# **ОСНОВНЫЕ СВОЙСТВА ПРОГРАММ. КОНЦЕПЦИИ ПРОГРАММИРОВАНИЯ.**

Современное программирование имеет арсенал из множества языков программирования:

* Языки высокого уровня
* Ассемблеры (низкий уровень)
* Специализированные
* Общие языки

Современное программирование происходит в IDE (среде программирования), например в *VS*, *Bilder* и т.п. Программа пишется на языке программирования, и он определяет ее свойства.

Программные вычисления – это последовательность шагов, т.е. алгоритмическое написание программ преследует цель выполнить то, что не может сделать человек или упростить это действие.

Цель написания эффективных программ часто не совпадает с целью сделать их понятными, так как понятных код не всегда эффективен. Из-за этого противоречия языков программирования начали изменяться: на ранних этапах развития программирование работало непосредственно с автоматами (процессорами), что было сложно и привело к смене парадигмы со структурной на ООП.

Задача программиста – писать понятные и неитерабельные (без повторов) коды [**принцип DON’T REPEAT YOURSEL**]

1. Хаотическая концепция
2. Процедурная парадигма
3. ООП

ООП основано на понятии объекта, который оперирует программа. Объект имеет свойства и разрешенные по отношению к нему методами; структуры таких методов называются *классами*, т.е. у класса есть свои объекты и свои методы работы с этими объектами.

# **СТРУКТУРЫ ПРОГРАММ**

Современные программы организованы в программные комплексы разного уровня сложности (библиотеки, фреймворки и т.п.).

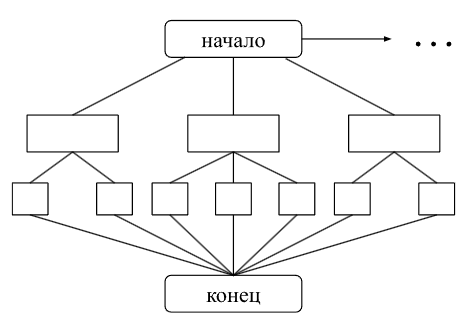
Классификация программ:

1. Без интерфейса
2. С интерфейсом
3. С машинным интерфейсом

Структуры

|  |  |
| --- | --- |
| 1. Последовательная | 1. Диспетчерная |

Сигнал диспетчеру - событие. Поэтому структура управления событием выглядит так:



Событие содержит информацию о том, как его обработать, откуда оно поступило и т.д. Вычисления происходят асинхронно. Начальный блок порождает события, которые тоже создают события, и программа закончится только после обработки всех событий.

Событие – структура в памяти.

Диспетчер событий извлекает события по очереди и раздает их на обработку в соответствующий блок. Как правило, он встроен в систему. Динамическое распределение памяти позволяет это сделать.

Управляющие структуры в программе

Современные языки программирования характеризуются большим разнообразием, однако, несмотря на различия синтаксиса, управляющие конструкции одинаковы. Это свойство процедурных языков. Языки программирования кодируют определенные структуры данных и управление вычислительных процессов. Программа есть совокупность управляющих конструкций, закодированных по правилам синтаксиса.

Правильность работы - важнейшее свойство программы, то есть она не содержит ошибок и выполняет свою функцию.

Одни способы реализации программы корректнее с точки зрения правильности, и они же позволяют это проверить. Следовательно, управляющие структуры должны быть такими: корректными и проверяемыми.

# **ОСНОВНАЯ СТРУКТУРНАЯ ТЕОРЕМА ДЕЙКСТРЫ**

Э. Дейкстра о правильных программах и способах их проверки: все интуитивные попытки проверки программы сводятся к ограниченному числу формальных рассуждений, а в программе - выделить соответствующие управляемые конструкции для проверки (доказательства).

Структуры по Дейкстре:

1. Следование (перечисление операторов через разделители)
2. Альтернативы в разных вариантах

*if (...) then do S1*

*else do S2*

или структура *case i of S1, S2, …*

1. Структуры повторения, циклы.

а) С предусловием *while (...) do S*

б) С постусловием *do repeat S until (...)*

Их основная особенность: 1 вход - 1 выход, последовательное соединение (ошибка в последовательности останавливает всю программу). Это привело к отказу от операторов типа *go to*, без них можно обойтись.

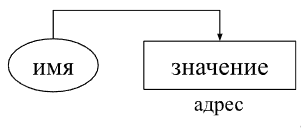
Структуры по Дейкстре - теоретически корректные инструменты уже изначально.

*go to* возможен при обработке исключений в программе.

По Липунову алгоритм реализует логику управления и логику вычисления.

Программные объекты: основные структуры данных (переменные, массивы, поле памяти).

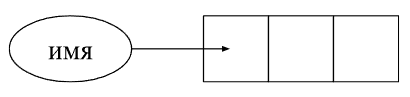
1. Поле памяти (константа или переменная)



, где имя означает адрес.

Размер поля различен, но вмещает только одно значение. Имя указывает на адрес.

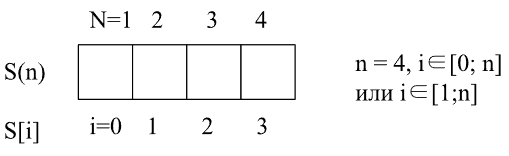
1. Массив



Массив - множество полей памяти одного типа, которые занимают сплошную область памяти, т.е. имя указывает на i-ый элемент.

Размерности массивов

А) Одномерный



индекс i – смещение адреса элемента массива относительно его начала (адреса 1-го элемента)

а) при индексации с i=1:

адрес i-го элемента = адрес 1-го + {(i-1)\*size\_t}\*

{...}\* – смещение

б) при индексации с i=0:

адр. i = адр. 0 + {i\*size\_t}\*

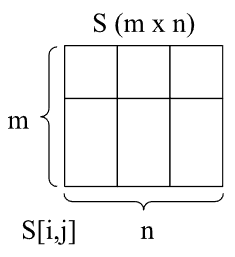
{}\* – смещение

номер элемента равен смещению

в) при инд. от a до b: i принадлежит [a; b]

адр. i = адр. a + (i-a) size\_t.

Б) Двумерные массивы (таблицы)

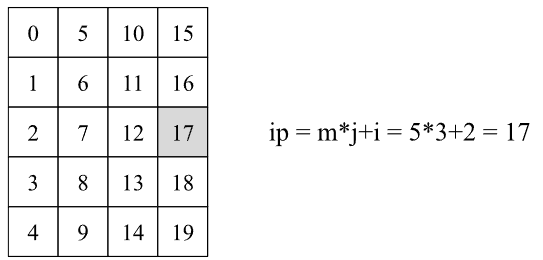


S (m x n), где m-строки, n-столбцы  
рассмотрим схему индексации с 0: таблица в памяти является последовательностью, так как адресная память располагает лишь началом адреса и количеством полей памяти.

