|  |  |
| --- | --- |
| С простейшим наследованием типа переменных программисты стал-киваются уже при формировании производных типов .  type  t=string[10];  tt=array of t ;  ранее описанный тип данного t наследуется типом tt.  Факт наследования полей данных и методов от предка фиксируется ссылкой в скобках после слова object на имя типа потомка.  Type  o1=object  A1,b1:real;  Procedure c1(…….) ;  Function ff (…..):char  End;  o2=object(o1)  A2:Boolean;  Procedure g2(……..);  End;  Var  ob1:o1;ob2:o2; | Имена полей данных у предка и потомка должны быть различными и описываться перед объявлением методов. Имена методов у предка и потомка могут совпадать. Существует несколько типов наследования полей методов:  Простое наследование, когда имена методов у предка и потомка не совпадают.  Наследование с переопределением методов, когда потомок использует метод с именем метода предка, то есть, переопределяет метод, чтобы иметь возможность работать именно со своим методом, не меняя описание методов предка.  Наследование при использовании виртуальных методов. |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Сложность: O(n)  Усовершенствованный вариант сортировки вставками. Сравниваются не только элементы, стоящие рядом, но и элементы, находящиеся на определённом расстоянии друг от друга.  Весь массив разбивается на несколько подмассивов, которые составляются из элементов основного массива, выбираемых с заданным шагом по индексу.  Все подмассивы сортируются независимо друг от друга одним из методов.  Частично отсортированный исходный массив снова разбивается на подмассивы уже с меньшим шагом разбиения по индексу и снова все массивы сортируются независимо.  Возьмем массив A= \{ 56, 43, 12, 78, 42, 93, 16, 55 \} и смещения предложенные Шеллом. Разбили массив на 4 списка. | |  |  | | --- | --- | | |  | | --- | | Отсортировали элементы списков сортировкой вставками. Количество обменов 2. |   Объединили списки в массив. Уменьшаем i на 1. i \geqslant 0, перейдем к шагу 1. | | Разбили массив на 2 списка. | | Отсортировали элементы списков сортировкой вставками. Количество обменов 4. | | Объединили списки в массив. Уменьшаем i на 1. i \geqslant 0, перейдем к шагу 1. | | Разбили массив на 1 список. | | Отсортировали элементы списков сортировкой вставками. Количество обменов 7. | | Объединили списки в массив. Уменьшаем i на 1. i<0. | |

|  |  |
| --- | --- |
| Стандартный Pascal обрабатывает процедуры и функции только как части программы, которые могут быть выполнены через обращение к процедуре или функции.  Borland Pascal имеет намного более широкую область применения процедур и функций. С помощью процедурных типов можно расценивать процедуры и функции как объекты, которые могут быть назначены переменным и переданы как параметры.  Объявление процедурного типа содержит параметры и, для функции, тип результата.  Синтаксис объявления процедурного типа идентичен заголовку процедуры или функции, за исключением того, что можно опускать идентификаторы после ключевого слова Procedure или Function. | Имена параметров в объявлении процедурного типа не влияют на значение объявления.  Borland Pascal не позволяет вам объявлять функции, которые возвращают значения процедурного типа.  Результат функции должен иметь тип String, Real, Integer, Char, Boolean, Pointer или определенный пользователем перечислимый тип.  Пример  Type Proc = Procedure;  SwapProc = Procedure(Var X, Y : Integer);  StrProc = Procedure(S : String);  MathFunc = Function(X : Real) : Real;  DeviceFunc = Function(Var F : Text) : Integer;  MaxFunc = Function(A, B : Real; F : MathFunc) : Real; |

|  |  |
| --- | --- |
| Сложность: O(n log n)  Алгоритм пирамидальной сортировки примечателен тем, что независимо от набора данных у него одна и та же сложность по времени – O(n log n).  Пирамидальная сортировка использует бинарное сортирующее дерево. Сортирующее дерево — это такое дерево, у которого выполнены условия: Каждый лист имеет глубину либо d, либо d-1, где d — максимальная глубина дерева. Значение в любой вершине не меньше (другой вариант — не больше) значения её потомков.  Достоинства  Имеет доказанную оценку худшего случая O(n log n).  Сортирует на месте, то есть требует всего O(1) дополнительной памяти (если дерево | организовывать так, как показано выше).  Недостатки  Сложен в реализации.  Неустойчив — для обеспечения устойчивости нужно расширять ключ.  На почти отсортированных массивах работает столь же долго, как и на хаотических данных.  На одном шаге выборку приходится делать хаотично по всей длине массива — поэтому алгоритм плохо сочетается с кэшированием и подкачкой памяти.  Методу требуется «мгновенный» прямой доступ; не работает на связанных списках и других структурах памяти последовательного доступа. Из-за сложности алгоритма выигрыш получается только на больших n. На небольших n (до нескольких тысяч) сортировка Шелла быстрее. |

