МИНОБРНАУКИ РОССИИ

–––––––—————–––—————–––––––

Санкт-Петербургский государственный
электротехнический университет «ЛЭТИ»

——————————————————

**АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ**

Методические указания к лабораторным работам,
практическим занятиям
и курсовому проектированию
Часть 2

Выпуск 1606

Санкт-Петербург
Издательство СПбГЭТУ «ЛЭТИ»
2016

УДК 004.424:004.422.63(075.8)

Алгоритмы и структуры данных: методические указания к лабораторным работам, практическим занятиям и курсовому проектированию. Ч. 2. Вып. 1606 / сост. П. Г. Колинько. — СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2016. — 48 с.: ил.

Описывается цикл лабораторных работ и практических занятий в компьютерном классе. Содержатся материалы для курсовой работы.

Предназначены для студентов бакалавриата по направлению 230100.62 «Информатика и вычислительная техника» дневной, очно-заочной и заочной форм обучения.

Утверждено
редакционно-издательским советом университета
в качестве методических указаний

© П. Г. Колинько, 2012–2016(0628)

© СПбГЭТУ «ЛЭТИ», 2014

# ВВЕДЕНИЕ

Цель методических указаний — завершение двухсеместрового курса «Алгоритмы и структуры данных». Методические указания содержат шесть разделов. Первые три раздела посвящены комбинированным структурам данных. В них рассматриваются способы размещения множеств в памяти ЭВМ, оптимизированные под различные задачи работы с ними. Изучаются алгоритмы эффективного выполнения двуместных операций над множествами в этих структурах данных, а также применение их для поддержки произвольных последовательностей.

Следующие три темы предусматривают полное знакомство с возможностями объектного программирования. На простом примере изучается иерархическая структура классов и механизм поддержки обработки особых ситуаций. Обучение завершается знакомством со стандартной библиотекой шаблонов.

Содержанием курсового проектирования является эксперимент по прямому измерению временной сложности алгоритма, построенного на использовании стандартной библиотеки шаблонов. Результат эксперимента позволит дать заключение об эффективности этой библиотеки.

В каждом разделе даны ссылки на литературу, которую следует проработать для изучения темы.

По каждой из тем и по курсовой работе должны быть представлены отчёты, содержащие название темы; текст индивидуального задания; тесты для проверки программ; результаты их работы; выводы, обязательно содержащие заключение о временной сложности использованных алгоритмов; список использованных источников; перечень приложений. Обязательное приложение: исходный текст отлаженной программы на машинном носителе. Допускается объединение нескольких тем в одном отчёте. Отчёт по теме 6 можно включить в пояснительную записку к курсовой работе.

Все примеры, имеющиеся в методических указаниях, проверены в оболочке Visual C++ 2012. Для обучения могут быть использованы любые программные оболочки с поддержкой С++, выпущенные не ранее 2005 г. Рекомендуется оболочка Visual C++ 2012 как поддерживающая стандарт С++11. Информация об этом стандарте имеется в Википедии. Нововведения подробно изложены в [10, c. 1049–1151]. Изучить программирование на С++ в новом стандарте можно также по [11]. С основами библиотеки STL можно ознакомиться по [7] и [15]. Типовые ошибки в программах на С++ обсуждаются в  [12], [13] и [14].

ОСОБЫЕ ТРЕБОВАНИЯ К ЗАОЧНИКАМ

Для заочников темы 1–3 факультативны, отчётности по ним нет. Материалы этих тем используются для подготовки отчётного задания по теме 6. Варианты индивидуальных заданий комбинируются из содержимого таблиц 1.1 и 3.1. Из табл. 1.1 берётся выражение с операциями над множествами, а из табл. 3.1 — набор операций над последовательностями, а также указание на выбор базового контейнера — для дерева двоичного поиска (set и т. п.) или для хеш-таблицы (unordered\_set). Из индивидуального набора операций с множествами и последовательностями составляется цепочка, для которой выполняется статистический эксперимент: измерение времени выполнения цепочки для изменяемых размеров множеств. Результаты измерений будут использованы в курсовой работе.

Отчёты по темам с текстами программ, составленных по индивидуальным заданиям, высылаются по адресу «clgn@mail.ru»:

— до восьмой недели семестра — по теме 4 «наследование и полиморфизм»;

— до двенадцатой недели — по теме 5 «исключительные ситуации»;

— до начала сессии — по теме 6 «стандартная библиотека шаблонов».

По этому же адресу можно получить консультации.

Обработка статистики, полученной в работе по теме 6, и оформление курсовой работы будет выполняться на занятиях во время сессии.

Вопросы к экзамену размещены в конце каждой темы и в приложении.

# 1. ХЕШ-ТАБЛИЦЫ

Хеш-таблица — это обобщение способа хранения множества целых чисел (ключей) в форме вектора битов на случай, когда мощность универсума *U* очень велика по отношению к мощности множеств, с которыми нужно работать. Функция отображения преобразует значения ключей к интервалу [0, *m* – 1], где *m* — размер хеш-таблицы, *m* ≪ |*U*|. Очевидно, что при этом каждому индексу хеш-таблицы будет соответствовать много различных значений ключей. Поэтому, во-первых, в хеш-таблице приходится хранить не биты, а сами значения ключей, а во-вторых, имеется возможность размещать в ней более одного ключа для каждого значения функции отображения (разрешать коллизии).

Количество возможных коллизий можно уменьшить, если выполнить два условия:

1) выбрать размер хеш-таблицы с запасом. Если размер таблицы превышает мощность хранимого множества более чем вдвое, вероятность коллизии становится меньше 0,5. Если мощность множества заранее неизвестна, то выбирают некоторый начальный размер, а когда его оказывается недостаточно, таблицу перестраивают с увеличением размера (обычно вдвое);

2) подобрать функцию отображения (хеш-функцию) такую, чтобы все ячейки таблицы были востребованы по возможности с равной вероятностью, независимо от того какое распределение имеют хранящиеся в таблице ключи.

По способу разрешения коллизий различают хеш-таблицы двух типов:

1) с открытой адресацией. Конфликтующие значения ключей размещаются в свободных ячейках таблицы;

2) с цепочками переполнения. Каждая ячейка таблицы содержит указатель на список конфликтующих ключей.

Подробнее о хеш-таблицах *см*. [1, с. 529–556], [2, с. 567–601], [3, с. 115–128], [4, с. 316–338].

Таблицы второго типа применяются чаще, потому что для них не существует проблемы переполнения. Если мощность хранимого множества становится слишком большой, таблица просто начинает работать как *m* списков. Если же таблица правильно построена и не переполнена, проверка принадлежности элемента множеству, а также вставка и удаление элемента выполняются в ней за постоянное время, примерно такое же, как и в массиве битов.

За постоянное время будут выполняться и двуместные операции над множествами: объединение, пересечение и разность: если хеш-функции для обоих множеств одинаковы, для этого пригоден такой же алгоритм попарного сравнения соответствующих ячеек, как и для массивов битов.

В хеш-таблице можно хранить и множество с повторениями: совпадающие значения ключей не создают никаких проблем, кроме гарантированных коллизий, которые разрешаются обычным образом.

К сожалению, всегда можно подобрать такие данные, что они все попадут в одну или несколько ячеек таблицы, образовав неупорядоченные списки (упорядочивание обычно не применяется). Это худший случай, для которого справедливы оценки временной сложности для списков: *O*(*n*) — для поиска и удаления элемента; *O*(*n*2) — для двуместной операции над множествами.

Подбор подходящей хеш-функции — в общем случае достаточно сложная задача. Но если ключи представляют собой целые числа (или сводятся к таковым), хорошие результаты можно получить с хеш-функцией вида

*h*(*x*) = (*a* \* *x* + *b*) % *m*,

где *m* – размер таблицы, а *a* и *b* — простые числа.

Обычно *a* выбирается близким по значению к *m*, а *b —* к 1. Так, при *m =*100 можно взять *a* = 97, *b* = 11. Такой выбор обеспечивает равномерное использование всех ячеек таблицы в большинстве практических случаев.

### 1.1. Практикум по теме

Составить и отладить программу для вычисления пятого множества по четырём заданным, представленным в форме хеш-таблиц. Элементы множеств — целые числа из интервала [0, *MAXINT*] (для удобства вывода на экран и увеличения вероятности появления общих элементов в множествах рекомендуется ограничиться интервалом [0, 100] или [0, 1000]). Формула для вычислений и средняя мощность множеств выбираются из табл. 1.1 в соответствии с номером индивидуального варианта. Подходящий размер хеш-таблиц определить по мощности обрабатываемых множеств.

Хеш-таблицу закодировать как класс, а операции с ней — как функции-члены этого класса. В качестве образца можно использовать программу для работы с множествами-объектами, составленную в предыдущем семестре.

Программа должна генерировать исходные данные и выводить на экран структуру всех хеш-таблиц таким образом, чтобы было видно распределение элементов множества по ячейкам.

Таблица 1.1

Индивидуальные задания к темам «Хеш-таблицы»
и «Деревья двоичного поиска»