а)построчное перечисление элементов 2-го массива



б) расположение по столбцам



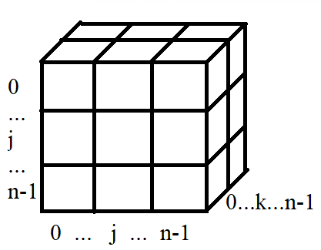
В а) и б) расстановка внутренних индексов будет различной(11 и 17), значит в памяти они хранятся по-разному: иногда это имеет значение при определенных значениях адреса элемента

при индексации с i = 1:

а) ip = n\*(i-1)+j

б) ip = m\*(j-1)+i

**Массивы большей размерности**

Рассмотрим подобным образом, например, трехмерный массив.

а) ip = k\*m\*n+i\*n+j по строкам

б) ip = k\*m\*n+j\*n+i по столбцам

Четырехмерный массив – последовательность кубов

S(m, n, l, f), S[i, j, k, q]

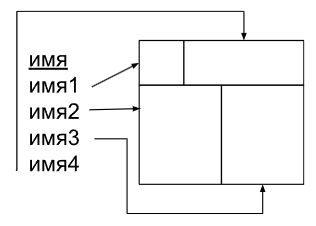
ip = f-m\*n\*k+m\*n\*k+i\*n+j

ip = f-m\*n\*k+m\*n\*k+j\*m+i

С произвольной размерностью используют схему Горнера, где размерность М, то S[i, j, k, …, x, y, z], тогда elem = (адрес начала массива + sizeT)\*(z + mz(y + x\*my((... + mk(j + imj)...))).

# **СТРУКТУРЫ**

**Структура** – совокупность полей разного типа.

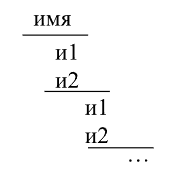
Структура (запись) разных полей памяти:

имя = {имя\_1, …, имя\_n}

Для адресации используется смещение специальным оператором (например точкой).

ip = имя.имя1

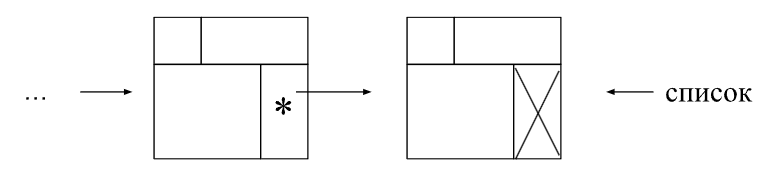
Элемент структуры может быть любого типа, кроме самой структуры (либо может быть только указатель на нее), иначе возникает рекурсия:



Если последний элемент адрес:

Структура конечна, т.к. рано или поздно \*u2 = null.

**Список** – цепочка элементов, связанных ссылками, которые могут занимать неизменную область памяти.



Если ссылка нулевая, то это последний элемент структуры.

Структура может быть представлена списком вида

имя = {u1, …, \*un};

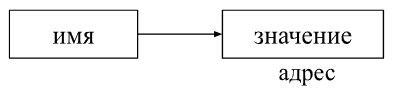
**Динамические структуры данных и причины их использования**

Основная причина: в момент объявления размер объекта неизвестен, неясно, сколько нужно памяти для него, какие в ней будут значения и сколько их; есть информация только о типе данных (структур).

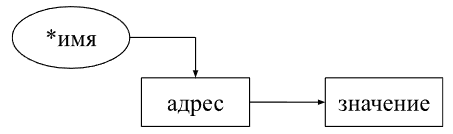
Наиболее простое решение – сознательно увеличить число типов данных.

Прямая и косвенная реализация

а) прямая



б) косвенно с помощью указателей



Если переменная является указателем, то:

1. можно узнать ее содержимое - адрес

*int\* a, \*a* - адрес

1. можно узнать само значение

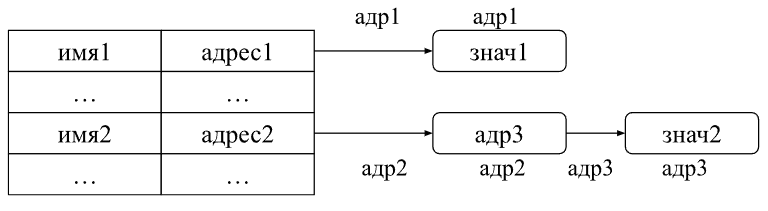
*int\* a, &a* - значение по адресу *a*

При работе с указателями важно заполнять их верными адресами.

*\*а => dereferencing*, адрес

*&a =>* значение по адресу

Во время разбора программы создается таблица имен.



имя1 -> знач1

&имя1 -> адр1

\*имя1 -> ошибка

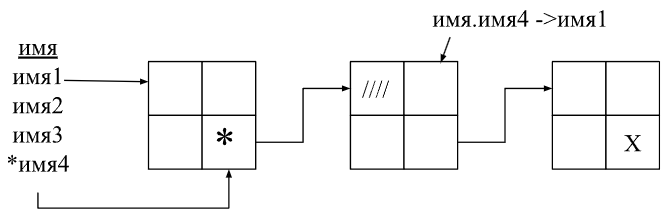
имя2 -> адр3

&имя2 -> адр2

\*имя2 -> знач2

**Рекурсивная структура**

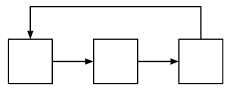
Механизм работы с указателями позволяет организовать рекурсии. В случае использования указателей не возникает бесконечной рекурсии, если в качестве указателей используется она сама.



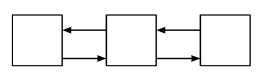
Всегда можно указать конец этой цепочки (О) во избежание рекурсии.

# **ЛИНЕЙНЫЙ СПИСОК (LIST)**

1. однонаправленный

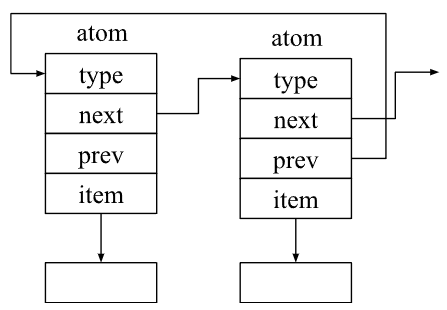


1. двунаправленный



Однородный – из элементов одного типа, разнородный из разных.

**Методы работы со списком**

****

atom - элемент списка атомарной структуры

atom {type, \*item, atom\* next, atom\* prev}

Весь список: list {atom\* head, \*cur}

1. create - создать - выделить память ~ new
2. next - движение вперед
3. prev - движение назад
4. head - движение в начало
5. last - движение в конец
6. add - добавление после текущего
7. addhead - добавить в начало
8. addlast - добавить в конец
9. cut - удаление текущего элемента из списка (обрыв связи) и очистка памяти при необходимости

При удалении элемента нельзя терять его адрес; следующий элемент станет текущим.

