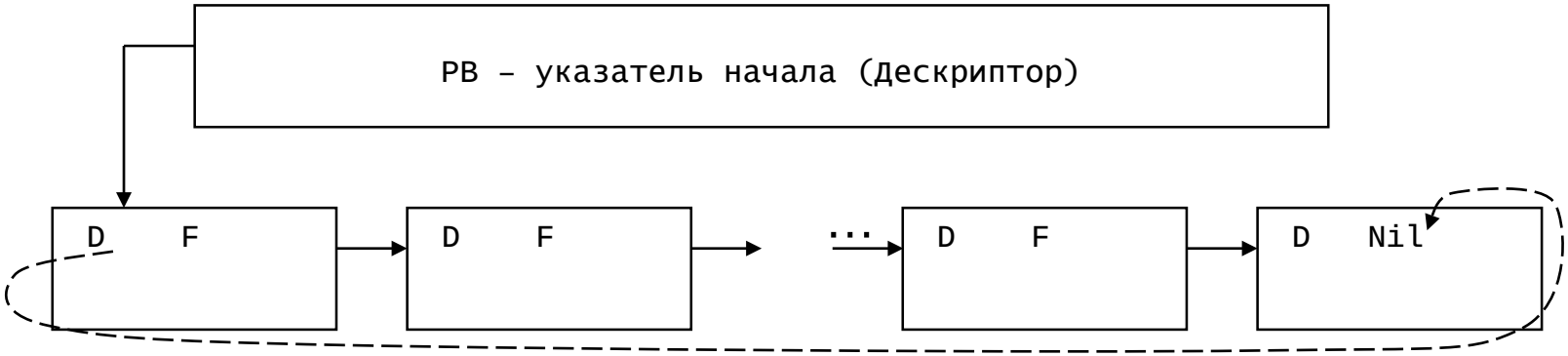
С п и с к и

Динамическая память

Односвязный линейный список

- **это совокупность элементов, содержащих два поля:**
- **поле D – запись с данными поле F – указатель «вперед», т. е., адрес следующего элемента. (**адрес последнего элемента – нулевой).
- PB – указатель начала (Дескриптор)
-
-



- Дескриптор: имя списка PB, количество элементов, описание структуры элементов и т.д.

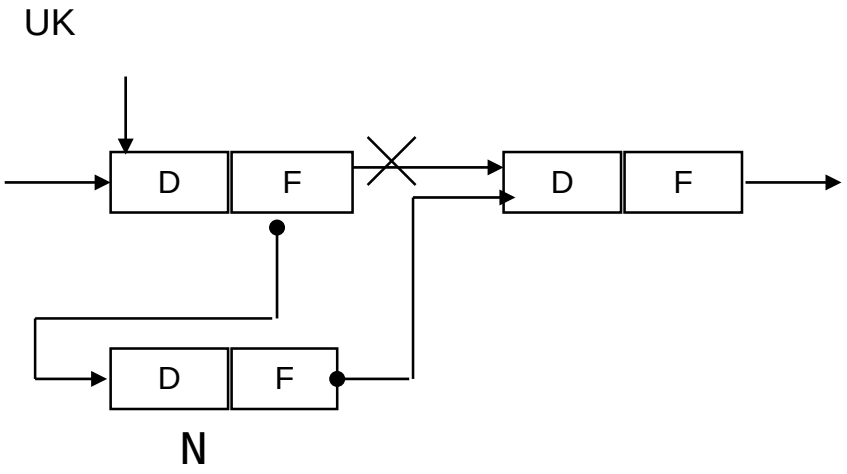
- **<u>Доступ к элементу списка - просмотр списка с головы</u>** (***последовательный***) - медленно.

- ***Кольцевой -*** просматривать можно с любого элемента, после доступа к нужному элементу в PB заносится адрес его последователя.

Главные операции над списками

- Вставка элемента в список;
- Исключение элемента из списка.

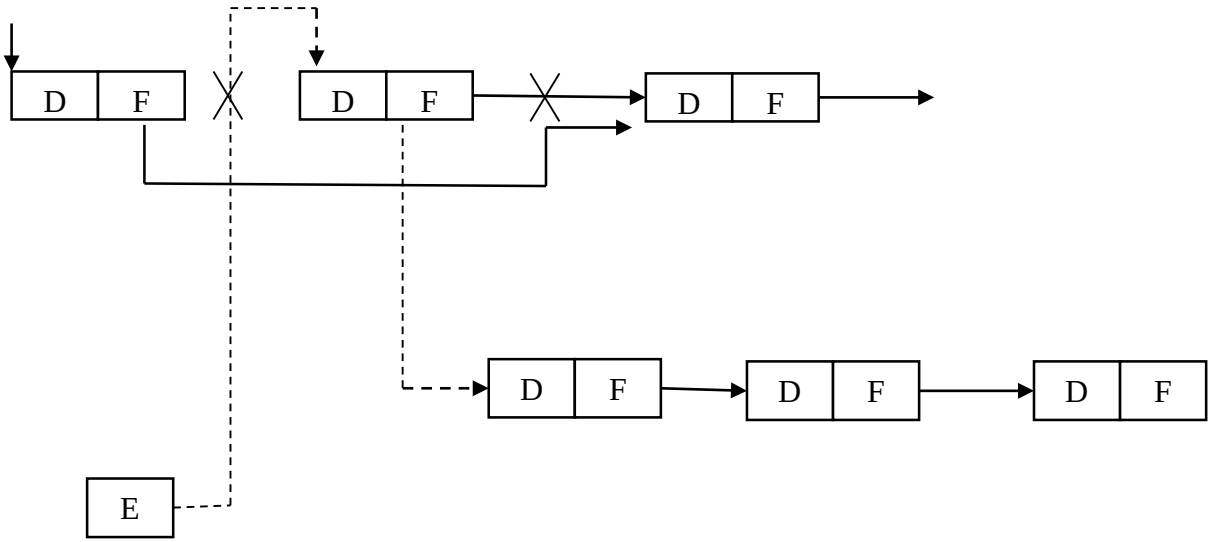
Вставка элемента с адресом N в односвязный список после элемента UK:



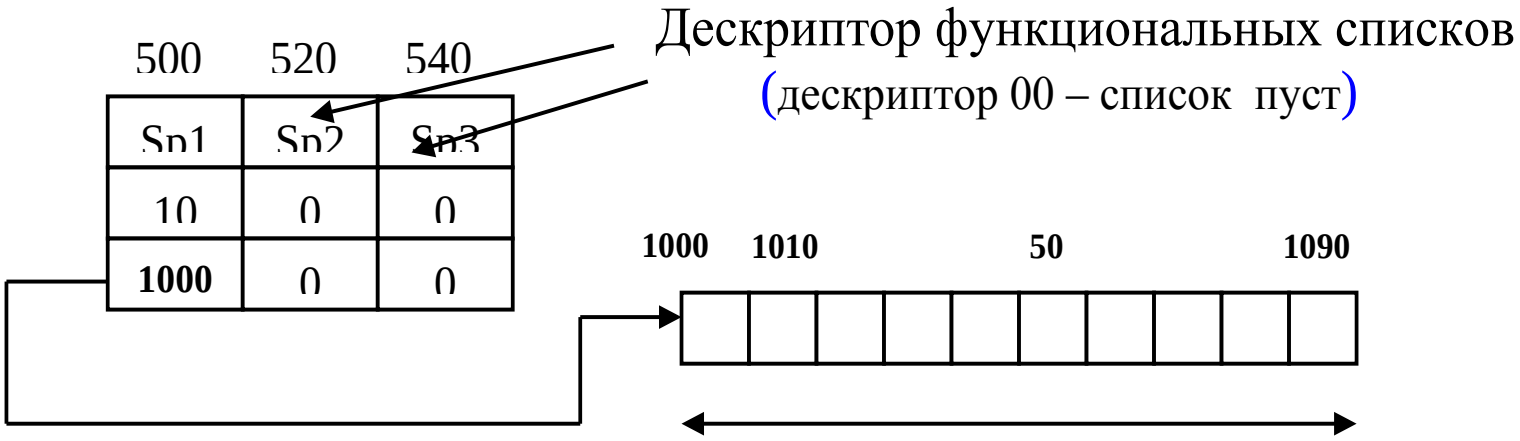

```
F(N) ← F(UK);
F(UK) ← N;
```

- Исключение элемента, стоящего после элемента UK из односвязного списка:
- **IF F(UK) <> Nil**
- **THEN**
- **P ← F(UK)** {запоминаем адрес
- исключаемого элемента}
- **F(UK) ← F(P)** {изменяем адрес указателя
- элемента с адресом UK}
- **ELSE** {исключение невозможно,
- так как список пуст}

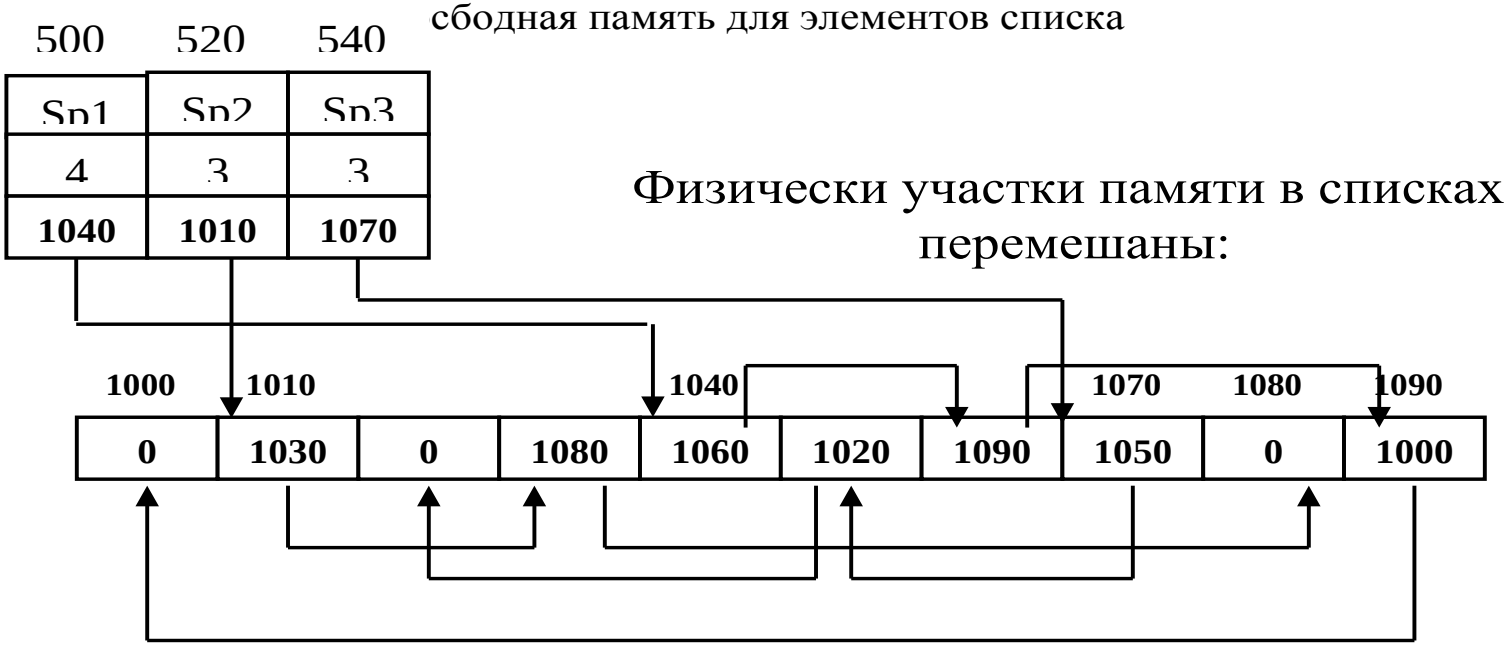
- удаленный элемент включается в начало списка свободных элементов:
- F(P) ← E; UK
- E ← P;
-



-
- При использовании нескольких списковых структур в системе создается не менее 2-х списков:
- включенных элементов (функциональный список с информацией) и
- исключенных (свободных элементов), причем второй – один на все существующие в программе списки.