|  |  |
| --- | --- |
| Идея сортировки файлов основана на понятии “отрезок” или “интер- вал”. Это такая последовательность элементов сортируемого файла, кото- рая уже отсортирована в соответствии с условиями задачи. Предположим , что требуется отсортировать по возрастанию последовательность чисел , находящихся в файле: . Тогда отрезки, уже отсортированные так , как требуется по условию, подчеркнуты стрелками. Всего здесь 4 отрезка. Разбивка исходного мно- жества на упорядоченные части называется распределением. Сортировка файлов предполагает два процесса , реализуемых пооче- редно: распределение отрезков по двум файлам в шахматном порядке и так называемое слияние содержимого двух файлов в один. Сортировку файлов можно проводить с применением 3 или 4 файлов. | Пусть исходная информация находится в файле A, и используются дополнительные файлы B и C. Тогда процедура распределения содержимого исходного файла по двум вспомогательным даст следующий результат: В процессе слияния содержимого двух файлов B и C производится чтение пары отрезков из этих файлов в порядке возрастания значений чи- сел и их объединение в новый отрезок. Если у отрезка нет пары в другом файле , то он снова считывается целиком в файл A. В нашем случае в ре- зультате слияния содержимого двух файлов B и C получим в файле A два новых отрезка.56 Рис.1.8.1 Рис.1.8.2 Распределение по файлам B и C содержимого файла A даст: Рис.1.8.3 Последнее слияние даст : Рис.1.8.4 Таким образом , в файле A все элементы оказываются полностью от- сортированными. При слиянии двух отрезков надо читать числа из обоих файлов и запи- сывать наименьшее из них в результирующий файл. При распределении отрезков во двум другим файлам признаком окон- чания отрезка является прекращение роста читаемых элементов. Поэтому схема алгоритма не приводится, так как вышеизложенная идея очень про- ста |

|  |
| --- |
| Хеш-таблица — это [структура данных](https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), реализующая интерфейс [ассоциативного массива](https://ru.wikipedia.org/wiki/%D0%90%D1%81%D1%81%D0%BE%D1%86%D0%B8%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D1%8B%D0%B9_%D0%BC%D0%B0%D1%81%D1%81%D0%B8%D0%B2), а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу.  В общем случае хеш-таблица позволяет организовать массив, специфика которого проявляется в связанности индексов по отношению к хеш-функции; индексы могут быть не только целого типа данных (как это было в простых массивах), но и любого другого, для которого вычислимы хеш-коды. Данные, хранящиеся в виде такой структуры, удобны в обработке: хеш-таблица позволяет за минимальное время O(1) выполнять операции поиска, вставки и удаления элементов. |