Создадим базовый набор этих функций.

create(list l) - создать пустой список

{

l.c = 0;

l.h = 0;

}

atom\* next(list l)

{

if (l.cur->next == null)

return null;

return l.cur = l.cur->next;

}

atom\* prev(list l)

{

if (l.cur->next = 0)

return 0;

return l.cur = l.cur->prev;

}

atom\* head(list l)

{

if (l.cur = l.head)

return 0;

return l.cur = l.head;

}

atom\* last(list l)

{

if (l.cur->next = 0)

return 0;

while (l.cur->next != 0)

l.cur = l.cur->next;

return l.cur;

}

Проверим ситуации:

|  |  |
| --- | --- |
| а) список пуст | б) список не пуст, добавление в конец |
| в) добавить не в конец | |

atom\* add(list l, atom\* a)

{

if (l.head = 0)

{

l.head = a;

a.prev = a.next = 0;

}

else

{

a->prev = l.cur;

a->next = l.cur->next;

l.cur->next = a;

if (a->next != 0)

a->next->prev = a;

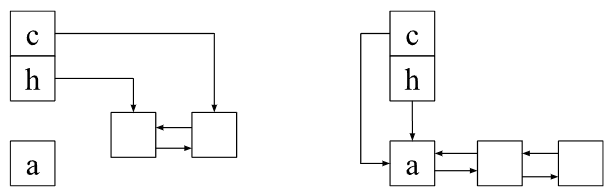
l.cur = a;

}

return l.cur;

}

atom\* addhead(list l, atom\* a)



{

if (l.head = 0)

a->next = 0;

else

{

a->next = head;

l.head->prev = a;

}

a->prev = 0;

l.head = l.cur = a;

return l.cur;

}

atom\* cut(list l)

{

if (l.head = 0)

return 0;

if (l.cur->next != 0)

l.cur->next->prev = l.cur->prev;

if (l.cur->prev != 0)

l.cur->prev->next = l.cur->next;

if (l.cur->next != 0)

l.cur = l.cur->next;

else

if (l.cur->prev != 0)

l.cur = l.cur->prev;

else

l.cur = l.head = 0;

if (l.cur->prev = 0)

l.head = l.cur;

return l.cur;

}

Операция cut не удаляет сам элемент из памяти, а только разрывает связь с остальными. Нужно запомнить его адрес и почистить его ссылки, либо удалить за ненадобностью. Адрес сохраняется в буфере.

atom\* del(list l)

{

l.clip = l.cur;

cut(l);

delete l.clip->item;

delete l.clip;

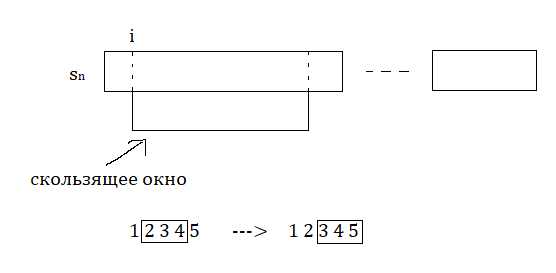
l.clip = 0;

return l.cur;

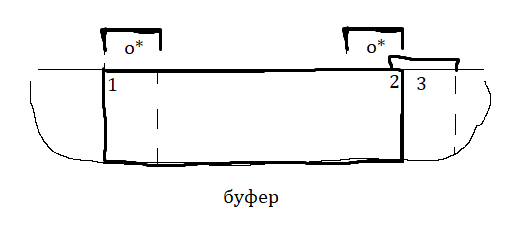
}

# **ДВОЙНОЙ КОЛЬЦЕВОЙ БУФЕР.**

Часто при обработке последовательностей возникает необходимость иметь в памяти несколько подряд идущих элементов (скользящее окно).

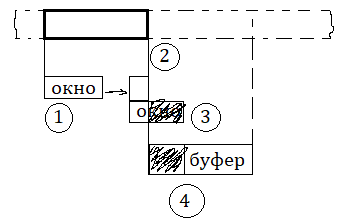


При сдвиге на одну позицию изменяются крайние элементы. С использованием буферизации возникает следующая ситуация:



, где o\* - окно, 1 и 2 – границы буфера

Со следующего шага (после второго) необходимо «закольцевать» последовательность. Перезагрузим буфер:



В случаях **1** и **2** все элементы последовательности доступны в памяти одновременно. Для доступа к ним за пределами буфера необходима перезагрузка буфера. Отсюда возникает биение (beating) / тряска (shaking) / вибрации (jarring) (активное движение элементов) считывающего механизма и замедление его работы.

Решением данной проблемы является двойной кольцевой буфер при условии, что окно меньше половины буфера. Одна из половин активна. Если требуемый элемент в ней, то берём оттуда, иначе переключаемся на другой. Если элемент вне буферов, то снова переключаемся и перезагружаем буфер. В итоге удобно переключаемся между двумя буферами.

Определим структуру вида:

object double ddbseq {dbseq b1, dbseq b2, act1, act2}

openddb (ddbseq n, len)  
{  
 opendb(n.b1, len)  
 opendb(n.b2, len)  
 n.act1 = 1  
 n.act2 = 0  
}

closeddb (ddbseq n)  
{  
 closedb(n.b1)  
 closedb(n.b2)  
}

resetddb (ddbseq n)  
{  
 resetdb(n.b1)  
 resetdb(n.b2)  
 n.act1 = 1  
 n.act2 = 0  
}

putddb (ddbseq n, item, pos) // функция вывода записи  
{  
 if (pos < 0) then error  
 if (n.act1 != 0) then // смотрим первый буфер  
 if (pos < n.b1.last && pos >= n.b1.first) then // мы внутри буфера [fisrt, last]  
 {  
 n.b1.last[pos - n.b1.first] = item  
 n.b1.update = 1  
 }  
 else  
 {  
 n.act1 = 0  
 n.act2 = 1  
 }  
 if (n.act2 != 0) then // смотрим второй буфер  
 {  
 if (pos <= n.b1.last && n.b2 >= n.b1.first) then  
 {  
 n.b2.put[pos - n.b2.first] = item  
 b.b2.update = 1  
 }  
 else // не попали ни в один из буферов  
 putdb(n.b2, item, pos)  
 }  
 return  
}  
// дальше идёт – хз что

if (n.act2 != 0) then  
{  
 if (pos <= n.b2.last && pos >= n.b2.first) then  
 { n.b2.put[pos – n.b2.first] = item  
 n.b2.update = 1 }  
 else { n.act2 = 0, n.act1 = 1 }  
}  
if (n.act1 != 0) then  
{  
 if (pos <= n.b1.last && pos >= n.b1.first) then  
 {  
 n.b1.put[pos – n.b1.first] = item  
 n.d1.update = 1  
 }  
 else { putdb(n.b1.item, pos) }  
}  
return  
} – конец