сбодная память для элементов списка

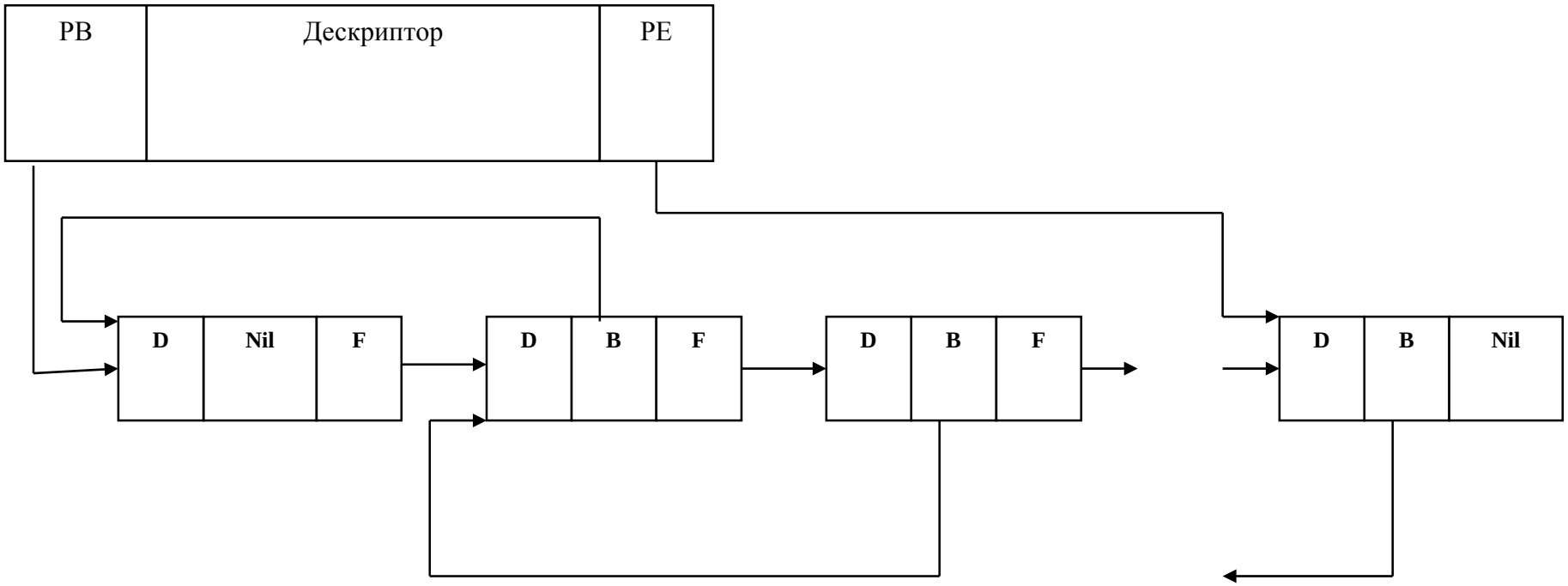


Своб. обл-ти: 1080, 1000, 1020

Двусвязные линейные списки

***Двусвязный* список,** в котором каждый элемент имеет два указателя: вперед (**F**) и назад (**B)**. В структуру этого списка добавляется указатель конца (**PE)**. Начало и конец в этом списке логически эквивалентны, так как доступ к элементу списка имеется с любого конца.

Кольцевой двусвязный список



- ***Кольцевой двусвязный список:*** объединить два пустых указателя, указатель конца не нужен, а указатель начала м. б. в любом месте:

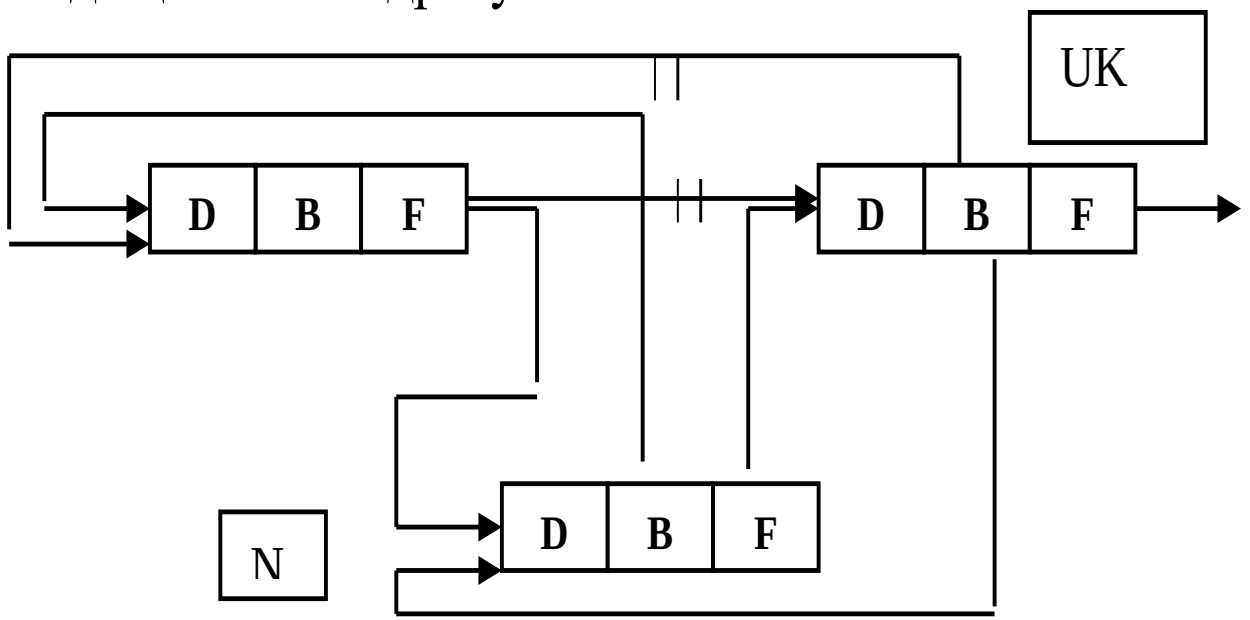
Включение элемента с адресом N в двусвязный список

Включение элемента в двусвязный список

а) перед элементом, находящимся по адресу UK:

B(N) ← B(UK)
B(UK) ← N
F(N) ← UK

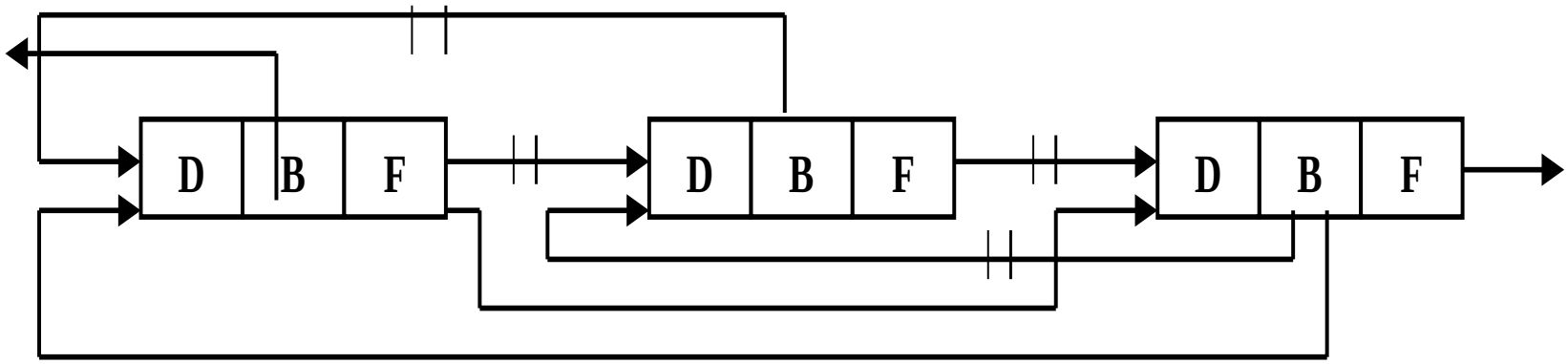
F(B(N) N



б) после элемента, находящегося по адресу UK:

F(N) ← F(UK)
B(N) ← UK
B(F(UK)← N;
F(UK) ← N;