|  |  |  |  |
| --- | --- | --- | --- |
| Коллизий следует избегать, выбирая «хорошую» хеш-функцию, и используя один из методов разрешения конфликтов: открытое (метод цепочек) или закрытое (открытая адресация) хеширование.  Открытое хеширование (метод цепочек)  Принцип организации хеш-таблицы методом открытого хеширования заключается в реализации логически связанных цепочек, начинающихся в ячейках хеш-таблицы. Под цепочками подразумеваются связанные списки, указатели на которые хранятся в ячейках хеш-таблицы. Каждый элемент связанного списка – блок данных, и если с некоторым указателем, хранящимся по адресу i, связаны n таких блоков (n>1), то это свидетельствует о том, что n ключей получили один и тот же хеш-код i, т. е. имеет место коллизия. Но метод открытого хеширования устраняет конфликт, поскольку данные хранятся не в самой таблице, а в связных списках, которые увеличиваются при возникновении конфликта.  Закрытое хеширование (метод с открытой адресацией)  Первый метод назывался открытым, потому что он позволял хранить сколь угодно много элементов, а при закрытом хешировании их количество ограниченно размером хеш-таблицы. | В отличие от открытого хеширования закрытое не требует каких-либо дополнительных структур данных. В ячейках таблицы хранятся не указатели, а элементы исходного массива, доступ к каждому из которых осуществляется по хеш-коду ключа, при этом одна ячейка может содержать только один элемент.  Сам процесс заполнения хеш-таблицы с использованием алгоритма закрытого хеширования осуществляется следующим образом:  имеется изначально пустая хеш-таблица T размера M, массив A размера N (M≥N) и хеш-функция h(), пригодная для обработки ключей массива A;  элемент xi, ключ которого keyi, помещается в одну из ячеек хеш-таблицы, руководствуясь следующим правилом:  если h(keyi) – номер свободной ячейки таблицы T, то в последнюю записывается xi;  если h(keyi) – номер уже занятой ячейки таблицы T, то на занятость проверяется другая ячейка, если она свободна то xi заноситься в нее, иначе вновь проверяется другая ячейка, и так до тех пор, пока не найдется свободная или окажется, что все M ячеек таблицы заполнены. | метод деления. Некоторый целый ключ делится на размер таблицы и остаток от деления берется в качестве значения хеш-функции. Эта хеш-функция обозначается h (key) := key mod m.  метод середины квадрата. Ключ умножается сам на себя и в качестве индекса используется несколько средних цифр этого квадрата.  Аддитивный метод для строк (размер таблицы равен 256). Для строк вполне разумные результаты дает сложение всех символов и возврат остатка от деления на 256.  Исключающее ИЛИ для строк (размер таблицы равен 256). Этот метод аналогичен аддитивному, но успешно различает схожие слова и анаграммы (аддитивный метод даст одно значение для XY и YX). Метод заключается в том, что к элементам строки последовательно применяется операция "исключающее или". В алгоритме добавляется случайная компонента, чтобы еще улучшить результат. | Полиморфизм позволяет по-разному использовать методы с одинаковыми именами в иерархии типов объектов.  Многообразие способов исполнять методы с одинаковыми именами проявляется в возможности переопределения статических методов в иерархии типов, в использовании виртуальных методов и в возможности вызова метода предка из потомка.  1. Поиск ключа в хеш-таблице с цепочками. 1.1. Вычисляем хеш ключа, находим соответствующую ячейку. Для открытого хеширования ячейка это указатель на (односвязный) список пар "ключ — значение". Ячеек в массиве столько, сколько значений выдаёт хеш-функция. 1.2. Проходим по списку в поисках элемента с совпадающим ключом. 1.3. Нашли — элемент есть. Прошли весь список и не нашли — такого ключа в хеше нет.  2. Вставка ключа в хеш-таблицу с цепочками. 2.1. Проверяем наличие ключа в таблице как в пункте 1. 2.2. Если ключа нет, то приписываем новый элемент в начало списка соответствующей ячейки. Если ключ есть, то варианты: 1) ругаемся; 2) записываем в элемент новое значение. |

|  |  |  |  |
| --- | --- | --- | --- |
| 1.Критерий покрытия операторов, т.е. набор тестов должен обеспечивать исполнение всех операторов . С этой точки зрения дос-таточно проверить ситуацию A=true и B=true, при которой исполняется и логический оператор IF (символ “Решение”), и оператор при-своения x:=y. Ясно, что при этом даже не все пути проверяются.  2. Критерий покрытия решений (или переходов).  При этом необходимо пройти обе ветви всех символов “Решение” . С этой точки зрения достаточной является проверка ситуации A=true, B=true и A=true, B=false. При этом возможны и другие альтернативные, но тоже достаточные варианты значений A и B. В данном примере вариант A=true,B=false и A=false,B=true также решает задачу проверки всех пере-ходов.  3 Критерий покрытия условий.  При этом логические условия A и B должны принять различные воз-можные значения (в наборе тестов и A и B должны принимать значения и true и false). Нетрудно видеть, что тест A=true, B=false и A=false, B=true удовлетворяют этому требованию, однако при этом даже не все “решения” покрыты, т.е. этот критерий может оказаться слабее предыдущего крите-рия покрытия переходов. | .  4.Покрытие решений и условий. Это более сильный критерий , чем 2 и 3. Например, ситуация A=true,B=true и A=false ,B=false вместе покры-вают все условия и все переходы.  Однако все предыдущие критерии не способны обеспечить гарантии прохождения всех путей алгоритма. Поэтому самым сильным критерием является так называемое комбинаторное покрытие или покрытие всех путей.  5.Комбинаторное покрытие условий. Оно обеспечивает выполнение требований всех предыдущих критериев. Чтобы убедиться в этом, развернем более детально предыдущий алгоритм в виде, приведенным ниже.  Этот алгоритм учитывает всевозможные значения переменных A и B и их различные сочетания. Всего таких сочетаний 4 (A может быть true и false , и B может также принимать два возможных значения). Поэтому, не-смотря на идентичность путей abdh и abeh , все же число условий будет равно 4 , что эквивалентно четырем различным путям : abdh, abeh, acfh, acgh. Задавая различные комбинации значений A и B , получим возмож-ность проверки всех путей алгоритма , тем самым проходя все операторы  **Белый ящик** | 1.Если входные данные по спецификации описывают область значе- ний исходных данных , например, 100 <= X<=200, то указанный диапазон представляет один правильный класс и два неправильных X200. 2.Если в спецификации оговорено конкретное множество значений входных данных, с которыми должна работать программа, то существует один правильный класс и один неправильный , состоящий из элементов, не принадлежащих заданному множеству. 3. Если есть основание полагать, что различные элементы класса мо- гут по-разному обрабатываться программой, то класс разбивается на под- классы, и эти элементы рассматриваются как принадлежащие к разным подклассам. В задаче с треугольником , если тест на L1=L2=L3=5 не обнаружил ошибки , то вряд ли разумно проверять программу на данных L1=L2=L3=2, так как эти данные принадлежат одному классу эквивалент- ности, в котором длины всех сторон равны и положительны. | Случай же L1=L2=L3=-6 принципиально отличается от предыдущего, так как эти данные относятся к неправильному классу, в этом случае выдача сообще- ния , что “ это равносторонний треугольник” явилось бы свидетельством ошибки в программе. Особо тщательно должны рассматриваться тесты , принадлежащие границам классов эквивалентности. Именно здесь и бывает основная часть ошибок. Например, в задаче о треугольнике , рассмотренной выше, в обяза- тельном порядке надо проверить правильность работы программы при 166 L1=L2=L3=0, поскольку у треугольника не может быть нулевой длины стороны. Точно так же при формировании линейного списка из элементов файла необходимо проверять ситуацию, когда в файле находится всего один элемент или файл пуст  **Черный ящик** |
|  |  |  |  |

|  |  |
| --- | --- |
| **Достоинства** тестирования «белого ящика» связаны с тем, что принцип «белого ящика» позволяет учесть особенности программных ошибок:  1. Количество ошибок минимально в «центре» и максимально на «периферии» программы.  2. Предварительные предположения о вероятности потока управления или данных в программе часто бывают некорректны. В результате типовым может стать маршрут, модель вычислений по которому проработана слабо.  3. При записи алгоритма ПО в виде текста на языке программирования возможно внесение типовых ошибок трансляции (синтаксических и семантических).  4. Некоторые результаты в программе зависят не от исходных данных, а от внутренних состояний программы.  Каждая из этих причин является аргументом для проведения тестирования по принципу «белого ящика». Тесты «черного ящика» не смогут реагировать на ошибки таких типов.  **Недостатки** тестирования «белого ящика»:  1. Количество независимых маршрутов может быть очень велико.  2. Исчерпывающее тестирование маршрутов не гарантирует соответствия программы исходным требованиям к ней.  3. В программе могут быть пропущены некоторые маршруты.  4. Нельзя обнаружить ошибки, появление которых зависит от обрабатываемых данных (это ошибки, обусловленные выражениями типа if abs (a-b) < eps..., if(a+b+c)/3=a...). | **Преимущества** тестирования “чёрного ящика”:  1. Тестирование с точки зрения пользователя  2. Не требует специальных знаний (например, конкретного языка программирования)  3. Позволяет найти проблемы в спецификациях  4. Можно создавать тесты параллельно с кодом  5. Тестировщик может быть отделен от разработчиков  **Недостатки** тестирования “чёрного ящика”:  1. Эффективность зависит от выбора конкретных тестовых значений  2. Необходимость наличия четких и полных спецификаций  3. Невозможность сконцентрироваться на особо сложных частях кода  4. Трудность локализации причины дефекта  5. Возможность не протестировать часть кода |

|  |  |
| --- | --- |
| Program ob35;{инициализация полей данных}  uses max\_crt;  type  t1=object  a1:integer;  a2:char;  procedure p1(pa1:integer;pa2:char);  Constructor Init(pa1:integer;pa2:char);  procedure pout; virtual;  end;  t2=object(t1)  c: string;  procedure p3(pa3:string);  procedure pout; virtual;  end;  procedure t1.p1(pa1:integer;pa2:char);  begin  a1:=pa1;a2:=pa2;  end;  Constructor t1.Init(pa1:integer;pa2:char);  Begin  a1:=pa1;a2:=pa2;  end; procedure t1.pout;  Begin  Writeln ('t1.pout',' a1=',a1,' a2=',a2);  end;  procedure t2.p3(pa3:string);  begin  c:=pa3  end;  procedure t2.pout;  Begin  writeln('t2.pout',' a1=',a1,' a2=',a2,' c=',c);  end;  var  o1:t1;o2:t2;  begin  clrscr;  o1.Init(5,'f');{инициализация с помощью конструктора}  o1.pout; {вывод своей виртуальной процедурой}  o2.Init(6,'g');{запуск конструктора, наследуемого от объекта o1} | o2.p1(7,'h');{повторная инициализация полей o2.a1 и o2.a2}  o2.p3('\*\*\*'); {инициализация поля o2.c}  o2.pout; {вывод полей объекта o2}  o1.a1:=100;  writeln('o1.a1=',o1.a1)  end.  Отметим важные особенности этого текста.  1. У потомка отсутствует свой конструктор , так как он наследует его от предка .  2. Инициализацию данного можно проводить как с помощью метода t1.p1, так и с помощью конструктора , который можно наследовать и обращаться к нему из потомка.  3. Без запуска конструкторов обоих объектов нельзя обращаться к их виртуальным методам pout.  4. Инициализация полей o2.a1 и o2.a2 проводится дважды (первый раз с помощью конструктора предка, а второй раз с помощью своего метода p1), поэтому на экране не увидим значений 6 и ‘g’.  5. В последних двух операторах программы поле o1.a1 инициализируется непосредственно присвоением, и при выводе происходит непосредственное обращение к этому полю , как в случае **record.**  Однако **идеология объектно-ориентированного программирования предполагает все действия по вводу и выводу значений полей данных осуществлять с помощью собственных методов**. В результате исполнения этой программы на экран будут выведены строки:  t1.pout a1=5 a2=f  t2.pout a1=7 a2=h c=\*\*\*  O1.a1=100 |

|  |  |
| --- | --- |
| Program ob36;{инициализация полей данных внешней процедурой}  uses max\_crt;  type  t1=object  a1:integer;  a2:char;  Constructor Init;  procedure pout;virtual;{для вывода своих полей данных}  end;  t2=object(t1)  c:string;  {процедура p3 обеспечивает инициализацию данных этого типа объектов}  procedure p3(pa1:integer;pa2:char;pa3:string);  procedure pout;virtual;{для вывода своих полей данных}  end;  Constructor t1.Init;  Begin  end;    procedure t1.pout;  Begin  writeln('t1.pout',' a1=',a1,' a2=',a2);  end;  procedure t2.p3(pa1:integer;pa2:char;pa3:string);  begina1:=pa1;a2:=pa2; c:=pa3  end;  procedure t2.pout;  Begin  writeln('t2.pout',' a1=',a1,' a2=',a2,' c=',c);  end;  {начало внешней процедуры инициализации полей объектов типа t1}  procedure p1(var ob:t1; pa1:integer; pa2:char);  begin  with ob do  begin  a1:=pa1;a2:=pa2;  end;  end; {конец внешней процедуры инициализации полей объектов типа t1} | var  o:array[1..3] of t1;o2:t2;i:integer;  begin  clrscr;  For i:=1 to 3 do  p1(o[i],I,chr(64+i));{инициализация полей массива объектов}  o2.p3(2,'D','???'); {инициализация полей объекта o2}  For i:=1 to 3 do  begin  o[i].Init;{запуск конструкторов массива объектов}  o[i].pout {вывод полей всех объектов массива}  end;  o2.Init; {запуск конструктора o2}  o2.pout;{вывод полей объекта o2}  end.  Особенности этой программы:  1. В ней используется массив из трех объектов типа t1 и один объект типа t2;  2. У этих двух типов есть виртуальные методы, поэтому должны быть и конструкторы, причем объект типа t2 наследует конструктор от типа t1.  Не запустив конструктор конкретного объекта, нельзя обратиться к его виртуальному методу вывода полей данных;  3. У внешней процедуры инициализации полей в объектах типа t1 сам тип объекта задается в качестве формального параметра **ob** , а при обращении к этой процедуре на месте первого параметра указывается фактическое имя инициализируемого объекта. Следовательно , переменные объектного типа ничем не отличаются от других типов с точки зрения их использования в качестве формальных и фактических параметров в процедурах и функциях.  При работе этой программы на экран выводятся строки:  t1.pout a1=1 a2=A  t1.pout a1=2 a2=B  t1.pout a1=3 a2=C  t2.pout a1=2 a2=D c=??? |