Чтение выглядит аналогично, только пересылка в обратном направлении (из буфера в item)

1. getddb (ddbseq n, item, pos)  
   {…}

Пример программы подсчёта среднего арифметического в окне:

ddbseq a // создать объект  
openddb(a, 0) // откроем как пустую  
putd(a.b1, 0, 1000) // запишем 1000 нулей  
resetddb(a)  
for (i = 0, i < 1000, i++)  
{  
 input(item)  
 putddb(a, item, i) // ввод из потока  
}  
resetddb(a) // позиция в начало  
if (wsize > a.b1.size) then error // проверка размера окна  
for (i = 0, i < 1000 – wsize + 1, i++) // чтобы не уйти за границы  
{  
 sum = 0  
 for (j = 0, j < wsize, j++) // вычисление в окне wsize  
 {   
 getddb(a, item, i+j)  
 sum += item  
 }  
 sum = sum / wsize // среднее в окне  
 output(sum)  
}  
closeddb(a)

При работе программы если не будет перестановки в начало после ввода, то самой системе придётся сделать это.

Файловый доступ:

- Низкоуровневый – хендлеры

- Высокоуровневый – работа со структурами и классами, работа с объектом типа file.

# **ХЭШИРОВАНИЕ**

Хэш-функция преобразует входную информацию в битовую строку определенного размера. Эта строка называется хэшем или просто хэш сумма. Хэш функции обычно используются для проверки целостности файлов или хранения паролей. Хеширование паролей применяется для безопасного хранения данных пользователей. Хорошая и надежная криптографическая хэш функция должна иметь несколько основных свойств:

- Устойчивость к коллизиям (без повторов хэшей)

- Однонаправленность (нельзя рассчитать обратно первичное значение)

- Быстрое вычисление хэш функции

- Уникальный ХЭШ для разных наборов информации

- Наличие лавинного эффекта(оч. разные хэши при малейшем изменении)

Хэширование – метод организации хранения эл. доступа для последующего быстрого поиска (расстановка ключей).

Операция поиска не является массовой только в случае битовой реализации, если число возможных индексов меньше диапазона возможных чисел, но это бывает редко.

Можно использовать функцию отображения «Многие к одному» - хэш-функцию - для разных ключей, так мы получим одинаковые индексы. Хэш-функция отображения множества ключей на множество индексов массива из p:

Получение индекса по ключу: i=h(k)

Обратное представление «один по многим». Инъекция: .

Проблема в отсутствии конкретики обратного преобразования.

Индексы делят множество на подмножества.

I=h(k) – однозначное

=h – неоднозначное, подмножество.

Исходное множество ключей разбивается на фиксированное число подмножеств. Если мощность множества ключей не превышает множество индексов, то это аналог битовой реализации, но чаще – наоборот.

Нужно, чтобы хэш-функция разбивала множество на равномерные подмножества (генератор равномерного распределения). В итоге массовость операции поиска снижается.

*Задание хэш-функции*

f принимает целые значения. Можно брать сумму цифр ключа: первый символ и т.д. Разные хэш-функции – разные степени равномерности.

*Разрешение конфликта, когда индексов меньше, чем ключей*

Для разных ключей возникает одинаковый индекс. Неясно, какое значение нужно изъять. Создается список соответствий ключей (прямое связывание), или повторно применяют хэш-функцию, пока не найдется совпадение по ключу или свободный индекс. Вопрос лишь в том, сколько мы будем так «прыгать» по индексам, I=h(k)

Известны разные способы поиска повторных индексов (проба).

Проба – попытка определить повторный индекс. (Бывает линейная и квадратичная).

- линейная

) mod P; i=1, p-1, (i>0)

Это позволяет проверить, не занят ли (i+1)-ый индекс.

- квадратичная

) mod P; i>0

Здесь индексы более разрежены. Но некоторые индексы точно будут не заняты (массив индексов избыточен).

*Сортировка прямым выбором*

При прямом выборе для поиска одного элемента с наименьшим ключом (или значением) просматриваются все элементы входной последовательности, и найденный элемент меняется местами (ссылками) с первым. Операция повторяется рекурсивно до следующего n-1 элемента, потом n-2и так далее, пока не останется один максимальный элемент.

Псевдокод:

Int temp, min

for(i=0;i<size-1;i++)

{

min=I; // запомнили текущий элемент

for(j=i+1;j<size;j++)

{

if(num[j]<num[min]) min = j

}

temp=num[i]; num[i]=num[min]

num[min]=temp

}

}

# **СОРТИРОВКИ**

*Сортировка вставками*

Список разбивается на две части: отсортированную и неотсортированную. Из неотсортированной части извлекается элемент и вставляется на свое место. Неотсортированная часть уменьшается на 1. Процесс повторяется, пока неотсортированная часть не закончится (последний элемент будет уже на своем месте).

**void** **Sort**(**int**\* arr,**int** n){

**int** counter=0;

**for**(**int** i=1;i<n;i++){

**for**(**int** j=i; j>0 && arr[j-1]>arr[j];j--){

counter++;

**int** tmp=arr[j-1];

arr[j-1]=arr[j];

arr[j]=tmp;

}

}

}

*Обменная сортировка*

Маленькое значение «всплывает», как пузырек

 for (int i = 0; i < size - 1; i++)  
  {  
    for (int j = (size - 1); j > i; j--) // для всех элементов после i-ого  
    {  
      if (num[j - 1] > num[j]) // если текущий элемент меньше предыдущего  
      {  
        int temp = num[j - 1]; // меняем их местами  
        num[j - 1] = num[j];  
        num[j] = temp;  
      }  
    }  
  }