Удаление элемента по адресу UK в двусвязном списке:



B(F(UK)) ← B(UK)

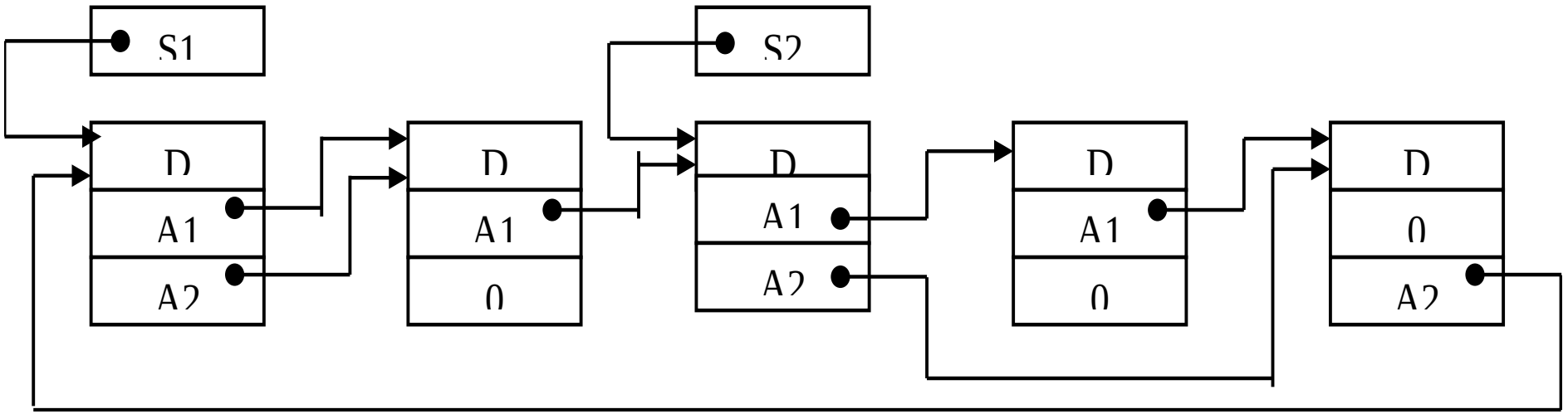
F(B(UK)^ ←F(UK)

Операции с линейными списками

1. Получить доступ к ***k***-му узлу списка.
2. Объединить два (или более) линейных списка в один список.
3. Разбить линейный список на два (или более) списка.
4. Сделать копию линейного списка.
5. Определить количество узлов в списке.
6. Выполнить сортировку узлов списка по некоторым полям в узлах.
7. Найти узел с заданным значением в некотором поле.

Нелинейные списки

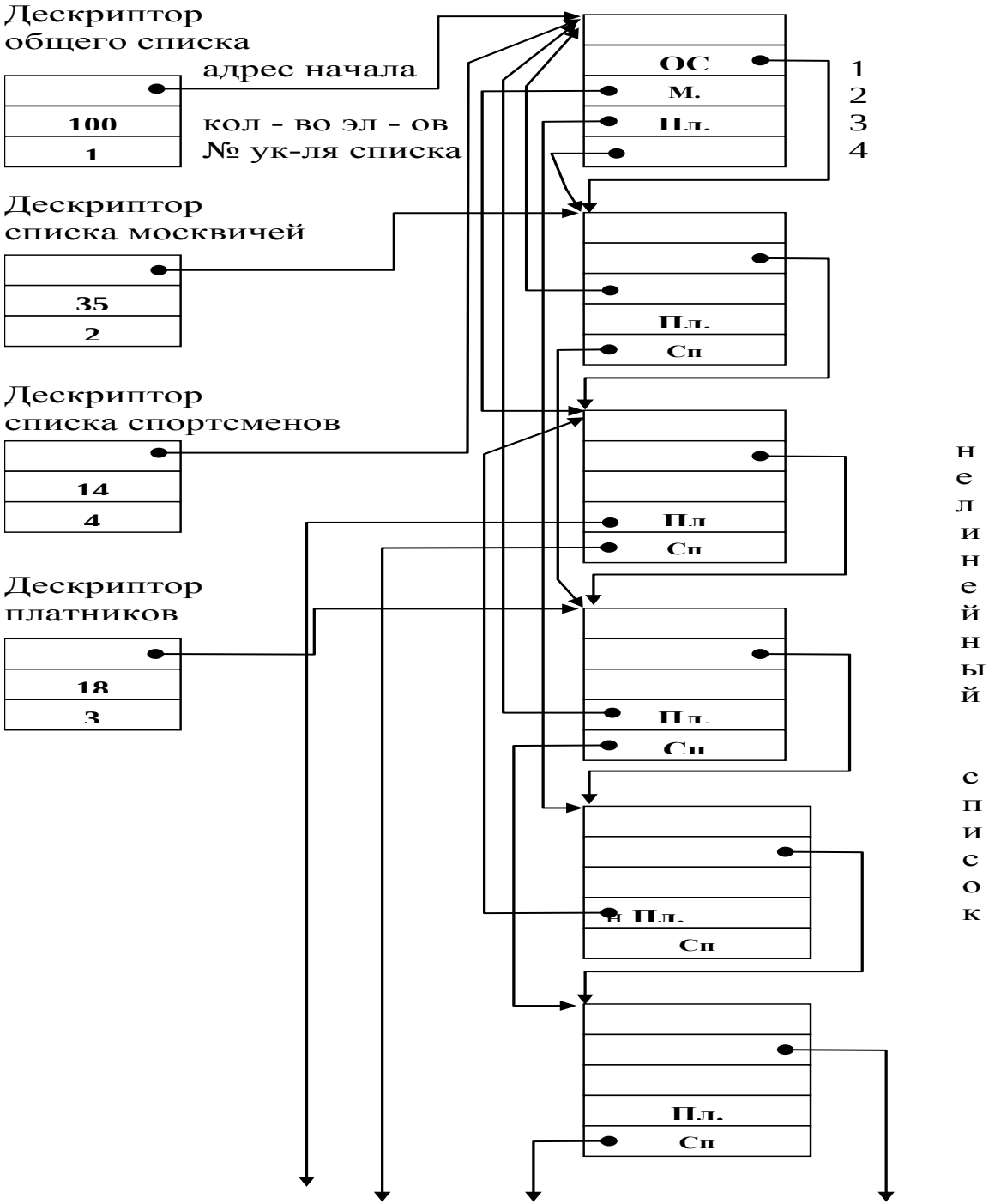
Двусвязный список может быть и **нелинейным**, т. е. второй указатель двусвязного списка задает произвольный порядок следования. Т.о., каждый элемент этого списка содержится в двух односвязных списках, при этом переменные S1 и S2 явл-ся указателями начала двух разных односвязных списков, например:



Многосвязные списки

- В более общем случае каждый элемент связного списка может содержать произвольное конечное число связок, причем, различное в различных элементах. В этом случае получается ***многосвязный список***, каждый элемент которого входит одновременно в несколько разных односвязных списков.
- Такие списки еще называют ***прошитыми (мультисписками).***
- Например, есть списки абитуриентов, содержащие общие сведения: фамилию, имя и отчество, год рождения и т.д. Приемную комиссию дополнительно интересует:
  - Москвич или нет (нуждается ли в общежитии);
  - Спортсмены;
  - Платники.

Пример многосвязного списка



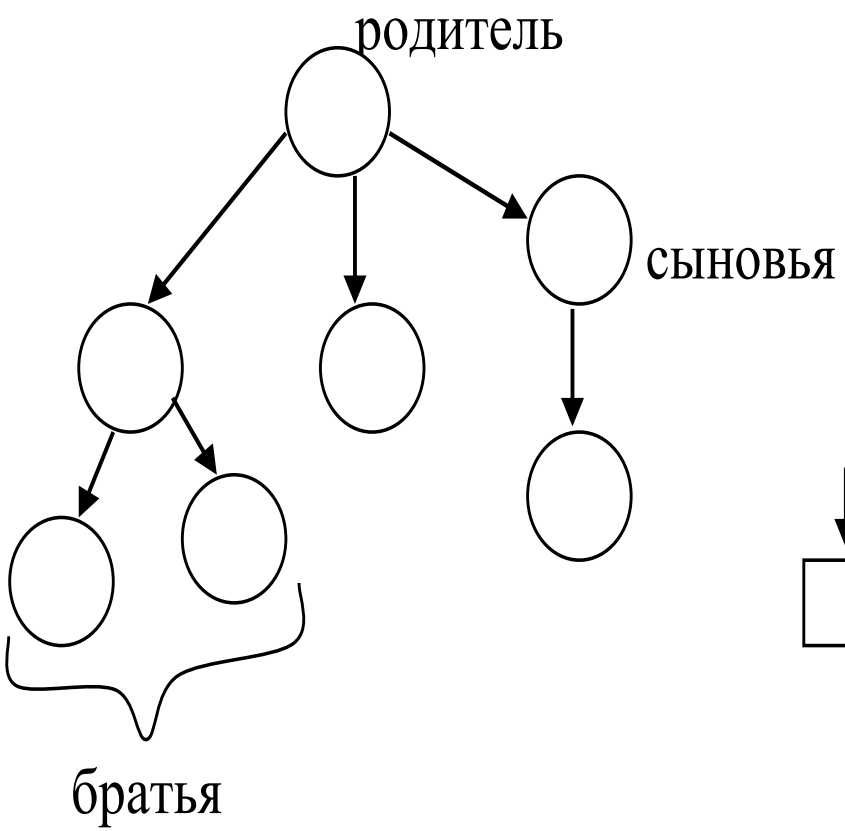
- Мультисписки экономят память. Реальные СД не сводятся к типовым структурам, а представляют некоторую комбинацию из них: и линейные, и циклические, и однонаправленные и двусвязные.

Наиболее общий вид многосвязной структуры

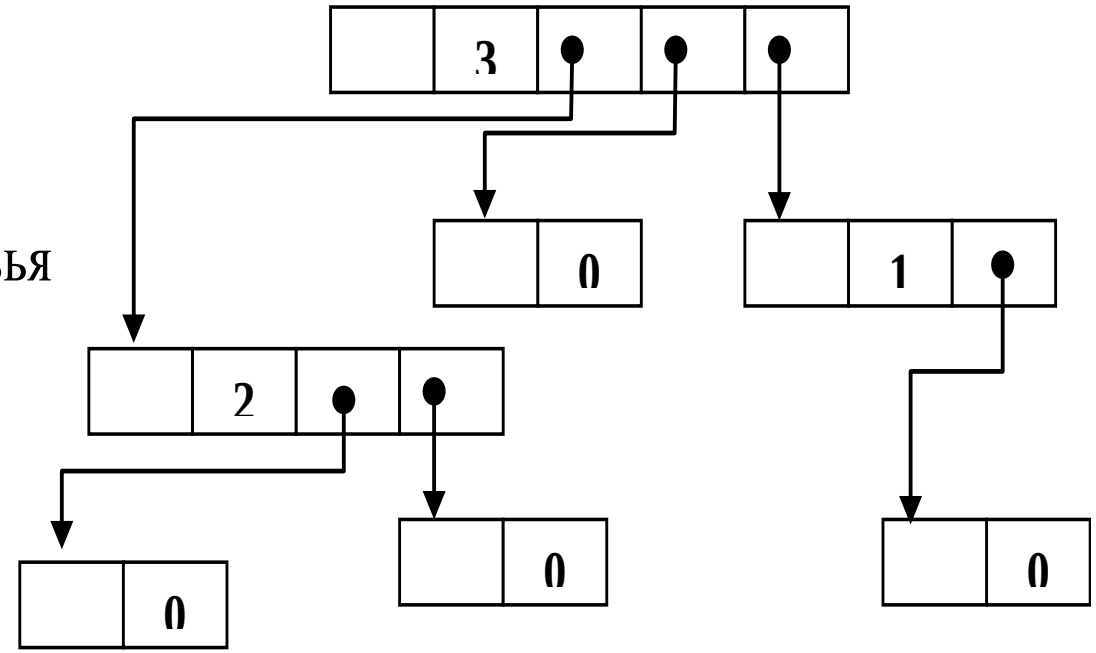
- характеризуется следующими свойствами:
  - Каждый элемент структуры содержит произвольное число направленных связок с другими элементами (ссылок на др. элементы);
  - С каждым элементом может связываться произвольное число других элементов;
  - Каждая связка имеет не только направление, но и вес.
- Такую структуру называют ***сетью***, логически она эквивалентна взвешенному ***орграфу*** общего вида.

Пример различного представления связного списка (т.е. различными структурами)

Дерево:

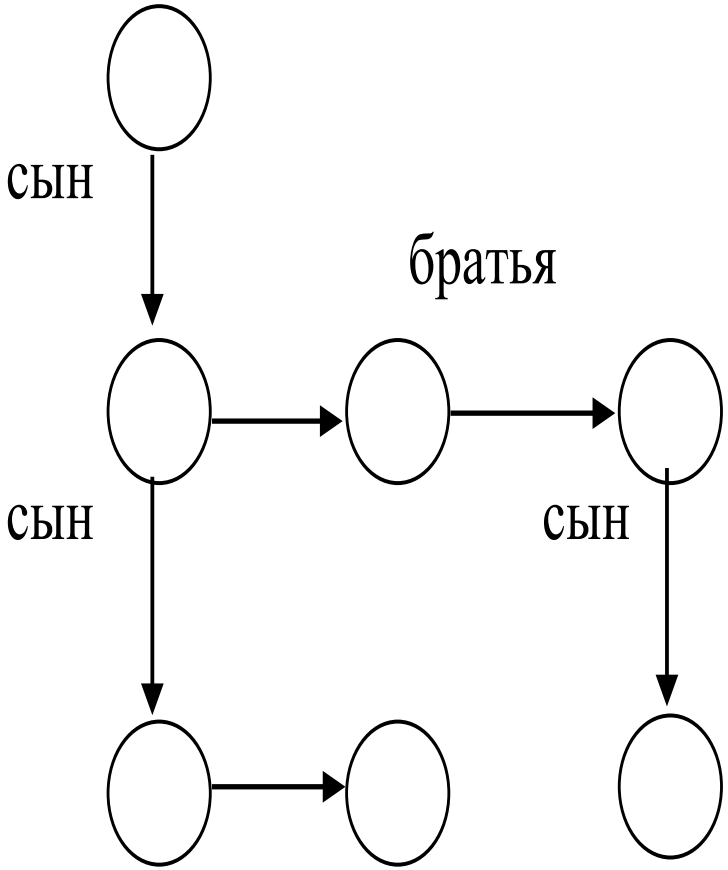


ссылки:

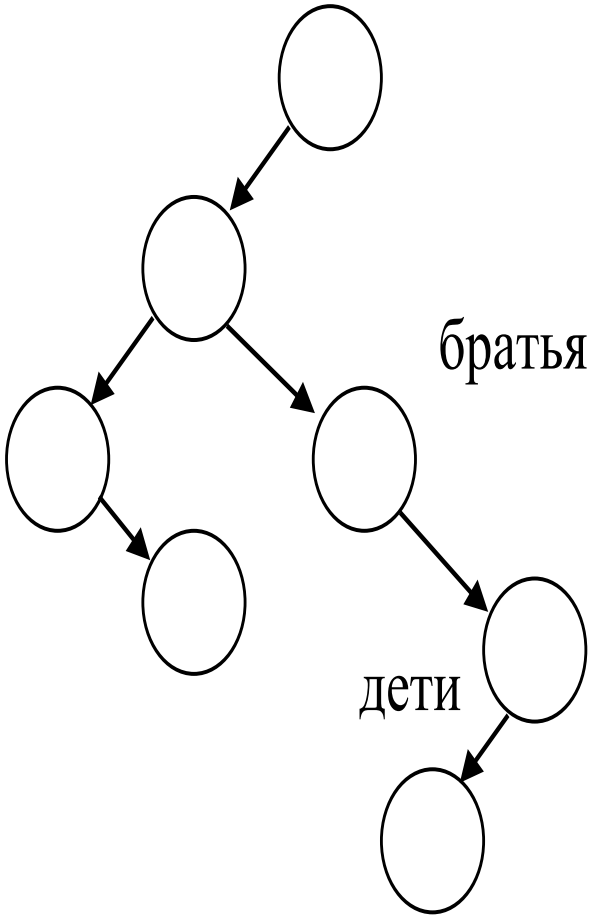


Бинарные деревья

Преобразование в бинарные



слева – сыновья, справа – братья



Сравнение реализаций списков:

- ЕСЛИ:
  - неизвестно кол-во элементов в списке, то реализовать его лучше посредством указателей;
  - частые операции включения или исключения элемента по позиции (с номера), реализуйте список посредством массивов, а не указателей, т.к. в массивах есть прямой доступ к элементу массива и он легче, а указатели вообще не отслеживают позицию.
  - с др. стороны, операции вкл -я или искл-я из массива происходят долго, а с помощью указателей реализовать эти операции и быстрее, и проще.
  - массивы – расточительное выделение памяти - сразу под весь массив, Указатели – дополнительная память в каждом элементе на указатель + время на ее выделение. + время на освобождение памяти.
- Т. е., каждый раз при реализации списков надо думать, какой способ реализации выгоднее в данном случае.

Выделение и освобождение динамической памяти

- В общем случае при распределении динамической памяти должны быть решены следующие вопросы:
- способ учета свободной памяти;
- дисциплины выделения памяти по запросу;
- обеспечение утилизации освобожденной памяти.

Распределение памяти
(на примере Турбо-Паскаля)

- Старшие адреса
- Стандарт-
- ные пере-
- менные

- Младшие адреса памяти
- Сплошной массив байт – куча (Heap)

Оперативная память

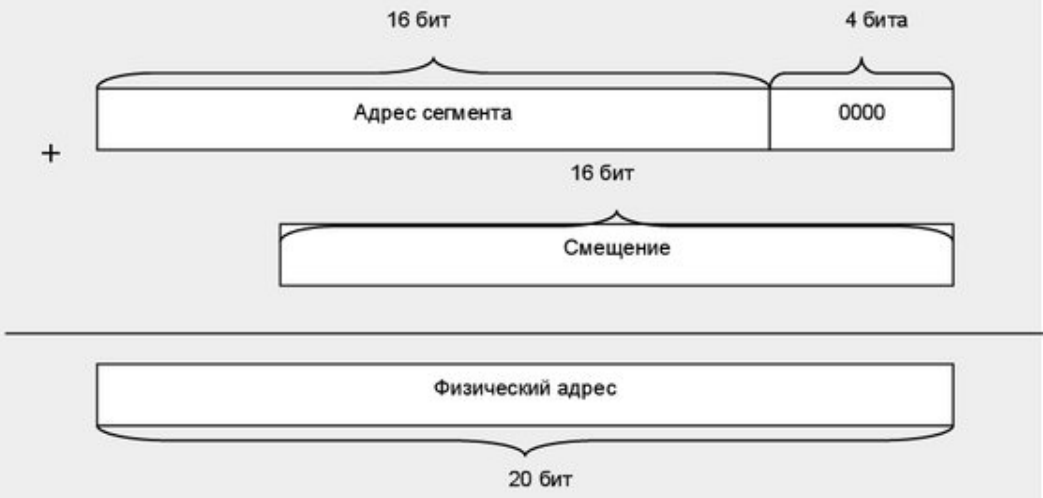
- HeapPtr – адрес нижней границы свободной кучи;
- FreeList – адрес описателя первого свободного блока памяти, который указывает на структуру:

Структура программы в оперативной памяти

В IBM PC-совместимых компьютерах память условно разделена на сегменты.

Адрес каждого байта составляется из номера сегмента и смещения.

Компилятор формирует **сегмент кода**, в котором хранится программа в виде машинных команд, и **сегмент данных**, в котором выделена память под глобальные переменные программы.



Структура описания свободных областей памяти:

Т.о. размер сегмента кратен 16 байтам (параграф - §), максимально – 64Кб, а смещение – адрес относительно параграфа.

- **Type**
- **PFreeRec = ^TFreeRec;** //указатель на запись (структуру)
- **TFreeRec = Record**
- **Next : Pointer**; // адрес описателя следующего по //списку свободного блока кучи
- **Size : Pointer;** // ненормированная длина //свободного блока, либо 0
- **End;**
- Структура используется для описания всех свободных областей памяти, расположенных **ниже** границы HeapPtr, (фрагментация).