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № вари-анта | Мощ­ностьмножества | Что надо вычислить | № вари-анта | Мощ­ностьмножества | Что надо вычислить |
| 1 | 10 | A ∩ B ∪ C ∪ (D ⊕ E) | 26 | 26 | A \ (B ∩ C ∩ D) ⊕ E |
| 2 | 26 | (A \ B \ C) ⊕ D ∪ E | 27 | 16 | (A ∪ B ⊕ C ∩ D) ∩ E |
| 3 | 16 | A ∩ B ⊕ C ∩ D ∩ E | 28 | 26 | A ⊕ (B ∪ C ∪ D) ∩ E |
| 4 | 26 | (A ⊕ B \ C) ∪ D ∩ E | 29 | 10 | (A \ B) ∪ C ⊕ D ∪ E |
| 5 | 10 | A ⊕ B \ (C ∪ D) \ E | 30 | 32 | A ∪ B ∩ C ∪ D ⊕ E |
| 6 | 32 | (A ∩ B) ⊕ C ∪ D \ E | 31 | 16 | (A ⊕ B) ∪ (C \ D) \ E |
| 7 | 16 | A ∪ B ∪ C ⊕ D ∩ E | 32 | 32 | (A ∪ B ∪ C) \ D ⊕ E |
| 8 | 26 | A \ (B ∩ C) ∪ D ⊕ E | 33 | 10 | (A ∩ B) ⊕ (C ∪ D) \ E |
| 9 | 10 | A ⊕ B ∩ C ∪ D ∪ E | 34 | 26 | A \ B \ (C ⊕ D) \ E |
| 10 | 26 | ((A ∪ B) \ C) ∩ D ⊕ E | 35 | 16 | (A ∪ B) \ (C ∪ D) ⊕ E |
| 11 | 16 | A ∪ B ⊕ C \ D \ E | 36 | 32 | (A ⊕ B ∩ C) \ D ∩ E |
| 12 | 26 | A ∪ B ∪ C ⊕ D \ E | 37 | 10 | A \ (B ∩ C ∪ D) ⊕ E |
| 13 | 10 | A ⊕ B \ C \ D ∩ E | 38 | 32 | (A ∪ B ⊕ C) \ D ∩ E |
| 14 | 32 | A \ B ⊕ (C \ D \ E) | 39 | 16 | A ∩ B ∪ C ∩ D ⊕ E |
| 15 | 16 | (A ∪ B) \ C \ (D ∪ E) | 40 | 52 | (A \ B) ∪ C ⊕ D ∩ E |
| 16 | 32 | (A ⊕ B ∩ C) \ D ∩ E | 41 | 10 | A ⊕ B ∪ C ∩ D ∪ E |
| 17 | 10 | A \ (B ∩ C) \ D ⊕ E | 42 | 66 | (A ∩ B) \ (C ∩ D) ⊕ E |
| 18 | 32 | A ∪ (B ⊕ C) \ D ∩ E | 43 | 16 | A \ (B ⊕ C ∩ D) ∪ E |
| 19 | 16 | A ∩ B ⊕ C ∩ D ∪ E | 44 | 52 | (A ∪ B) ⊕ (C ∩ D) \ E |
| 20 | 26 | (A \ B) ∪ C ∩ D ⊕ E | 45 | 10 | A ∩ B ∩ C ⊕ D ∩ E |
| 21 | 10 | A ⊕ B ∪ C ∩ D ∪ E | 46 | 26 | A \ (B ∩ C ∩ D) ⊕ E |
| 22 | 32 | (A ∩ B) \ (C ∩ D) ⊕ E | 47 | 16 | A ∪ B ⊕ C ∩ D ∩ E |
| 23 | 16 | A ⊕ B \ C ∩ D \ E | 48 | 26 | A ∩ (B ∪ C ∪ D) ⊕ E |
| 24 | 32 | A ∪ B ⊕ (C ∩ D \ E) | 49 | 10 | (A \ B) ⊕ C ∪ D ∪ E |
| 25 | 10 | A ∩ B ⊕ C ∩ D ∩ E | 50 | 40 | A ∪ B ∩ C ∪ D ⊕ E |

### 1.2. Требования к отчёту

Отчёт по теме составляется по стандартной форме. В него следует поместить обоснование по выбору размера хеш-таблицы и коэффициентов хеш-функции, а также оценку временной сложности алгоритма обработки множеств — в худшем случае и в среднем. В выводах постарайтесь дать ответ на контрольные вопросы.

### 1.3. Контрольные вопросы

1. Какой объём памяти нужно выделять под хеш-таблицу для хранения множеств со средней мощностью 50?

2. Хеш-таблица какого типа расходует больше памяти для хранения множества — с открытой адресацией или с цепочками переполнения?

3. Каким требованиям должна удовлетворять «хорошая» хеш-функция?

4. Можно ли построить хеш-таблицу, в которой не будет коллизий?

5. Каков оптимальный алгоритм выполнения двуместной операции над множествами в хеш-таблице? Какова его временная сложность?

6. Можно ли хранить в хеш-таблице множество с повторениями?

7. Какова временная сложность операций вставки и удаления элемента для хеш-таблицы?

8. Почему для операций с хеш-таблицей дают две оценки временной сложности — сложность в худшем случае и сложность в среднем?

9. Что такое вырождение хеш-таблицы и как его избежать?

10. Что нужно делать, если хеш-таблица переполнилась?

# 2. ДЕРЕВЬЯ ДВОИЧНОГО ПОИСКА

Дерево двоичного поиска (ДДП) — это способ хранения множества в форме расширяемого упорядоченного списка с сохранением упорядоченности при вставке новых элементов без перемещения уже имеющихся.

ДДП — это дерево с нагруженными узлами, вес в любом узле которого больше любого веса в левом его поддереве и не больше любого веса в правом поддереве. Количество шагов алгоритма поиска элемента множества в таком дереве не превышает его высоты, т. е. имеет сложность *O*(log *n*). Такую же сложность имеют операции вставки нового элемента в дерево и удаления элемента.

ДДП можно получить из упорядоченной последовательности ключей, если двоичное дерево соответствующей мощности разметить внутренним (симметричным) способом, а затем заменить номера узлов соответствующими элементами последовательности. Если последовательность хранится в массиве, то построить ДДП можно методом деления пополам: поместить в корень дерева средний по порядку элемент, затем рекурсивно создать левое поддерево из первой половины последовательности, а правое — из второй.

Node \* Build(int a, int b, int \* data) //Сборка ДДП из массива узлов data

{ if (b <= a) return 0;

 int c = (a + b) /2;

 Node \* s = new Node (data[ c ]);

 s->left = Build(a, c, data);

 s->right = Build(c+1, b, data);

 return s;

}

Недостаток ДДП — в том, что оно хорошо работает только в том случае, если сбалансировано, т. е. длины путей из корня в любой лист примерно одинаковы. Однако при поэлементной вставке в дерево упорядоченной последовательности дерево вырождается, превращаясь в линейный список. Поиск, вставка и удаление в таком дереве будут выполняться не за логарифмическое, а за линейное время. Вероятность вырождения весьма велика. Так, из 7 узлов можно образовать только одно полностью сбалансированное дерево, а полностью вырожденных — 64. Поэтому алгоритмы работы с ДДП часто дополняют автобалансировкой после вставки и удаления. Наиболее употребительны следующие схемы:

1) АВЛ-деревья. Разность высот поддеревьев любого узла дерева не превышает 1. Информация о разности высот хранится в узле и используется для его перестройки после вставки и удаления узла;

2) красно-чёрные деревья. Узлы красятся в один из двух цветов — чёрный или красный. Если узел — красный, его сыновья — обязательно чёрные. Вставляемый узел — всегда красный. При появлении цепочки из двух красных узлов дерево перестраивается;

3) 2–3-деревья. Данные хранятся в листьях, над которыми делается надстройка из управляющих узлов, каждый из которых может иметь 2 или 3 сына и содержит наибольшие значения ключей в левом и в среднем поддеревьях, что необходимо для операции поиска. Если в результате вставки или удаления у управляющего узла оказывается 1 или 4 сына, дерево перестраивается.

Подробнее о ДДП и деревьях с автобалансировкой *см*. [1, с. 457–468], [2, с. 523–566], [5, с. 341–344].

Двуместные операции над множествами в ДДП можно выполнять, используя примитивы проверка–вставка–удаление. Очевидно, что временная сложность двуместной операции при этом будет в среднем *O*(*n* log *n*). В худшем случае, при вырожденных деревьях, двуместная операция потребует *O*(*n*2) времени. Более того, если алгоритм порождает упорядоченную последовательность ключей, ДДП, полученное в результате последовательности вставок, всегда будет вырожденным.

Однако можно использовать тот факт, что из ДДП легко получить упорядоченную последовательность ключей внутренним обходом. Применив такой обход к двум ДДП одновременно, можно обработать последовательности алгоритмом слияния, модифицировав его для нужной операции над множествами. Результат в виде упорядоченной последовательности записывается в массив, из которого затем описанным ранее методом деления пополам строится новое ДДП. Этот способ обеспечивает получение результата за время *O*(*n*) независимо от формы исходных ДДП, а результат всегда получается сбалансированным.

ДДП сами по себе мало пригодны для хранения множеств с повторениями, поскольку дубликаты ключей искажают форму дерева, образуя в нём мёртвые зоны: группы указателей, которые никогда не используются. Ситуацию можно улучшить, если вместо дубликатов ключей хранить в каждом узле значение кратности (1 или больше). Если же дубликаты должны быть представлены явно, их можно хранить в узлах как цепочки переполнения, по аналогии с хеш-таблицей.

### 2.1. Практикум по теме

Переделать программу, составленную по теме «Хеш-таблицы», под использование деревьев двоичного поиска. Вид дерева определить из табл. 2.1 по номеру варианта индивидуального задания.

Таблица 2.1

Варианты вида дерева для реализации в теме «Деревья двоичного поиска»

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Вариант | Дерево | Вариант | Дерево | Вариант | Дерево | Вариант | Дерево |
| 1 | ДДП | 14 | АВЛ-д | 27 | 2–3-д | 40 | К-ч-д |
| 2 | АВЛ-д | 15 | 2–3-д | 28 | К-ч-д | 41 | ДДП |
| 3 | 2–3-д | 16 | К-ч-д | 29 | ДДП | 42 | АВЛ-д |
| 4 | К-ч-д | 17 | ДДП | 30 | АВЛ-д | 43 | 2–3-д |
| 5 | ДДП | 18 | АВЛ-д | 31 | 2–3-д | 44 | К-ч-д |
| 6 | АВЛ-д | 19 | 2–3-д | 32 | К-ч-д | 45 | ДДП |
| 7 | 2–3-д | 20 | К-ч-д | 33 | ДДП | 46 | АВЛ-д |
| 8 | К-ч-д | 21 | ДДП | 34 | АВЛ-д | 47 | 2–3-д |
| 9 | ДДП | 22 | АВЛ-д | 35 | 2–3-д | 48 | К-ч-д |
| 10 | АВЛ-д | 23 | 2–3-д | 36 | К-ч-д | 49 | ДДП |
| 11 | 2–3-д | 24 | К-ч-д | 37 | ДДП | 50 | АВЛ-д |
| 12 | К-ч-д | 25 | ДДП | 38 | АВЛ-д |  |  |
| 13 | ДДП | 26 | АВЛ-д | 39 | 2–3-д |  |  |

*Примечание*. В таблице использованы следующие обозначения:

ДДП — дерево двоичного поиска (без автобалансировки);

АВЛ-д — АВЛ-дерево (с автобалансировкой);

К-ч-д — красно-чёрное дерево (с автобалансировкой);

2–3-д — 2–3-дерево (всегда сбалансированное).

### 2.2. Требования к отчёту

В выводах отчёта по теме «Деревья двоичного поиска» попробуйте оценить, насколько удачен выбор типа ДДП для вашего варианта задания на обработку множеств.

### 2.3. Контрольные вопросы

1. Каким способом следует разметить дерево, чтобы в нём был возможен двоичный поиск? Как его следует нагрузить?