# **ДЕРЕВЬЯ (ИЕРАРХИЧЕСКИЕ СПИСКИ)**

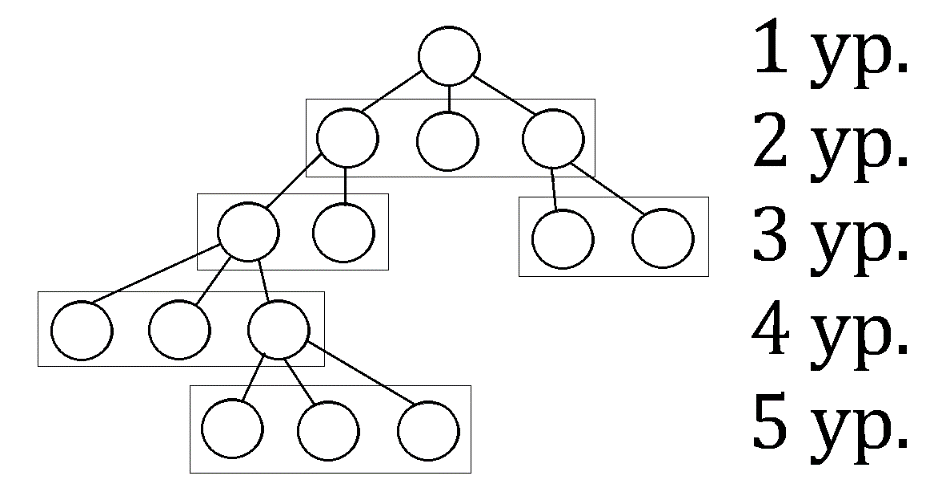
Структура в виде дерева – один из способов организации ссылок элементов друг на друга в некотором множестве. В них отсутствуют циклические ссылки элементов (если элемент пройден – обратно не попасть)

Каждый элемент может быть связан более, чем с одним. При этом подмножества элементов, связанных с ним, не пересекаются, то есть дерево с вершинами и рёбрами в виде ссылок   
(предок 🡪(ссылка)🡪 потомок). При этом перекрестий нет.

Деревья – частный случай однонаправленного потока.

Дерево можно понимать, как комбинация списков, оно имеет начало (корень) и концы (терминалы / листья). Каждая ветвь является линейным списком.

Способы организации древовидных списков:

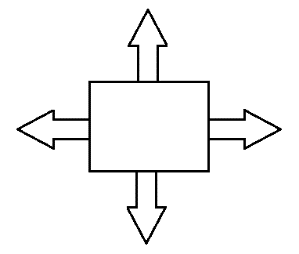


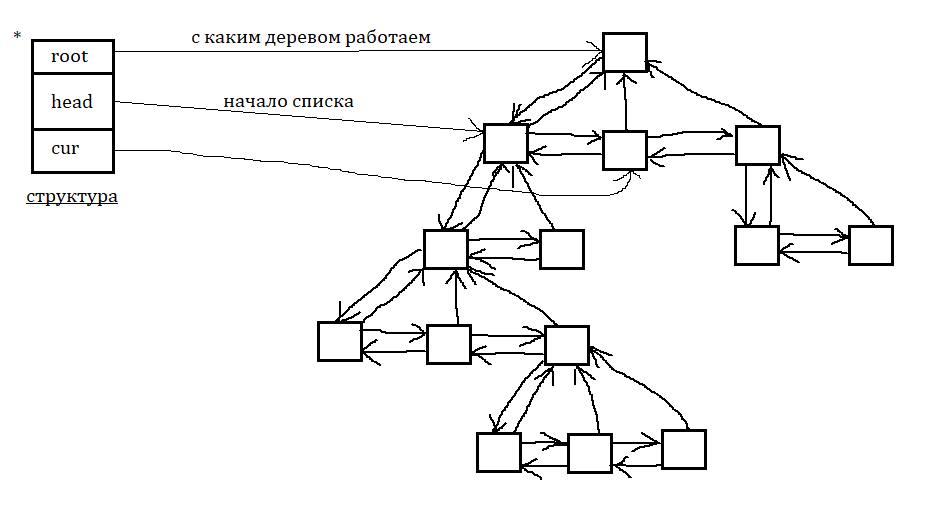
1 уровень – корневой

Как связать элементы ссылками? Проблема в том, что число потомков переменно и не определено. Каждый уровень можно рассматривать как отдельный список.

1 уровень – список из 1 элемента

список 2-го уровня: из каждого элемента теперь не 2, а по 4 ссылки:

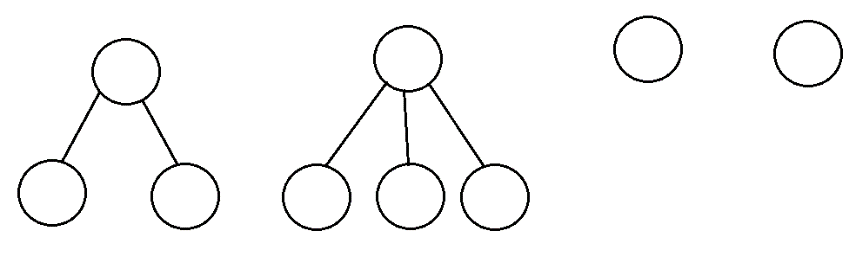
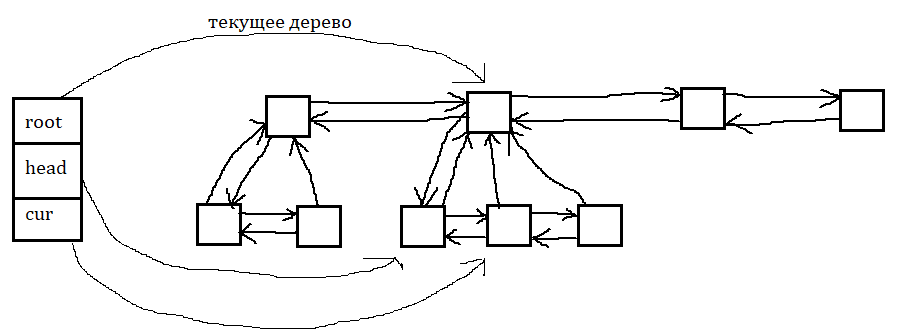


Движение вниз и влево – вправо.

\* нужна структура, которая управляет и следит за состоянием дерева

Удобно организовать объект, расширив линейный список. Пусть каждый элемент линейного списка может быть корневым и (ссылка вниз) ссылаться на другой список. Каждый элемент может ссылаться на элемент другого списка, который будет для него корневым (ссылка вверх). Таким образом, структура состоит из горизонтальных списков с вертикальными связями.

Также можно создать «коллекции» деревьев – лес:



Построим доступ для одиночного дерева.

atom {type, item, atom\* prev, atom\* next, atom\* up, atom\* down}

Далее используем именно эту структуру.