Ненормированная длина:

- в старшем слове содержится количество свободных параграфов (§ –16 байт),
- в младшем – количество свободных байт от 0 до 15.

- Функция, преобразующая ненормированную длину свободного блока в байты:
- **Function BlockSize (Size : Pointer) : Longint;**
- **Type**
- **PtrRec = Record**
- **Lo : Word; {свободные байты}**
- **Hi : Word; {свободные §-фы }**
- **End;**
- **Var**
- **LenghtBlock : Longint;**
- **Begin**
- **BlockSize := Longint (PtrRec(Size).Hi * 16 + PtrRec(Size). Lo);**
- **End;**

Сразу после загрузки получим: HeapPtr = FreeList = HeapOrg. При этом в первых 8 байтах кучи хранится запись типа **TFreeRec**, а Next=HeapEnd, Size=0.

При освобождении памяти уменьшается значение HeapPtr, FreeList начинает ссылаться на него и в его начале будет запись **TFreeRec**. Используя FreeList как начало списка, а дминистратор кучи всегда может просмотреть весь список и модифицировать его.Т.К. в любой ссвободный блокпомещается описатель блока, равный 8б, то он (блок) не м.б. <8б, даже если программа запросит 1 байт! Это надо учитывать, чтобы минимизировать возможные потери памяти. Если не находится необходимый участок, то функция выдаст **HeapErrorCode.**

При неоднократном выделении и освобождении памяти администратор кучи может выделять память как из самой кучи, так и из освобожденных ранее участков (**ниже** границы HeapPtr). При этом может возникать фрагментация памяти – участки свободной и занятой памяти перемешаны.

способ учета свободной памяти

Методы учета свободной памяти основываются на принципах:

- битовой карты
- списков свободных блоков

В методах битовой карты создается "карта" памяти - массив бит, в кот. каждый однобитовый элемент соответствует единице доступной памяти и отражает ее состояние: 0 - свободна, 1 - занята.

Если единица распределения это единица адресации (байт), то карта памяти занимает 1/8 часть всей памяти. При единице распределения 16 байт -1/128. Карта может рассматриваться как строка бит, тогда поиск участка памяти для выделения выполняется как поиск в этой строке подстроки нулей, требуемой длины.

- При использовании связных списков в системе есть переменная, в кот. хранится адрес первого свободного участка.
- В начале первого свободного участка записывается его размер и адрес следующего свободного участка.

Дисциплины выделения памяти

Две основные:

"самый подходящий (best fit) и

"первый подходящий" (first fit)

По дисциплине "самый подходящий" выделяется тот свободный участок, размер которого равен запрошенному или превышает его на минимальную величину.

По дисциплине "первый подходящий" выделяется первый же найденный свободный участок, размер которого не меньше запрошенного (эффективнее).

Проблема фрагментации (дробления) памяти

Возникновении "дыр" - участков памяти, кот. не м. б. использованы.

Различаются дыры внутренние и внешние.

- Внутренняя дыра - неиспользуемая часть выделенного блока (характерны для выделения памяти блоками фиксированной длины).
-
- Внешняя дыра - свободный блок, но он мал для удовлетворения запроса (характерны для выделения блоками переменной длины).
- **Управление памятью должно быть построено таким образом, чтобы минимизировать суммарный объем дыр.**

Best fit – увеличивает фрагментацию памяти

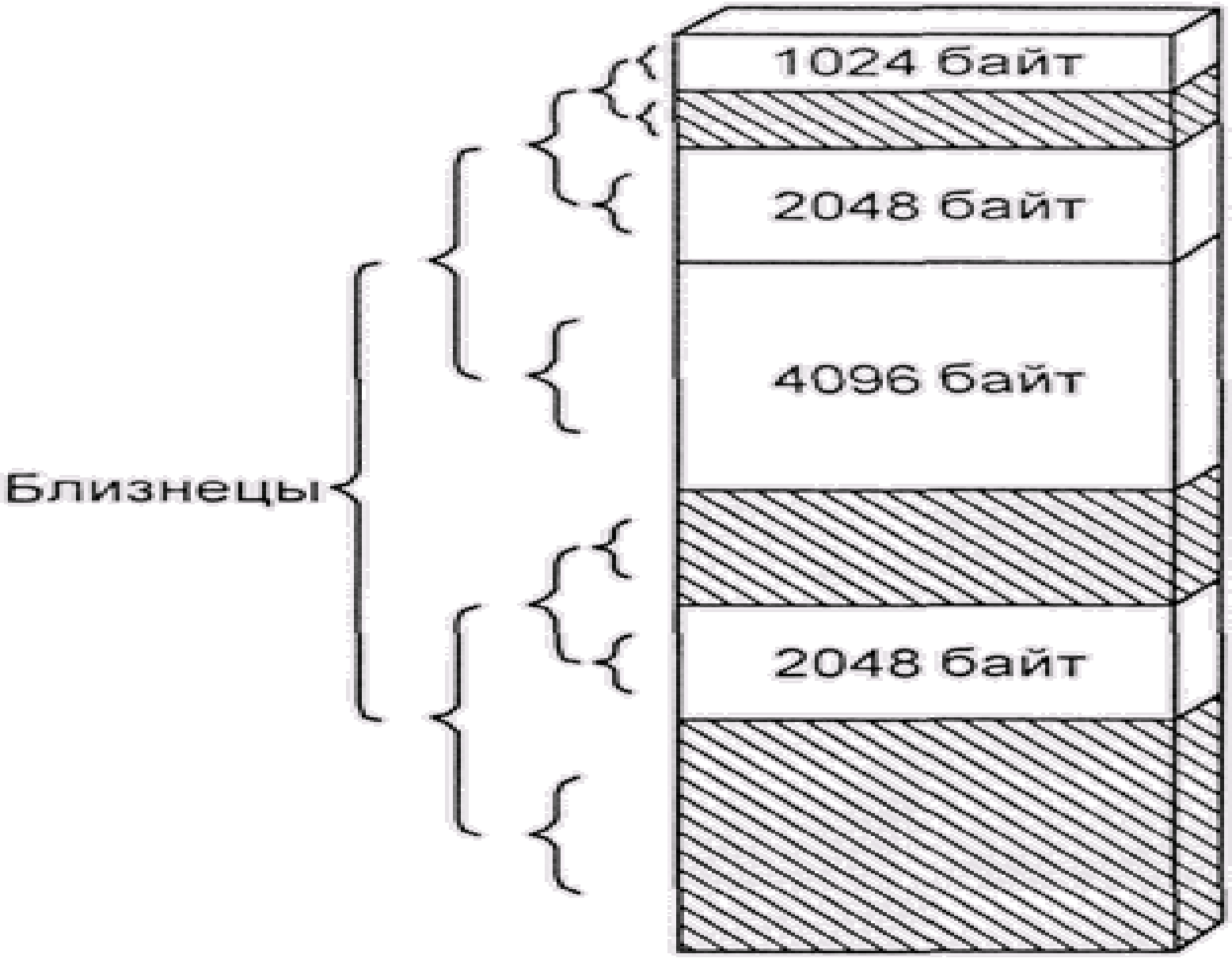
При использовании **first fit** с линейным двунаправленным списком возникает специфическая проблема:

- при просмотре списка с одного и того же места большие блоки, расположенные ближе к началу, будут чаще удаляться, т.о., мелкие блоки будут иметь тенденцию скапливаться в начале списка - увеличивается среднее t-мя поиска.
- Для устранения - просматривать список то в одном направлении, то в другом. Еще проще – сделать список кольцевым, каждый поиск начинать с того места, где мы остановились в прошлый раз. В это же место добавляются освободившиеся блоки. В результате список очень эффективно перемешивается и никакой «антисортировки» не возникает.

- В ситуациях, когда мы размещаем блоки нескольких фиксированных размеров, алгоритмы **best fit** оказываются лучше. Однако библиотеки распределения памяти рассчитывают на худший случай, и в них обычно используются алгоритмы **first fit**.
- В случае работы с блоками нескольких фиксированных размеров напрашивается такое решение: создать для каждого типоразмера свой список. Это избавляет программиста от необходимости выбирать между **first** и **best fit**, устраняет поиск в списках как явление... короче, решает сразу много проблем.

алгоритм близнецов

Вариант подхода для случая, когда различные размеры блоков являются степенями 2-ки (512 байт, 1Кб, 2К и т.д.) Мы ищем блок требуемого размера в соответствующем списке. Если этот список пуст, то берем список блоков вдвое большего размера. Получив блок вдвое большего размера, мы делим его пополам. Ненужную половину помещаем в соответствующий список свободных блоков. Любопытно, что нам совершенно неважно, получили ли мы этот блок просто из соответствующего списка, или же делением пополам вчетверо большего блока, и далее по рекурсии. Одно из преимуществ этого метода состоит в простоте объединения блоков при их освобождении. Действительно, адрес блока-близнеца получается простым инвертированием соответствующего бита в адресе нашего блока. Нужно только проверить, свободен ли этот близнец. Если он свободен, то мы объединяем братьев в блок вдвое большего размера, и т.д.


1024 байт
2048 байт
4096 байт
2048 байт
Близнецы

- Алгоритм близнецов значительно снижает фрагментацию памяти и резко ускоряет поиск блоков. Наиболее важным преимуществом этого подхода является то, что даже в наихудшем случае время поиска не превышает O(log(Smax)-log(Smin)), где Smax Smin - соответственно максимальный и минимальный размеры используемых блоков. Это делает алгоритм близнецов труднозаменимым для ситуаций, когда необходимо гарантированное время реакции – напр., для задач реального времени. Часто этот алгоритм или его варианты используются для выделения памяти внутри ядра ОС. Например, функция kmalloc, используемая в ядре ОС Linux, основана именно на алгоритме близнецов.

- Если мы знаем о блоке его начальный адрес и размер (при выделении памяти «второго рода», т.е. когда размер освобождаемого блока передается как параметр процедуры FreeMem) или когда дескриптор блока содержит только его длину, то это очень плохая ситуация. Т.к. для объединения блока с соседями необходимо найти их в списке свободных блоков, или же убедиться, что там их нет – т.е. просмотр всего списка.

- Гораздо проще запоминать в дескрипторе блока указатели на дескрипторы соседних блоков. Такой метод называется алгоритмом парных меток и состоит в том, что мы добавляем к каждому блоку по два **слова** памяти.
- Именно слова, а не байта. Дело в том, что требуется добавить достаточно места, чтобы хранить там размер блока в байтах или словах. Обычно такое число занимает столько же места, сколько и адрес, а размер слова обычно равен размеру адреса.

утилизация освобожденной памяти

- **При явных запросах** память должна быть освобождена явным образом.
- При представлении памяти на битовой карте: сбросить в **0** биты, соответствующие освобожденным кадрам.
- При исп-нии списков блоков: освобожденный участок д. б. включен в список + при образовании в памяти двух смежных свободных блоков необходимо слить их в один своб. блок суммарного размера. (Задача упрощается при упорядочении списка своб. блоков по адресам памяти).

- **В системах без явного освобождения памяти:**
- **1)** система не приступает к освобождению, пока свободной памяти совсем не останется. («**сборка мусора**»). Алгоритм сборки мусора обычно бывает двухэтапным. На первом этапе осуществляется маркировка (пометка) всех блоков, на которые указывает хотя бы один указатель. На втором этапе все неотмеченные блоки возвращаются в свободный список, а метки стираются. (недостаток метода - расходы на него увеличиваются по мере уменьшения размеров свободной области памяти).

- **2) освобождается любой блок, как только он перестает использоваться**. Реализуется посредством счетчиков ссылок: при 0 - блок не используется. Блок возвращается в свободный список.
- **+** предотвращает накопление мусора, не требует большого числа оперативных проверок во время обработки данных;
- **-** 1) когда все связи, идущие извне блоков в циклическую структуру, будут уничтожены, НО если зарезервированные блоки образуют циклическую структуру, то счетчик ссылок каждого из них не равен 0 ????; 2) требуются лишние затраты времен и памяти на ведение счетчиков ссылок.

- **3) метод «Уплотнение»:** осуществляется путем физического передвижения блоков данных с целью сбора всех свободных блоков в один большой блок.
- **+** после применения выделение памяти по запросам упрощается;
- **-** серьезная проблема - переопределение указателей

метод «Уплотнение»:

- Механизма освобождения памяти в методе уплотнения (восстановления) совсем нет. Используется механизм маркировки, отмечающий блоки, используемые в данный момент, затем используется уплотнитель, кот. собирает неотмеченные блоки в один большой блок в одном конце области памяти.
- **-** 3 просмотра памяти >> затраты времени
- **+** повышенная скорость резервирования в опред. условиях м. компенсировать этот недостаток

- Практическая **эффективность методов** зависит от частоты запросов, статистического распределения размеров запрашиваемых блоков, способа использования системы - групповая обработка или стратегия обслуживания при управлении вычислительным центром.

Плохое использование динамической памяти !!!

- **Example:**
- **while(TRUE) {**
- **void * b1 = malloc(random(10));**
- /* Случайный размер от 0 до 10 байт */
- **void * b2 = malloc(random(10)+10);**
- /* ..... от 10 до 20 байт */
- **if(b1 == NULL && b2 == NULL)**
- /* Если памяти нет */
- **break;** /* выйти из цикла */
- **free(b1);**
- **}**
- **void * b3 = malloc(150);**
- /* Скорее всего, память не будет выделена */