2. Почему для операций с двоичным деревом дают две оценки сложности — «в худшем случае» и «в среднем»? Почему не рассматривается «лучший» случай?

3. Отчего деревья двоичного поиска вырождаются?

4. Можно ли воспрепятствовать вырождению ДДП?

5. Каков оптимальный алгоритм двуместной операции над множествами, представленными деревьями двоичного поиска?

6. Какова временная сложность такой операции и как сказывается на ней возможное вырождение деревьев?

7. Может ли при двуместной операции над множествами в ДДП получиться вырожденное дерево-результат?

8. Можно ли хранить в дереве двоичного поиска множество с пов­торе­ни­ями?

9. Какая структура данных требует больше памяти для хранения множества — хеш-таблица или дерево двоичного поиска?

10. Какая из них быстрее работает?

11. Как сделать не вырождающееся ДДП? Зачем оно может понадобиться?

# 3. ПОДДЕРЖКА ПРОИЗВОЛЬНОЙ ПОСЛЕДОВАТЕЛЬНОСТИ В СТРУКТУРЕ ДАННЫХ ДЛЯ МНОЖЕСТВ

К одной структуре данных могут применяться операции как для множества, так и для последовательности. Иногда требуется поддерживать в структуре данных для множеств произвольную последовательность элементов этих множеств. Например, можно фиксировать порядок появления элементов в множестве при его создании и работать с этим порядком.

Последовательность из структуры данных для множеств может быть получена как результат её обхода. Часто этого бывает достаточно: порядок элементов для результата операции над множествами можно назначить произвольно. Однако для множества мощностью *n* это будет только одна из *n*! возможных последовательностей. Если нужно поддерживать любую последовательность, возможны следующие подходы:

1) присоединить к каждому ключу дополнительное поле для хранения порядкового номера. Способ не создаёт проблем при поиске номера по значению ключа и при вставке новых ключей, они просто нумеруются по порядку. Удаление ключа требует просмотра всей структуры данных для корректировки номеров, следующих за удаляемым. То же приходится делать при поиске ключа по порядковому номеру (сложность *O*(*n*));

2) с помощью дополнительных указателей для каждого ключа сформировать из них список, возможно, двунаправленный. Проходом по этому списку можно как восстановить хранящуюся последовательность, так и получить номер для каждого ключа. Доступ к ключу по номеру и наоборот имеет в этом случае линейную сложность. Зато как вставка, так и удаление ключа требуют минимальных накладных расходов;

3) создать массив указателей на ключи. Если одновременно поддерживать в ключах дополнительное поле с обратным указателем на соответствующие элементы массива, можно избежать дополнительных расходов как для определения ключа по номеру, так и номера по ключу. Недостаток способа — необходимо заранее знать объём памяти для создания массива и перемещать часть массива в случае удаления ключей.

Операции над последовательностями, в отличие от операций с множествами, могут приводить к появлению дубликатов ключей. Структура данных для множеств должна обеспечивать соответствующую возможность.

Операции над последовательностями:

1. Слияние (*MERGE*). Объединение двух упорядоченных последовательностей в третью с сохранением упорядоченности. От операции объединения множеств отличается только возможностью появления дубликатов ключей. Если исходные последовательности не упорядочены, можно после их слияния просто упорядочить результат. Исходный порядок ключей в множествах в результате не сохраняется.

2. Сцепление (*CONCAT*). Вторая последовательность подсоединяется к концу первой, образуя её продолжение.

3. Размножение (*MUL*). Последовательность сцепляется сама с собой заданное количество раз.

4. Укорачивание (*ERASE*). Из последовательности исключается часть, ограниченная порядковыми номерами от *p*1 до *p*2.

5. Исключение (*EXCL*). Вторая последовательность исключается из первой, если она является её частью.

6. Включение (*SUBST*). Вторая последовательность включается в первую с указанной позиции *p*. Операция похожа на конкатенацию. Сперва берётся начало первой последовательности до позиции *p*, затем идёт вторая последовательность, а за ней — остаток первой.

7. Замена (*CHANGE*). Вторая последовательность заменяет элементы первой, начиная с заданной позиции *p*.

***Пример***. Пусть имеются две последовательности *A* = <5, 3, 2, 4, 6, 7, 9, 1> и *B* = <6, 7, 9>. Позиции считаются от 0.

Тогда операция *A.MERGE*(*B*) даст результат <1, 2, 3, 4, 5, 6, 6, 7, 7, 9, 9>;

*A.CONCAT*(*B*) — <5, 3, 2, 4, 6, 7, 9, 1, 6, 7, 9>;

*B.MUL*(3) — <6, 7, 9, 6, 7, 9, 6, 7, 9>;

*A.ERASE*(2, 4) — <5, 3, 7, 9, 1>;

*A.EXCL*(*B*) — <5, 3, 2, 4, 1>;

*A.SUBST*(*B,* 3) — <5, 3, 2, 6, 7, 9, 4, 6, 7, 9, 1>;

*A.CHANGE*(*B,* 2) — <5, 3, 6, 7, 9, 7, 9, 1>.

### 3.1. Практикум по теме

Дополнить программу, составленную по теме «Хеш-таблицы» или «Деревья двоичного поиска», операциями над последовательностями, указанными в табл. 3.1 в соответствии с номером задания. Выбрать такой способ доработки структуры данных, чтобы получились эффективные алгоритмы.

Таблица 3.1

Индивидуальные задания к теме «Последовательности»

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № вари-анта | Базо­ваяСД | Дополнительные операции | № вари-анта | Базо­ваяСД | Дополнительные операции |
| 1 | ДДП | MERGE, EXCL, CHANGE | 26 | ХТ | CONCAT, EXCL, SUBST |
| 2 | ХТ | CONCAT, EXCL, MUL | 27 | ДДП | MERGE, EXCL, SUBST |
| 3 | ДДП | EXCL, SUBST, MUL | 28 | ХТ | CONCAT, SUBST, CHANGE |
| 4 | ДДП | MERGE, ERASE, SUBST | 29 | ХТ | MERGE, CONCAT, CHANGE |
| 5 | ХТ | MERGE, CONCAT, SUBST | 30 | ДДП | MERGE, CONCAT, MUL |
| 6 | ХТ | MERGE, EXCL, SUBST | 31 | ДДП | EXCL, SUBST, CHANGE |
| 7 | ДДП | CONCAT, EXCL, SUBST | 32 | ХТ | ERASE, EXCL, MUL |
| 8 | ХТ | MERGE, ERASE, EXCL | 33 | ДДП | MERGE, CONCAT, SUBST |
| 9 | ХТ | MERGE, CONCAT, EXCL | 34 | ДДП | ERASE, EXCL, CHANGE |
| 10 | ХТ | CONCAT, ERASE, CHANGE | 35 | ДДП | CONCAT, ERASE, EXCL |
| 11 | ДДП | CONCAT, SUBST, CHANGE | 36 | ХТ | EXCL, SUBST, MUL |
| 12 | ДДП | ERASE, EXCL, SUBST | 37 | ХТ | CONCAT, EXCL, CHANGE |
| 13 | ХТ | CONCAT, EXCL, SUBST | 38 | ДДП | MERGE, CONCAT, EXCL |
| 14 | ДДП | CONCAT, EXCL, CHANGE | 39 | ХТ | MERGE, EXCL, CHANGE |
| 15 | ХТ | MERGE, CONCAT, MUL | 40 | ДДП | CONCAT, EXCL, MUL |
| 16 | ДДП | ERASE, SUBST, CHANGE | 41 | ХТ | MERGE, CONCAT, ERASE |
| 17 | ХТ | MERGE, SUBST, MUL | 42 | ДДП | CONCAT, EXCL, SUBST |
| 18 | ДДП | MERGE, SUBST, CHANGE | 43 | ХТ | ERASE, EXCL, SUBST |
| 19 | ХТ | CONCAT, ERASE, EXCL | 44 | ДДП | CONCAT, ERASE, CHANGE |
| 20 | ДДП | MERGE, CONCAT, CHANGE | 45 | ХТ | MERGE, ERASE, SUBST |
| 21 | ХТ | EXCL, SUBST, CHANGE | 46 | ДДП | MERGE, SUBST, MUL |
| 22 | ХТ | ERASE, EXCL, CHANGE | 47 | ДДП | MERGE, ERASE, CHANGE |
| 23 | ДДП | MERGE, CONCAT, ERASE | 48 | ХТ | MERGE, SUBST, CHANGE |
| 24 | ДДП | ERASE, EXCL, MUL | 49 | ХТ | ERASE, SUBST, CHANGE |
| 25 | ХТ | MERGE, ERASE, CHANGE  | 50 | ДДП | MERGE, ERASE, EXCL |

### 3.2. Требования к отчёту

В отчёте по этой теме должно быть обоснование выбора способа дополнения базовой структуры данных для обеспечения возможности выполнения операций с последовательностями. В выводах можно дать заключение, насколько выбор оказался удачным.

### 3.3. Контрольные вопросы

1. Почему для хранения произвольной последовательности структуру данных для множества (хеш-таблицу или ДДП) приходится дорабатывать?

2. Какие доработки возможны?

3. Можно ли предложить оптимальный вариант доработки?

4. Влияет ли доработка структур данных для множеств для поддержки последовательностей на временную сложность операций над множествами?

5. Какую структуру данных проще дорабатывать — хеш-таблицу или ДДП?

6. Какова оптимальная доработка структуры данных и временная сложность для операции исключения части последовательности между указанными позициями?

7. То же — для операции вставки с указанной позиции?

8. То же — для замены?

# 4. РАБОТА С ИЕРАРХИЕЙ ОБЪЕКТОВ: 4 НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ

Производные классы — это простое, гибкое и эффективное средство определения класса с целью повторного использования готового программного кода. Новые возможности добавляются к уже существующему классу, не требуя его перепрограммирования или перекомпиляции. С помощью производных классов можно организовать общий интерфейс с несколькими различными классами так, что в других частях программы можно будет единообразно работать с объектами этих классов. Понятие виртуальной функции позволяет использовать объекты надлежащим образом даже в тех случаях, когда их тип на стадии трансляции неизвестен. Основное назначение производных классов — упростить программисту задачу выражения общности классов.

Любое понятие не существует изолированно, оно существует во взаимосвязи с другими понятиями, и мощность данного понятия во многом определяется наличием таких связей. Раз класс служит для представления понятий, встаёт вопрос, как представить взаимосвязь понятий. Понятие производного класса и поддерживающие его языковые средства служат для представления иерархических связей, иными словами, для выражения общности между классами. Например, понятия окружности и треугольника связаны между собой, так как оба они представляют ещё понятие фигуры, т. е. содержат более общее понятие. Чтобы представлять в программе окружности и треугольники и при этом не упускать из вида, что они являются фигурами, надо явно определять классы окружность и треугольник так, чтобы было видно, что у них есть общий класс — фигура. Эта простая идея по сути является основой того, что обычно называется объектно-ориентированным программированием.

Подробнее *см*. [6, с. 149–180], [7, с. 200–210].

Рассмотрим учебную программу, использующую некоторые из этих идей, прототип которой взят из [6]. Программа предназначена для вывода на экран картинок, составленных из набора заготовок — «фигур».

В программе объявлен абстрактный класс «фигура» (*shape*). Все конкретные фигуры — линия, прямоугольник и т. п. — являются производными от этого класса. Класс «фигура» поддерживает действия, необходимые для всех фигур: он создаёт из всех объявляемых фигур общий список, который может быть обработан программой рисования при выдаче фигур на экран. Кроме того, в классе «фигура» объявлен набор функций-членов, которые должны поддерживать все фигуры, чтобы из них можно было создавать картинки. Это функции, возвращающие координаты всех крайних точек фигуры, по которым их можно будет стыковать. Эти функции — чисто виртуальные, они должны быть обязательно определены затем отдельно в каждой фигуре. Имеются также два дополнительных класса, уточняющие свойства фигур. Некоторые фигуры можно поворачивать. Для таких фигур имеется базовый класс *rotatable*, производный от *shape*. Для других фигур возможна операция отражения относительно горизонтальной или вертикальной оси. Эти фигуры можно строить на базе класса *reflectable*. Если фигура имеет оба этих свойства, то она должна быть производной от обоих классов.

Класс «фигура» является ядром библиотеки фигур *snape.h*. Имеется также библиотека поддержки работы с экраном *screen.h*, в которой определены размеры экрана, введено понятие точки и перечислены утилиты работы с экраном, конкретизированные затем в *shape.h*. Для простоты и универсальности работа с экраном реализована как формирование и построчный вывод матрицы символов.

Предполагается, что файлы *screen.h* и *shape.h* — покупные, а разработчик создаёт только третий файл — с самой прикладной программой.

### 4.1. Учебная программа «Библиотека фигур»

//Прикладная программа для работы с библиотекой фигур

//=== Файл screen.h - поддержка работы с экраном

const int XMAX=80; //Размер экрана

const int YMAX=40;

class point { //Точка на экране

public:

 int x, y;

 point( ) { };

 point(int a, int b) : x(a), y(b){ }

};

// Набор утилит для работы с экраном

void put\_point(int a, int b); // Вывод точки

void put\_point(point p) { put\_point(p.x, p.y); }

void put\_line(int, int, int, int); // Вывод линии

void put\_line(point a, point b)

 { put\_line(a.x, a.y, b.x, b.y); }

extern void screen\_init( ); // Создание экрана

extern void screen\_destroy( ); // Удаление экрана

extern void screen\_refresh( ); // Обновление

extern void screen\_clear( ); // Очистка

//————————————————————————————————

//=== Файл shape.h -- библиотека фигур

//==1. Поддержка экрана в форме матрицы символов ==

char screen[XMAX] [YMAX];

enum color { black='\*', white='.' };

void screen\_init( )

{

 for (int y=0; y<YMAX; y++)

 for (int x=0; x<XMAX; x++)

 screen[x] [y] = white;

}

int on\_screen(int a, int b) // проверка попадания на экран

{ return 0 <= a && a < XMAX && 0 <= b && b < YMAX; }

void put\_point(int a, int b)

{ if (on\_screen(a,b)) screen[a] [b] = black; }

void put\_line(int x0, int y0, int x1, int y1)

/\*

Рисование отрезка прямой (x0,y0) - (x1,y1).

Уравнение прямой: b(x-x0) + a(y-y0) = 0.

Минимизируется величина abs(eps),

где eps = 2\*(b(x-x0)) + a(y-y0).

\*/

{

 int dx = 1;

 int a = x1 - x0;

 if (a < 0) dx = -1, a = -a;

 int dy = 1;

 int b = y1 - y0;

 if (b < 0) dy = -1, b = -b;

 int two\_a = 2\*a;

 int two\_b = 2\*b;

 int xcrit = -b + two\_a;

 int eps = 0;

 for (;;) {

 put\_point(x0, y0);

 if (x0 == x1 && y0 == y1) break;

 if (eps <= xcrit) x0 += dx, eps += two\_b;

 if (eps >= a || a < b) y0 += dy, eps -= two\_a;

 }

}

void screen\_clear( ) { screen\_init( ); } //Очистка экрана

void screen\_refresh( ) // Обновление экрана

{

 for (int y = YMAX-1; 0 <= y; --y) { // с верхней строки до нижней

 for (int x = 0; x < XMAX; ++x) // от левого столбца до правого

 cout << screen[x] [y];

 cout << '\n';

 }

}

//==2. Библиотека фигур ==

struct shape { // Виртуальный базовый класс "фигура"

 static shape\* list;

 shape\* next;

 shape( ) { next = list; list = this; }

 virtual point north( ) const = 0;

 virtual point south( ) const = 0;

 virtual point east( ) const = 0;

 virtual point west( ) const = 0;

 virtual point neast( ) const = 0;

 virtual point seast( ) const = 0;

 virtual point nwest( ) const = 0;

 virtual point swest( ) const = 0;

 virtual void draw( ) = 0;

 virtual void move(int, int) = 0;

};

shape \* shape :: list = nullptr; //Инициализация списка фигур

class rotatable : public shape { //Фигуры, пригодные к повороту

public:

 virtual void rotate\_left( ) = 0; //Повернуть влево

 virtual void rotate\_right( ) = 0; //Повернуть вправо

};

class reflectable : public shape { // Фигуры, пригодные

 // к зеркальному отражению

public:

 virtual void flip\_horisontally( ) = 0; // Отразить горизонтально

 virtual void flip\_vertically( ) = 0; // Отразить вертикально

};

class line : public shape {

/\* отрезок прямой ["w", "e" ].

north( ) определяет точку "выше центра отрезка и так далеко

на север, как самая его северная точка", и т. п. \*/

 point w, e;

public:

 line(point a, point b) : w(a), e(b) { };

 line(point a, int L) : w(point(a.x + L - 1, a.y)), e(a) { };

 point north( ) const { return point((w.x+e.x)/2, e.y<w.y? w.y : e.y); }

 point south( ) const { return point((w.x+e.x)/2, e.y<w.y? e.y : w.y); }

 point east( ) const { return point((w.x+e.x)/2, e.y<w.y? e.y : w.y); }

 point west( ) const { return point((w.x+e.x)/2, e.y<w.y? e.y : w.y); }

 point neast( ) const { return point((w.x+e.x)/2, e.y<w.y? e.y : w.y); }

 point seast( ) const { return point((w.x+e.x)/2, e.y<w.y? e.y : w.y); }

 point nwest( ) const { return point((w.x+e.x)/2, e.y<w.y? e.y : w.y); }

 point swest( ) const { return point((w.x+e.x)/2, e.y<w.y? e.y : w.y); }

 void move(int a, int b) { w.x += a; w.y += b; e.x += a; e.y += b; }

 void draw( ) { put\_line(w, e); }

};

// Прямоугольник

class rectangle : public rotatable {

/\* nw ------ n ------ ne

 | |

 | |

 w c e

 | |

 | |

 sw ------ s ------ se \*/

 point sw, ne;

public:

 rectangle(point, point);

 point north( ) const { return point((sw.x + ne.x) / 2, ne.y); }

 point south( ) const { return point((sw.x + ne.x) / 2, sw.y); }

 point east( ) const { return point(sw.x, (sw.y + ne.y) / 2); }

 point west( ) const { return point(ne.x, (sw.y + ne.y) / 2); }

 point neast( ) const { return ne; }

 point seast( ) const { return point(sw.x, ne.y); }

 point nwest( ) const { return point(ne.x, sw.y); }

 point swest( ) const { return sw; }

 void rotate\_right() // Поворот вправо относительно se

 { int w = ne.x - sw.x, h = ne.y - sw.y;

 sw.x = ne.x – h \* 2; ne.y = sw.y + w / 2; }

 void rotate\_left() //Поворот влево относительно sw

 { int w = ne.x - sw.x, h = ne.y - sw.y;

 ne.x = sw.x + h \* 2; ne.y = sw.y + w / 2; }

 void move(int a, int b)

 { sw.x += a; sw.y += b; ne.x += a; ne.y += b; }

 void draw( );

};

rectangle::rectangle(point a, point b)

{ if (a.x <= b.x) {

 if (a.y <= b.y) sw = a, ne = b;

 else sw = point(a.x, b.y), ne = point(b.x, a.y);

 }

 else {

 if (a.y <= b.y) sw = point(b.x, a.y), ne = point(a.x, b.y);

 else sw = b, ne = a;

 }

}

void rectangle::draw( )

{ point nw(sw.x, ne.y);

 point se(ne.x, sw.y);

 put\_line(nw, ne);

 put\_line(ne, se);

 put\_line(se, sw);

 put\_line(sw, nw);

}

void shape\_refresh( ) // Перерисовка всех фигур

{

 screen\_clear( );

 for (shape\* p = shape :: list; p; p = p->next) p->draw( );

 screen\_refresh( );

}

void up(shape\* p, const shape\* q) // поместить p над q

{

 point n = q->north( );

 point s = p->south( );

 p->move(n.x - s.x, n.y - s.y + 1);

}

//========================================================

// Прикладная программа:

// пополнение и использование библиотеки фигур

#include "stdafx.h"

#include <conio.h>

#include <iostream>

#include "screen.h"

#include "shape.h"

// Дополнительная "сборная" фигура

class myshape : public rectangle {

 line\* l\_eye; // левый глаз

 line\* r\_eye; // правый глаз

 line\* mouth; // рот

public:

 myshape(point, point);

 void draw( );

 void move(int, int);

};

myshape::myshape(point a, point b) : rectangle(a, b)

{

 int ll = neast( ).x - swest( ).x + 1;

 int hh = neast( ).y - swest( ).y + 1;

 l\_eye = new line(point(swest( ).x + 2, swest( ).y + hh \* 3 / 4), 2);

 r\_eye = new line(point(swest( ).x + ll - 4, swest( ).y + hh \* 3 / 4), 2);

 mouth = new line(point(swest( ).x + 2, swest( ).y + hh / 4), ll - 4);

}

void myshape::draw( )

{

 rectangle :: draw( );

 int a = (swest( ).x + neast( ).x) / 2;

 int b = (swest( ).y + neast( ).y) / 2;

 put\_point(point(a, b));

}

void myshape :: move(int a, int b)

{

 rectangle :: move(a, b);

 l\_eye->move(a, b);

 r\_eye->move(a, b);

 mouth->move(a, b);

}

int \_tmain( )

{

 screen\_init( );

//== 1.Объявление набора фигур ==

 rotatable\* p1 = new rectangle(point(0, 0), point(14, 5));

 shape\* p2 = new line(point(0,15),17);

 shape\* p3 = new myshape(point(15,10), point(27,18));



*Рис*. *4.1*. Результат
работы программы

 shape\_refresh( );

 \_getch( );

//== 2.Ориентация ==

 p1->rotate\_right( );

 shape\_refresh( );

 \_getch( );

//== 3.Сборка изображения ==

 p3->move(-10, -10);

 up(p2, p3);

 up(p1, p2);

 shape\_refresh( );

 system(“pause”);

// screen\_destroy( );

 return 0;

}

При запуске программы на экран сначала выводится объявленная коллекция фигур. Затем демонстрируется результат поворота/отражения некоторых фигур как подготовка к их использованию. Далее фигуры перемещаются и образуют заданную картинку: физиономия в шляпе (рис. 4.1). Для рисования использованы прямоугольники, линии и точки. Физиономия является пользовательской фигурой.

### 4.2. Практикум по теме

В табл. 4.1 собрана коллекция фигур, которыми можно дополнить рассмотренную прикладную программу.

Таблица 4.1

Коллекция дополнительных фигур

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № п/п | Наименование | Вид | Отражение | Поворот |
| 1 | Прямоугольник |  | Нет | Да |
| 2 | Квадрат |  | Нет | Нет |
| 3 | Ромб |  | Нет | Нет |
| 4 | Параллелограмм |  | Да | Да |
| 5 | Трапеция |  | Да | Да |
| 6 | Крест |  | Нет | Нет |
| 7 | Косой крест |  | Нет | Нет |

Окончание табл. 4.1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| № п/п | Наименование | Вид | Отражение | Поворот |
| 8 | Треугольник |  | Да | Да |
| 9 | Кружок |  | Нет | Нет |
| 10 | Квадрат с крестом |  | Нет | Нет |
| 11 | Ромб с крестом |  | Нет | Нет |
| 12 | Кружок с крестом |  | Нет | Нет |
| 13 | Треугольникс крестом |  | Да | Да |
| 14 | Зачёркнутый квадрат  |  | Нет | Нет |
| 15 | Зачёркнутый кружок |  | Нет | Нет |
| 16 | Зачёркнутыйтреугольник |  | Да | Да |

Для некоторых фигур возможны поворот на 90o вправо или влево или отражение относительно горизонтальной и/или вертикальной оси симметрии, причём для части из них имеются обе возможности. Некоторые фигуры строятся как составные из более простых. Эти идеи можно отобразить показанной на рис. 4.2 иерархией классов.

Доработать учебную программу: добавить в коллекцию ещё одну фигуру, номер которой указан в табл. 4.2. Для этой фигуры нужно будет оп­ределить подходящее место в иерархии классов и написать необходимые функ­ции-члены. Функции-члены, использование которых не предполагается, можно определить так, чтобы они были недоступны. Разработанной фигурой нужно дополнить картинку в указанных в варианте позициях. Позиция *1* обозначает галстук или воротник, *2* и *3* — бакенбарды, *4* и *5* — уши, *6* — кокарду, *7* и *8* — рога, *9* — нос, *10* и *11* — глаза, *12* — шляпу в целом (*см*. рис. 4.3). Возможно, некоторые из фигур нужно будет повернуть или отразить. Для примыкания фигур должны использоваться их габаритные точки. Необходимо написать аналоги функции up (поместить *p* над *q*), обеспечивающие примыкание очередной фигуры *p* с нужной стороны по отношению к уже размещённой *q*.



*Рис. 4.2*. Фрагмент иерархии классов фигур



*Рис*. *4.3*. Позиции на результате работы программы
 для возможной вставки дополнительной фигуры

Таблица 4.2

Индивидуальные задания к теме «Наследование и полиморфизм»

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № вари-анта | Фигура | Расположение | № вари-анта | Фигура | Расположение |
| 1 | 2 | 2, 3, 10, 11 | 26 | 15 | 6, 7, 8 |
| 2 | 7 | 1, 4, 5 | 27 | 12 | 9, 10 11 |
| 3 | 6 | 4, 5 | 28 | 11 | 7, 8, 12 |
| 4 | 3 | 10, 11, 12 | 29 | 2 | 4, 5, 9 |
| 5 | 8 | 4, 5, 6 | 30 | 11 | 1, 7, 8 |
| 6 | 5 | 1 | 31 | 12 | 1, 2, 3 |
| 7 | 4 | 7, 8 | 32 | 3 | 1, 4, 5 |
| 8 | 9 | 10, 11, 12 | 33 | 10 | 4, 5 |
| 9 | 4 | 2, 3 | 34 | 13 | 1, 7, 8 |
| 10 | 5 | 4, 5 | 35 | 4 | 7, 8 |
| 11 | 10 | 1, 12 | 36 | 9 | 1, 4, 5 |
| 12 | 3 | 4, 5, 9 | 37 | 14 | 4, 5, 6 |
| 13 | 6 | 1, 12 | 38 | 5 | 4, 5 |
| 14 | 11 | 10, 11 | 39 | 8 | 1, 6, 9 |
| 15 | 2 | 1, 7, 8, 12 | 40 | 15 | 1, 2, 3 |
| 16 | 7 | 6, 7, 8 | 41 | 6 | 2, 3, 9 |
| 17 | 12 | 2, 3, 7, 8 | 42 | 7 | 10, 11 |
| 18 | 15 | 10, 11 | 43 | 2 | 1, 2, 3, 7, 8 |
| 19 | 8 | 2, 3, 7, 8 | 44 | 7 | 2, 3, 12 |
| 20 | 13 | 1, 12 | 45 | 6 | 10, 11 |
| 21 | 14 | 2, 3, 12 | 46 | 3 | 2, 3, 6 |
| 22 | 9 | 2, 3, 9 | 47 | 8 | 9, 10, 11 |
| 23 | 14 | 7, 8, 9 | 48 | 5 | 12 |
| 24 | 13 | 1, 4, 5 | 49 | 4 | 4, 5 |
| 25 | 10 | 7, 8 | 50 | 9 | 4, 5, 7, 8 |

### 4.3. Требования к отчёту

В отчёт по этой теме включите описание получившейся иерархии классов и пояснения:

1) какие классы пришлось добавить;

2) какие функции-члены пришлось переопределить и почему;

3) какие функции-члены сделаны недоступными и каким образом это осуществлено.

### 4.4. Контрольные вопросы

1. Какой базовый класс лучше всего использовать для производного класса «треугольник»?

2. То же — для класса «кружок»?

3. То же — для класса «крестик»?

4. Какой тип наследования следует выбрать: private, public или protected?

5. Можно ли вообще не указывать тип наследования?

6. В чём смыл объявления функций в базовом классе как виртуальных?

7. Что такое «чисто виртуальная функция»?

8. Обязательно ли переопределять все функции-члены базового класса в производном классе?

9. Зачем может понадобиться создание набора (массива или списка) указателей на разные типы объектов в пределах некоторой иерархии?

10. Как запретить для объекта вызов конструктора по умолчанию?

11. Как запретить вызов конструктора для использования в качестве неявного преобразователя типа?

12. Каким образом можно установить значение переменных объекта, объявленных с модификаторами const?

13. Каким образом следует инициализировать объект базового класса в конструкторе производного класса? Всегда ли это нужно делать?

# 5. ПОДДЕРЖКА ОБРАБОТКИ ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

Механизм исключительных ситуаций предоставляет пользователю возможность контроля за ходом выполнения программы и механизм нейтрализации возможных ошибок. Главное в этом механизме — поддержка действий по нейтрализации последствий события, препятствующего продолжению нормальной работы. Очень часто исключительная ситуация — это не ошибка, а просто исчерпание какого-то ресурса, например, доступной памяти или времени ожидания сигнала или даже один из предусмотренных вариантов завершения процесса. Механизм используется, если ситуация не может быть разрешена в той точке, где была выявлена, и требует перехода на более высокий уровень, с завершением каких-то активных функций и освобождением ресурсов.

Для задействования механизма особых ситуаций в программе нужно проделать следующее:

— обнаружив в программе место, где особая ситуация может возникнуть, надо придумать для неё уникальное название, например *My\_Error*, и объявить соответствующий класс ошибок, возможно, пустой:

class My\_Error {};

— в точке программы, в которой обнаружена особая ситуация, поместить утверждение

throw My\_Error( );

Это утверждение создаёт объект класса *My\_Error*. Если в объекте предусмотрены поля для данных, через них можно передать информацию обработчику ошибок: аргумент утверждения *throw* — это конструктор объекта;

— точку вызова функции, которая может создать исключения, нужно поместить в блок контроля, за которым следуют обработчики особых ситуаций:

try{ //начало блока контроля;

 вызов функции, которая может создать особую ситуацию

 (содержит утверждения throw My\_Error);

}

catch (My\_Error)

{ обработка особой ситуации }

Предложение *catch* размещается на том уровне, где обработка ситуации *My\_Error* возможна. Если нужно обрабатывать несколько различных ошибок, после блока *try* последовательно размещаются соответствующие обработчики. При возникновении любой особой ситуации в блоке *try* его работа прерывается: происходит принудительный выход из всех функций, которые были активны в точке особой ситуации, и вызов деструкторов для всех созданных при этом объектов, как это происходит при выходе из блока (области видимости). Этот процесс называется раскруткой стека: стек возвращается в состояние, в котором он был при входе в блок *try*.

Далее просматриваются блоки *catch* в том порядке, в каком они объявлены. Как только обнаруживается блок обработки ошибок нужного типа, управление передаётся ему. Остальные блоки *catch* не используются.

Если же выполнение блока *try* завершилось успешно, все блоки catch после него игнорируются.

Если для некоторого типа ошибки не обнаружено соответствующего блока *catch*, программа завершается аварийно. Чтобы этого избежать, последним в цепочке можно разместить блок *catch*(…), перехватывающий ошибки любого типа.

Как только подходящий блок *catch* будет вызван, особая ситуация будет считаться обработанной, даже если этот блок пуст. Однако чаще всего в него помещают выдачу на экран или в специальный файл (журнал) содержательного сообщения об ошибке. Возможно также одно из следующих действий:

— устранение причины ошибки (уменьшение запроса на выделение памяти, отказ от обработки несуществующего или испорченного файла и т. п.);

— аварийное завершение программы (вызов *abort*( ));

— перевозбуждение особой ситуации для передачи на следующий уровень иерархии (вызов *throw* без аргумента).

Подробнее об особых ситуациях и их обработке *см*. [6, с. 232–256], [7, с. 222–230], [8, с. 399–414].

Правильный выбор уровня для размещения блока контроля позволяет сделать программу безопасной в смысле исключений (*см*. [14, с. 105–174]). Так, в учебном примере имеется следующая цепочка вызовов функций:

main( ) → screen\_refresh( ) → myshape ∷ draw( ) → rectangle ∷ draw ( ) → put\_line(a, b) → put\_point(x, y) → on\_screen(x, y).

Выход точки за пределы буферного массива SCREEN (экрана) выявляется функцией on\_screen( ). Блок контроля вокруг вызова этой функции (или вызывающей её put\_point) не имеет смысла: на этом уровне ничего, кроме выдачи сообщения об ошибке, сделать нельзя, а такое сообщение можно выдать непосредственно, не прибегая к механизму throw — cath. На уровне main( ) или screen\_refresh( ) обрабатывать ошибку поздно, можно только прервать выполнение программы, содержательное сообщение о месте ошибки получить нельзя. В то же время блок контроля внутри функции myshape ∷ draw( ) позволит локализовать ошибку при выводе прямоугольника — контура фигуры myshape и, возможно, попробовать изменить его размер. В общем случае проектирование реакции программной системы на ошибки должно выполняться одновременно с проектированием её самой. Так, в программе, рассмотренной в учебном примере, можно снабдить каждую фигуру автоматически формируемым порядковым номером, значение которого можно выводить как часть сообщения об ошибке «выход за пределы экрана».

Если ошибка выявлена в конструкторе фигуры, фигура не создаётся. Для правильной обработки такой ситуации нужно дополнить библиотеку деструктором для класса shape. Вариант: заменить ошибочную фигуру специальным значком «ошибка», выводимым на картинку.

### 5.1. Практикум по теме

Дополнить программу работы с библиотекой фигур механизмом контроля исключительных ситуаций. Предусмотреть выявление следующих ошибок:

— непопадание точки на экран;

— некорректные параметры при формировании фигуры;

— нехватка места на экране для размещения фигуры.

Можно использовать в качестве базы исключений стандартное исключение exception. Aргумент его конструктора — строка сообщения об ошибке. В блоке catch эта строка может быть получена вызовом функции-члена what( ).

Организовать перехват исключений следует таким образом, чтобы искажения итоговой картинки были минимальны.

Протестировать исключительные ситуации, результаты эксперимента поместить в отчёт.

### 5.2. Требования к отчёту

В отчёте по теме обоснуйте набор и вид классов для фиксации особых ситуаций, место расположения операторов throw и блоков контроля с целью получения безопасного кода.

### 5.3. Контрольные вопросы

1. Что такое исключительная ситуация при выполнении программы?

2. Как можно выявить исключительную ситуацию?

3. Что можно предпринять в случае, когда исключительная ситуация выявлена?

4. Могут ли в одной программе появляться исключительные ситуации разных типов и если «да», то как их можно различить?

5. В каком месте программы следует поместить обработчик особых ситуаций?

6. Как можно вызвать обработчик особых ситуаций?

7. Можно ли передать в обработчик особых ситуаций какую-либо информацию о произошедшем событии?

8. Можно ли обработать неизвестную особую ситуацию?

9. Можно ли сделать обработчик ситуации пустым?

10. Что можно предпринять, если выяснилось, что для корректной обработки ситуации в данном месте программы у обработчика оказалось недостаточно данных?

11. Если требуется несколько обработчиков особых ситуаций, в каком порядке их следует размещать в программе?

12. Как действуют обработчики в случае, когда никакой особой ситуации не произошло?

13. Как следует размещать блоки контроля, чтобы получить безопасный программный код?

# 6. ИСПОЛЬЗОВАНИЕ СТАНДАРТНОЙ БИБЛИОТЕКИ ШАБЛОНОВ

Стандартная библиотека шаблонов (STL) поддерживает большинство типовых операций со структурами данных.

В первую очередь понадобятся последовательные контейнеры *vector*, *list* и *deque* и их производные (адаптеры) *stack* и *queue*, а также ассоциативные контейнеры *map*, *set*, *multimap*, *multiset* — для деревьев двоичного поиска и *unordered\_map, unordered\_set, unordered\_multimap, unordered\_multiset* — для хеш-таблиц*.* Для обработки данных следует использовать возможности библиотеки алгоритмов (*algorithm*).

Каждому контейнеру соответствует заголовочный файл, который нужно подключать директивой #*include*.

Контейнеры *map* и *set* хранят множества в виде дерева двоичного поиска с автобалансировкой (красно-чёрное дерево). Контейнер *set* хранит множество ключей, а *map* — пары <ключ, значение>, причём все ключи в них уникальны. Для множеств с повторениями используются контейнеры *multimap* и *multiset*. При просмотре всех этих контейнеров их содержимое выдаётся в виде упорядоченной последовательности (внутренний обход дерева двоичного поиска).

При просмотре *unordered* контейнеров будет выдана неупорядоченная последовательность ключей.

Возможно много вариантов приспособления контейнеров для работы с последовательностями: использование *map* (или *multimap*) вместо *set*, чтобы хранить вместе с ключами их порядковые номера, комбинирование контейнера для множеств с контейнером последовательностей (*vector* или *forward\_list*), хранящим итераторы, и т. п.

Конструкторы контейнеров позволяют уже при их объявлении сформировать множество заданной мощности. Для этого достаточно в качестве инициализатора содержимого контейнера использовать датчик случайных чисел.

 Всё необходимое для операций с контейнерами можно найти в библиотеке алгоритмов (*algorithm*). В частности, в ней имеются функции *set\_union*, *set\_intersection* и *set\_difference*, выполняющие объединение, пересечение и вычитание множеств. Функции принимают в качестве аргументов отрезки из двух контейнеров и формируют новый контейнер с результатом. В них реализуется схема слияния, поэтому входные отрезки должны быть упорядочены. Это справедливо по умолчанию для контейнеров *set*, *map* и аналогичных. Для *unordered\_set* аналогичные результаты даёт одновременный просмотр двух контейнеров с применением функций проверки наличия и вставки элемента множества в результат. В библиотеке *STL* имеются функции для выполнения любых операций с последовательностями.

Подробнее *см*. [7, с. 295–368], [8, с. 623–702], [10, с. 835–962]. Полезно также посмотреть библиотечные файлы в каталоге include компилятора C++: только там содержится исчерпывающая информация о том, какие на самом деле объявляются классы и какие функции-члены они содержат. Информация в литературных источниках, как правило, запаздывает и содержит неточности. Важно и то, что в сообщениях компилятора об ошибках обычно присутствует информация из текстов каталога include, поскольку эти тексты компилируются вместе с программой пользователя. Требуется также знакомство с механизмом шаблонов языка С++.

### 6.1. Практикум по теме

Переделать программу, составленную при выполнении темы «Последовательности», под использование контейнеров из стандартной библиотеки шаблонов. Для хранения множеств выбрать контейнер подходящего типа (*set* или *unordered\_set* и т. п.) и доработать его для поддержки операций с последовательностями. Для реализации операций с контейнерами использовать возможности библиотеки алгоритмов. Программа должна реализовывать цепочку операций над множествами в соответствии с заданием по теме 1 (2) и операций с последовательностями — по теме 3. Результат каждого шага цепочки операций выводится на экран.

### 6.2. Требования к отчёту

В отчёте по теме опишите набор подходящих контейнеров и функции STL, использованные для работы с ними. Приведите результат выполнения цепочки операций для случайного набора данных заданной мощности. Оцените предполагаемую временную сложность выполнения цепочки операций.

### 6.3. Контрольные вопросы

1. Что такое стандартный контейнер библиотеки *STL*? Чем он отличается от обычного объекта?

2. Какой стандартный контейнер можно считать наиболее подходящим для работы с множествами?

3. Можно ли использовать стандартные контейнеры для множеств, на которых не определено отношение полного порядка?

4. Существуют ли ограничения на применение стандартных алгоритмов двуместных операций над множествами в контейнерах?

5. Можно ли реализовать двуместную операцию над множествами в контейнерах без применения стандартного алгоритма?

6. Можно ли выполнять операции над последовательностями для множеств, хранящихся в стандартном контейнере?

7. Можно ли обеспечить поддержку произвольных последовательностей в контейнере для множеств?

8. Какова ожидаемая временная сложность при выполнении стандартным алгоритмом операции объединения двух множеств в стандартных контейнерах *set*?

# 7. КУРСОВАЯ РАБОТА. Измерение временной сложности алгоритма

Выполнить статистический эксперимент по измерению временной сложности алгоритма обработки данных, использующего стандартную библиотеку шаблонов (тема 6).

Доработать программу таким образом, чтобы она генерировала множества мощностью, меняющейся, например, от 10 до 200, измеряла время выполнения цепочки операций над множествами и последовательностями и выводила результат в текстовый файл. Первая строка этого файла должна содержать количество опытов, а последующие — по паре значений «размер входа — время» для каждого опыта. Затем эти данные обрабатываются, и по результатам обработки делается заключение о временной сложности алгоритма.

Для повышения надёжности эксперимента предусмотреть в программе перехват исключительных ситуаций таким образом, чтобы сбой сводился просто к пропуску очередного шага эксперимента. В частности, рекомендуется перехватывать ситуацию *bad\_alloc*, возбуждаемую конструктором при нехватке памяти.

### 7.1. Пример программы для эксперимента

//Demo\_01.cpp: Измерение времени обращением к счётчику тактов // процессора.

// Эксперимент по сбору статистики для оценки временной сложности

// Демонстрационная программа--------------- (c) Clgn, 17.05.13

#include "stdafx.h"

#include <stdlib.h>

#include <conio.h>

#include <fstream>

#include <iostream>

#include <intrin.h>

#include <algorithm>

#include <set>

using namespace std;

typedef set<int> MySet;

typedef set<int>::const\_iterator MyIt;

int set\_make(const int p, MySet & A)

{

 A.clear( );

 for (int i = 0; i < p; i++)

 A.insert(rand( )%1000);

 //sort(A.begin( ), A.end( ));

return A.size();

}

void set\_out(char N, MySet & A)

{ size\_t p = A.size( );

 cout << "\n" << N << '(' << p << ")=[ ";

 for(MyIt i = A.begin(); i != A.end( ); ++i) cout << \*i << ' ';

 cout << ']';

}

int set\_and(const MySet & A, const MySet & B, MySet & C)

{

 set\_intersection<MyIt, MyIt, insert\_iterator<MySet> >(A.begin( ), A.end( ), B.begin( ), B.end( ), inserter(C, C.begin()));

/\* Вариант: MyIt a = A.begin( ), b = B.begin( );

 C.clear( );

 while( (a != A.end( )) && (b != B.end( )) )

 { if(\*a < \*b) ++a;

 else if (\*b < \*a) ++b;

 else C.insert(\*a), ++a, ++b;

 }

\*/

 return C.size();

 }

int set\_or(const MySet & A, const MySet & B, MySet & C)

{

 set\_union<MyIt, MyIt, insert\_iterator<MySet> >(A.begin( ), A.end( ), B.begin( ), B.end( ), inserter(C, C.begin( )));

 /\* Вариант: MyIt a = A.begin( ), b = B.begin( );

 C.clear( );

 while( (a != A.end( )) && (b != B.end( )) )

 { if(\*a < \*b) C.insert(\*a), ++a;

 else if(\*b < \*a) C.insert(\*b), ++b;

 else C.insert(\*a), ++a, ++b;

 }

 while( a != A.end( ) )C.insert(\*a), ++a;

 while( b != B.end( ) )C.insert(\*b), ++b;

 \*/

 return C.size();

 }

void set\_sub(const MySet & A, const MySet & B, MySet & C)

{ set\_difference<MyIt, MyIt, insert\_iterator<MySet> >(A.begin( ), A.end( ), B.begin( ), B.end( ), inserter(C, C.begin( )));

/\* MyIt a = A.begin( ), b = B.begin( );

 C.clear( );

 while( (a != A.end( )) && (b != B.end( )) )

 { if (\*a < \*b) C.insert(\*a), ++a;

 else if(\*b < \*a) ++b;

 else ++a, ++b;

 }

 while( a != A.end( ) ) C.insert(\*a), ++a;

 \*/

 }

int main( )

{

 ofstream out("in.txt"); // Выходной файл in.txt

 if(!out) { cout << "\nОшибка открытия выходного файла!"; return(1); }

 unsigned \_\_int64 t1, t2;

 int p=5, q=205, dp=2; //Параметры эксперимента

 MySet A, B, C, D, E;

 out << static\_cast<int>((q - p) / dp + 1); // Количество опытов

 do { // Операции над множествами мощностью p

 try {

 size\_t k = 0, sets = 0;

 k += set\_make(p, A); sets++;

 k += set\_make(p, B); sets++;

 k += set\_make(p, C); sets++;

 k += set\_make(p, D); sets++;

 set\_out('A', A);

 set\_out('B', B);

 set\_out('C', C);

 set\_out('D', D);

 // A.clear();

 // A.insert<vector<int> :: iterator>(Q.begin( ), Q.end( ));

 t1 = \_\_rdtsc( );

 k += set\_and(A, B, E); sets++;

// set\_out('E', E);

 k += set\_or(C, E, A); sets++;

// set\_out('E', A);

 k += set\_sub(A, D, E); sets++;

//...

 t2 = \_\_rdtsc( );

 set\_out('E', E);

 k /= sets;

 cout << "\np=" << p << " k=" << k << " Dt=" << t2 - t1;

 out << '\n' << k << ' ' << t2 - t1 ;

 }

 catch(...) { cout << "\nСбой"; }

 } while ((p += dp) <= q);

 cout << "\n\n=== The End ===\n"); system("pause");

 return 0;

}

### 7.2. Обработка результатов эксперимента

По результатам эксперимента выполняется регрессионный анализ: подбор уравнения регрессии, наиболее соответствующего полученным данным. Для выполнения этого пункта следует получить у преподавателя набор данных «*stat*», состоящий из программы *RG32*, файлов с примерами её входа и выхода (*in.txt*, *out.txt*) и заготовки электронной таблицы *EXAMPLE*.

Программа *RG32* представляет собой консольное *Windows*-приложение и запускается из-под интерфейса командной строки (*CMD*). В качестве аргумента программа принимает имя файла с результатами измерений. Имя и расширение этого файла могут быть любыми. Рекомендуется поместить программу и файл измерений в один каталог. Если файл назван «*IN.txt*», строка запуска будет выглядеть так:

RG32 in.txt

В программе *RG32* имя входа «in.txt» принято по умолчанию и его можно опустить. В этом случае программу *RG32* можно запускать и в окне проводника.

Первая строка текста с данными измерений содержит общее количество опытов. Каждая из последующих строк содержит результат одного опыта: размер входа и время, разделённые хотя бы одним пробелом. Упорядочивать данные по размеру входа необязательно. Можно выполнить по несколько опытов для некоторых или для всех размеров входа. Если общее количество опытов окажется меньше указанного в первой строке, будет сделано предупреждение об этом, и программа использует фактически имеющееся количество. Если данных окажется больше – лишние игнорируются. Программа откажется работать, если опытов менее 10.

Программа *RG32* выдаст на экран значения коэффициентов регрессии для набора функций, начиная от константы (отсутствие регрессии) и кончая полиномом четвёртой степени. Общий вид уравнения регрессии выводится программой в первой строке. Далее выводятся результаты подбора с поочерёдным подключением коэффициентов слева направо. Не используемые в уравнении коэффициенты выводятся нулями. Кроме коэффициентов даётся значение выборочной дисперсии (и среднеквадратичного отклонения —СКО). Результат работы программы следует включить в отчёт в виде таблицы.

Одновременно с выводом на экран программа *RG32* создаёт текстовый файл *OUT.txt*, пригодный для импорта в электронную таблицу *EXCEL*.

Для упрощения обработки результатов следует получить у преподавателя заготовку электронной таблицы *EXAMPLE.xls*. В эту таблицу импортируются файлы с результатами измерений и с коэффициентами уравнений регрессии — ввод и вывод программы *RG32*. Данные из файлов переносятся на отведённые для них места. Программа *RG32* может завершиться аварийно (не выдать результата), если данных недостаточно или они недостоверны.

Вся необходимая обработка данных уже заложена в таблицу *EXAMPLE*. Содержимое файла с измерениями и файла *OUT.txt* нужно импортировать на свободное место в таблице (закладка «Данные», команда «Из текста»), а затем скопировать на штатное: измерения — в две крайние левые колонки, а коэффициенты уравнений регрессии — на соответствующее место в правой части.

Для корректного построения графиков измерения должны быть упорядочены по возрастанию мощности (выделить область с измерениями, в закладке «Данные» выполнить команду «А->Я»).

Рядом с колонками результатов эксперимента находятся расчёты значений уравнений регрессии, по которым уже построены графики, а правее места расположения коэффициентов регрессии находится расчёт отношений каждой выборочной дисперсии ко всем остальным. По значениям отношений дисперсий можно сразу указать наиболее подходящее уравнение регрессии, самый старший коэффициент которого определит временную сложность алгоритма. Наиболее подходящим будет то уравнение, при дальнейшем усложнении которого уменьшение выборочной дисперсии прекращается или перестаёт быть значимым.

Поскольку выборочные дисперсии являются случайными величинами. значения дисперсий можно считать различными только в том случае, если их отношение превосходит значение квантиля распределения Фишера (рис. 7.1).

*Рис*. *7.1.* Квантили распределения Фишера

Входом в таблицу является количество степеней свободы выборки, равное количеству опытов минус количество оцениваемых коэффициентов регрессии. Во второй строке дано значение отношения выборочных дисперсий, которое можно считать значимым с ошибкой не более 5 %, в третьей — не более 1 %.

Если с учётом уровня значимости из всех полученных уравнений регрессии невозможно однозначно указать наиболее подходящее, следует попробовать увеличить количество опытов.

### 7.3. Оформление результатов эксперимента

Результаты эксперимента следует оформить в виде графика зависимости времени решения задачи как функции размера входа. На графике должны быть представлены:

— точки, соответствующие измеренным значениям;

— кривая регрессии для уравнения, отобранного по результатам сравнения выборочных дисперсий;

— границы доверительного интервала (регрессия ±3 СКО).

Значения СКО выдаются программой *RG32*. Ожидается, что все измеренные значения попадут в доверительную область (с вероятностью 99,7 %), а кривая регрессии будет усреднять их.

График можно взять из таблицы *EXAMPLE*. Возможно, таблицу придётся подкорректировать под фактический объём обработанных измерений (размножить строки в расчётной части и исправить диапазоны данных в графиках. Достаточно доработать только тот график, который пойдёт в отчёт.

### 7.4. Выводы

По итогам эксперимента даётся заключение о том, насколько экспериментальная оценка временной сложности алгоритма обработки данных соответствует теоретической. Исследуются особенности реализации алгоритмов, позволяющие объяснить полученные результаты.

## Список литературы

1. Седжвик Р. Алгоритмы на С++ / пер. с англ. — М.: И. Д. Вильямс, 2011. — 1156 с.: ил.

2. Седжвик Р. Фундаментальные алгоритмы С++. — К.: Диасофт, 2001. — 484 с.

3. Ахо Дж., Хопкрофт А., Ульман Дж. Построение и анализ вычислительных алгоритмов. — М.: Мир, 1979.

4. Кнут Д. Искусство программирования для ЭВМ. Т. 3: Сортировка и поиск. — М.: Мир, 2013.

5. Новиков Ф. А. Дискретная математика: учеб. для вузов. 2-е изд. Стандарт третьего поколения. — СПб.: Питер, 2013. — 432 с.: ил.

6. Страуструп Б. Язык программирования С++. 2‑е доп. изд. — М.: Бином-пресс, 2001. – 1098 с.

7. Павловская Т. А. С/С++. Процедурное и объектно-ориентированное программирование: Учеб. для вузов. — СПб.: Питер, 2015. — 496 с.: ил.

8. Шилдт Г. Полный справочник по С++. 4-е изд. Пер. с англ. — М.: И. Д. Вильямс, 2004. — 800 с.: ил.

9. Страуструп Б. Программирование: принципы и практика использования С++. Испр. изд. / пер. с англ. — М.: И. Д. Вильямс, 2013. —1248 с.: ил.

10. Прата С. Язык программирования C++. 6‑е изд. — М.: И. Д. Вильямс, 2011. — 1244 с.: ил.

11. Липпман С. Б., Лакойе Ж., Му Б. Э. Язык программирования С++. Базовый курс. 5-е изд. Пер. с англ. — М.: И. Д. Вильямс, 2014. — 1120 с.: ил.

12. Дьюхэрст Стефан К. Скользкие места С++. Как избежать проблем при проектировании и компиляции ваших программ. — М.: ДМК Пресс, 2012. — 264 с.: ил.

13. Мейерс С. Наиболее эффективное использование С++. 35 новых рекомендаций по улучшению ваших программ и проектов. — М.: ДМК Пресс, 2012. — 294 с.: ил.

14. Саттер Г. Решение сложных задач на С++. Пер. с англ. — М.: И. Д. Вильямс, 2015. — 400 с.: ил.

15. Джоссатис Н. М. Стандартная библиотека С++: справочное руководство. 2-е изд. Пер. с англ. — М.: И. Д. Вильямс, 2014. — 1136 с.: ил.

## ПРИЛОЖЕНИЕ

## Измерение времени запросом внутреннего счётчика тактов процессора

Современные процессоры (начиная с *Pentium* II последних серий) поддерживают команду *RDTSC*, возвращающую 64-битное значение внутреннего счётчика тактов. Это достойная альтернатива применению функции *clock*( ): можно измерить время выполнения даже одной команды процессора. Надёжный способ добраться до счётчика тактов состоит в применении ассемблерной вставки в код на С++. Можно написать функцию, аналогичную *clock*( ). В программе под *Borland* C++ 3.1 эта функция может возвращать отсчёт в формате *double*, как наиболее информативном, и дублировать его массивом из 8 байт, содержащим само значение отсчёта для дополнительного контроля. Функция может выглядеть так:

#include <dos.h>

#include <mem.h>

double Ti (unsigned char \*u) // u – массив из 8 байт для контроля

{ struct W { unsigned long P, Q; }; // Структура из двух длинных целых

 union { unsigned char tb[8]; // Объединение структуры и массива байтов

 W tt; } T;

asm { // Ассемблерная вставка:

 // команда RTDSC не поддерживается, задана кодом

 db 0x0f, 0x31, 0x66; mov WORD PTR T.tt.P, AX; // Младшие 32 бита

 db 0x66; mov WORD PTR T.tt.Q, DX; // Старшие 32 бита

 }

 memcpy(u, T.tb, 8); // Копирование в контрольный массив

 return (double) T.tt.P/65536 + T.tt.Q\*65536; // Вычисление результата

}

В оболочках *Visual* C++ 6.0 и более современных (32-битных) ассемблерная вставка будет выглядеть иначе: нет нужды задавать команду *RTDSC* кодом и использовать префиксы 32-битной операции.

asm { // Ассемблерная вставка для 32-битной программной оболочки

 RTDSC; mov DWORD PTR T.tt.P, EAX; // Младшие 32 бита

 mov DWORD PTR T.tt.Q, EDX; // Старшие 32 бита

 }

Этим способом время измеряется с максимально возможной точностью. Способ хорош не только тем, что не требует зацикливания исследуемого фрагмента программы. Он позволяет исключать из измеряемого интервала части кода, не относящиеся собственно к алгоритму, например, генерацию или вывод данных.

В системах программирования *Visual* С++ от *Microsoft* можно также воспользоваться функцией *QueryPerformanceCounter*(*long \*lpPerformanceCount*) из библиотеки *winbase.h*

Следует отметить, что частота, на которой работает процессор в современных ЭВМ, особенно в портативных, не постоянна, она может искусственно уменьшаться для экономии ресурса аккумуляторной батареи. Запросить текущую частоту можно с помощью функции *QueryPerformanceFrequency*(*long* \**lpFrequency*). Функция формирует указатель на количество тактов процессора в секунду.

Подробности — в справочнике *MSDN* (*https://msdn.microsoft.com/en-us/library/ms644904.aspx*).

Можно использовать функции из класса *std∷chrono∷highresolutionclock*( ) из библиотеки *STL*. Этот способ особенно рекомендуется использовать в системах программирования, работающих не под *MS Windows*,

Ещё один способ измерения времени использован в примере программы для статистического эксперимента.

## Вопросы к экзамену

1. Способы хранения множеств, облегчающие произвольный доступ к данным.

2. Значение упорядоченности для двуместных операций над множествами.

3. Дерево двоичного поиска как расширяемый список с сохранением упорядоченности.

4. Упорядоченный массив как способ хранения дерева двоичного поиска.

5. Хеширование как обобщение для хранения множества в массиве битов.

6. Открытые и закрытые хеш-таблицы: способы разрешения коллизий.

7. Рекомендации по составлению хеш-функций.

8. Двуместные операции с множествами в форме хеш-таблиц.

9. Дерево двоичного поиска: алгоритмы вставки и удаления.

10. Вырождение ДДП при произвольной вставке последовательности узлов.

11. АВЛ-дерево: алгоритмы балансировки при вставке и удалении.

12. Красно-чёрное дерево: балансировка при вставке и удалении.

13. 2-3-дерево. Алгоритмы вставки и удаления. Рекомендация по кодированию в виде варианта ДДП.

14. Получение ДДП с автобалансировкой из упорядоченного массива.

15. Получение 2-3-дерева из упорядоченной последовательности.

16. Двуместные операции с множествами в форме ДДП.

17. Алгоритм пошагового обхода ДДП.

18. Множества с повторениями в хеш-таблице, ДДП, АВЛ-дереве, 2-3-дереве.

19. Множество и последовательность. Хранение последовательностей в структуре данных для множеств. Варианты: обход, разметка, связи, массив ссылок.

20. Алгоритм реализации двуместной операции с АВЛ-деревьями: схема слияния с использованием массивов ссылок на узлы дерева.

21. Алгоритм слияния для двуместной операции: чистка хвостов и восстановление дерева-результата.

22. Иерархия классов. Производные классы — что это даёт. Функции-члены. Конструкторы и деструкторы. Пример иерархии классов.

23. Управление доступом к объектам в иерархии. Виртуальные функции. Абстрактные классы. Идентификаторы override и final.

24. Пример программы с иерархией классов. Монитор экрана. Библиотека фигур. Возможная иерархия классов и её смысл.

25. Множественное наследование. Множественное вхождение базового класса. Разрешение неоднозначности.

26. Закрытое наследование и его альтернатива — включение.

27. Виртуальные базовые классы.

28. Контроль доступа к членам базовых классов. Защищённые члены.

29. Шаблоны. Обобщённые функции.

30. Шаблоны. Обобщённые классы.

31. Исключительные ситуации: основные понятия.

32. Исключительные ситуации: иерархия и способы передачи информации в обработчик.

33. Исключительные ситуации — неожиданные. Возможности для их обработки.

34. Проектирование системы обработки особых ситуаций. Базовые и расширенные гарантии.

35. Идиома «захват ресурса через инициализацию».

36. Функции, не генерирующие исключений. Функции, прозрачные для исключений.

37. Преобразования типов: в стиле Си, статические, константные, произвольные.

38. Преобразования типов: динамические; неожиданные. Блокировка неявного вызова конструктора или функции преобразования типа.

39. Перегрузка функций для предотвращения преобразования типа. Функции-члены для преобразования типа.

40. Свободная память (перегрузка new и delete). Явное указание размещения. Размещение, не вырабатывающее исключений.

41. STL: Библиотека string как (плохой) образец контейнера.

42. STL: Перегрузка разыменования и взятия адреса. Итераторы: сущность и классификация.

43. STL: Функциональные объекты: указатели, функторы, лямбда.

44. STL: Полезные алгоритмы. Особенности применения библиотеки algorithm.

45. Последовательные контейнеры: vector, list, deque.

46. Адаптеры последовательного контейнера: стек и очередь; очередь с приоритетами.

47. Ассоциативные контейнеры — краткий обзор. Множество и отображение.

48. Ассоциативные контейнеры — мультимножество и мультиотображение.

49. Контейнеры — хеш-таблицы.

50. Умные указатели: уникальный указатель.

51. Умные указатели: разделяемый и слабый указатели.

52. Измерение временной сложности алгоритма с помощью ЭВМ.

**СОДЕРЖАНИЕ**

ВВЕДЕНИЕ 3

1. ХЕШ-ТАБЛИЦЫ 4

1.1. Практикум по теме 5

1.2. Требования к отчёту 7

1.3. Контрольные вопросы 7

2. ДЕРЕВЬЯ ДВОИЧНОГО ПОИСКА 8

2.1. Практикум по теме 10

2.2. Требования к отчёту 11

2.3. Контрольные вопросы 11

3. ПОДДЕРЖКА ПРОИЗВОЛЬНОЙ ПОСЛЕДОВАТЕЛЬНОСТИ В СТРУКТУРЕ ДАННЫХ ДЛЯ МНОЖЕСТВ 12

3.1. Практикум по теме 13

3.2. Требования к отчёту 15

3.3. Контрольные вопросы 15

4. РАБОТА С ИЕРАРХИЕЙ ОБЪЕКТОВ: НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ 16

4.1. Учебная программа «Библиотека фигур» 17

4.2. Практикум по теме 24

4.3. Требования к отчёту 29

4.4. Контрольные вопросы 29

5. ПОДДЕРЖКА ОБРАБОТКИ ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ 30

5.1. Практикум по теме 32

5.2. Требования к отчёту 32

5.3. Контрольные вопросы 32

6. ИСПОЛЬЗОВАНИЕ СТАНДАРТНОЙ БИБЛИОТЕКИ ШАБЛОНОВ 34

6.1. Практикум по теме 35

6.2. Требования к отчёту 35

6.3. Контрольные вопросы 36

7. КУРСОВАЯ РАБОТА. Измерение временной сложности алгоритма 37

7.1. Пример программы для эксперимента 37

7.2. Обработка результатов эксперимента 40

7.3. Оформление результатов эксперимента 43

7.4. Выводы 43

Список литературы 44

ПРИЛОЖЕНИЕ. Измерение времени запросом внутреннего счётчика тактов процессора 45

Редактор И. Г. Скачек

––––––––––––––––––––––––––––––––––––––––––––––––––––––––––

Комплектация 1 CD-ROM. Объём данных 1,3 Мб.

Гарнитура «Times New Roman». Заказ

––––––––––––––––––––––––––––––––––––––––––––––––––––––––––

Издательство СПбГЭТУ «ЛЭТИ»

197376, С.-Петербург, ул. Проф. Попова, 5