Структуру для отслеживания состояния дерева назовём tree:

tree {list c{…}, atom\* root}

Операции над деревом на основе раннее определения для списка. Изменение будет в новых операциях движения (up, down, встать в корень ~ root, добавить ветвь ~ add branch)

tcreate (tree t) // если система выделила память  
{  
 create(t)  
 t.root = 0  
}

atom\* tnext (tree t) // движение вправо  
{  
 if (t.root = 0) then return 0  
 return next(t)  
}

atom\* tprev (tree t)  
{  
 if (t.root = 0) then return 0  
 return prev(t)  
}

atom\* thead (tree t)  
{  
 if (t.root = 0) then return 0  
 return head(t)  
}

atom\* tlast (tree t)  
{  
 if (t.root = 0) then return 0  
 return last(t)  
}

atom\* tadd (tree t, atom\* a) // добавить потомка в горизонт-м напр-ии  
{  
 if (t.root = 0) then  
 {   
 t.root = a  
 a->up = 0  
 }  
 else  
 a->up = t.cur->a  
 a->down = 0  
 return add(t, a)  
}

atom\* addhead (tree t, atom\* a)   
{  
 if (t.root = 0) then  
 {  
 t.root = a  
 a->up = null  
 }  
 else  
 {  
 a->up = t.head->up  
 if t.head->up != 0) then  
 t.head->up->down = a  
 }  
 a->down = 0  
 return addhead(t, a)  
}

atom\* up (tree t)  
{  
 if (t.root = 0) then return 0  
 if (t.cur->up = 0) then return 0  
 t.cur = t.cur->up  
 t.head = t.cur  
 while (t.head->prev != 0)  
 t.head = t.head->prev  
 return t.cur  
}

atom\* down (tree t)  
{  
 if (t.root = 0) then return 0  
 if (t.cur->down = 0) then return 0  
 t.cur = t.cur->down  
 t.head = t.cur  
 return t.cur  
}

atom\* root (tree t)  
{  
 if (t.root = 0) then return 0  
 if (t.root = t.cur) then return 0  
 t.cur = t.head = t.root  
 return t.cur  
}

**Поиск элементов множества. Поиск по ключу**

Частая практическая задача. Условие поиска можно сформулировать по-разному, в зависимости от содержимого. Если ищется структура, то её ищут по ключу, удовлетворяющему некоторым условиям или являющемуся частью этой структуры. Поиск часто требуется при работе с Базами Данных (БД).

Ключевое поле – индекс БД. А сортировка по ним называется переиндексацией. Сортировка БД…

Множество элементов представимо разными структурами:

1. Массив (непрерывный кусок памяти)
2. Ссылки элементов друг на друга, то есть список (несмежные участки памяти)

Обычно говорят, что множество имеет либо 1 – непрерывную реализацию, либо 2 – ссылочную.

На базе обоих можно представить все основные структуры:

1. Неупорядоченное и упорядоченное множество
2. Стек, дек, очередь
3. Деревья

Поиск по ключу или по соответствию ключа некоторому логическому условию. Конечная цель – указать адрес элемента или оповестить о его отсутствии.

**Массовые и одиночные операции доступа**

Все операции над программным объектом, представляющим множество в том или ином виде, либо массовые, либо одиночные.

одиночная 🡪 к одному элементу или к очень малому подмножеству

массовая 🡪 просмотр более одного элемента множества, и количество зависит от размера множества

Часто одна и та же операция может оказаться и той, и той в зависимости от реализации структур. Массовая операция, очевидно, потребует больше времени, поэтому предпочтительнее использовать одиночные. В случае использования массовой операции можно попытаться оптимизировать её.

Удачи, читатель, ты обязательно всё сдашь!

Упорядоченность элементов множества часто влияет на массовость операции. Вставка одного элемента в неупорядоченное множество всегда является одиночной (вставляется в конец списка). А вот при соблюдении порядка и добавления в середину нужен сдвиг (при непрерывной реализации). В ссылочной же это одиночная операция, так как нужно лишь скорректировать ссылки. В непрерывной реализации одиночной является операция доступа. В списке – наоборот (как у итераторов проход по ссылкам).

У специальных видов списков (дек, очередь, стек) добавление элемента – одиночная операция, так как вес строго фиксирован в обеих (непрерывной (кольцевой буфер) и ссылочной) реализациях.

**Операции поиска**

1. Определить положение элемента
2. Выбрать его (взять, достать)

В стеке / очереди / деке место выбора (взятие элемента) фиксировано, то есть операция поиска одиночна.

Для упорядоченных и неупорядоченных множеств, то массово. Число просмотров ~ числу элементов, в худшем случае равняется ему, то есть в среднем: n/2 просмотров.

**Бинарный (двоичный) поиск**

- метод деления диапазона напополам

Имеет сложность O(log2N) в отл. O(N/2)

A(n), a1 = a [n/2]

Пусть k – искомый ключ. Встаём в середину a1 [n/2]

Если k = a1, значит ключ найден.

Если k < a1, то ищем левее, то есть от [1] до [n/2 – 1], иначе – правее.

Действия повторяются рекурсивно, пока элемент не будет найден, либо пока не кончатся индексы (пока L < v).

Прототип BinSearch:

index find(k, a(n))

{

l = 0  
 r = n  
 while (l < r)  
 {  
 m = (l + r) / 2  
 if (a[m] < k) then l = m + 1  
 else r = m  
 if (a[m] = k) then break  
 }  
 return m  
 }

Существует возможность для упорядоченной и неупорядоченной операции поиска, когда число различных значений невелико. Тогда множество элементов можно будет представить массивом из 0 и 1. Если a[x] = 1, то элемент есть.

Этот способ называется битовой реализацией множества.

# **ПРИМЕНЕНИЕ ПОСТРОЕНИЯ ОБЪЕКТОВ**

Стек = «стакан»

Доступ LIFO (Last Input, First Output)

Стек может быть представлен списком; его особенность – обработка элементов в строго определённом порядке (сверху стека).

Stack {list c {…}, size}

Операции работы со стеком:

1. Создать (newstack)

2. Добавить элемент.

3. Удалить.

4. Очистить полностью.

5. Узнать размер.

1. newstack (stack S) {create (S), s.size=0}

2. push (stack S, atom\* a) {addhead (S, a), s.size++}

3. atom\* pop (stack S) {head (S), s.clip=s.head, cut(S), s.size--, ret s.clip}

4. clear (stack S) {while (s.size≠0) {det(S), s.size--}}

5. int size (stack S) {ret s.size}

Вычисление выражений с помощью стека.

- при синтаксическом разборе формул;

- при вычислении по формулам;

- передача списков параметров;

– математически;

(а-в)/(а+в) – инфиксная форма

/(а-в)(а+в) – префиксная форма

/ -ав +ав – бесскобочная

(ав-)(ав+)/ - постфиксная скобочная

ав- ав+ / - постфиксная бесскобочная

Последняя форма может использоваться без (), т.к. не нарушает иерархии операций.

Постфиксная форма упорядочивает последовательность элементов, т.е. список.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Движение | а | в | - |  |  | а | в | + |  |  | / |  |  | Рез |
| Вычисление |  |  | -в | -ва | а-в |  |  | +в | +ва | а+в | /  (а+в) | /  (а-в)  (а+в) |  |  |
| Стек | а | в  а | а |  | а-в | а  а-в | в  а  а-в | а  а-в | а-в | а+в  а-в | а-в |  | а+в /  а-в | СТОП |

Программа- прототип для вычислений.

