков компонент типа T, который можно характеризовать следующими операциями:

создать (всегда создаётся пустой стек)
проверка на пустоту
подсчёт глубины
постановка в стек нового элемента
удаление из стека первого элемента
чтение первого элемента
уничтожение стека

12. Стек. Логическое описание.

Самый свежий элемент стека, т. е. последний введенный и еще не уничтоженный играет особую роль; именно его можно рассмотреть или уничтожить. Этот элемент называет- ся верхушкой стека. Оставшуюся часть можно назвать телом стека и оно само является, по существу, стеком; если снять со стека верхушку, то тело превращается в стек. Таким образом, стек, как и очередь, является рекурсивной структурой данных.
ДЛЯ обеспечения эффективной вычислимости необходим терминатор, ограничивающий глуби- ну рекурсии описания, роль которого, как в файлах и очередях, играет пустой стек

13. Стек. Физическое представление (массив).(ст. 239)

Создаём структуру, в которой храним массив из n элементов выбранного типа и заполненность стека (переменная) {функция получения элемента – returns->data[s->size – 1];}
ограничим максимальную глубину стека ве- личиной *POOL SIZE.* Все элементы стека в таком случае размещаются в массиве *data* длины *POOL SIZE.* Переменная *size* играет двойную роль: это длина стека, а величина *size* — 1 является индексом его вершины.

```
const int POOL SIZE = 100;
typedef struct{
int size;
```

```
T data[POOL_SIZE];} stack;

void Create(stack* s){
s –>size = 0;}
bool Empty (stack* s){
return s–>size == 0;}

int Size(stack* s){
return s–>size;}

bool Push(stack* s, T t){
if(s–>size >= POOL SIZE)
  return false;
s–>data[s–>size++] = t;
return true;}

bool Pop(stack* s){
if (! s –>size)
  return false;
s –>size--;
return true;}

T Top (stack* s){
if (s–>size)
  return s–>data[s–>size - i ] ;}

void Destroy (stack* s) {}
```

14. Стек. Физическое представление (динамические объекты). (240)

понадобится тип «элемент стека», который содержит ссылку на следующую компоненту.

```
struct Item {
T data;
struct Item* prev;}
```

Самоопределение стека заключается в описании двух переменных: указателя на вер-шину стека и целой переменной-глубины стека.

```
typedef struct {
struct Item* top;
int size ; } stack;

void Create (stack* s) {
s–>top = 0;
s –>size = 0;}

bool Empty(stack* s) {
return s–>top == 0;}

int Size(stack* s) {
return s–>size;}

bool Push(stack* s, T t) {
struct Item* i = malloc(sizeof(struct Item));
if(!i)
  return false;
i –>data = t;
i –>prev = s–>top;
s–>top = i;
s–>size++;
return true;}

bool Pop(stack* s){
if (! s–>size)
  return false;
struct Item* i = s–>top;
s –>top = s–>top–>prev;
s–>size free (i);
return true;}

T Top (stack* s){
```

```
if (s—>top)
  return s—>top—>data;}

void Destroy (stack* s){
while(s—>top) {
  struct Item* i = s—>top;
  s—>top = s—>top—>prev;
  free (i);}
s—>top = 0;
s—>size = 0;}
```

15. Линейный список. Функциональная спецификация.(ст. 243)

Линейный список – конечное линейно упорядоченное динамическое множество элементов типа T (точнее мультимножество, т.к. В нём могут находиться элементы с одинаковыми значениями), в котором каждый элемент доступен без извлекания всех предыдущих (как в деке).
{порядок определяется не номерами, а относительно расположения элементов, т.е. Каждый последующий указывает место предыдущего (если и наоборот, то список двусвязный)}
{можно закольцевать – сказав, что после последнего элемента идёт первый, а перед первым - последний}

(к примеру – список программ к выполнению, расположенных по убыванию приоретета ← добавление новой программы в список относительно её приоретета)
Линейные списки естественно использовать всякий раз, когда встречаются упорядоченные множества переменного размера, где включение, поиск и удаление элементов должны выполняться не систематически в голове или хвосте, как для файлов или стеков, а в произвольных последовательно достигаемых местах, *но с сохранением* порядка следования остальных элементов.
Для *поиска* элемента в списке надо просматривать его с начала, сравнивая искомое значение со значением, содержащимся в очередном хранимом элементе.
Список, как очередь и стек, является последовательной структурой с линей- ным временем доступа — максимальным для линейных структур.
Для *добавления* нового элемента в список необходимо указать значение, *перед* (или после) которого надо сделать вставку.

Функциональная спецификация: Обозначим Lt – линейный список компонент типа T, характерезуемый следующими операциями:

- создать (всегда создаётся пустой список)
- проверка на пустоту
- подсчёт длины
- получение первого / последнего элемента
- постановка в стек нового элемента
- получение следующего / предыдущего элемента
- вставка нового элемента относительно какого-то
- удаление из линейного списка элемента
- уничтожение линейного списка

16. Линейный список. Логическое описание.(ст. 246)

Поскольку в Си встроенных списков нет, а есть только библиотечные (std::list в STL), их логическое описание тесно переплетено с их физическим представлением.

17. Линейный список. Физическое представление. Итераторы.

(пока список не изменяется, его удобно представлять в виде цельного куска памяти → при удаление/вставке внутрь элемента возникает проблемма из-за боьших затрат на поддержание целостности[, в чём превосходство динамических структур])
Для удобства организации списка и обеспечения единообразия доступа к нему определим объекты, обладающие функциями перехода от данного элемента списка к соседним. Зададим для них отношения равенства и неравенства. Два таких объекта равны тогда и только тогда, когда они указывают на один и тот же элемент списка. Также предоставим возможность чтения и записи элемента списка посредством введенных объектов. Такие объекты принято называть *итераторами,* и, конечно же, для них надо определить соответствующий тип данных.
Итератор предназначен для навигации по списку, чтения и перезаписи элементов списка Возвращаемым значением функции ПЕРВЫЙ будет не сам элемент, а итератор, который на него указывает. А вот функция ПОСЛЕДНИЙ в этом случае будет выдавать не последний элемент списка, а итератор, *указывающий на фиктивный элемент после конца списка — терминатор.*

Каждый элемент списка содержит не только собственное значение, но и указатели на предыдущий и следующий элементы

```
struct Item{
    struct Item *prev;
struct Item *next;
    T * data;  //Т-типэлементоввнашемсписке }

struct Iterator{ // "бегунок"
    Item * node;}

bool Equal(const Iterator* lhs, const Iterator* rhs){
return lhs—>hode == rhs—>node;}

bool NotEqual(const Iterator* lhs, const Iterator* rhs){
return  !Equal(lhs,  rhs);}  //Функцияпроясняетусловияветвленияициклов,
ноиупрощаетэтиусловияудаляяизнихунарныйоператор.

Iterator Next (Iterator* i){
i —>node = i—>node—>next;
return i;}

Iterator Prev(Iterator* i){
i —>node = i—>node—>prev;
return i;}

T Fetch (const Iterator*i) { // FetchQ нужнадлячтенияэлементасписка,
накоторыйуказываетитератор.
return i—>node—>data;}

void   Store(const   Iterator*   i,   const   T   t)   //   Store()
позволяетзаписатьданныевэлементсписка
{i —>node—>data = t;}
```

Изменяя функциональность итераторов путем запрещения функций *FetchQ* или *Store()* можно получить списки *только для чтения* или *только для записи*
Аналогично описанным *прямым* итераторам, можно дополнительно определить *обратные* итераторы, которые позволят пройти список с последнего элемента до первого, полностью сохраняя семантику и идиомы прямых итераторов.

18. Линейный список. Физическое представление (массив).

Если необходима боьшая быстродействненность и известна максимально возможная длинна списка (POOL_SIZE), то прибегают к более эффективной реализацие списка на массиве.

{к перечисленным в п. 19 функциям необхдимо дописать следующие}

```
  const int POOL_SIZE = 100; //допустиммакс. Размер- 100 элементов

  typedef struct{
    int size;
  struct Item *top;  //указательнапервыйсвободныйэлемент
  struct Item data[POOL_SIZE + 1];
  }List;

  void Create(List *l){  //созданиесписка
    int I;
    for(i = 0; I < POOL_SIZE,; i++)
      l->data[i].next = &(l->data[i+1]);  //унасодносвязныйсписок
    l->data[POOL_SIZE - 1].next = 0;  //терминальныйэлемент-нуль
    l->head                  =                  &(l->data[POOL_SIZE]);
//анаходитсятерминальныйэлементздесь (вконцесписка)
    l->head->next      =            l->head->prev      =      l->head;
//исноватерминальныйэлемент
  l->top = &(l->data[0]);
    l->size = 0;
  }

  Iterator Insert(List *l,  Iterator *i, const T){
  Iterator res = { l->top}; //выделениепамятиподновыйэлемент
    if(!res.node)  //еслинеудалосьвыделитьпамять (массивзаполнен)
  return Last(1);
  l->top = l->top->next;  //передвигаем top наследующуюпозицию
    res.node->data = t;
  res.node->next           =           i->node->next;//унасоднонаправ-
ыйсписок(вметодичкепроф.Зайцевадля 2направ-ого)
  res.node->prev = i->nod;
  l->size++;
  }
```

+ операции удаления элемента и уничтожения списка

19. Линейный список. Физическое представление (динамические объекты).

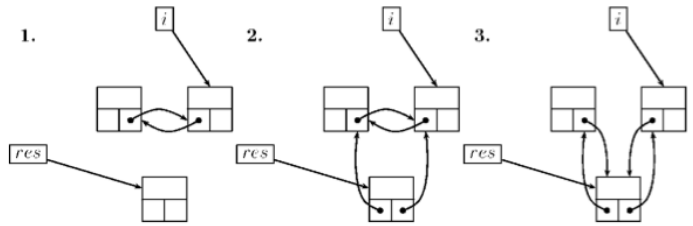
Поскольку список двунаправленный, каждый его элемент содержит ровно по две ссылки: на предыдущий и на последующий элементы списка. =>введём терминатор – элемент *next* которого указывает на первый элемент списка, а *prev* – на последний (сведя тем самым колличество указателей до 1го, т.е. Для обращения к списку идём в терминатор). Сначала обращаемся к терминатору, через него, в свою оче- редь, — к первому и последнему элементам списка, а через них — строго последовательно и к остальным.
{эти функции – дополнения к перечисленным в п.17}

```
typedef struct{
    struct Item *head;
int size;
}List;
```

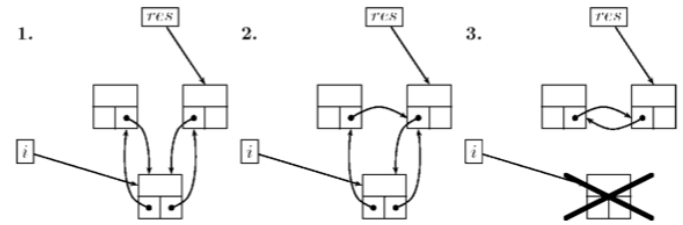
CreateQ выделяет память под терминатор, создает пустой кольцевой список путем присваивания компонентам prev и next значения указателя на терминатор и уста- навливает равной 0 длину списка.

```
viod Create(List *l){  //созданиесписка
    l->head = malloc(sizeof(struct Item));
    l->head->next =  l->head->prev = l->head;
    l->size = 0
  }
Iterator First (const List* 1)
{Iterator res = { 1—>head—>next }; return res;}

Iterator Last (const List* 1)
{ Iterator res = { 1—>head }; return res;}
```

```
bool Empty(const List* 1)
{ Iterator fst =First(l);
Iterator 1st = LastQ);
return Equal(&fst, &lst);}

int Size(const List* 1)
{ return 1—>size;}
```




```
Iterator Insert (List* 1, Iterator* i, const T t){
Iterator res = { malloc(sizeof(struct Item)) };
if (! res .node)
  return Last(l);
res .node—>data = t;
res .node—>next = i—>node;
res .node—>prev = i—>node—>prev;
res .node— >prev—>next = res.node;
i —>node—>prev = res.node;
1 —>size++;
return res}
```




```
Iterator Delete(List* 1, Iterator* i) {
Iterator res =Last(l);
if(Equal(i, &res))
  return res;
res. node = i—>node—>next;
res .node—>prev = i—>node—>prev;
i —>prev—>next = res.node;
1—>size--;
free (i —>node);
i —>node = 0;
return res;}

void Destroy(List* 1) {
struct Item* i = 1—>head—>next;
while(i != 1—>head){
  struct Item* pi = 1—>head;
  i = i—>next;
  free (pi);}
free (1 — >head);
1—>head=0;
1 —>size = 0;}
```

20. Списки общего вида. Представление и обработка графов.

Граф – это совокупность непустого множества вершин и множества пар вершин (дуг (для неор.) или ребер).

2 вершины называются смежными, если в графе есть дуга (ребро), соединяющая эти вершины.

Граф пердставляется в виде матрицы смежности.

Матрица смежности для графа A — матрица M(A) = [aij] порадка *n*

{ aij = 1, if есть ребро (vi , vj); elseaij = 0 }
(аналогично для неор. Графа)

Основное применение графа в информатике – поиск путей (алгоритм Уоршала).
Существует путь из i в j ←→ существуетaij в матрице M+ = MvM^2 vM^3 v … vM^n.

Другие представления графа в ЭВМ:

а) целочисленный массив из n столбцов и m сторок (m – макс. Число вершин, смежных с произвольной вершиной графа) – удобен, когда у многих вершин графа колличество смежных близко к m

б) граф описывается в виде *основного списка вершин* и *списков дуг*; в основном списке перечислены узлы (вершины графа), каждый из которых указывает на свой список дуг; каждый узел списка дуг указывает на узел основного списка, входящего в соответствующую дугу

Поиск в глубину
Алгоритм поиска описывается следующим образом: для каждой непройденной вершины необходимо найти все не пройденные смежные вершины и повторить поиск для них. Используется в качестве подпрограммы в алгоритмах поиска одно- и двусвязных компонент, топологической сортировки.

Пусть задан граф G = (V,E), где V — множество вершин графа, E — множество ребер графа. Предположим, что в начальный момент времени все вершины графа окрашены в белый цвет. Выполним следующие действия:

Из множества всех белых вершин выберем любую вершину, обозначим её v1.
Выполняем для неё процедуру DFS(v1).
Перекрашиваем её в чёрный цвет.
Повторяем шаги 1-3 до тех пор, пока множество белых вершин не пусто.
Процедура DFS (параметр — вершина)

Перекрашиваем вершину u в серый цвет.
Для всякой вершины w, смежной с вершиной u, выполняем следующие два шага:
Если вершина w окрашена в белый цвет, выполняем процедуру DFS(w).
Окрашиваем w в чёрный цвет.

Поиск в ширину
Поиск в ширину выполняется в следующем порядке: началу обхода s приписывается метка 0, смежным с ней вершинам – метка 1. Затем поочередно рассматривается окружение всех вершин с метками 1, и каждой из входящих в эти окружения вершин приписываем метку 2 и т.д.
{реализуется через очередь}

21. Понятие рекурсии. Рекурсия и итерация. Примеры.

Рекурсивным называют объект, частично состоящий или определяемый с помощью самого себя.
{направить камеру на монитор, на который выводится изображение с этой камеры}

Рекурсия в программирование – вызов функциеё самой себя.
Т.к. Мы используем рекурсию для сведения программы к более простой (как и цикл) рекурсия должна заканчиваться! (необходимо условие вывода из рекурсии)
вызов подпрограммоиˇ самоиˇ себя по имени. Такая рекурсия называется *прямой.* Если программа *P*вызывает себя посредством другоиˇ *Q,* то рекурсия *косвенная.*
Существует более сложная классификация рекурсииˇ: линейная, повторная (концевая, хвостовая), каскадная, удаленная, взаимная

определение факториала от n: n! = n * (n-1)! через рекурсию

```
int f (int i){
return (i > 0)  ?  i * f(i – 1) : 1;
}
```

Итерация – многократно повторяющиеся действия, не приводящие к вызовам самих себя (шаг цикла).
{итерация намноо дешевле рекурсии}
определение факториала от n: n! = n * (n-1)! через итерации

```
int i = 0;
int f = 1;
while(i < n)
f *= i++;
```

22. Деревья. Двоичные деревья.(ст. 267)

Пусть T – некоторый тип данных. Деревом типа T называется структура, образованная элементом типа T (корнем) и конечным (возможно пустым) множеством с переменным числом элементов – деревьев ипа T (поддеревьев) {не пустое}
{дерево с нулевым разветвлением – линейная структура}
{для описания взаимного расположения элементов взята терминология из генеалогичекого дерева (отец и сын; потомок и предок; НО дочерняя вершина); сыновья одного дерева – братья}

Приведём определения, относящиеся к дереву:

а) элементы типа T, входящие в дерево называют узлами (вершинами)

б) число поддеревьев данного узла называют его степенью

в) узел, не имеющий поддеревьев, называют листом (концевым узлом)

г) уровень в дереве определяется рекурсивно: уровень корня каждого поддерева данного узла на “1” больше уровня донного узла; уровень корня всего дерева = 1.

д) уровень дерева – макс. Уровень его вершин.

Если порядок сыновей существенен, то сыновей упорядочивают “по старшенству” (подобно очереди), т.е. чем больше номер, тем дальше от предка.

Определённые таким образом деревья называют деревьями общего вида.

Двоичное (бинарное) дерево – конечное множество узлов, которое или пусто, или состоит из корня и 2х непересекающихсяф поддеревьев (левое и правое) {может быть пустым} (схема спортивного турнира)

Предком для узла ***x*** называется узел дерева, из которого существует путь в узел ***x***.
Потомком узла ***x*** называется узел дерева, в который существует путь (по стрелкам) из узла ***x***.
Родителем для узла ***x*** называется узел дерева, из которого существует непосредственная дуга в узел ***x***.
Сыном узла ***x*** называется узел дерева, в который существует непосредственная дуга из узла ***x***. **Уровнем** узла ***x*** называется длина пути (количество дуг) от корня к данному узлу. Считается,
что корень находится на уровне 0.
Листом дерева называется узел, не имеющий потомков.
Внутренней вершиной называется узел, имеющий потомков.
Высотой дерева называется максимальный уровень листа дерева.
Упорядоченным деревом называется дерево, все вершины которого упорядочены (то есть имеет значение последовательность перечисления потомков каждого узла).

23. Двоичное дерево. Функциональная спецификация.(ст. 273)

Обозначим BTt – двоичное дерево компонент типа T, характерезуемый следующими операциями:

а) создать (всегда создаётся пустое дерево)
б) проверка на пустоту
в) построить дерево (из корня и 2х поддеревьев)
г) получение корня
д) получение левого/правого поддеревьев
е) уничтожение дерева

Операции:
а) чтение данных из узла
б) создание дерева, состоящего из одного корня
в) построение дерева из заданных корня и поддеревьев
г) присоединение нового поддерева к узлу
д) замена поддерева на новое
е) удаление поддерева
ё) получение узла, следующего за данным в определённом порядке

24. Двоичное дерево. Логическое описание. Построение и визуализация.(ст. 268 + 274)

Каждый элемент дерева имеет (в структуре) собственное значение и 2 указателя: на левого и правого сыновей.

{задача – постричь минимальное двоичное дерево из n элементов; решение – *взять вершину в качестве корня* и построить тем же способом левое поддерево из n**div**2 элементов и правое (из оставшихся)}

Способы визуалезации: граф (обычно); вложенные диаграммы эйлера; иерархическая И многоуровневая ступенчатая запись.

иерархическая: (A (B (D (I), E (J , K , L)) , C ((O) , G (M , N) , H (P))))

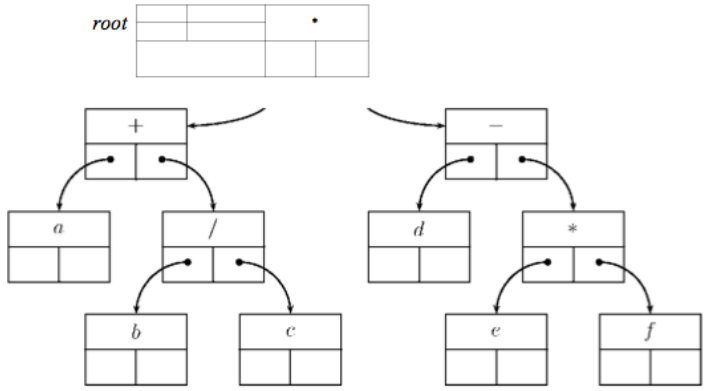
многоуровневая ступенчатая запись:

```
A       //корень
B       //левое поддерево
D
        I
E
        J
        K
        L
C       //правое поддерево
F
        O
G
        M
        N
H
        P
```

Предком для узла ***x*** называется узел дерева, из которого существует путь в узел ***x***.
Потомком узла ***x*** называется узел дерева, в который существует путь (по стрелкам) из узла ***x***.
Родителем для узла ***x*** называется узел дерева, из которого существует непосредственная дуга в узел ***x***.

Сыном узла ***x*** называется узел дерева, в который существует непосредственная дуга из узла ***x***. **Уровнем** узла ***x*** называется длина пути (количество дуг) от корня к данному узлу. Считается,
что корень находится на уровне 0.
Листом дерева называется узел, не имеющий потомков.
Внутренней вершиной называется узел, имеющий потомков.
Высотой дерева называется максимальный уровень листа дерева.
Упорядоченным деревом называется дерево, все вершины которого упорядочены (то есть имеет значение последовательность перечисления потомков каждого узла).

Тип данных дерево обычно отсутствует в универсальных языках программирования, и нам придется его моделировать программно. Так же, как и в списках, цепная структура дерева будет состоять из динамически порождаемых элементов, в которых предусмотрены ссылки на очередные компоненты структуры. В двунаправленных списках были указатели вперед и назад, а здесь будут
ссылки влево вниз и вправо вниз.
struct node {
char key; struct node***** 1**;** struct node***** r; }



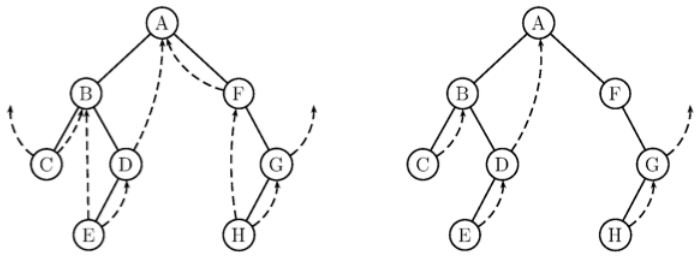
25. Двоичное дерево. Физическое представление. Прошивка.(ст. 299)

На массиве: цельный кусок памяти, на первом месте корень, на 2м и 3м – его левый и правый сыновья, а далее – по паром сыновья каждого родителя, т.е. Сыновья i-го родителя лежат на местах 2i и 2i+1 (j-ый элемент i-го уровня находится на месте 2^(j-1) + j – 1.
Основным неудобством сплошного представления дерева является высокая цена вставки и удаления элементов + перерасход памяти.
Для деревьев можно использовать рекурсивные ссылочные представления, которые были разработаны для списков, но с той лишь разницей, что указатели вперёд и назад по линейной структуре теперь направляются к левому и к правому поддеревьям соответственно.

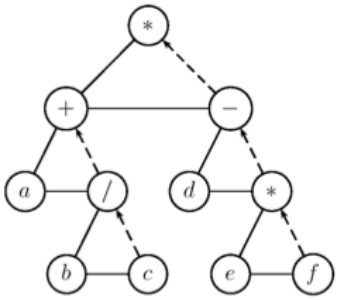
Динамическая структура: каждый элемент указывает на место своих сыновей
{сложнее читать/искать, но проще добавлять/удалять}

Прошитые бинарные деревья – В прошитых бинарных деревьях вместо пустых указателей используются специальные связи-нити и каждый указатель в узле дерева дополняется однобитовым признаком *ltag* и *rtag*, соответственно. Признак определяет, содержится ли в соответствующем указателе обычная ссылка на поддерево или в нем содержится связь-нить.
Связь-нить – в поле *l* указывает на узел-предшественник при обратном обходе, а в поле *r* – на узел-приемник.
Для прошитых деревьев можно выполнить *нерекурсивный* обход *без использования стека.*
Прошитое дерево линеаризовано и может быть помещено в очередь или список => можно применить итераторы для обработки, т.к функциональность совпадает.

Среди прошитых деревьев важный класс составляют правопрошитые деревья, т. е. прошитые бинарные деревья, у которых используется только правая связь-нить, а в поле / содержится либо обычный указатель, либо пустой указатель. Поле ltag в таком случае не используется.



Такие деревья хорошо применять для арифметических операций



26. Алгоритмы обхода деревьев.

Обход дерево – просматривание дерева по определённому алгоритму так, что все вершины будут просмотрены.

Для бинарного дерева:

1) прямой обход (сверху вниз; КЛП; корень прежде; preorder)
   (а) если дерево пусто – конец обхода
   (б) берём корень
   (в) обходим левое поддерево
   (г) обходим правое поддерево

2) обратный обход (слева направо; ЛКП; корень между; inorder)
   (а) если дерево пусто – конец обхода
   (б) обходим левое поддерево
   (в) берём корень
   (г) обходим правое поддерево

3) концевой обход (снизу вверх; ЛПК; корень в конце; posorder)
   (а) если дерево пусто – конец обхода
   (б) обходим левое поддерево
   (в) обходим правое поддерево
   (г) берём корень

Для обычного дерева:

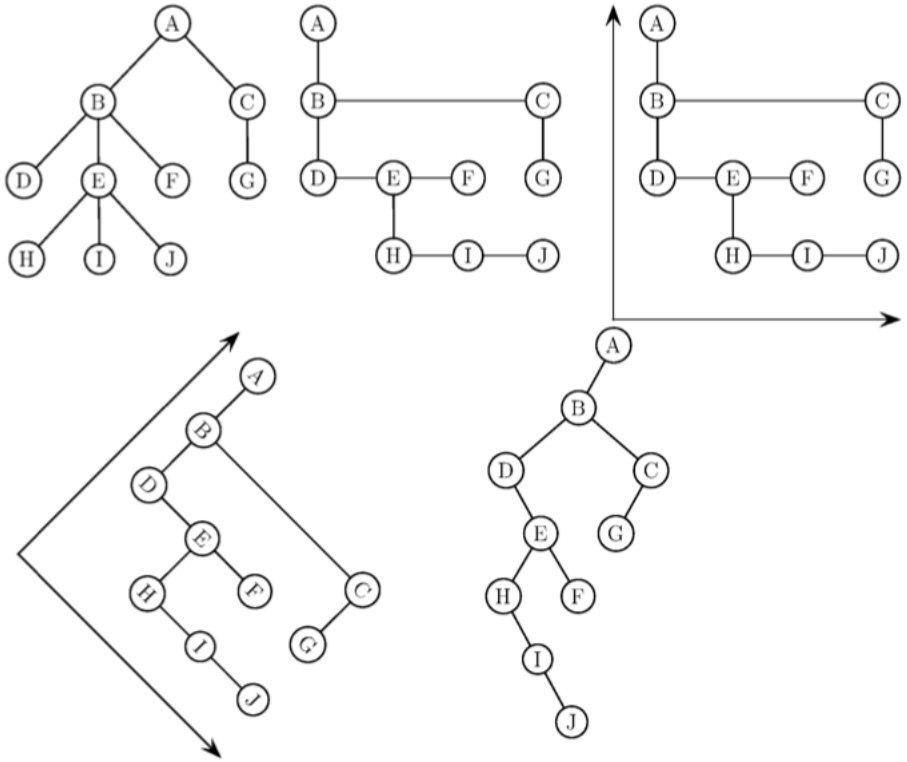
1) поиск в глубину:
   (а) если дерево пусто – конец обхода
   (б) берём корень
   (в) ищем в глубину для поддерева старшего сына
   (г) ищем в глубину для поддерева следующего брата

2) поиск в ширину

(а) поместить в пустую очередь корень дерева
(б) если очередь узлов пуста – конец обхода
(в) извлечь первый элемент из очереди и поместить в конец всех его сыновей
(г) вернуться к п. б

27. Особенности представления и обработки деревьев общего вида (преобразование к двоичному, ...).

Для перевода обычного дерева в бинарное надо соединить братьев предка, у которого их больше 2х и убрать все лишние связи (полученные на предыдущем шаге)



28. Деревья выражений.

Деревья выражений – деревья, в узлах которых стоят элементы выражения (+, _, *, /, a = const, x != const)

при прямом обходе получаем префиксную (польскую) форму: * + a / bc – d * ef (КЛП – Левый-Корень-Правый)
при обратном обходе получаем инфиксную (обычную) форму без скобок: a + b / c * d – e * f(ЛКП)
при концовом обходе получаем постфиксную (обратную польскую) форму: abc / + def * - *(ЛПК)

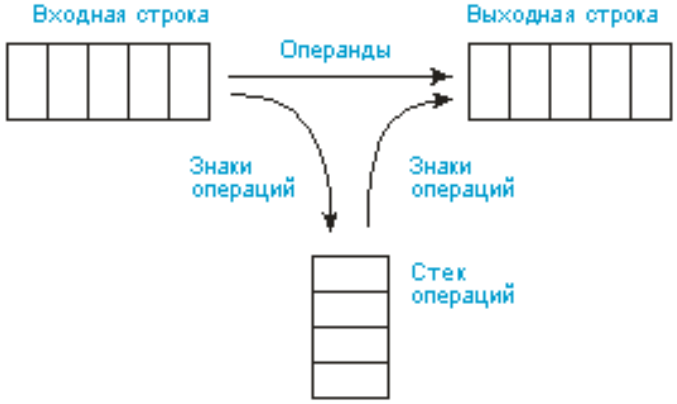
Изобретение польской формы: в 1920г Ян Лукашевич (математик, логик) печатал свою дисертацию на старой немецкой пишушей машинке с дополнительным регистром для скобок и спец знаков. К тому же эта вондервафля заедала при переключение регистров → Лукашевич придумал запись без скобок, заменив привычную [*операнд* ***оператор*** *операнд*] на [***оператор****операнд операнд*]

Выражение (1+2)*4+3 в ОПН может быть записано так: 1 2 + 4 × 3 +
Вычисление производится следующим образом (указано состояние стека после выполнения операции):

Ввод	Операция	Стек
1	поместить в стек	1

2 поместить в стек 1, 2
+ сложение 3
4 поместить в стек 3, 4
* умножение 12
3 поместить в стек 12, 3
+ сложение 15
Результат, 15, в конце вычислений находится на вершине стека.

{при преобразование из инфиксной формы в постфиксную помимо выноса знака оператора назад, необходимо упорядочивать операторы по приоритету (не забывай о том, что скобки повышают приоритет), т.е. нужен стек}
К примеру для вычисления деревья выражений можно применить алгоритм Дейкстры основанный на применении стеков с приоритетами и польской записью.



29. Деревья поиска.
Дерево поиска – это такое дерево, для каждого узла *t* которого выполняются следующие условия:

1) ключи всех узлов в левом поддереве меньше значения *t*
2) ключи всех узлов в правом поддереве больше значения *t*

{эффективно для поиска ключей, т.к. при переходе из узла дерева к одному из поддеревьев мы автоматически исключаем просмотр другого поддерева → O(lnn).что меньше чем O(n)}

{при удаление из дерева узла с 2я сыновьями – 3 варианта: поставить на его место самого *правого* потомка его левого сына (самого *левого* потомка правого) или поставить его *правого* сына}

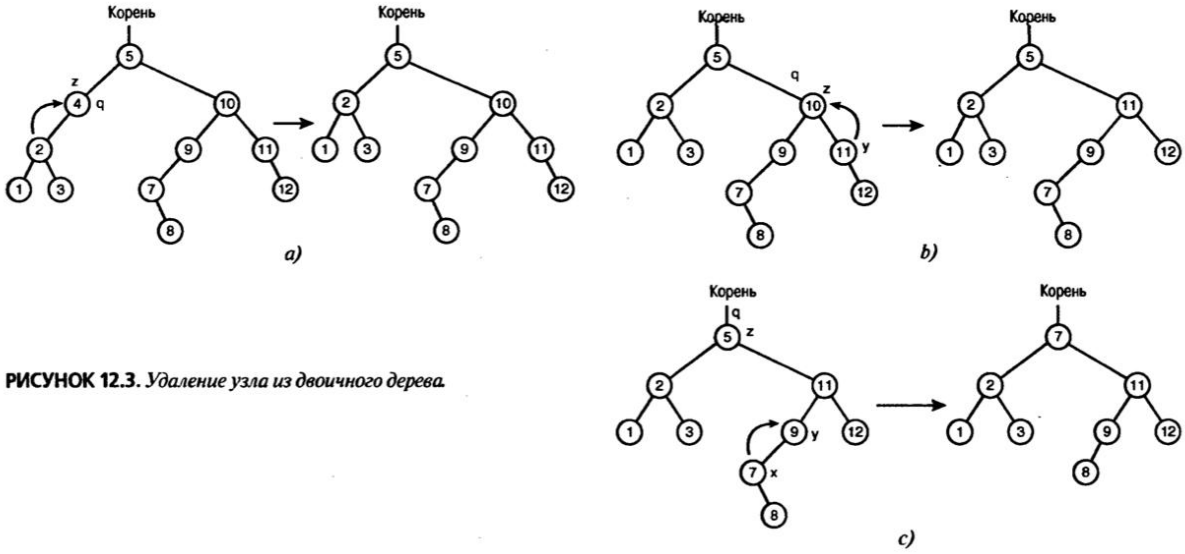


РИСУНОК 12.3. *Удаление узла из двоичного дерева.*

Для большей эфективности деревья поиска приводят к сбалансированным

В эту процедуру поиска может быть внесено усовершенствование. Для упрощения условия поиска можно, также как и в списках, ввести терминирующий элемент s к которому гамаком [54] привязать пустые (неиспользуемые, кроме как для прошивки) ссылки всех концевых вершин дерева поиска. В результате из условия поиска удаляются два из трёх сравнений. В предусловии цикла остаётся только сравнение с барьерным элементом. Правда, в случае неудачного поиска функция примет значение не **nil,** а будет указывать на якорь нашей древовидной структуры, что придётся учесть при использовании

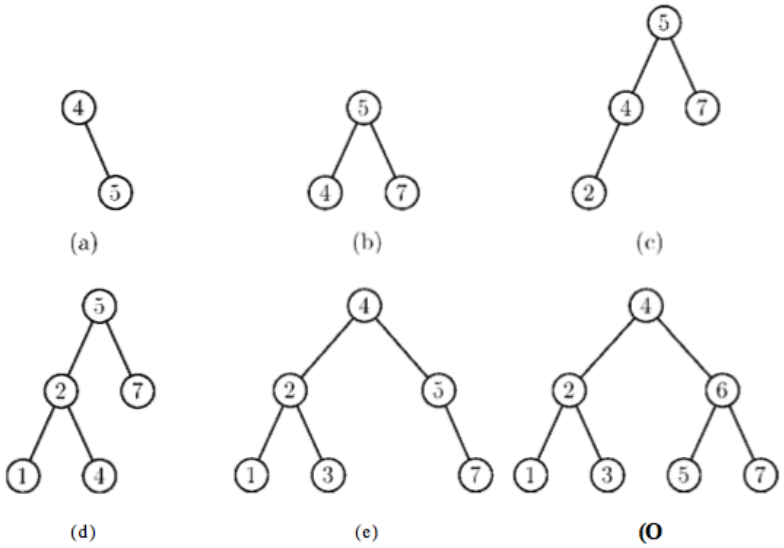
Дерево поиска может быть использовано для упорядочивания данных. Для этого надо разместить эти данные в дереве поиска, а потом составить в результате его обхода упорядоченную последовательность.

30. Сбалансированные деревья поиска.(ст. 293)

Т.к. введение новой вершины в идеально сбалансированное дерево его разбалансирует, то решили делать балансировку после вставки.
AVL-дерево – то дерево, высота поддеревьев каждой из вершин которого отличается не более, чем на единицу.
(идеально сбалансированное дерево является AVL-деревом)

В AVL-дереве за сложность O(lnn) можно найти искомую вершину, добавить новую или удалить старую.

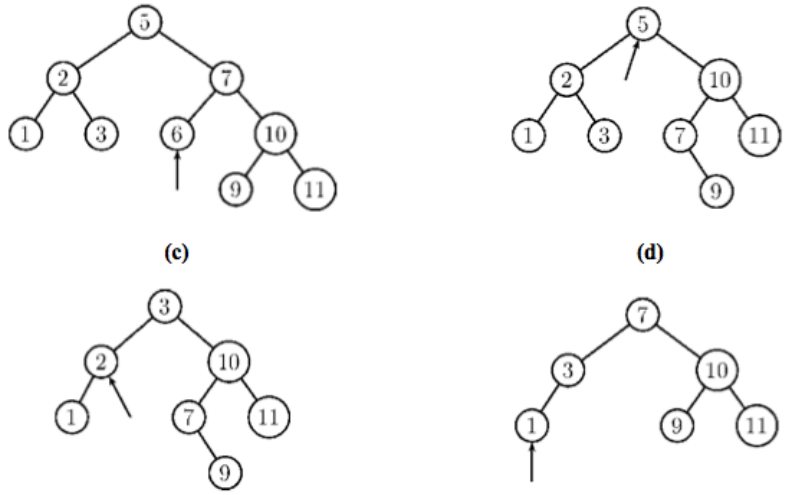
Схема алгоритма включения в сбалансированное дерево такова:
1. поиск элемента в дереве (неудачный!);
2. включение новой вершины и определение результирующего показателя сбалансированности;
3. отход по пути поиска с проверкой показателя сбалансированности для каждой проходимой вершины, балансируя в необходимых случаях соответствующие поддеревья.



Ожидаемая высота AVL-дерева: h = lnn + c, где c примерно равно 0,25.
{постоянно проверять, сбалансированно ли дерево, ДОРОГО → храним (в структуре) [2х битный] элемент, хранящий разность высот правого и левого поддеревьев каждого узла}

К тому же, затраты на поддержание AVL-дерева значительно меньше затрат на поддержание идеально сбалансированного дерева

При удаление двухдетного узла – на его место встанет самый правый элемент его левого поддерева.
Исключение любого элемента из сбалансированного дерева оценивается O(logN).



31. Простые методы поиска.(ст. 308)

Линейный поиск – последовательный просмотр массива как списка (цикл). Поиск происходит за O(N). Индексация начинается с 0. Ускорить поиск можно с помощью барьерного элемента. Его следует поместить в N+1-ый элемент массива. Этот метод поиска единственный для структур данных со сторого последовательным доступом.

```
int function LinearSearch (Array A, int L, int R, int Key);
begin
  for X = L to R do
    if A[X] = Key then
      return X
return -1; // элемент не найден
```

end;
Двоичный поиск – метод дихтомии (делим упорядоченный массив по середине: попали – ура, не попали – смотрим, в которой половине находится искомый элемент и ищем там)
При выборе среднего значения т максимальное число сравнений равно [log2 *N]* + 1. То есть, поиск в упорядоченном множестве с прямым доступом к эле- ментам также эффективен, как и в дереве поиска, и намного быстрее линейного поиска с числом сравнений *N/2.* Однако вставка и удаление элемента в упорядоченную таблицу дороже, чем в дереве поиска.

```
while((low <= high)&&(!found)) {
        mid=(low+high)/2;
        if(keys[mid] == key) found = 1;
else if(key > keys[mid]) low = mid + 1;
        else high = mid - 1;
 }
if (found==1) return mid;
if(found==0) {
        printf("Record not found!\n");
        return -1;
 }
```

32. Поиск по образцу в последовательностях и таблицах.(ст. 312)

Поиск в последовательности – поиск в последовательности *s* из H элементов последовательности *p* из M, меньше или равно H, элементов (выдаёт первое вхождение, а точнее индекс начала первого вхожления).

Прямолинейный метод поиска: а) ищем вхождение первого элемента
б) поэлетерно сравниваем, если не совпадает → переходим к элементу из п.1, берём следующий за ним и возвращаемся в п.1
{максимальная сложность - O(M*H)}
В таблицах поиск отличается от поиска в массиве тем, что элемент сам по себе составной (“ключ”, по которому мы ищем, и значение, соответствующее данному ключу)
{сам ключ может быть словом или строкой проверка на совпадение строк – политерное сравнение до первого расхождения}
Реализация строк переменной длины: а) размер строки хранится в первом (нулевом) элементе строки как сверхкороткое число (проще узнать длину, но ограничение возможной длинны строки)
б) размер НЕ задаётся (указывается), а сама строка простирается от начала до терминатора (ограничивающего элемента) (предпочтительнее)

33. Алгоритм Кнута-Морриса-Пратта.(ст. 317)

В 1977 году Д.Кнут, В.Морис и В.Пратт опубликовали алгоритм, требующий только H сравнений даже в самом плохом случае, вместо H*M (Морис разработал алгоритм отдельно от Кнута и Пратта, но печатались они вместе)

Идея: начиная каждый раз сравнение объекта со строкой мы не забываем ценную информацию (как в прямолинейный метод поиска из п. 32) о тех элементах, которые уже просмотрели и сравнили.
Технически это просмотр объекта на самоподобие (фрактальность) и составление таблицы D, в которой для каждого случая не полного прохода по объекту (нашли первое не полное вхождение, в котором на ***j*-том** месте расхождения) было число, на которое нужно было вернуться назат (с ***j*-того** места) для нового сравнения (возможно и “-1”, т.е. идти вперёт)

Вычисление D само является поиском в строке, для чего можно использовать КМП алгоритм.

Алгоритм: а) вычисление таблицы D

б) ищем вхождение первого элемента

в) поэлетерно сравниваем, если не совпадает на *j*-той позиции → подставляем*j* в таблицу D и продолжаем проверять, сдвинув начало первого вхождения на $j - d[j]$

г) если в строке ещё может быть слово (*j*!= H – M) возвращаемся к п.б, ELSE – вхождений нет.

Достоинством КМП является однопроходность, ведь по строке мы не возвращаемся назад, что выгодно для электромеханических устройств запоминания (не нужен реверс) и для удалённого поиска по сети.
(В начале 3го тыс-летия А.Л.Калинин предложил обобщённый КМП алгоритм на случай одновременного поиска нескольких образцов. Алгоритм Кнута-Морриса-Пратта-Калинина позволил повысить эффективность антиспамового фильтра антивируса “Касперский”)
Минус – подлинный выигрышь только если перед неудачей было некоторое число совпадений (скорее исключение, чем правило).

34. Алгоритм Бойера-Мура.(ст. 323)

Р.Боер и Д.Мур предложили алгоритм, который не только улучшал (относительно КМП) обработку самого худшего случая (когда в образце и строке многократно повторяется не большое число букв), но и даёт выигрыш в промежуточных ситуациях.
БМ алгоритм поиска строки считается наиболее быстрым среди алгоритмов общего назначения, предназначенных для поиска подстроки в строке.

Идея: сканирование *слева направо*, сравнение *справа налево* (характерно для семитских языков). Совмещается начало строки и образца, проверка начинается с последнего символа образца.
Если символы совпадают, производится сравнение предпоследнего символа шаблона и т.д.
Если все символы образца совпали с наложенными символами строки, значит, подстрока найдена, и поиск окончен.
Если же какой-то символ образца не совпадает с соответствующим символом строки, шаблон сдвигается на *“несколько”*символов вправо, и проверка снова начинается с последнего символа (эти *“несколько”* обычно равно *разнице* между [колличеством литеров в образце] и [первой встречи (справа) данного символа в образце]).

{опять прибегаем к таблице D}

Достоинствами БМ являестя то, почти всегда, кроме специально построенных случаев, он требует значительно меньше H сравнений, проходя “саженью образа” по строке. Лучшая оценка, когда последний символ образа всегда не совпадает с соответствующими литерами строки, проходимой аршином M, равно H/M.
В своей публикацие (которая была после Кнута, Морриса и Пратта) Боер и Мур приводили соображения о дальнейшем улучшение алгоритма. Одно из которых – объединение стратегий БМ (больший сдвиг во время несовпадений) и КМП (существенный сдвиг при частичном совпадение), которая основывалась на 2х таблицах, из которых выбиралось большее смещение.

35. Алгоритм Рабина-Карпа.(ст. 327)

Разработаный в 1987 году Майклом рабином и Ричардом Карпом алгоритм поиска подстроки эффективен на практике и обобщаем для многомерного случая (например, поиск в 2мерном массиве).
Хотя в худшем случае время работы составляет O (M * (N – M + 1)), в серднем он работает достаточно быстро.

Идея: без ограничения общности предположим, что алфавит строки и образца – цифровой (а строки имеют простую цифровую интерпретацию).
Если p[0 … M – 1] – образец, то число P – десятичная запись образца.
В последовательности s[0 … n – 1] через числа Sj (для всех j от 0 до N – M – 1) обозначим десятичное представление всех последовательностей s[j … j + M – 1] (длина такой подстроки равно длине образца)
P вычисляется по схеме Горнера: P = p[M – 1] + 10(p[M – 2] + 10(… + 10(p1 + 10 * p0) …))
Если известна Sj, то вычисление S[j+1] есть вычёркивание первой цыфры в Sj и приписывание в конец s[j+M]: S[j+1] = 10(Sj – 10^(m-1) * sj) + s[j+M]

Если Sj = P, то Sj – вхождение образца в строку.
{числа бывают огроменные (не влезают ни в один тип) → вычисляем не сами P и S, а их остаток от деления на простое число q, такое, чтобы для алфавита {0, 1, …, d} число dq помещалось в машинное слово, т.е.
S[j+1] = (d(Sj – h * sj) + s[j+M]) mod q, h = d^M-1}

36. Таблицы с прямым доступом.(ст.362)

В 2х словах: ускорение работы с таблицей за счёт хеширования (перевода ключей элементов из литерной (символьной) формы в число, которое является указателем в памяти на этот элемент).
{ведь обращение к элементу (чтение, запись, удаление) в конечном итоге есть обращение к памяти, так почему бы не начать сразу обращаться к памяти, пропуская сравнения ключей и всякое остальное}
{для такого финта ушами нужно поместить таблицу с массив (доступ за постоянное время)}

Для этого перевода нужна хеш-функция (отображение Н*ключей* элементов в *адреса* памяти).

Простейший пример хеш-функции: функция получения числового кода литера (если литеров несколько, то можно воспользоваться схемой Горнера: A = б + (i – 1) * sizeof(T), где б – адрес начала массива, i – индекс компоненты вектора, отсчитываемый от “1”, sizeof(T) – размер компоненты вектора)

Проблема в том, что разные ключи могут соответствовать одному адресу. Такие случаи называются *коллизиями*.
На такой случай нужно предусмотреть операцию рехеширования – переадресация, в случае занятости адреса (запись нового элемента) или в случая, когда по данному адресу находится элемент с другим ключом (чтение).
{2 способа:
организовать список строк с идентичным первичным ключом H(k) (этот список может расположить где душе угодно, но такой способ вызовет увеличение расхода памяти);
или, при занятости данной ячейки памяти, “впихнуть” наш элемент куда-нибудь рядом}

Достоинства таблицы с прямым доступом (хеш-таблицы) в том, что при линейных сложностях операций хеширования и рехеширования можно обрабатывать таблицу (читать, искать, добавлять новый элемент и удалять старые) за постоянное время.

Недостаток – нельзя напечатать так быстро заполненную таблицу в упорядоченном виде, да ещё и сохраняя естественный хронологический порядок равнозначных элементов (нужна сортировка → что было до хеша).

37. Алгоритмы сортировки.(ст.329 + 358)

Трудно найти какую-либо другую задачу, для которой было бы изобретено столько алгоритмов, как для сортировки.

Сортировка называется устойчивым (стабильным), если в процессе сортировки относительное расположение элементов с равными ключами не изменилось.

Характеристики методов сортировки
Время — основной параметр, характеризующий быстродействие алгоритма. Называется также вычислительной сложностью. Для упорядочения важны худшее, среднее и лучшее поведение алгоритма в терминах размера списка (n). Для типичного алгоритма хорошее поведение — это O(nlogn) и плохое поведение — это $O(n^2)$. Идеальное поведение для упорядочения — O(n).
Память — ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных.
Устойчивость (stability) — устойчивая сортировка не меняет взаимного расположения равных элементов.
Естественность поведения — эффективность метода при обработке уже упорядоченных, или частично упорядоченных данных. Алгоритм ведёт себя естественно, если учитывает эту характеристику входной последовательности и работает лучше.
Использование операции сравнения. Алгоритмы, использующие для сортировки сравнение элементов между собой, называются основанными на сравнениях.
Внутренняя сортировка оперирует с массивами, целиком помещающимися в оперативной памяти с произвольным доступом к любой ячейке. Данные обычно упорядочиваются на том же месте, без дополнительных затрат.
В современных архитектурах персональных компьютеров широко применяется подкачка и кэширование памяти. Алгоритм сортировки должен хорошо сочетаться с применяемыми алгоритмами кэширования и подкачки.
Внешняя сортировка оперирует с запоминающими устройствами большого объёма, но с доступом не произвольным, а последовательным (упорядочение файлов),

Категория сортировок входящие в раздел “внутренние” имеют прямые методы сортировок.

Сортировка вставкой (устойчивая). Вставка элемента в на нужную позицию в уже отсортированном массиве

Сортировка выборокой (не устойчивая). 1-ый элемент меняется местами с минимальным, далее 2-ой так же..

Пузырьковая сортировка. 1-ый элемент сравнивается с соседним, если >, то меняются местами, если <, то берется тот элемент с кем происходило сравнение и сравнивают его, так пока не дойдем до конца. В 1-ой итерации всегда всплывет максимальный элемент.

Сортировка Шелла. Выбирается интервал d, через которые будут сравниваться элементы в первый проход, к примеру 5, после полного прохода, берется новый интервал 3 и опять проход и так далее.

Турнирная сортировка.

38. Сортировка вставкой (прямым включением).(ст. 331)

{устойчавая}
На примере карт: у нас есть колода на столе и отсортированный веер карт в руке; берём из колоды карту и вставляем в веере в нужное место; и так до того, пока карты не закончатся.

{т.е. у нас есть 2 части массива: отсортированная и не отсортированная}
{массив из одного или нуля элементов отсортирован!}

сложность O(n^2)
Вход: массив A, состоящий из элементов A[1], A[2], ..., A[n]

```
for i = 2, 3, ..., n:
key := A[i]
j := i - 1
while j > 0 and A[j] > key:
A[j + 1] := A[j]
j := j - 1
A[j + 1] := key
```

39. Сортировка выборкой.

Так же, как и в сортировке вставкой, у нас 2 части массива: отсортированная и не отсортированная.
Алгоритм:

1) отыскиваем элемент с наименьшим ключём в неотсортированной части массива
2) меняем его местами с первым в неотсортированной части массива (отсортировали ещё один элемент)
3) данный элемент теперь входит в отсортированную часть массива; переходим к п.1

Обычно сортировка выборкой быстрее, чем сортировка вставкой, но если входной массив [почти] упорядочен, то вставкой работает быстрее.

```
void selectSort(T a[], long size) {
long i, j, k;
  T x;

for( i=0; i < size; i++) {       // i - номер текущего шага
   k=i; x=a[i];
for( j=i+1; j < size; j++)       // цикл выбора наименьшего элемента
if (  a[j] < x ) {
       k=j; x=a[j];              // k - индекс наименьшего элемента
}
a[k] = a[i]; a[i] = x;           // меняем местами наименьший с a[i]
  }
}
```

40. Обменные сортировки (пузырьковая).(ст. 338)

{устойчавая}
Алгоритм:

1) берём первый элемент
2) сравниваем со следующим
3) если следующий больше – наш оставляем на месте и берём следующий; ELSE меняем местами
4) п.2, пока не дошли до конца (эффективнее держать номер начала отсортированной части)
5) п.1пока есть эементы

Улучшение – сортировка Шеёкер = 2пузырька, направленные в разные стороны (один тащит наименьтшие к началу, а другой – наибольшие к концу).

Пример – труба, в которой находятся разные по плотности не смешиваемые жидкости: более лёгкие всплывут на поверхность, а более тяжёлые уйдут на дно.

Прост в программирование, но сложность O(n^2) + один не удачно расположенный элемент будет “просачиваться” на своё место со скоростью 1 позиция за проход.

```
emplate<class T>
void bubbleSort(T a[], long size) {
long i, j;
  T x;

for( i=0; i < size; i++) {            // i - номер прохода
for(j = size-1; j>i; j-- ) {     // внутренний цикл прохода
if ( a[j-1] > a[j] ) {
      x=a[j-1]; a[j-1]=a[j]; a[j]=x;
    }
  }
}
}
```

41. Сортировка Шелла.(ст. 342)

первое улучшение алгоритма заключается в запоминании, производился ли на данном проходе какой-либо обмен. Если нет - алгоритм заканчивает работу.

Процесс улучшения можно продолжить, если запоминать не только сам факт обмена, но и индекс последнего обмена k. Действительно: все пары соседних элементов с индексами, меньшими k, уже расположены в нужном порядке. Дальнейшие проходы можно заканчивать на индексе k, вместо того чтобы двигаться до установленной заранее верхней границы i.

Качественно другое улучшение алгоритма можно получить из следующего наблюдения. Хотя легкий пузырек снизу поднимется наверх за один проход, тяжелые пузырьки опускаются со минимальной скоростью: один шаг за итерацию. Так что массив 2 3 4 5 6 1 будет отсортирован за 1 проход, а сортировка последовательности 6 1 2 3 4 5 потребует 5 проходов.

Чтобы избежать подобного эффекта, можно менять направление следующих один за другим проходов. Получившийся алгоритм иногда называют "шейкер-сортировкой". Выигрыш Шелла в том, что на каждом шаге либо сортируется мало элементов, либо они досортировываются.

Недостатки – не устойчивость и сложны математический анализ (до сих пор не найдена универсальная последовательность шагов, дающая минимальную сложность). Сложность O (n ^ (1+б)), где б от “0” до “1”, что хуже, чем n * lnn.

```
template<class T>
void shakerSort(T a[], long size) {
longj, k = size-1;
longlb=1, ub = size-1; // границы неотсортированной части массива
Tx;
```

```
do {
        // проход снизу вверх
for( j=ub; j>0; j-- ) {
if ( a[j-1] > a[j] ) {
       x=a[j-1]; a[j-1]=a[j]; a[j]=x;
       k=j;
     }
   }
lb = k+1;
   // проход сверху вниз
for (j=1; j<=ub; j++) {
if ( a[j-1] > a[j] ) {
       x=a[j-1]; a[j-1]=a[j]; a[j]=x;
       k=j;
     }
   }
ub = k-1;
  } while ( lb< ub );
}
```

42. Турнирные сортировки.(ст. 344)

В сортируемом множестве можно сравнить пары соседних элементов → из n/2 сравнений получаем меньшую по значению ключей половину (победители); также из получаем четверть и т.д., пока не получим один элемент.
Результат – дерево выбора (турнирная таблица – двоичное дерево, в котором для каждого двудутного узла один из сыновей – он сам (победитель), а другой больше своего отца) (в корне – самый маленький элемент)

Перевода в множество: а) возьмём корень и запишем его в множество

б) спускаемся от победителя к призёрам по пути победителя, опустошая вершины

в) поднимаясь по этому пути, выполняем перераспределение мест, как будто победителя из п.а не было (элемент, соревнующийся с пустым местом, автоматом переходит в следующий тур)

г) теперь место победителя занял новый элемент; переходим к п.1

Достоинство – сложность O(n * lnn) ВСЕГДА!!!.
Недостаток – двойной расход памяти (2n – 1)
{для уменьшения расхода памяти необходимо применить пирамидольное улучшение традиционных древовидных сортировок (hi =<h2ihi =<h[2i+1]); в пирамиде нет повторов элементов сортируемого множества}

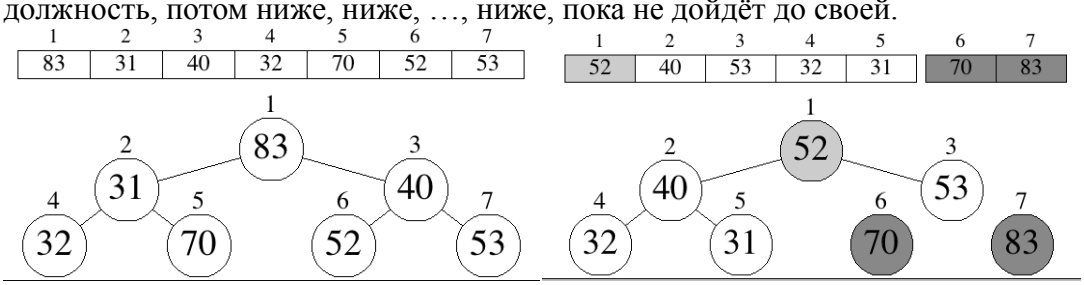
Строительство пирамиды: 1) первоначально элемент помещается в вершину, i = 0

2) рассмотрим тройку элементов с индексами i, 2i, 2i+1 (узел и его дети)

3) выберем наименьший из п.2: если он на позиции i, элемент вставлен; ELSE обозначим позицию наименьшего буквой j, где j=j2i или j = 2i + 1, меняем местами элементы i и j

4) переходим к п.2

Пример – шутка про то, что в каждой иерархии любой член занимает своё место согласно своей [не]компетентности. Новичок, попав туда, в начале примеряет к себя высшую должность, потом ниже, ниже, …, ниже, пока не дойдёт до своей.



43. Гладкая сортировка.

В 1981 году Э.Дейкстар пердложил алгоритм гладкой сортировки, имеющий время работы порядка $n\ln n$ в наихудшем случае и порядка n в случае почти отсортированной последовательности (название как бы укажывает на плевность изменения времяни работы в зависимости от случая)

Подобно пирамидальной, плавная обрабатывает массив в 2 прохода. После первого получаем несколько частично упорядоченных деревьев (пирамид). При втором проходе элементы занимают свои места.

44. Сортировка Хоара.(ст.351)

Быстрые сортировки – те, в которых осуществляются перестановки на большие расстояния.

Алгоритм:

В методе Хоара первоначально выделяют базовый элемент, относительно которого ключи с большим весом перебрасываются вправо, а с меньшим влево. Базовый элемент сравнивается с противоположным элементом.

В качестве базового элемента очень удобно брать крайние элементы.

Давайте рассмотрим пример:
Дано множество
{9,6,3,4,10,8,2,7}

Берем 9 в качестве базового элемента. Сравниваем 9 с противоположностоящим элементом, в данном случае это 7. 7 меньше, чем 9, следовательно элементы меняются местами.
{7,6,3,4,10,8,2,9}

Далее начинаем последовательно сравнивать элементы с 9, и менять их местами в зависимости от сравнения.
{7,6,3,4,10,8,2,9}
{7,6,3,4,10,8,2,9}
{7,6,3,4,10,8,2,9}
{7,6,3,4,9,8,2,10} - 9 и 10 меняем местами.
{7,6,3,4,8,9,2,10} - 9 и 8 меняем местами.
{7,6,3,4,8,2,9,10} - 2 и 9 меняем местами.

После такого перебрасывания элементов весь массив разбивается на два подмножетсва, разделенных элементом 9.
{7,6,3,4,8,2}
{10}

Далее по уже отработанному алгоритму сортируются эти подмножества. Подмножество из одного элемента естественно можно не сортировать. Выбираем в первом подмножестве базовый элемент 7.
{7,6,3,4,8,2}

{2,6,3,4,8,7} - меняем местами 2 и 7.
{2,6,3,4,8,7}
{2,6,3,4,8,7}
{2,6,3,4,8,7}
{2,6,3,4,7,8}- меняем местами 7 и 8

Получили снова два подмножества.
{2,6,3,4}
{8}

Достоинства – легко распараллеливается, что полезно в нашь век многопроцессорных/ядерных/потоковых ЭВМ
(лучший случай - когда каждыё раз *медиана* выбирается *зерном*)

Есть ещё и итеративное решение, в котором мы запоминаем границы подпоследовательностей, которые нужно отсортировать, но на практике оба варианта этого алгоритма занимают одинаковое время.

Недостаток – плохо работает при малых n (в прочем, как и все усовершенствованные методы)

45. Сортировки слиянием.(ст.358)

Решения задачи сортировки:

1) последовательность a разбивается на 2 половины b и c
2) каждая из получившихся частей сортируется отдельно тем же самым алгоритмом
3) два упорядоченных массива половинного размера соединяются в один

2) части b и c сливаются; при этом одиночные элементы из разных частей образуют упорядоченные пары в выходной последовательности, т.е. первым в ней оказывается меньший из 2х первых элементов

3) полученная последовательность более упорядочена чем изначальная и она снова подвергается *разделению* и *слиянию*; при этом упорядоченные пары переходят в четверки, те – в 8ки и т.д.

Пример: 44 55 12 42 94 18 06 67 → 44 55 12 42 → 44 94 18 55 06 12 42 67 → 44 94 18 55 →
94 18 06 67
06 12 42 67
→ 06 12 44 94 18 42 55 67 → 06 12 44 94 → 06 12 18 42 44 55 67 94
18 42 55 67

При разделение элементы не изменяют своей позиции → опирация разделения (занимающая половину) не эффективна → можно от неё избавиться, сразу записывая выходной массив в 2 (один за другим), которые снова подаются на вход при следующем просмотре. Цена вопроса – ещё один “магнитофон” (4ый) → вместо 2хфазного 3хленточного получаем 1нофазный 4хленточный алгоритм.

46. Модульное программирование. Реализация на языке Си.(ст. 368)

Модульное прогр. - это способ разработки программ, при котором программа разбивается на относительно независимые составные части - программные модули. При этом каждый модуль может разрабатываться, программироваться, транслироваться и тестироваться независимо от других. Внутреннее строение модуля для функционирования всей программы, как правило, значения не имеет. При модификации алгоритма, реализуемого модулем, структура программы не должна меняться

Абстрактный тип данных (АТД) – это “новый” тип, реализованный набором процедур и функций и не поддерживаемый ни аппаратурой, ни системеой программирования.

Но АТД, введённым внутри программы нельзя воспользоваться вне программы (в других программах), т.к. то, что создано внутри программы есть неотемлеммые части программой единицы.

Для того, чтобы программное обеспечение АТД было удобно для использования вне данной программы, необходимо добиться его автономного описания и функционирования в виде некой программной единицы
Для реализации этого в С предусмотренны внешние отдельно компилируемые процедуры (функции).
{т.е. функция может быть описана отдельно от самой программы (объектный/загодовочный файл)}

Связь таких подпрограмм осуществляется после компиляции системным компоновщиком (линковщик).

В модулях прописывается *интерфейс модуля,* который дает доступ к своим объектам. Для взаимообмена между объектами происходит *экспорт и импорт*. Т.е экспортируемые объекты модуля — это константы, типы, переменные и процедуры, которые доступны для использования в других модулях. Импорт объек- тов — это получение доступа к экспортируемым объектам других модулей. Списки объ- ектов экспорта-импорта перечисляются в интерфейсной части модуля.

В Си поддержка модульной архитектуры реализуется просто: каждый .c файл компилируется независимо, но при этом они могут связываться с помощью подключения заголовочных .h-файлов. В сущности, .c-файл - это реализация модуля, а .h-файл - его интерфейс, то есть доступная клиентскому коду часть функциональности. При этом разные реализации могут использовать один интерфейс: пусть есть заголовочный файл list.h, описывающий интерфейс списочной структуры данных, и файлы реализации list_array.c, list_dynamic.c, его подключающие. Тогда управляющий модуль main.c может быть скомпилирован и как gcclist_array.cmain.c, и как list_dynamic.c, и при этом в коде не придётся менять абсолютно ничего.

Примечательно, что разрешает связи между модулями линковщик, и они могут храниться в виде объектного кода, что позволяет перекомпилировать каждый раз не всю систему, а только изменённые модули.
Но стоит отметить, что ручное перечисление файлов реализации при компиляции весьма утомительно, как и ручной сбор зависимостей и управление объектными файлами. Для автоматизации этих рутинных операций удобно использовать утилиту make, разрешающую зависимости между модулями и исполняющую заданные пользователем команды.

47. Абстракции в языках программирования.(ст. 384)

Абстракция – отвлечение от несущественного, помогающее лучше отобразить суть дела.

Абстракция, как и типы, в той или иной степени присуща любому языку программирования сколь-нибудь высокого уровня.
{“Даже понятие переменной представляет собой абстракцию соответствующего текущего значения” Дейкстар}

был осуществлен переход к более высоко- му уровню абстракции, связанному с понятием *абстрактного типа данных.* АТД— это, по существу, определение некоторого понятия в виде класса (одного или более) объектов с некоторыми свойствами и операциями.

В самой развитой форме в определение АТД входят следующие четыре части:
1. внешность (видимая часть, сопряжение, интерфейс), содержащая имя определяемого типа (понятия), имена операций с указанием типов их аргументов и значений и т. п.;
2. абстрактное описание операций и обьектов, с которыми они работают, средства- ми некоторого языка спецификаций, допускающего, в частности, формулирование свойств;
3. конкретное (логическое) описание этих операций на обычном распространенном язы- ке программирования предыдущего поколения;
4. описание связи между 2 и 3, обьясняющее, в каком смысле часть 3 корректно пред- ставляет часть 2.

Наиболее распространённый механизм создания абстракций в программирование – использование процедур.
Разделяя в программе тело процедуры и обращаясь к ней, язык высокого уровня реализует тем самым 2 важных метода абстракций: 1) абстракция через параметризацию и 2) абстракция через спецификацию.

Зачастую программист использует абстракция через параметризацию даже не замечая этого.
Например: необходима процедура, сортирующая массив A; в дальнейшем вероятно надо будет отсортировать другой массив, значит нам нужна процедура, сортирующая произвольный массив, а не только массив A.

Абстракция через спецификацию позволяет нам абстрагироваться от процесса вычислений, описанных в теле процедуры, до уровня знания лишь того, что данная процедура должна в итоге реализовать.
{нам не важно, как кофеварка варить кофе, нам важно знать, что мы получим на выходе, засыпав то или иное}

Простейший пример абстракции в программировании - понятие переменной, сопоставляющей какой-то последовательности ячеек в памяти имя, по которому к ней можно обратиться, и в случае типизированного языка - интерпретации этой последовательности. Ясно, что это намного удобнее, чем непосредственная работа с ячейками памяти.

Это же можно сказать и о типизации: программисту не надо думать о том, как правильно интерпретировать последовательность ячеек памяти, и как при этом ничего не испортить. Среда самостоятельно, из контекста (динамическая типизация) или из текста программа (статическая типизация), определяет интерпетацию именованной совокупности байт памяти и допустимые для неё операции.
Всё это - своего рода абстракция данных.

Другая значимая абстракция - это абстракция кода, известная как функциональная (процедурная) абстракция: вместо того, чтобы каждый раз писать объёмный код выполнения какой-то операции каждый раз, когда это требуется, описывается функция, эти действия инкапсулирующая. Это очень удобно: во-первых, композиция сложных

последовательностей действий сильно запутывает код, а во-вторых - любое изменение придётся многократно дублируя, при этом почти гарантированно создавая ошибки.

Существует и другой механизм абстракции: абстрактные типы данных. Это - объединение воедино данных и операций, определённых над ними, а также их свойств. В сущности, АТД - это прообраз классов в ОО-языках. Как правило, у АТД имеется интерфейс, предназначенный для клиентского кода, и обобщённое описание, достаточное для понимания свойств и задания АТД.

Простой пример - АТД Файл: к примеру, имея обобщённые операции чтения и записи, можно с лёгкостью применять их как для сетевых сокетов, так и для оцифрованного сигнала с микрофона.
При использовании механизмов абстракции важно осознавать, что если одно описание шире другого по объёму, то первое беднее второго по содержанию (пример диалектического закона о переходе количества в качество).
Абстракция через параметризацию (параметры функций позволяют организовывать огромное количество однотипных вычислений над разными параметрыми).
Абстракция через спецификацию - это такое описание абстракции, из которого становится ясно, что она из себя представляет.
Существует также абстракция через итерацию, позволяющая не рассматривать информацию, не имеющую прямого отношения к управляющему потоку или циклу.

48. Абстрактные типы данных. Пример модуля АТД ОЧЕРЕДЬ.(ст. 368 + 372)

см. п.46 + 7 + 10

АТД - это определение некоторого понятия в виде класса объектов с некоторыми свойствами и операциями. В сущности, АТД - это прообраз классов в ОО-языках. Как правило, у АТД имеется интерфейс, предназначенный для клиентского кода, и обобщённое описание, достаточное для понимания свойств и задания АТД.
Рассмотрим для примера АТД очередь, подходящую для описания любого явления, удовлетворяющего описанию "первый пришёл - первым ушёл": это могут быть процессы, ожидающие своей доли процессорного времени, люди, стоящие в банке, или даже пары газа в прямой цилиндрической турбине.

Модуль АТД ОЧЕРЕДЬ:

```
#ifndef __functions_h_
#define __functions_h_
#include <stdio.h>

typedef struct elem elem;
struct elem{
int val;
elem *next;
};
typedef struct{
elem *top;
elem *bottom;
} stack;

void stack_init(stack *s);     // создание
bool empty(stack *s);                  // “пусто?”
voidpop(stack *s);             // удалить первый элемент
```

```
inttop(stack *s);               // значение первого элемента
voidpush(stack *s, intval);     // добавить элемент в конец
voidprint_list(stack *s);       // распечатать содержимое
intcount(stack *s);             // “сколько элементов?”

#endif

#include "functions.h"

void stack_init(stack *s){
s->top = 0;
}
bool empty(stack *s){
return s->top == 0;
}
…
```

49. Экспорт и импорт объектов. Инкапсулированные АТД.(ст. 369 + 384)

Инкапсуляция – отделение абстрактного типа от реализации, с сокрытием её подробностей.

Предположим, что мы хотим написать модуль для чтения из командной строки не зависящий от операционной системы. Для этого мы спрячем подробности реализаций в тело модуля, которое будет меняться от системы к системе, сохраняя при этом функциональную спецификацию.
{т.е., когда мы вызываем функцию *printf*, нам не важно, на какой ОС мы работаем}

Модуль – программная единица, изолированная от объектов других программ, что предохраняет её внутрнности от нежелательного доступа.
К некоторым данным и услугам модуля доступ может быть разрешён; такие разрывы в оболочке называются *интерфейсом модуля*. В соответствии с направлением они называются *экспорт*(из модуля) и *импорт* (в модуль).

50. Типизация языка программирования. Контроль типов.(ст. 394)

Типизация языка программирования – это наделения языка той или иной системой типо. Существуют бестиповые языки (Lisp), т.е. те, в которых все объекты одного и того же типа.

Строгая типизация – это статическая фиксация во время компиляции типа каждой переменной, массива, и т.д.

Деятельность по определению типов переменных, функций и т.п. в программе называют контролем типов (во время компиляции – статический (Си); во время выполнения - динамический).

Типизация языка и контроль типов – важнейшие элементы аппарата защиты языка программирования. Они объявляют: свойства поведения объектов, принадлежность к определённому типу, указание области действия и области допустимых значений в определённых контекстах.

51. Средства ослабления типового контроля. Преобразование и передача типов.

Из практических соображений большинство языков программирования разрешает одновременное использовании данных нескольких типов (например, в одном выражении или разных частях оператора присваивания: было бы неудобно, если бы при записи значения в переменную типа double ВСЕХ участников выражения пришлось бы явно приводить к типу double от типа float, int, longint и т.д; неудобна и невозможность сравнить с помощью < целое число с машинно-рациональным).
Однако для этого требуется механизм согласования типов. Если он отсутствует, то происходит передача типа: внутреннее представление объекта одного типа предаётся функции интерпретации другого типа. Иногда это удобно: к примеру, для записи в файл данных без преобразования значений в изображение (== участка памяти как последовательности байт): в Си для этого достаточно преобразовать указатель на переменную некоторого типа к типу char*, после чего записать байты в файл. В случае родственности (например, математической: целые \sub рациональные \sub действительные) типов может иметь смысл функция преобразования всех значений одного типа в допустимые значения другого и наоборот. При этом ясно, что преобразование может быть неточным (ex.: разномощные множества (биекция невозможна)), необратимым, требовать немалого времени (к примеру, линейного (схема Горнера)).
В отличие от преобразования значений согласованных типов передача типа - это трактовка данных одного типа как значений другого типа (см. пример с записью участка памяти в файл). При этом едва ли в машинном представлении 3.0 побитово равно представлению целого числа 3.
Возможность передачи типов зачастую можно трактовать как способ ослабления типового контроля.
+ См. вопрос 53.

52. Полиморфизм операций, отношений и процедур. Родовые модули.

Родовые модули – одна из идей, важных для коммерческого программирования. К примеру, родовой модуль, реализующий стек, способен вталкивать и выталкивать элементы различных типов данных. Один программист может импортировать этот модуль и использовать его в качестве стека целых значений, а другой – применять его как стек текстовых строк.
Полиморфизм в широком смысле – это возможность объектов с одинаковой спецификацией иметь различную реализацию. Полиморфизм позволяет писать более абстрактные программы и повысить коэффициент повторного использования кода.
Пример полиморфизма функций – функции последовательной передачи данных, определённые для любых потоков (потоком может быть нечто, вводимое пользователем с терминала, обмен данными по компьютерной сети, файл и т.д), и реализующие таким образом единый интерфейс работы с потоками.
Другой полезный пример – функция сортировки, упорядочивающая любые массивы, для элементов которых определено отношение порядка <.
Существует возможность введения полиморфных процедур с параметрами-типами. В данном контексте хочется отметить обобщённое (generic) программирование, хорошо представленное в языке C++: для реализации полиморфных функций, зависящих от типа передаваемых аргументов, в C++ используется понятие шаблона функции.
В языке Си полиморфизм аргументов функций может быть реализован с помощью указателей.
Рассмотрим идею на примере функции стандартной библиотеки языка Си qsort(void * base, size_tnum, size_tsize, int (* comparator) (constvoid *, constvoid *)). Путём задания указателя на начало сортируемого массива, количества его элементов, размера каждого элемента в байтах и указания функции сравнения двух элементов, можно сортировать практически любые объекты.
Аналогично можно описать функцию void* bsearch(constvoid *key, constvoid *base, size_tnum, size_tsize, int (* comparator) (constvoid *, constvoid *)), осуществляющую двоичный поиск ключа *key в массиве num элементов размером size байт каждый,

начинающемся в base. При этом для сравнения используется функция-параметр comparator.
Можно заметить, что это довольно общая идея: сплошной объект в памяти задаётся своей первой и последней ячейками (или аналогично – первой ячейкой и размером элемента в сравнении с размером ячейки).

53. Адресный тип. Реализация полиморфизма с помощью адресного типа на языке Си.

Для обеспечения процессов разнотипной интерпретации памяти необходимо применение родовых типов. Родовые типы данных уже заложены в конструкцию универсальных ЭВМ, память которых представляет собой слова, предназначенные для хранения данных произвольных типов. Ослабляя ограничения строго типизированных языков, можно ввести родовой тип данных word, не предполагающий какого-либо особенного использования памяти, выделенной для его объектов. Для этого типа определены лишь операция присваивания и отношение равенства. Поскольку он внутримашинный, он непечатаемый и в нём нет констант, изображающих конкретные значения. Основное его применение – ослабление контроля типов при передаче параметров. Любой формальный параметр типа word считается совместимым с любым фактическим параметром, занимающим ровно одно слово в памяти ЭВМ.
Более универсальный способ ослабления типового контроля – применение родовых (бестиповых) указателей. В соответствии с описанным выше типом word их можно ввести как указатели на word, распространяя типовую свободу word на мир ссылок и указателей.
В Си бестиповый указатель может быть объявлен как void*. Фактически тип родового указателя вводит в обиход языка адреса соответствующей ЭВМ, единообразно ссылающиеся на данные любых типов, размещённые в ячейках её памяти. Поэтому он называется адресным типом.
Он имеет множеством значений диапазон допустимых адресов ЭВМ (возможно, суженный до сегмента памяти в системах с защитой памяти). Объекты адресного типа совместимы по присваиванию и сравнению с любыми ссылочными переменными, а формальные переменные – с любыми ссылочными фактическими. Кроме того, адресный тип совместим с целым и к нему применимы некоторые арифметические операции, такие как сложение и вычитание.

54. Процедурный тип данных. Реализация полиморфизма с помощью процедурного типа на языке Си.

Это – ещё одно средство реализации родовых модулей. Этот тип позволяет трактовать процедуры и функции как значения, которые можно присваивать переменным или передавать в качестве параметров другим подпрограммам. Фактически, процед. Тип данных представляет собой целый класс типов, отдельные типы которого есть множества глобально определённых процедур с идентичной спецификацией, включая и тип результата для функций. Константами процедурных типов являются имена глобальных процедур. Переменные процед. Типа принимают значения из соответствующего спецификации заголовка множества процедур. Разумеется, её можно и вызвать.
Проц. Тип данных может играть адрес функции, поскольку в конечном счёте для её вызова его надо знать. Примечательно, что процед. Типы в раздельно компилируемых модулях делают эти модули родовыми по отношению к процедурам, а не к типам данных.
Процедурный тип позволяет описывать более универсальные родовые модули с хранимыми процедурами, поскольку подстановка параметров процедурного типа (интегрируемой функции – в «функцию» интегрирования) существенно повышает параметризацию такого модуля.
В Си и C++ просто предусмотрены указатели на функции (с возможностью приведения их к обобщённому указателю void*).
См. код в ЗВЕ.

55. Понятие об объекте. Наследование. Реализация полиморфизма в объектной форме на языке Си.

До сих пор мы интересовались созданием программ «с нуля», не учитывая потребность ис- пользовать имеющиеся программные услуги при создании новых. Соответствующее свой- ство языка принято называть *развиваемостъю* [21]. В частности, к развиваемости относят любые средства расширения языка: процедурность, модульность, наследуемость и объект- ная ориентация.

Основными аспектами развиваемости являются:
Модульность. Она обеспечивает развиваемость за счет фиксации сопряжения (ин- терфейса) между создателем и потребителем услуг.
Стандартизация. В дополнение к модульности, устраняет нерациональное разно- образие сопряжений.
Наследуемость. Подразумевает гибкий аппарат развития, заимствования и защиты, действующий на уровне практически произвольных языковых объектов, а не только на уровне заранее предусмотренных модулей.

Такой уровень гибкости позволяет легко приспособить программу к обслуживанию объектов, тип которых неизвестен не только при ее создании, но и при трансляции (с гарантией статического контроля типов).

Модульность обеспечивает упаковку программных услуг в модули-контейнеры, стандартизация — доставку упакованных услуг потребителю в работоспособном состоя- нии, а наследуемость — изготовление контейнера новых услуг с минимальными затратами, с минимальным риском и в рамках законности.

идеал наследуемости с учетом типизации языковых объектов: должно быть дозволено:
Определять новый тип, наследующий те и только те атрибуты исходного типа, которые желательны.
Пополнять перечни атрибутов нового типа по сравнению с атрибутами исходного типа (обогащение типов (возможность вводить дополнительные поля при объявле- нии производного типа) и виртуальные операции, заменяющие старые операции при действиях с обогащенными значениями, даже в старых программах).
Гарантировать применимость сохраняемых операций исходного типа к объектам но- вого типа.

Основные понятия и неформальные аксиомы наследования: • Основные понятия:
—Отношение наследования между родителем и наследником
— Атрибуты наследования
— Накопление атрибутов у наследника по отношению к родителю
— Типы объектов и экземпляры (объектов) определенных типов
Наследник обладает всеми атрибутами родителей, обратное неверно.
Право участвовать в операциях определенного класса — это атрибут наследования!
Все экземпляры (объекты) одного типа обладают идентичными атрибутами. Следствие: индивидуальные свойства объектов не наследуются и по наследству не передаются.
Наличие наследников не влияет на атрибуты родителей.
Атрибуты могут быть абстрактными (виртуальными), и тем самым требовать кон- кретизации

Преимущества развитой наследуемости:
Гармония открытости и защиты.
Поддерживаемая технология развития программной системы
Поддерживаемый стиль мышления адекватен естественному развитию от простого к сложному, от общего — к частному, от единого корня — к разнообразию,

развитое наследование обеспечивает расширяемость объек- тов, типов и операций с защитой авторского права и с гарантией сохранности старых программ

56. Парадигма функционального программирования.

ФП процесс вычисления трактуется как вычисление значений функций в их математическом понимании.
Принято противопоставлять декларативное (в частности, фукнциональное) и императивное программирование. В декларативном программист описывает, ЧТО он хочет получить, а
в императивном - КАК (процесс вычисления как последовательность изменения состояний).
ФП не предполагает явного состояния программы, а предлагает обходиться вычислением результатов функций от исходных данных и результатов других функций. В частности, не предполагается и изменение состояния (а в импер. парадигме переменная - одна из базовых концепций).
На практике отличие математич. функции от "функции" в императивном программиовании заключается в "чистоте" первых: они опираются только на свои аргументы. В императивном же программировании функции могут иметь побочные эффекты и влиять на (и зависеть от!) внешние по отношению к функции переменные.
Считается, что \lambda-исчисления - это основа для функционального программирования, и многие языки могут считать "надстроками" над ним.
Особенности:
функции высших порядков (интеграл и производная - функции от других функций, возвращающие тоже функции),
чистые функции (без побочных эффектов ввода-вывода и памяти), очень полезные для оптимизации и анализа кода: о коде с чистыми функциями гораздо проще строить предположения, его корректность проще доказать, легко реализовать кэширование результатов вызова функций, порядок вычислений чистых функций можно менять (очень полезно для распараллеливания, и при этом совершенно тривиально!), компилятор может использовать любую поилитку вычислений (к примеру, комбинировать и реоргранизовывать вычислений выражений, например исключать неиспользуемые древовидные структуры).
рекурсия (обычно заменяет цикл, нередко обобщается с помощью функций высш. порядка: к примеру, "свёртка" и "развёртка"),
способ вычисления аргументов (строгий и нестрогий (ленивые вычисления): при первом значении всех аргументов подсчитываются заранее, а при нестрогом - только когда это действительно нужно, реализуется обычно редукцией графа),
Плюсы: надёжность, удобство модульного тестирования, возможности оптимизации и параллелизма.

57. Парадигма логического программирования.

В широком смысле - использование математической логики в программировании.
Основана на автоматическом доказательстве теорем и связана с разделом дискретки, изучающим логический вывод информации на основе заданных фактов и правил вывода.
Логическое программирование основано на теории и аппарате математической логики с использованием математических принципов резолюций (resolve, т.е. "разрешать" в смысле "разрешить зависимость"). Известный язык ЛП - Prolog.
Написать здесь всё можно, что знаешь о мат. логике.
Порассуждать на темы "можно попробовать вывести в обратном порядке" и т.д.