Float calc(list l)

{

stack S; newstack(S);

Push(S, head(l));

While(next(l) < > 0)

{

if (l.cur->type <> symbol) then

push(S, l.cur);

else

push(S, operation(\*l.cur->item, pop(S)));

}

ret\* (pop(S)->item)

}

Atom\* operation (symbol, atom\*, atom\*)

Этот прототип может не работать из-за сложного модуля operation, который работает с памятью лучше, чтоб он был type oper (symbol, type, type).

Также проблема может быть в 1) push9S, head (l)), который может вернуть нуль, если мы уже в начале, а также может возникнуть утечка памяти: модуль operation кладёт элемент в стек, а потом элемент извлекается оттуда и повисает в памяти после вычислений.

Очередь – другой вид списка (FIFO)

В очереди добавление происходит с одной стороны, а удаление – с другой.

Определим очередь как структуру вида:

Queue {list c {…}, size}

1. создать очередь

Newqueue (queue q) {create (q), q.size=0}

2. Добавить

Put q (queue q, atom\* a) [last (q), add(q, a), q.size++}

3. Удалить

atom\* get q (queue q) {head (q), q.dip=q.cur, cut(q), q.size--, ret q.clip}

4. Очистить очередь

Clear (queue q) {while q.size<>0 {del(q), q.size--}

5. Размер очереди

Int size (queue q) {ret q.size}

Дек (DEQ) – double ended queue – очередь с двумя концами

Дек образован двумя стеками. Если добавляем слева – левый стек, очередь движется влево.

Deq {list c {…}, size}

1. Создать дек

New deq (deq d) {create (d), d.size=0}

2. Добавить слева

Pushl (deq d, atom\* a) {addhead (d, a), d.size++}

3. Добавит справа

Pushr (deq d, atom\* a) {last (d), add(d, a), size++}

4. Удалить слева

Atom\* popl (deq d) {head (d), d.clip=d.cur, cut(d), d.size--, ret d.clip}

5. Удаление справа

Atom\* popr (deq d) {last (d), d.clip=d.cur, cut(d), d.size--, ret d.clip}

6. Очистить

Clear (deq d) {while (d.size<>0) { del(d), d.size--}}

7. Размер

Int size (deq d) {ret d.size}

Если необходимо сделать pushl и pushr симметричными, то необходимо изменять (дорабатывать) список.

Физический доступ к файлам.

Последовательность похожа на одномерный массив, но отличие в том, что, (1) как правило, длина последовательности может меняться в процессе обр. и как правило, (2) велика (не помещается в памяти). (3) В памяти остаётся только часть последовательности, и нужный элемент может быть необходимо загрузить.

(4) Если последовательности нет в памяти, то она находится на внешнем носителе. (5) Для того, чтобы дойти до середины последовательности, нужно прочитать все предыдущие (как у ленты), т.е. строго последовательный доступ. (6) Часть реальных данных пользователя организована как последовательность, при этом их порядок важен (нужно рассматривать в фиксированном порядке).

Следствие (1) – (6):

1. Послед-ый доступ имеет существенные ограничения.

(7) Размер последовательности не зафиксирован. Добавление новых элементов – в конец. Вставка в середину – нестандартная операция. Это естественно для динамических структур, где ничего не нужно раздвигать, а вот для последовательности неестественно.

Файловый доступ осуществляется с помощью последовательностей.

Рассмотрим базовые операции с последовательностями:

1. Открыть – создать структуру, которая отслеживает текущее состояние последовательности.

2. Закрыть – удалить её.

3. Записать элемент.

4. Прочитать элемент (движение вперёд по очереди)

5. «Перемотать» в начало.

Common sequence -> cseq

Cseq {pos(позиция), len(длина), eof(конец ф), seq(буфер)}

Open s (cseq n) {n.pos=0, n.len=0, n.eof=0}

2. close s (cseq n) {}

3. write s (cseq n, item) // запись в текущую позицию и изменение длины

{n.seq=item, put(n.seq), n.pos++, n.len=n.pos}

4. read s (cseq n, item) // чтение из текущей позиции и сдвиг вправо

{get (n, seq), if (n.eof=0) then {item=n.seq, n.pos++}}

5. reset s (cseq n) {n.pos=0, n.eof=0}

Такая реализация обладает существенными недостатками. Неясно, как работать с существующими последовательностями. Сперва нужно подряд записать элементы, и только после этого подряд читать. Обращение к конкретным элементам замедляет работу.

При реализации методов доступа используются программно-аппаратные функции get, put, предоставляемые в ПО. Их вызов синхронизирует состояние оперативной памяти и носителя. Такая система уже позволяет написать прогр. Обработки вида:

Cseq a;

Opens (a);

….

For (i=1, i<=100, i++)

Writes (a, i);

Resets (a);

While not (a eof)

{reads (a, i), ….}

….

Close (a)

Для произвольного доступа нужно реализовать псевдопрямой доступ, его имитацию со следующими свойствами:

1. Обращение к произвольным элементам.

2. Выполнять чтение / запись в произвольном порядке.

3. Делать это быстро.

Нужен новый набор операций и новая структура:

Dseq {cseq {…}}

1. Открыть

Opend (dseq n) {opens (n)}

2. Закрыть

Closed (dseq n) {closes (n)}

3. Запись putd

4. get

3. запись в текущую позицию и нумерация с нуля.

put d(dseq n, item, pos)

{

resets(n)

while (n.pos < pos)

{

if not (n.eof) then reads(n, cur)

else

write (n, o) // средство удлинение позиции

}

n.seq = item;

put(n.seq)

}

При записи за пределами последовательности она удлиняется. Это средство орг-ии за одно обращение. Очевидно, такой «прямой» доступ весьма медленен.

Можно придумать разные наборы функции, реализующие движение по носителю:

Forward – вперёд и backward – назад.

Пусть вызов функции физического уровня get() и getl() без аргументов выполняет перемещение. Тогда функция перемотки вперёд:

Forward (dseq n) {getl(); if (not n.eof) then n.pos++}

Один вызов – один шаг

Backward (dseq n) {getl(); if (not n.eof) then n.pos--}

Функция прямого вывода и прямого чтения на носитель.

putd(dseq n, item, pos)

{

if pos < 0 then error

while (n.pos < pos && not n.eof)

forward(n)

while (n.pos>pos && not n.eof)

backward(n)

while (n.pos<pos)

writes(n, 0) // удлинение данной последовательности

n.seq=item // запись в поле

put(n.seq) // переселыаем на носитель

}

Получить (getd)

getd(dseq n, item, pos)

{

if pos < 0 then error

while (n.pos < pos && not n.eof)

forward(n)

while (n.pos > pos && not n.eof)

backward(n)

get(n.seq)

if n.eof then

item =n.eof

else item=n.seq

}

Необходимо ускорить процесс доступа. Здесь поможет буферизованная последовательность. Т.к. носитель сильно медленнее, к нему нужно обращаться реже и не брать по одному элементу, нужно считывать целый буфер.

Буфер хранит часть последовательности.

Усовершенствуем структуру последовательности.

Dbseq – буферизованная последовательность

Dbseq {cseq{…}, buf (size), update, first, last}

Снова рассмотрим базовые операции:

1. Открыть opendb

2. Закрыть closedb

3. Вывести putdb

4. Прочитать getdb

5. reset db

Нам потребуется функция сохранения (saveb) и чтения (loadb) буфера.

Чтобы размер буфера не выходил за пределы последовательности, нужно перепозиционировать его. Иначе он будет заполнен частично.

save b(dbseq n)

{

if n.last> n.len-1 then

n.last = n.len-1

if n.first<0 then

n.first=0

while (n.pos<n.first & not n.eof)

forward(n)

while (n.pos > n.first & not n.eof)

backward(n)

if (n.eof) then error

while (n.pos <= n.last)

{

n.seq = n.buf[n.pos - n.first]

put(n.seq), n.pos++

}

n.update=0

}

load b(dbseq n)

{

if n.last> n.len-1 then

n.last = n.len-1

if n.first<0 then

n.first=0

while (n.pos<n.first & not n.eof)

forward(n) // идём вправо

while (n.pos > n.first & not n.eof)

backward(n) // идём влево

if (n.eof) then error

while (n.pos <= n.last)

{

get(n.seq)

n.buf[n.pos - n.first] = n.seq

n.pos++

}

}

putdb(dbseq n, item, pos)

{

if pos<0 then error

if (pos<=n.last & pos>=n.first) then

{

n.buf[pos-n.first]=item,

n.update=1,

ret

}

if (n.update<>0) then saveb(n) // сохранить

if (pos<n.first) then

{

while (pos < n.first)

{

n.last=n.first-1

n.first=n.last-n.size

loadb (n)

}

n.buf[pos-n.first]=item

n.update=1, ret

}

if (pos>n.last) the

{

while (pos > n.last)

{

n.first=n.last+1

n.last=n.first+n.size

loadb (n)

}

n.buf[pos-n.first]=item

n.update=1, ret

}

}

Функция get аналогична, только пересылка в обратном направлении

getdb (dbseq n, item, pos) {…}

Теперь нужна функция открытия. При открытии последовательности мы можем изменить уже существующий объект, поэтому надо его указать или взять размер с носителя. Нужно знать длину.

open db(dbseq n, len)

{

opens(n), n.update=0, n.first=n.last=-1,

n.len=len

}

Закрытие

close db(dbseq n)

{

if n.update<>0 then saveb(n),

closes(n)

}

Перемотка:

reset db(dbseq n)

{

if n.update<>0 then saveb(n),

resets(n)

n.first=n.last=-1

}

Если открыт существующий файл, то мы можем выполнить буфериз-ый ввод и вывод. Если же файл (последовательность) пуст, нужно инициализировать нулями.

Программа обработки для подсчёта среднего в скользящем окне.

dbseq a

opendb(a, 0)

putd(a, 0, 1000)

resetdb (a)

for (i = 0, i < 1000, i++)

{

input(item)

putdb(a, item, i)

}

closedb(a)

opendb(a, 1000)

sum=0

for (i = 999, i >= 0, i--)

{

getdb(a, item, i), sum+=item

}

sum=sum/1000,

output(sum)

closedb(a)

## **Программа поиска с однократными включениями**

(ключ в бинарном дереве не повторяется)

pos search (key, btree t)

{

atom \*a;

if (t.root = null)

{

a = new atom,

a->item = new type {key, pos, LR}

\*(a->item) = {key, 0, none};

insert(t,a);

ret t.size

}

else

{

rootb(l);

if (key = t.cur->item->key)

ret t.cur->item->pos;

while

{

if (key > t.cur->item->key)

{

if (downl(t) = null) break;

}

else if (key > t.cur->item->key)

{

if (downr(t) = null) break;

}

} //не было ret, значит элемент не найден, добавим его

a = new atom;

a-> item = new type{key, pos, LP};

\*(a->item) = {key, 0, none};

if (key < t.cur->item->key)

{

insertl(t,a);

}

else if (key > t.cur->item->key)

{

insertr(t,a);

}

ret t.size;

}

Эта программа последовательно достраивает дерево поиска при добавлении нового элемента. После этого возможно выполнить поиск. Фактически выполняется сортировка элементов множества, и упорядоченная выборка может быть выполнена **слева направо**, но она НЕ является полноценной программой поиска, поскольку ключ не может повторяться. Но можно обобщить её и для случая с повторениями.

## **Добавление в дерево повторяющихся ключей (сортировка бинарного дерева)**

(примем, что повторения добавляются в правое поддерево)

sort btree (btree t, s(n))

{

for (newpos = 1, n, step = 1)

{

if (pos != newpos) //взят повторный ключ

{

key = s[newpos-1];

a = new atom;

a->item = new type {key, pos, LR};

\*(a->item) = {key, newpos, R};

if (downr(t) = 0)

{

insertr(t, a);

}

}

else //есть правое поддерево

{

while (downr(t) != null)

{

if (t.cur->item->key != key) break;

t.clip = t.cur; //подвесили дерево

a = new atom;

a->item = new type {key, pos, LR};

\*(a->item) = {key, newpos, R};

t.add(t,a);

t.prev(t);

t.cut(t);

atom\* b = t.clip->down;

insertr(t,t.clip);

t.cur->down = b; //ссылка вниз

}

}

}

}

Эта программа имеет недостаток: она создаёт третью ссылку, хотя дерево бинарное.

## **Исключение вершин из бинарного дерева**

Данная задача сложнее добавления, поскольку при удалении вершин из середины дерева может нарушиться сама структура дерева. Возможны следующие ситуации:

1. Удалять нечего (ключа нет в дереве)
2. Вершина терминальная (т.е. не имеет потомков), удалить легко



1. У вершины есть 1 потомок (вершина заменяется своим потомком)



[ Снизу рисунок случая повтора ключа ]



1. Вершина имеет больше 1 потомка:

А) Удаляемый эл. заменяется на самый **правый** элемент **левого** поддерева



Б) Удаляемый эл. заменяется на самый **левый** элемент **правого** поддерева (в частности, при повторе ключа)



[ Снизу ситуация А) для случая повтора ключа ]



[ Снизу ситуация Б ) для случая повтора ключа ]



Возьмём за основу вариант Б. Тогда можно написать следующую программу для удаления узла: