МИНОБРНАУКИ РОССИИ

Санкт-Петербургский государственный
электротехнический университет «ЛЭТИ»

П. Г. Колинько

ПОЛЬЗОВАТЕЛЬСКИЕ СТРУКТУРЫ ДАННЫХ

Методические указания
по дисциплине «Алгоритмы и структуры данных, часть 1».
Вып. 2408

Санкт-Петербург
СПбГЭТУ «ЛЭТИ»
2024

УДК 004.424:004.422.63(075.8)

Колинько П. Г. Пользовательские структуры данных: Методические указания по дисциплине «Алгоритмы и структуры данных, часть 1». — СПб.: СПбГЭТУ «ЛЭТИ», 2024. — 64 с. (вып.2408).

Описывается цикл зачётных самостоятельных работ на ПЭВМ. Содержатся материалы для курсовой работы.

Пособие предназначено для студентов бакалавриата по направлению 09.03.01 «Информатика и вычислительная техника» всех форм обучения.

Одобрено
Методической комиссией факультета
информатики и вычислительной техники
СПбГЭТУ «ЛЭТИ»
в качестве методических указаний

ВВЕДЕНИЕ

Цель курса «Алгоритмы и структуры данных» — развитие навыков программирования, полученных студентами при изучении предмета «Программирование» и теоретических сведений из курса «Дискретная математика». Используются также сведения из параллельно изучаемых курсов «Математическая логика и теория алгоритмов» и «Теория вероятностей и математическая статистика». Основное внимание уделяется изучению способов реализации в ЭВМ абстрактных данных и вытекающих из этих способов свойств алгоритмов обработки этих данных. В качестве примеров рассматриваются популярные алгоритмы на ненагруженных и нагруженных графах, жадные алгоритмы, эмпирические алгоритмы для переборных задач. Изучаются способы организации данных в реальных задачах, когда к одному и тому же набору данных могут применяться одновременно несколько абстрактных моделей.

Настоящее пособие покрывает первый семестр двухсеместрового курса и состоит из четырёх разделов. Каждый раздел соответствует четырём учебным неделям.

Тема первого раздела «Множества» является вводной. В ней показывается, что абстрактные данные могут быть реализованы в программе разными способами и что от способа реализации зависит существенная характеристика алгоритма — его временная сложность. Во второй теме вводится понятие класса как естественного расширения языка C++ для поддержки пользовательских типов данных.

Изучение обеих тем разбито на этапы по схеме от простого — к сложному. Самостоятельная работа состоит в изучении учебных примеров, имеющихся в лекциях или прилагаемых к ним, а также в постановке опытов с программным кодом и исследовании алгоритмов.

Третья тема «Деревья» акцентирует внимание студентов на свойствах рекурсивного определения данных и рекурсивных алгоритмов. Она предусматривает также совершенствование техники работы с объектами: создание и уничтожение, копирование, совместное использование в программе объектов разных типов (дружественные функции) и т. п. Вводится понятие шаблона функции и класса, в качестве иллюстрации для применения которого используются абстрактные данные «очередь» и «стек».

Четвёртая тема «Графы» выносится на курсовое проектирование для закрепления навыков, полученных при изучении трёх первых тем.

На каждую тему отводится один месяц, в течение которого студенты должны подготовить и продемонстрировать в работе программы для эксперимента

и сдать отчёт (в электронном виде) по теме с указанием, какие эксперименты проведены и какие свойства структур данных фактически наблюдались.

Объём теоретических сведений об абстрактных данных и алгоритмах в методических указаниях минимален. Практические работы должны быть итогом изучения курса лекций, доступного также в электронном виде на сайте vec.etu.ru. Дополнительная информация может быть получена из параллельно изучаемых курсов «Дискретная математика» и «Математическая логика и теория алгоритмов», а также из рекомендованной литературы.

Предполагается, что студенты уже знакомы с такими элементарными структурами данных, как массивы, списки, очереди и стеки.

Все примеры проверены в оболочке Visual C++ 2017, используемой в учебном процессе СПбГЭТУ «ЛЭТИ». Для самостоятельной работы можно использовать компилятор из пакета Visual Studio 2017, 2019 или 2022 от Microsoft, который можно получить с сайта *visualstudio.microsoft.com* (рекомендуется вариант "*community*", доступный без лицензии, достаточно зарегистрироваться на сайте, в том числе — в «личном кабинете» пользователя *Windows* 10). Рекомендуемая альтернатива — оболочка *Code-Blocks*. Для её использования нужно с сайта *www.codeblocks.org -> Downloads -> Download the binary release* скачать и установить *codeblocks*-20.*03mingw-setup.exe* и при первом запуске *Code*::*Blocks* (сразу после установки) в главном меню выбрать «*Setting -> Compiler*…» и пометить галочкой пункт «*Have g++ follow* the *coming C++1z* (*aka C++*17) *ISO C++ language standard*».

Предполагается использование компьютеров по управлением *Windows*-10 и оформление зачётных работ с помощью *MS Word* 2013, 2016 или 2019. Можно использовать компьютеры под управлением различных вариантов *Linux*, а также *MAC OS*. Для этих ОС доступны свои версии как *Visual Studio*, так и *Code-Blocks*. Отчёты могут быть оформлены с помощью пакета *Open Office* или *Libre Office* (или аналогичных средств *MAC OS*) и сданы в формате *odt* или *pdf*.

Актуальную справку о языке *C++*11/14/17/20/23, шаблонах, библиотечных функциях с примерами их использования можно получить на сайте *ru.cppreference.com*.

Адрес для консультаций и сдачи зачётных работ: **valencem@mail.ru**

Тема 1. МНОЖЕСТВА

Множество — абстрактная структура данных, для которой определена операция принадлежности. Новые множества создаются из существующих с помощью операций объединения, пересечения, разности, симметрической разности.

Цель работы: сравнительное исследование четырёх способов хранения множеств в памяти ЭВМ.

Содержание работы: поэтапная разработка программы для обработки множеств четырьмя способами и тестирование её сначала на тесте-константе, потом — на тесте, вводимом с клавиатуры, затем — на тестах, генерируемых машинным способом, и измерение времени, необходимого для решения задачи каждым из способов. Предлагаемый набор этапов — универсальный, опытные программисты могут его сократить и сразу перейти к автоматической генерации тестов с измерением времени, по результатам которой готовится отчёт.

1.1. Представление множества набором элементов

Одним из способов задания множества является простое перечисление входящих в него элементов. В памяти ЭВМ элементы такого множества могут быть расположены в последовательности ячеек, т. е. образуют массив. Это самый экономный способ хранения множества в тех случаях, когда мощность множества известна, а его элементы — данные одного типа (во всех случаях, когда это не так, элементы множества можно расположить в памяти произвольным образом и работать с ними через указатели). Память под массив удобно выделить статически. В этом случае её можно сразу инициализировать.

Пример. Объявление и инициализация массива для множества десятичных цифр.

```
const int Nmax = 10;
char A[ Nmax+1 ] = {'1', '3', '4', '7'};
```

В память, выделяемую под универсум, помещено множество-константа. Множество символов предполагается обрабатывать как строку, поэтому пришлось позаботиться о месте для ограничивающего нуля (компилятор заполнит нулями не использованную часть массива). Если множество расширяться не будет, размер памяти можно не указывать:

```
char A[ ] = {"1347"};
```

При инициализации массива строкой выделяется память под множество (из четырёх элементов) и ограничивающий нуль.

Мощность множества, заданного константой, при желании можно

вычислить:

```
int nA = strlen(A);
```

А можно этого не делать: обрабатывать массив A до появления нуля.

Если множество создаётся в результате вычислений, его мощность может быть неизвестна. Поскольку память под массив должна быть выделена до начала работы с ним, приходится делать это с запасом. Проще всего выделить память сразу под универсум, а если универсум слишком велик, то под множество максимальной мощности, которое может быть реально получено. Если же память оказалась исчерпана до окончания вычислений, можно попытаться заказать область памяти большего размера и перенести в неё накопленные элементы множества. Это дорогая операция, поскольку приходится полностью копировать множество. Она может быть невыполнима, если в памяти нет непрерывного участка достаточного размера. Освобождаемый участок памяти тоже не всегда можно использовать. Если же оценка мощности результата слишком пессимистична, память под множество используется нерационально. Так, её приходится выделять даже для результата — пустого множества.

Таких проблем нет, если для представления множества в памяти используется односвязный список. Память в этом случае выделяется под каждый элемент множества отдельно, и её ровно столько, сколько необходимо. Так, под пустое множество-список память вообще не выделяется. Память от удаляемых элементов списка легко использовать под вновь создаваемые элементы. Перенос элемента из одного множества в другое вообще не требует дополнительной памяти.

Главный недостаток способа — необходимость хранить с каждым элементом множества указатель на следующий элемент и тратить время на работу с ним. Так, создание копии списка в другом месте памяти требует не только отдельной обработки каждого элемента множества, но и создания вновь всех указателей. Имеются и скрытые потери: минимальная область, выделяемая в динамической памяти, часто больше, чем требуется для хранения одного элемента множества вместе с указателем.

Реализация алгоритмов для работы с множествами, представленным массивами или списками, отличается только способом перебора этих структур данных. Подтвердим это примером реализации операции принадлежности элемента множеству десятичных цифр.

Для массива используем объявление из предыдущего примера. Множество-список объявим так:

```
struct Set { char el;
```

```
  Set * next;
  Set(char  e, Set * n = nullptr) : el(e), next(n) { }
  ~Set( ) { delete next; }
}
Set *LA;    // Указатель на начало списка для множества A.
```

Результатом вычислений будет значение булевской переменной *b*.

```
bool b = false;
char s = cin.get( ); // Символ вводится с клавиатуры
for (int i = 0; A[ i ]; ++i) // Вариант 1: перебор элементов массива
    if (A[ i ] == s) b = true;
for (Set * p = LA; p; p = p->next) // Вариант 2: просмотр списка
  if (p->el == s) b = true;
```

Очевидно, что оба варианта реализации имеют линейную временную сложность по мощности множества: $O(nA)$. Оценка не меняется и в том случае, если при обнаружении элемента в множестве цикл прерывается.

1.1.1. Практикум по теме

Составить и отладить программу, реализующую обработку множеств по предложенному заданию (табл. П.2.1).

1. Уточнить задание: записать его в виде формулы для получения пятого множества по заданным четырём, используя знаки операций над множествами. Результат может выглядеть так:

$$E = A \cup B \cap C \setminus D$$

2. Предложить контрольный тест в соответствии с заданным типом универсума, например, такой

$$A = \{a, b, c, d, e\}$$
$$B = \{b, d, f\}$$
$$C = \{d, f, g, h\}$$
$$D = \{a, c, i\}$$
$$E = ? \text{ (вычислить результат!)}$$

3. Составить программу для вычисления пятого множества по четырём заданным, используя для представления множеств в памяти массивы символов. Для задания исходных множеств использовать инициализацию. Добиться прохождения теста.

4. Добавить в программу ввод исходных множеств и протестировать её на нескольких тестах: на пустых, полных, не пересекающихся, совпадающих множествах и т. п. Рекомендуется вводить символы строкой в стиле Си, используя

для этого буфер подходящего размера:

```
char A[80];  cout << "A= "; cin >> A;
```

5. Дополнить программу так, чтобы исходные множества из массивов преобразовывались в линейные списки, и получение результата достигалось обработкой этих списков.

1.1.2. Контрольные вопросы

1. Какой объём памяти нужно выделить под массив-результат?

2. Можно ли сэкономить память, если обрабатывать множества не попарно, а все четыре сразу?

3. Какова временная сложность получения результата одновременной обработкой четырёх множеств?

4. Какая временная сложность получилась у вас? Можно ли считать ваш алгоритм оптимальным?

1.2. Представление множества отображением на универсум

Если элементы универсума упорядочить, т. е. представить в виде последовательности $U = <u_0, u_1, u_2 \ldots u_{m-1}>$, то любое его подмножество $A \subseteq U$ может быть задано вектором логических значений $C = <c_0, c_1, c_2 \ldots c_{m-1}>$, где $c_i = \{u_i \in A\}$, или вектором битов

$$c_i = \begin{cases} 1, u_i \in A \\ 0, u_i \notin A \end{cases}$$

Такой способ представления множеств в памяти имеет практическое значение, если мощность универсума $m = |U|$ не очень велика и существует простая функция $f: U \to [0 \ldots m-1]$ отображения элемента множества в соответствующий ему порядковый номер бита.

Так, например, если U — множество десятичных цифр, подходящей функцией будет $f(a) = a - '0'$, где a — символьная переменная с кодом цифры, поскольку известно, что коды цифр образуют монотонную последовательность. Аналогично, для множества прописных латинских букв можно взять $f(a) = a - 'A'$. Для шестнадцатеричных цифр, коды которых образуют два интервала, функция будет сложнее: $f(a) = a \le '9' \ ? \ a - '0' \ : \ a - 'A' + 10$. В общем случае можно работать с полным множеством символов, положив $m = 256$ и $f(a) = a$. В общем случае можно создать словарь из элементов универсума: char U[ ] = "0123456789ABCDEF". Элемент множества *s* по номеру бита *i* можно получить из словаря непосредственно: char s = U[i]. Обратное преобразование не так

удобно, потребуется поиск символа в словаре — за линейное (или логарифмическое, если применить дихотомию) время.

Операции над множествами в форме вектора битов сводятся к логическим операциям над соответствующими битами множеств. Для вычисления объединения $A \cup B$ следует выполнить $a[i] \,||\, b[i]$, для пересечения $A \cap B$ — $a[i]$ && $b[i]$, для разности $A \setminus B$ — $a[i]$ && !$b[i]$ для всех битов от $i = 0$ до $i = m - 1$. Следовательно, временная сложность двуместной операции с множествами A и B в форме вектора битов будет $O(m)$, что при фиксированном m соответствует $O(1)$, т. е. не зависит от мощности этих множеств.

Для получения вектора битов *bA* из строки символов *A* следует заполнить вектор *bA* нулями, а затем установить в 1 биты, соответствующие каждому символу из *A*:

```
for (int i = 0; A[ i ]; ++i) bA [ f ( A[ i ] )] = 1.
```

Обратное преобразование очевидно:

```
for (int i = 0, k = 0; i < m; ++i) if ( bA[ i ] ) A[ k++ ] = f -1( i ),
```

где $f^{-1}(i)$ — функция, обратная для $f(a)$. Так, если $f(a) = a - '0'$, то $f^{-1}(i) = i + '0'$.

Использование массива битов в качестве промежуточной памяти при работе с элементами множества — самый простой способ устранить дубликаты.

Вектор битов может быть представлен в памяти в компактной форме — форме машинного слова, в качестве которого на языке C++ могут использоваться переменные типа *char, int, long* или *long long*. Для таких переменных в языке предусмотрены поразрядные логические операции:

— логическое сложение (поразрядное «ИЛИ») $A \mid B$, реализующее объединение множеств $A \cup B$;

— логическое умножение (поразрядное «И») $A \,\&\, B$ — пересечение $A \cap B$;

— поразрядное сложение по модулю 2 (исключающее «ИЛИ», сравнение кодов) $A \,\hat{}\, B$ — симметрическая разность $A \oplus B = (A \cup B) \setminus (A \cap B)$;

— инвертирование ~*A*, соответствующее $\bar{A}$ — дополнению до универсума.

Операции над множествами в форме машинного слова выполняются за один шаг алгоритма независимо от мощности множеств, т. е. имеют временную сложность $O(1)$. Например, вычисление $E = (A \cup B \cap C) \setminus D$ реализуется оператором $wE = (wA \mid wB \,\&\, wC) \,\&\, \sim wD$, где *wA*, *wB*, *wC*, *wD* — машинные слова, хранящие соответствующие множества.

Способ применим, если размер универсума *m* не превосходит разрядности переменной (8 для *char*, 16 для *short*, 16 или 32 для *int*, 32 для *long*, 64 для *long long*).

Если $m > 64$, можно использовать несколько слов. Если m не равно в точности 64 (32, 16 или 8), часть битов слова не используется. Обычно это не вызывает проблем. Исключение: если переменная сравнивается с нулём для выявления пустого множества, нужно, чтобы неиспользуемые биты содержали 0.

Недостаток способа — в отсутствии удобного доступа к каждому биту машинного слова, как к элементу массива. Вместо этого приходится генерировать множество из одного элемента $\{a\}$ сдвигом 1 на $f(a)$ битов влево. Далее с помощью поразрядного «ИЛИ» можно добавить элемент в множество, а с помощью поразрядного «И» — проверить его наличие в нём. Так, для преобразования множества из строки символов в машинное слово можно использовать алгоритм

```
wA = 0;
for ( int i = 0; A[ i ]; ++i ) wA |= (1 << f(A));
```

Примечание. Если программа пишется для компилятора, поддерживающего разрядность данных *int* — 16, а мощность универсума больше (переменная *wA* имеет тип *long*), вместо константы 1 следует использовать 1*L*.

Для обратного преобразования (из машинного слова в строку символов) удобнее использовать сдвиг слова вправо и логическое умножение на 1:

```
for ( int i = 0, k = 0; i < m; ++i) if ( (wA >> i) & 1) A[k++] = f -1(i).
```

Отметим, что элементы массива битов нумеруются справа налево, а биты машинного слова — слева направо. Поэтому, если для массива битов и для машинного слова используется одна и та же функция отображения $f(a)$, порядок битов в этих структурах данных будет противоположный.

1.2.1. Практикум по теме

Добавьте в ранее составленную программу для работы с массивами и со списками такое же, как и ранее, вычисление пятого множества по четырём заданным, но с использованием представления множеств в форме (1) массива битов и (2) машинного слова. Для преобразования множества из массива символов в массив битов и в машинное слово напишите соответствующие функции. Результат преобразуйте обратно в последовательность символов. Все четыре способа обработки множеств должны давать одинаковый результат — с точностью до перестановки.

1.2.2. Контрольные вопросы

1. Какова временная сложность обработки множеств в случае представления их массивами битов?

2. Отличается ли от неё оценка временной сложности обработки машинных слов?

3. Можно ли получить выгоду, обрабатывая множества попарно?

4. Можно ли считать отображение на универсум универсальным приёмом, применимым для любых задач?

1.3. Генерация тестов

1.3.1. Генерация случайного подмножества

Программу, проверенную на тестах, введённых вручную или из специально подготовленного файла, можно затем дополнительно проверить подачей на вход некоторого количества случайных тестов, генерацию которых разумно поручить машине. Достаточно просто получить случайное множество в форме машинного слова: для этого можно использовать функцию *rand*( ) из стандартной библиотеки *stdlib.h*. Функция возвращает псевдослучайное целое в интервале 0…*MAXINT*. Случайное слово из *n* битов можно получить, выделив его из возвращаемого значения подходящей маской из единиц. Так, для $n = 10$ это будет:

```
w  = rand( ) %0x3FF.
```

А можно просто положить *w* = *rand*( ) и игнорировать лишние биты.

Если разрядность *int* равна 16, для получения слова типа *long* можно использовать функцию дважды и объединить возвращаемые значения:

```
w = (long) (rand( ) % 16) | rand( ).
```

Массив из *n* случайных битов получить ещё проще:

```
for ( int i = 0; i < n; ++i) X[ i ] = rand( ) % 2.
```

Следует отметить, что датчик *rand* ( ) даёт не случайные, а псевдослучайные числа. При каждом новом запуске программы последовательность этих чисел будет повторяться. Это очень удобно для отладки, но когда отладка закончена, в начало функции *main*( ) нужно вставить строку *srand*(*time*(0)), которая обеспечит запуск датчика со случайной точки, зависящей от текущего времени. Время запрашивается функцией *time*( ) из стандартной библиотеки *time.h*. Возможен также промежуточный вариант: подобрать аргумент функции *srand* таким образом, чтобы получился хороший тест, т. е. испытать *srand*(1), *srand*(2) и т. д. Такой приём может быть использован как альтернатива вводу тестов с клавиатуры.

Получить случайную строку тоже проще всего генерированием последовательности битов:

```
int k = 0;
for ( int i = 0; i < m; ++i) if (rand( )%2) S[ k++ ] = f( i ); S[k] = 0;.
```

Результат — ограниченная нулём строка символов *S* — множество со случайной мощностью $k \in [0 \ldots m - 1]$.

Все рассмотренные генераторы создают множества, в которых каждый

элемент универсума появляется с вероятностью 0,5. Может получиться и пустое, и полное множество, но в среднем мощность получается близкой к $m / 2$. Если требуется получать множества почти пустые или почти полные, нужно сделать так, чтобы вероятности появления 0 или 1 различались. В общем случае генератор массива битов может выглядеть так:

```
for ( int i = 0; i < m; ++i) X[ i ] = (rand( ) % p > q).
```

В этом генераторе вероятность появления 1 зависит от соотношения значений констант p и q. Так, например, при $p = 5$ датчик будет давать с равной вероятностью элементы множества $\{0, 1, 2, 3, 4\}$, и при $q = 3$ вероятность генерации 1 будет 0,2, а при $q = 0$ — 0,8.

1.3.2. Случайное подмножество заданной мощности

Все рассмотренные ранее датчики генерировали множество случайной мощности. Если же требуется случайное подмножество заданной мощности k, например, в форме массива, его иногда пытаются получить следующим алгоритмом:

```
for ( int i  = 0; i < k; ++i )  X[ i ] = rand( ) % m.
```

Этот способ не годится, потому что он даёт не множество, а последовательность, в которой возможны повторы, и их будет много, если k близко к m, т. е. фактическая мощность множества будет меньше заданного k.

Можно усовершенствовать этот алгоритм: повторять генерацию очередного элемента множества до тех пор, пока не кончатся совпадения с уже имеющимися. Способ рекомендуется при больших m ($k \ll m$). Если же m не намного больше k, то с ростом k вероятность получить новый элемент множества очень быстро уменьшается, а при $k = m$ алгоритм может вообще никогда не остановиться.

Способ, рекомендуемый для небольших m: сформировать в памяти для результата массив — универсум, на каждом шаге убирать сгенерированный элемент множества в его начало и разыгрывать оставшиеся. Результат будет получен за время $O(k)$.

```
for ( int i = 0; i < m; ++i ) X[ i ] = i + 1; //Формирование универсума
for ( int i = 0; i < k; ++i ) // Генерация подмножества мощностью k
    { int p = rand( ) % (m – i); // Случайный выбор среди оставшихся
    if (p) swap ( X[ i ], X [ i + p ] ); } // Если p ≠ 0, обменять местами.
```

Результат — первые k элементов массива X. Способ легко приспособить для генерации последовательности подмножеств нарастающей мощности: использовать массив X в качестве теста после каждого добавления в него очередного элемента. Если взять $k = m – 1$, получается *алгоритм генерации случайной перестановки*. Без ограничения общности он может использовать очередную переста-

новку как исходные данные для генерации следующей.

1.3.3. Генерация последовательности всех подмножеств заданного множества

Подача на вход алгоритма последовательности всех подмножеств некоторого множества X может потребоваться для полного тестирования алгоритма или для решения задачи полным перебором в случаях, когда эффективного алгоритма не существует. Если мощность множества $|X| = n$, мощность множества всех его подмножеств — булеана (общее количество тестов) $|2^X| = 2^n$.

Если $n \leq 32$, последовательность подмножеств проще всего получить в форме машинных слов по очевидному алгоритму:

```
for ( w = 0; w < 2^n; ++w) yield(w).
```

Здесь и далее ***yield***(*w*) — некоторая функция, использующая множество *w*.

Для практических целей часто бывает удобнее, чтобы каждое подмножество в последовательности отличалось от предыдущего появлением или исчезновением ровно одного элемента (n-битный код Грея). Такую последовательность можно получить небольшой модификацией предыдущего алгоритма:

```
for ( int i = 0; i < 2n; ++i) { w = i ^ (i >> 1); yield(w); }.
```

1.3.4. Генерация перестановок

Некоторые алгоритмы требуют подачи на вход полного множества X в виде последовательности, отличающейся порядком расположения элементов. Пример такого алгоритма — проверка двух графов одинаковой мощности на изоморфизм, заключающаяся в подборе такой нумерации вершин второго графа, чтобы его рёбра совпали с рёбрами первого графа. Функция *Neith*( ) генерирует все перестановки множества чисел от 1 до n в виде последовательности, в которой на каждом шаге меняются местами два смежных элемента.

```
inline void swap( int &p, int &q ) { int r (p); p = q; q = r; }
void Neith( int n )
{ int *X = new int[ n ], *C = new int[ n + 1], *D = new int[ n + 1], i, j, k, x;
   for (i = 0; i < n; ++i) // Инициализация
     { X[ i ] = i + 1; C[ i ] = 1; D[ i ] = 1; }
yield (X); //Использование вектора X (исходная перестановка)
C[ n ] = 0; i = 1; D[ n ] = 1;
while ( i < n ) // Цикл перестановок
{    i = 1; x = 0;
     while (C[ i ] == (n – i + 1))
       { D[ i ] = !D[ i ];   C[ i ] = 1;   if (D[ i ]) ++x;  ++i; }
```

```
        if ( i < n ) // Вычисление позиции k и перестановка смежных
        {   k = D[ i ] ? C[ i ] + x - 1 : n – i – C[ i ] + x; Swap( X[ k ], X[k + 1] ); }
            yield (X); //Использование вектора X (очередная перестановка)
            ++C[ i ];
        }
    }.
```

1.3.5. Практикум по теме

Преобразовать ранее созданную программу так, чтобы исходные множества генерировались автоматически.

1.3.6. Контрольные вопросы

1. Можно ли применить для тестирования вашего алгоритма генератор множества всех подмножеств?

2. Какой из способов генерации случайного множества вы считаете самым удобным?

3. Целесообразно ли для вашего варианта обработки множеств применять несимметричный генератор тестов?

4. Можно ли применять для генерации подмножества заданной мощности генерацию случайных битов с остановкой по достижении нужного их количества?

1.4. Измерение времени решения задачи с помощью ЭВМ

Измерение времени решения задачи для разных объёмов исходных данных — важная составляющая процедуры тестирования любой программы. Для измерения времени можно применять специальные приборы (секундомер) или средства, предоставляемые системой программирования (профайлер). Но для исследования алгоритма удобнее всего организовать измерение времени в самой программе тестирования, используя для этого средства, предоставляемые системой программирования.

1.4.1. Использование функции clock( ) или new( )

Функция *clock*( ) возвращает значение счётчика тиков внутренних часов ПЭВМ как 32-битное целое типа *clock_t*, что соответствует *unsigned long*. Для измерения времени обработки множеств нужно вызвать функцию *clock*( ) в момент, когда в памяти готовы исходные данные, и в момент, когда получен результат, а затем найти разность двух отсчётов.

Каждый тик соответствует 1/50 с, т. е. 0,017 с, следовательно, о какой-либо точности измерения времени функцией *clock*( ) можно говорить, если измеряемый интервал времени — порядка нескольких секунд. Чтобы добиться этого,

измеряемый процесс приходится многократно повторять (до 1 000 000 раз и даже более). Полученную разность отсчётов времени можно затем разделить на количество повторений. Способ везде доступен, он унаследован из первых версий языка Си.

Чтобы измерить время таким способом, важно приспособить процесс вычислений к многократному повторению, выполнив два условия:

1) исходные множества не должны искажаться, их память нельзя использовать для получения результата вычислений;

2) не должно быть «утечки памяти» — выделения в динамической памяти большего объёма, чем освобождается после вычислений. Так, в варианте со списками до начала вычислений следует освобождать память из-под всех результатов, полученных предыдущим проходом.

Другие функции стандартной библиотеки *time.h* (например, функцию *time*( ) и т. п.) использовать нет смысла, поскольку обычно источник информации о времени у них всех один, а следовательно, точность измерения времени не больше, чем у *clock*( ). Возможно, в вашей системе программирования это не так. Проверьте!

Существует альтернативный способ измерения времени решения задачи, не требующий зацикливания измеряемого процесса — использование часов высокого разрешения. В программу для этого добавляется строка #include <chrono>.

Для получения очередного отсчёта времени, например, *t*1, используйте утверждение `auto t1 = std::chrono::high_resolution_clock::now( );`

Для получения разности двух отсчётов *dt* с приведением её к типу *double*:

```
auto dt = duration_cast<duration<double,micro>>(t2-t1).count( );
```

1.4.2. Практикум по теме

Добавить в ранее созданные программы с генерацией исходных данных измерение времени вычисления множества по четырём исходным отдельно для каждого из способов представления множеств в памяти. Измерить время и зафиксировать результат для отчёта. Проверить, наблюдается ли в действительности зависимость времени решения задачи от средней мощности обрабатываемых множеств.

1.4.3. Контрольные вопросы

1. Как правильно организовать эксперимент для сравнения фактического быстродействия разных способов представления множеств?

2. Сколько раз нужно повторять тест при измерении времени его выполнения функцией *clock*( )?

1.5. Отчёт по теме

По теме должен быть оформлен сводный отчёт следующего содержания:

1. Цель работы: исследование четырёх способов хранения множеств в памяти ЭВМ.

2. Задание на обработку множеств (формулировка из пособия).

3. Формализация задания: формула для вычисления пятого множества по четырём заданным.

4. Контрольные тесты. Можно представить рисунки — скриншоты с результатами прогона тестов, при необходимости дополнив их пояснениями: где на рисунке исходные данные, где результат, что проверялось, какая структура данных использована и т. п. Рисунки не заменяют поясняющего текста, а прилагаются к нему.

5. Временная сложность (ожидаемая и фактическая) для каждого из четырёх способов представления множеств.

6. Результаты измерения времени обработки каждым из способов, с пометкой, наблюдалась ли зависимость времени обработки от размера данных. Рекомендуется составить сводную таблицу.

7. Выводы о результатах испытания способов представления множеств в памяти и рекомендации по их применению в программах для ЭВМ. Необходимо указать достоинства, недостатки и возможную область применения для каждого из способов.

8. Список использованных источников.

9. Приложение: исходные тексты всех программ (на машинном носителе) и файлы с тестами (если они готовились). В тексте каждого модуля программы обязательно должны быть сведения об авторе, решаемой задаче и использованной системе программирования (особенно если она не стандартная). Допускается включение текстов программ в состав отчёта.

При составлении списка использованных источников следует придерживаться правил, установленных для научной литературы. Для книг приводится библиографическое описание (*см*. в качестве примера описание настоящего пособия на обороте титула и оформление списка литературы на с. 51).

При ссылке на курс лекций следует указывать тему и дату лекции:

1. Множества в памяти ЭВМ // Алгоритмы и структуры данных. — Лекция от 10.09.2015.

При ссылке на Интернет-источник указывается наименование и адрес страницы:

2. Пирамидальная сортировка. http://algolist.manual.ru/sort/pyramid_sort.php.

Все использованные страницы подлежат проверке. Ссылки на Интернет-ресурс в целом не допускаются.

Помощь друга (консультанта) обязательно оформляется следующим образом:

3. Студент Петров А. В. Частное сообщение.

Использование литературных источников для заимствования описаний алгоритмов и текстов программ всячески приветствуется, если выполнены следующие условия (особенно это важно для темы «Графы»):

— заимствование сделано с указанием на источник (поз. в списке, страница);

— источник надёжный и достоверный (например, из рекомендованной литературы);

— в источнике решается именно та задача, по которой составляется отчёт, а не какая-нибудь близкая или похожая;

— программа написана на языке C++ (C++11) с использованием объектов (а не на каком-нибудь другом языке);

— термины и обозначения величин — те же, что используются в настоящем пособии. Если это не так, их следует пояснить и затем последовательно придерживаться и в тексте программы (в комментариях) и в пояснениях к алгоритму.

Тема 2. МНОЖЕСТВО КАК ОБЪЕКТ

Если некоторая структура данных, например, массив, используется как реализация множества, это означает, что программист просто устанавливает для себя некоторые правила для работы с этим массивом и последовательно их придерживается. Часто большего и не требуется. Однако можно рассматривать множество как абстрактную структуру данных — область памяти, доступ к которой возможен только через некоторый интерфейс, т. е. набор функций, специально созданных для работы с этой памятью. Язык C++ поддерживает работу с абстрактными данными через механизм классов: абстрактная структура данных определяется как класс, в котором задаются как данные, так и связанные с ними операции. Определение класса позволяет расширить язык C++, включив в него множество как пользовательский тип данных и набор операций с этими данными.

Рассмотрим пример — класс для работы с множеством, представленным массивом символов (строкой):

```
class Set {
private: // Закрытая часть класса — данные
static int N, cnt; // мощность универсума и счётчик множеств
```

```
  int n; // мощность множества
  char S, *A; // тег и память для множества
public: // Открытая часть — функции для работы с множеством
  Set operator | (const Set&) const; // объединение
  Set operator & (const Set&) const; // пересечение
  Set operator ~ ( ) const; // дополнение до универсума
  void Show( ); // вывод множества на экран
  int power( ) { return n; } // получение мощности
  Set(char); // конструктор множества
  Set( ); // ещё конструктор — по умолчанию
  Set(const Set &); // конструктор копии
  Set operator = (const Set &); // оператор присваивания
  ~Set( ) { delete [ ] A; } // деструктор
};
```

Имя класса *Set* — это имя нового типа данных. С его помощью мы будем объявлять в программе множества-объекты.

Память для множества находится в закрытой части класса и доступна через член *A* — указатель на символы. Размер памяти не определён. Кроме этого, в закрытую часть помещены вспомогательные переменные-члены: мощность универсума *N*, счётчик множеств *cnt*, текущая мощность множества *n* и символ-тег *S*, с помощью которого можно различать объекты-множества. Мощность универсума *N* и счётчик *cnt* объявлены со спецификатором «*static*». Это означает, что все объекты класса *Set* будут использовать единственную копию этих переменных. Переменные *N* и *cnt* должны быть дополнительно объявлены вне всех функций, чтобы им была выделена память. При этом требуется установить и их значения:

```
int Set :: N = 26; // Мощность универсума (пример для латинских букв)
int Set :: cnt = 0; // Начальное значение счётчика  множеств
```

В открытой части класса объявлены функции-члены, с помощью которых в программе-клиенте можно работать с множеством. Каждая функция-член имеет в качестве обязательного аргумента объект, для которого она вызывается. Данные-члены из закрытой части класса доступны в ней как обычные глобальные переменные, и их тоже не нужно передавать как аргументы. Всё это позволяет свести количество аргументов функций-членов к минимуму или даже совсем от них отказаться, не засоряя при этом пространство глобальных имён.

Для работы с множествами-массивами предполагается использовать такой

же синтаксис, как для машинных слов. С этой целью функции объединения, пересечения и дополнения множеств объявлены с именами, содержащими ключевое слово «*operator*», после которого следует знак соответствующей операции. Операции языка C++ «|», «&» и «~» определены так, чтобы их можно было использовать в выражениях, состоящих из данных типа *Set*. Такой приём называется перегрузкой операций. Чтобы это действительно было возможно, функции объявлены так, чтобы была обеспечена совместимость со встроенными операциями языка C++: все функции возвращают объект типа *Set*, а двуместные операции в качестве аргумента (второго, потому что первый — это сам объект) имеют константную ссылку на объект типа *Set*. Функции не меняют объект, для которого вызываются. Для контроля за этим в каждом из объявлений после списка параметров помещён спецификатор *const*.

Если мы объявляем для своего типа данных для операции пересечения перегрузку знака «&», а также перегрузку присваивания «=», то это не делает автоматически доступной комбинированную операцию «&=» — пересечение и присваивание. Такую операцию нужно тоже перегружать явно. Более того, рекомендуется обязательно сделать это и сделать согласованно. Проще всего этого добиться, реализуя двуместную операцию через комбинированную:

```
Set& Set :: operator &= (const Set & B)
{ Set C(*this);
  n = 0;
  for (int i = 0; i < C.n; ++i) {
       for (int j = 0; j < B.n; j++)
            if (C.A[ i ] == B.A[ j ]) A[ n++ ] = C.A[ i ];
  }
  A[ n ] = 0; // ограничитель строки
  return *this;
}
Set Set :: operator & (const Set & B) const
{ Set C(*this);
  return (C &= B);
}
```

В первой функции объявляется множество *C*, которое конструктор копии заменяет текущим множеством, для которого вызвана операция (множеством слева от присваивания). Затем текущее множество делается пустым, и в нём формируется результат пересечения временного объекта *C* и множества *B*. Поскольку для

этого используется двойной цикл по мощности множеств, временная сложность операции — квадратичная. Результат — текущий объект. Во избежание лишнего копирования можно в качестве результата вернуть ссылку на него.

Вторая функция — двуместная операция пересечения множеств — тоже сперва создаёт копию текущего объекта, а затем возвращает результат комбинированной операции с временным объектом в левой части. Возврат ссылки здесь недопустим: по выходе из функции временный объект автоматически уничтожится, и возвращённая ссылка станет недействительна.

Операция объединения множеств «|» реализуется похожим алгоритмом:

```
Set & Set :: operator |= (const Set & B)
{ for(int i = 0; i < B.n; ++i) {
        bool f = true;
        for (int j = 0; j < n; ++j)
            if (B.A[ i ] == A[ j ]) f = false;
        if (f) A[ n++ ] = B.A[ i ];
  }
  A[ n ] = 0;
  return *this;
}
Set Set :: operator | (const Set & B) const
{ Set C(*this);
  return (C |= B);
}
```

В текущий объект-множество добавляются недостающие элементы из *B*.

Операция вычисления дополнения может быть реализована так:

```
Set Set :: operator ~ ( ) const
{ Set C;
for (char c = 'A'; c <= 'Z'; ++c) { // Цикл по универсуму
        bool f = true;
        for (int j = 0; j < n; ++j)
            if (c == A[ j ]) { f = false; break; }
        if (f) C.A[ C.n++ ] = c;
  }
  C.A[ C->n ] = 0;
  return C;
}
```

Здесь в качестве одного из операндов выступает множество-универсум. Результат — элементы универсума, которых нет в исходном множестве.

Поскольку количество повторений цикла по элементам универсума постоянно, временная сложность операции — $O(n)$. Однако, если учесть, что мощность универсума не может быть меньше мощности его подмножеств: $|U| \geq n$, более точной будет оценка $O(|U| * n)$, более пессимистическая по сравнению с $O(n^2)$.

Разумеется, нельзя обойтись без функции *Show*( ) для вывода множества на экран: это единственный способ увидеть результат обработки, поскольку сами множества из вызывающей программы недоступны.

```
void Set :: Show( ) { cout << '\n' << S << " = [" << A << "]"; }
```

Для получения мощности множества нужна специальная функция *power*(). Эта функция просто возвращает значение закрытой переменной *n*, в которой другие функции поддерживают значение текущей мощности множества. Поскольку функция не только объявлена, но и определена внутри класса, она по умолчанию является *встроенной*. К объявлению функции неявно добавляется спецификатор *inline*. Это означает, что никакой функции не создаётся, вместо этого в каждую точку вызова просто подставляется значение закрытой переменной *n*. Таким образом, запрет на доступ к *n* не приводит к дополнительным расходам на вызов функции.

Если класс не является контроллером, т. е. не управляет ресурсом, как, например, класс для множества — машинного слова, объявлять и определять для него конструктор копии, перегрузку присваивания и деструктор не нужно, компилятор сделает это автоматически. Но в данном случае имеется ресурс — память для множества, и автоматически определяемые функции не подходят.

Так, при создании объекта класса *Set* под него выделяется память, после чего вызывается функция-конструктор *Set*( ). По умолчанию эта функция — пустая, она ничего с памятью не делает, и использовать такой объект невозможно. Поэтому конструктор надо определить явно. Сделаем так, чтобы он создавал пустое множество латинских букв, представленное строкой символов:

```
Set :: Set( ): n(0), S ('A' + cnt++), A(new char[ N+1 ])   { A[ 0 ] = 0; }
```

В этом примере одни переменные инициализируются в заголовке, другие — в теле конструктора. Оба способа можно комбинировать произвольным образом, но нужно учитывать, что порядок инициализации переменных полностью определяется порядком их объявления в классе и не может быть изменён. Массив символов объявляется на 1 символ длиннее мощности универсума, чтобы резервировать место под ограничивающий нуль. Поскольку строка должна быть

пустой, ограничивающий нуль записывается в её начало. Инициализировать остальную часть массива не обязательно.

Если требуется иметь несколько способов создания объекта, для каждого способа объявляется свой конструктор, отличающийся от других типом и/или количеством аргументов. В примере объявлен конструктор с одним символьным аргументом. Это может быть конструктор, генерирующий случайную строку латинских букв. Аргумент нужен для указания, что множество требуется не пустое. С помощью датчика случайных чисел генерируется *N* битов, и для каждого единичного бита соответствующий ему элемент добавляется в множество. Одновременно подсчитывается фактическая мощность множества *n*. По окончании генерации в строку добавляется ограничитель. Сгенерированное множество выводится на экран.

```
Set :: Set(char): S('A' + cnt++), n(0), A(new char[ N+1 ])
{
  for (int i = 0; i < N; ++i)
      if (rand() % 2) A[ n++ ] = i + 'A';
  A[n] = 0;
  cout << '\n' << S << " = [" << A << "]";
}
```

Следующие две функции-члена — конструктор копирования и перегрузку присваивания — определяют только для класса-контроллера. Дело в том, что обе эти функции по умолчанию копируют один объект в другой по принципу «байт в байт». Если ничего другого не требуется, определять эти функции не нужно, так как компилятор наверняка сделает это лучше. В данном же случае такое копирование не годится, потому что в классе есть указатель на дополнительную память, и копирование приведёт к тому, что указатели *A* в обоих объектах будут указывать на одну и ту же строку.

Конструктор копирования имеет единственный аргумент — константную ссылку на объект того же класса. Определить конструктор можно так:

```
Set :: Set(const Set & B) : S('A' + cnt++), n(B.n), A(new char[N+1])
{  char *dst(A), *src(B.A); //Инициализация адресов/счётчиков
  while(*dst++ = *src++);  //Копирование символов до обнаружения 0
}
```

Здесь переменные *N*, *S* и *n* копируются обычным способом, а для указателя *A* создаётся новая строка, куда затем копируется содержимое старой.

Функция-член для перегрузки присваивания отличается от копирования тем,

что объект в левой части оператора уже существует. Более того, он может совпадать с аргументом (самоприсваивание). Поэтому первое, что функция должна сделать — проверить это. Затем текущий объект уничтожается и создаётся новый. В данном случае это делать не надо, можно ограничиться просто переносом содержимого строки в имеющуюся память. Поскольку результат операции присваивания может быть использован в выражении, например, в цепочке присваиваний, функция должна возвращать значение объекта, для которого она вызвана. Это делается с помощью встроенного указателя *this*. Тег вычисляется с использованием счётчика множеств *cnt*.

```
Set & Set :: operator = (const Set& B)
{    if (this != &B)
     { char *dst(A), *src(B.A); n = B.n;
               while(*dst++ = *src++); S = 'A' + cnt++; }
     return *this;
}
```

Конструктор копирования используется при инициализации объекта содержимым другого объекта в момент объявления, а также при передаче аргумента в функцию как параметра по значению и при возврате объекта как результата работы функции. Функция перегрузки присваивания вызывается соответствующим оператором. Бывают ситуации, когда эти функции в программе не нужны. Чтобы исключить трудно выявляемую ошибку в программе из-за использования функций по умолчанию, рекомендуется объявить ненужные функции в закрытой части класса. Можно даже не определять их. Невольное создание в программе ситуации, когда такая функция вызывается, будет ошибкой, выявляемой компилятором.

В стандарте C++11 добавлены конструктор копирования и перегрузка присваивания в варианте «с переносом». Они используются, если источником данных является временный объект. Вместо того чтобы создавать копию данных, принадлежащих объекту, в варианте «с переносом» эти данные просто передаются объекту-приёмнику:

```
Set :: Set(Set && B) : S('A' + cnt++), n(B.n), A(B.A)
{ B.A = nullptr; }    //Копирование с переносом
Set & Set :: operator = (Set&& B) //Присваивание с переносом
{    if (this != &B)
{ n = B.n; A = B.A;  S = 'A' + cnt++; B.A = nullptr; }
     return *this;
```

```
}
```

Указатель на строку в объекте-источнике обнуляется, чтобы сделать удаление этого объекта безопасным.

Последняя функция-член в объявлении класса — это деструктор, который автоматически вызывается при уничтожении объекта. В нём указано дополнительное действие, которое нужно выполнить перед освобождением памяти из-под объекта: уничтожить строку *A*. Поскольку деструктор определён внутри класса, он, как и *power*( ), тоже является встроенной функцией. Впрочем, для деструктора это не важно. Его всё равно нельзя использовать как обычную функцию, например, получить его адрес. Деструктор вызывается явно в операторе *delete* или неявно — при выходе из блока, в котором объект был определён. Объекты уничтожаются в порядке, обратном порядку их создания.

Программа, использующая объекты класса *Set* (программа-клиент), может выглядеть так:

```
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <iostream>
using namespace std;
#include "Set.h"
int Set :: N = 26, Set::cnt = 0; // определение статических членов класса
const  long q0 = 100000; // количество повторений цикла времени
int main()
{ srand(time(nullptr));
  Set A('A'), B('B'), C('C'), D('D'), E;
  clock_t  begin = clock( );
  for(long q =0; q < q0; q++)
   { E = (A | B) & (C & ~D); }
  clock_t  end = clock( );
  E.Out( );
  cout << " Middle power =" <<
  (A.power() + B.power() + C.power() + D.power() + E.power()) / 5 <<
            " Time=" << end – begin<< " / " << q0 << endl;
  cin.get( );
  return 0;
}
```

В программе определяются пять множеств. Для исходных множеств *A*, *B*, *C* и *D* используется конструктор, генерирующий случайное множество с выводом на экран результата. Множество *E* генерируется конструктором по умолчанию как пустое. Затем множество вычисляется с использованием перегруженных операций, и результат выводится на экран. Далее вычисляются и выводятся средняя мощность всех множеств и время решения задачи.

Объявление класса *Set* и определения всех функций-членов находятся в подключаемом модуле *Set.h*. Определение статических членов — переменных *N* и *cnt* — помещено сразу после модуля. На самом деле оно является его частью. В самой программе никакой информации об устройстве модуля *Set.h* не имеется. Чтобы использовать другой способ хранения множеств в памяти, достаточно просто подменить модуль.

Вычисление множества *E* выполняется одним оператором присваивания, как в варианте для машинных слов. Но временная сложность этого вычисления не будет константной, она определяется функциями, реализующими операции над множествами и, следовательно, по-прежнему зависит от способа их хранения в памяти.

Результат работы программы может выглядеть так:

```
A = [ABDFOPSXZ]
B = [DEIJKLNOQRUUWXYZ]
C = [BCDFHIJKLPQSUWXZ]
D = [ABCDEGHKLOTUWZ]
R = [FPSXIJQU] Middle power =11 Time=1776/100000
```

2.1. Практикум по теме

Цель работы: сравнение процедурного и объектно-ориентированного подходов на примере задачи обработки множеств.

1. Преобразовать программы, созданные по п. 1.4.2, так, чтобы множества были объектами некоторого класса, а операции над ними — функциями-членами этого класса. Добиться, чтобы функция *main*( ) во всех вариантах была одинакова, менялось только определение классов. Этого можно добиться вынесением определения класса и функций-членов в отдельный h-файл, сделать 4 варианта h-файлов и подменять их в проекте. Второй способ — собрать все варианты в одном h-файле и исключать ненужные включением в комментарий или с помощью препроцессорной переменной.

2. В варианте «списки» перегрузить операции new и delete для элемента списка и зафиксировать изменение времени решения задачи для этого случая.

3. Провести эксперимент по отслеживанию вызовов функций при вычислении пятого множества по четырём исходным. Для этого:

— определить для класса все служебные функции, возможно, пустые;

— вставить в каждую из них вывод сообщения о том, какое действие выполняется и для какого объекта.

Рекомендуется отследить, какие множества создаются, используются или уничтожаются каждой из функций. Для этого нужно создать для каждого множества уникальный тег, например, с помощью общего для всех множеств счётчика тегов. Чтобы увидеть уничтожение объектов, объявленных в функции *main*( ), необходимо заключить её содержимое в дополнительные фигурные скобки и предусмотреть паузу (cin.get() или system("pause")) после них.

4. Провести эксперимент для определения, какие функции-члены класса *Set* реально используются в программе *main*( ) при работе с объектами этого типа.

2.2. Контрольные вопросы

1. Класс какого типа можно использовать для создания объектов-множеств каждого из реализованных вами способов кодирования? Можно ли во всех случаях использовать класс-значение или класс-контроллер?

2. Какую выгоду можно получить от применения объектов в программе обработки множеств?

3. Как повлияло применение объектов на время вычисления множества-результата? Можно ли исключить такое влияние?

4. Все ли созданные в программе множества действительно уничтожаются?

5. В каком порядке происходит уничтожение множеств?

6. Встречается ли в программе факт использования уже уничтоженного или ещё не созданного множества?

7. Можно ли ожидать в программе использования конструктора копии или присваивания в варианте «с переносом»?

8. Наблюдалось ли такое использование, а если нет, что нужно изменить в программе, чтобы оно произошло?

2.3. Отчёт по теме

По теме должен быть оформлен отчёт следующего содержания:

1. Цель работы: исследование эффекта от использования классов.

2. Задание на обработку множеств (можно сослаться на отчёт по теме 1).

3. Результаты эксперимента с четырьмя структурами данных на основе классов (рисунки с тестами и таблица результатов измерения времени).

4. Результат эксперимента с отслеживанием вызовов функций-членов.

5. Выводы о результатах испытания способов представления множеств в памяти. Заключение о целесообразности и эффекте от использования классов.

6. Список использованных источников.

7. Приложение: исходные тексты всех программ.

Тема 3. ДЕРЕВЬЯ

Дерево в общем случае — это связный граф без циклов, абстрактная структура данных, представляющая собой множество *вершин*, или *узлов*, на которых определены попарные связи — *рёбра*. Будем рассматривать частный случай — корневые упорядоченные деревья. У таких деревьев рёбра становятся ориентированными, поскольку у любой последовательности попарно связанных вершин — *пути*, включающем корень, появляется направление от корня или к корню. Для некоторого узла *v* все вершины дерева на пути в корень, находящиеся ближе к корню, называется *предками*, а дальше от корня — *потомками*. Из пары узлов, связанных ребром, узел ближе к корню — *отец*, дальше от корня — *сын*. У каждого узла может быть несколько сыновей, которые называются *братьями*, или *дочерними узлами*, но только один отец. Корень — это единственный в дереве узел, у которого нет отца. Узлы, у которых нет сыновей, называются *листьями*.

Количество рёбер на пути из корня в узел дерева называется *глубиной* узла, количество рёбер на самом длинном пути в лист — *высотой*. Высота дерева — это высота его корня. Разность между высотой дерева и глубиной узла — это *уровень* узла.

Дерево упорядочено, если упорядочены сыновья любого его узла. Из корневых упорядоченных деревьев наиболее часто используются двоичные, или бинарные. Каждый узел двоичного дерева может иметь не более двух сыновей — левого и правого, причём единственный сын узла — обязательно левый или правый. Более сложный вариант — троичное дерево, где у каждого узла — не более трёх сыновей: левый, средний, правый — в любой комбинации. Каждый из сыновей может рассматриваться как корень соответствующего поддерева, возможно, пустого.

Для представления дерева в памяти можно предложить естественный способ — разветвляющийся список. Следует отметить, что этот способ — не единственный и не самый эффективный. Другие возможные варианты хранения дерева в памяти обсуждаются ниже в п. 3.5.

Узлы дерева — объекты, связи между которыми будут осуществляться через

указатели. Для создания дерева достаточно объявить класс «узел дерева», членами которого должны быть указатели на узлы того же типа: «левый» и «правый» (у троичного дерева — «левый», «средний» и «правый»). В узле могут быть и другие данные-члены. Минимально необходимым является тег — метка или номер узла, с помощью которого можно различать узлы в процессе их обработки. Однако для работы с деревом в целом удобнее иметь особый класс «дерево», в котором собираются данные, относящиеся к дереву в целом, и функции-члены для работы с деревом. Чтобы эти функции имели доступ к данным узла, достаточно объявить класс «дерево» дружественным для класса «узел».

```
 // Класс «узел дерева»
class Node { char d;    //тег узла
        Node * lft;   // левый сын
    //   Node  * mdl;   — средний сын (если нужно)
        Node * rgt;  // правый сын
public:
         Node() : lft(nullptr), rgt(nullptr) { } // конструктор узла
         ~Node(){ if(lft) delete lft; // деструктор (уничтожает поддерево)
                  if (rgt) delete rgt; }
friend class Tree;  // дружественный класс «дерево»
      } ;
// Класс «дерево в целом»
class Tree
{    Node * root;    // указатель на корень дерева
char num, maxnum;        //счётчик тегов и максимальный тег
   int maxrow, offset;       //максимальная глубина, смещение корня
   char ** SCREEN;   // память для выдачи на экран
   void clrscr();      // очистка рабочей памяти
   Node* MakeNode(int depth);   // создание поддерева
   void OutNodes(Node * v, int r, int c); // выдача поддерева
   Tree (const Tree &);     // фиктивный конструктор копии
   Tree (Tree &&);      //копия с переносом (C++11)
   Tree operator = (const Tree &) const = delete;     // присваивание
   Tree operator = (Tree &&) const = delete; // то же,  с переносом
public:
    Tree(char num, char maxnum, int maxrow);
    ~Tree();
```

```
    void MakeTree() // ввод — генерация дерева
    { root = MakeNode(0); }
    bool exist() { return root != nullptr; } // проверка «дерево не пусто»
    int DFS();  // обход дерева «в глубину»
    int BFS();  // обход «в ширину»
    void OutTree();     // выдача на экран
};
```

Кроме данных, в классе *Tree* объявлены скрытые функции-члены: вспомогательные функции, которые не входят в интерфейс и предназначены только для вызова из других функций-членов. Конструкторы копирования и перегрузки присваивания сделаны скрытыми умышленно: попытка создать в программе ситуацию, в которой эти функции могут быть вызваны, приведёт к ошибке на этапе компиляции «нарушение защиты».

Конструктор дерева инициализирует параметры разметки и создаёт рабочую память — матрицу символов, необходимую для выдачи изображения дерева на экран.

```
Tree :: Tree(char nm, char mnm, int mxr):
      num(nm), maxnum(mnm), maxrow(mxr), offset(40), root(nullptr),
      SCREEN(new char * [maxrow])
      {   for(int i = 0; i < maxrow; ++i ) SCREEN[ i ] = new char[ 80 ]; }
```

Деструктор дерева уничтожает матрицу символов и запускает деструктор узла для корня.

```
Tree :: ~Tree( ) { for(int i = 0; i < maxrow; ++i) delete [ ]SCREEN[ i ];
                   delete [ ]SCREEN; delete root; }
```

Обратите внимание на то, как создаётся и уничтожается матрица.

3.1. Обходы дерева как рекурсивной структуры данных

Чтобы обработать каким-либо образом множество узлов дерева, его нужно обойти. Каждый узел дерева является корнем поддерева, а его сыновья — тоже корнями поддеревьев. Поэтому алгоритм обхода, запускаясь для узла, должен обработать информацию в узле и запустить такой же алгоритм для каждого из непустых поддеревьев. Существует три способа сделать это, отличающиеся лишь порядком шагов:

1. Прямой обход:

— обработать узел;

— посетить (в прямом порядке) каждого сына.

2. Обратный обход:

— посетить (в обратном порядке) каждого сына;

— обработать узел.

3. Внутренний, или симметричный обход:

— посетить (во внутреннем порядке) левого сына;

— обработать узел;

— посетить во внутреннем порядке правого сына (остальных сыновей).

Минимальная обработка узла может состоять в присвоении соответствующему в нём полю номера в порядке посещения (разметка) или в выдаче номеров на экран, если они уже имеются, или в формировании последовательности из номеров посещённых узлов. Очевидно, что не существует иных способов отличить один порядок обхода узлов от другого.

При разметке дерева в прямом порядке номер любого узла — наименьший, а при обратном — наибольший в соответствующем поддереве, а диапазон использованных номеров равен мощности поддерева. При разметке внутренним способом номер узла больше любого номера в левом поддереве и меньше любого номера в правом.

3.2. Создание дерева

Для создания дерева в памяти тоже применяется алгоритм обхода. Первым шагом этого алгоритма является проверка необходимости создания узла.

Если ответ положительный, узел создаётся, и в нём заполняются информационные поля. В частности, может быть выполнен шаг разметки. Далее заполняются поля указателей на каждого сына: для получения значения указателя алгоритм запускается рекурсивно. Результат — указатель на вновь созданный узел или нуль, если узел не создан.

Проверка необходимости создания узла может быть выполнена тремя способами:

1. Запрос на ввод с клавиатуры. Приглашение ко вводу может содержать какую-либо информацию о месте предполагаемого узла в дереве. Ожидаемый ответ — «да» или «нет» (1 или 0, *Y* или *N*, и т. п.).

2. Чтение очередного элемента заранее заготовленной последовательности из массива, линейного списка, файла или битов из машинного слова. Такая последовательность сама по себе тоже является способом размещения дерева в памяти, а алгоритм ввода просто преобразует её в форму разветвляющегося списка.

3. Обращение к датчику случайных чисел с целью генерации дерева. Датчик должен быть управляемым. Простой датчик с равновероятной выдачей 0 или 1

будет создавать пустые или очень маломощные деревья — из 1, 2, 3 узлов, так как вероятность того, что узел будет создан, очень быстро падает с ростом его глубины: для корня она составляет всего 0.5, для сыновей — 0.25 и т. д. Нужен датчик, который бы обеспечивал вероятность создания корня близкую к 1 и уменьшал её с ростом глубины узла.

Пример такого датчика:

```
Y = depth < rand() % 6 + 1,
```

где *depth* — глубина узла: для корня она 0, для произвольного узла — на 1 больше, чем у отца. Очевидно, что для корня *Y* всегда 1, а для узла на глубине больше 5 — всегда 0.

Функция-член для генерации случайного дерева может выглядеть так:

```
Node * Tree :: MakeNode(int depth)
{ Node * v = nullptr;
  int Y = (depth < rand( )%6+1) && (num <= 'z');
//Вариант: cout << "Node (" << num << ',' << depth << ")1/0: "; cin >> Y;
if (Y) {  // создание узла, если Y = 1
      v = new Node;
      v->d = num++;      // разметка в прямом порядке (= «в глубину»)
      v->lft = MakeNode(depth+1);
//    v->d = num++;          //вариант — во внутреннем
      v->rgt = MakeNode(depth+1);
//    v->d = num++;          // вариант — в обратном
  }
  return v;
}
```

Эта функция запускается из встраиваемой функции-члена *MakeTree*(), результат её работы присваивается полю *root*.

Вместо генерации случайного значения *Y* можно организовать ввод его с клавиатуры. Соответствующая альтернатива помещена в комментарий.

Функция создаёт дерево прямым обходом по той простой причине, что невозможно создать узел дерева, если не создан его отец. Но вот считать узел «пройдённым» можно когда угодно. Поэтому для разметки узла в алгоритме можно использовать три точки (две из них закомментированы): до обхода поддеревьев, после левого поддерева и перед правым и по окончании обхода поддеревьев. Нужный вариант разметки можно обеспечить, включив инициализацию в соответствующей точке и выключив — в остальных.

Значение глубины узла *depth*, необходимое для датчика, известно при входе в функцию и может быть использовано в любом месте. А вот данные, зависящие от поддеревьев: высота узла, количество листьев, количество потомков и т. п., могут быть известны только тогда, когда оба поддерева уже обработаны, т. е. они доступны только при обратном обходе.

3.3. Вывод изображения дерева на экран монитора

Чтобы получить наглядное представление о способе разметки дерева, нужно вывести его на экран в виде диаграммы. Можно обойтись для этого текстовым режимом, если принять следующее соглашение. В середине первой строки текста вывести метку корня дерева. В следующей строке — расположить метки левого и правого сыновей в серединах левой и правой половины строки и т. д. Если дерево — троичное, метку среднего сына можно разместить прямо под корнем, и т. д., уменьшая смещение сыновей относительно корня в два раза по отношению к предыдущему ряду. Удобно воспользоваться рекурсивной функцией обхода дерева, которая выдаёт метку узла в некоторой точке экрана (*r*, *c*), а для сыновей добавляет 1 к номеру ряда и смещение к номеру столбца. Смещение удобно вычислять сдвигом некоторой константы *offset* на номер ряда, который совпадает с глубиной узла.

Для выдачи метки в нужную точку экрана можно использовать функцию позиционирования курсора *gotoxy*(*r*, *c*) из библиотеки *conio.h*, предварительно очистив экран функцией *clrscr*(). Но поскольку эти функции есть не во всех оболочках, можно обойтись без них, использовав промежуточную буферную память в виде матрицы символов, как это сделано ниже в примере.

Для того чтобы понять разметку дерева, достаточно вывести узлы пяти — шести верхних уровней. Для улучшения читабельности картинки рекомендуется вместо числовых меток использовать буквы латинского алфавита.

Функция-член для вывода изображения дерева на экран может выглядеть так:

```
void Tree :: OutTree( )
{    clrscr( );
     OutNodes(root, 1, offset);
     for (int i = 0; i < maxrow; i++)
     { SCREEN[ i ][ 79 ] = 0;
          cout << '\n' << SCREEN[ i ];
     }
     cout << '\n';
```

```
}
```

Она запускает закрытую функцию-член *clrscr*( ), которая готовит матрицу символов, заполняя её точками:

```
void Tree :: clrscr( )
{ for(int i = 0; i < maxrow; i++)
    memset(SCREEN[i], '.', 80);
}
```

Далее выполняется закрытая функция *OutNodes*( ), расставляющая метки вершин дерева в матрице символов:

```
void Tree :: OutNodes(Node * v, int r, int c)
{ if (r && c && (c<80)) SCREEN[ r – 1 ][ c – 1 ] = v->d; // вывод метки
if (r < maxrow) {
    if (v->lft) OutNodes(v->lft, r + 1, c – (offset >> r)); //левый сын
    // if (v->mdl) OutNode(v->mdl, r + 1, c);  – средний сын (если нужно)
    if (v->rgt) OutNodes(v->rgt, r + 1, c + (offset >> r)); //правый сын
    }
}
```

Затем матрица символов построчно выводится на экран.

3.4. Шаблоны классов для очереди и стека и нерекурсивные алгоритмы обхода дерева

Шаблон для класса «стек»:

```
 template <class Item> class STACK
 { Item * S; int t;
 public:
    STACK(int maxt) : S(new Item[ maxt ]), t( 0 ) { }
    int empty( ) const { return t == 0; }
    void push(Item item) { S[t++] = item; }
    Item pop( ) {return ( t ? S[ --t ] : 0 ); }
 };
```

Шаблон для класса «очередь»:

```
 template <class Item> class QUEUE
 {   Item * Q; int h, t, N;
 public:
    QUEUE(int maxQ): h(0), t(0), N(maxQ), Q(new Item[maxQ + 1]) { }
    int empty( ) const { return (h % N) == t; }
```

```
    void push(Item item) { Q[ t++ ] = item; t %= N; }
    Item pop( ) { h %= N; return Q[ h++ ]; }
};
```

Нерекурсивный обход дерева способом «в глубину»:

```
int Tree :: DFS( )
{ const int MaxS = 20; // максимальный размер стека
  int count = 0;
  STACK <Node *> S(MaxS);  //создание стека указателей на узлы
  S.push(root);          // STACK <- root
  while (!S.empty( ))  // Пока стек не пуст…
{   Node * v = S.pop( );                  // поднять узел из стека
    cout << v->d <<  '_'; count++;       // выдать тег, счёт узлов
    if (v->rgt) S.push(v->rgt);           // STACK <- (правый сын)
    if (v->lft) S.push(v->lft);           // STACK <- (левый сын)
  }
  return count;
}
```

Замена стека очередью — нерекурсивный обход «в ширину»:

```
int Tree :: BFS( )
{ const int MaxQ = 20; //максимальный размер очереди
  int count = 0;
  QUEUE < Node * > Q(MaxQ);  //создание очереди указателей на узлы
  Q.push(root); // QUEUE <- root поместить в очередь корень дерева
  while (!Q.empty( )) //пока очередь не пуста
{   Node * v = Q.pop( );// взять из очереди,
cout << v->d << '_'; count++; // выдать тег, счёт узлов
    if (v->lft) Q.push(v->lft); // QUEUE <- (левый сын)
    if (v->rgt) Q.push(v->rgt); // QUEUE <- (правый сын)
  }
  return count;
}
```

Пример программы для создания случайного дерева, выдачи его на экран и обхода двумя способами с подсчётом мощности дерева:

```
int main( )
{ int n = 0;
  Tree Tr('a', 'z', 8);
```

```
  srand(time(nullptr));
  setlocale(LC_ALL, "Russian");
  Tr.MakeTree( );
 if(Tr.exist( )) {
    Tr.OutTree( );
    cout << '\n' << "Обход в глубину: ";
    n = Tr.DFS( );
    cout << " Пройдено узлов = " << n;
    cout << '\n' << "Обход в ширину: ";
    n = Tr.BFS( );
    cout << " Пройдено узлов = " << n;
  }
  else cout << "Дерево пусто!";
  cout << '\n' << "=== Конец ===";   cin.get( );
}
```

Результат работы программы может выглядеть так:

```
.....a.....
.....b.....h.....
.....c.....g.....i.....m.....
.....d.....j.....k.....
.....e...f.....l.....
.....
.....
.....

Обход в глубину: a_b_c_d_e_f_g_h_i_j_k_l_m_ Пройдено узлов = 13
 Обход в ширину: a_b_h_c_g_i_m_d_j_k_e_f_l_ Пройдено узлов = 13
=== Конец ===_
```

3.5. Другие способы хранения и обхода дерева

Поскольку дерево — частный случай графа, способы хранения графа в памяти в принципе пригодны и для дерева (см. гл. 4). Деревом также можно объявить массив с произвольным содержимым, если принять соглашение: для любого i-того элемента массива-дерева его сыновьями будут элементы в позициях $(i+1)*2-1$ и $(i+1)*2$ (счёт позиций от 0), а отцом — элемент в позиции $\llcorner(i/2)$. Такое дерево (двоичное) однозначно определяется его мощностью n. Для троичного дерева в формулах нужно 2 заменить на 3.

Для дерева с такой формой представления возможен обход как простой перебор элементов массива циклом по позициям от 0 до n.

При применении произвольной перестановки индексов массива получается случайный обход. О генерации перестановок *см*. выше пп. 1.3.3 и 1.3.4.

Отметим, что создавать разветвляющийся список для хранения дерева совершенно не обязательно. Вместо этого функция *MakeNode* может просто запоминать генерируемые или вводимые биты *Y* в специальном векторе *Tr*, хранящемся в классе *Tree*:

```
class Tree
{  char num = 'a', maxnum = 'z';        //счётчик тегов и максимальный тег
   int pos = 0;             //счётчик позиций при обходе
   vector <int> Tr;       //вектор для хранения дерева
//…
};
```

Класс *Node* при этом не нужен вовсе, а функция создания узла может выглядеть так:

```
void Tree :: MakeNode(int depth)
{   int p = pos++;  //запоминание и счёт позиции узла в векторе
    int Y = (depth < rand( )%6+1) && (num <= 'z');
//Вариант: cout << "Node (" << num << ',' << depth << ")1/0: "; cin >> Y;
    Tr.push_back(Y); //запоминание бита наличия узла
    if (Y) {  // если Y = 1, то узел создан, создаём сыновей
        Tr[p] = num;    //прямая разметка
        MakeNode(depth+1);
    //    Tr[p] = num;  //вариант — симметричная
        MakeNode(depth+1);
     //   Tr[p] = num;  //вариант — обратная
      }
    }
```

Вектор битов *Tr* можно использовать как обобщённый, т. е. хранить в нём не 0 и 1, а информацию для узла, например, разметку, как делается в примере выше.

В функциях обхода значение бита не генерируется, а проверяется, с инкрементированием счётчика битов *pos*, который следует обнулять перед каждым обходом: *if*(*Tr*(*pos*++)) … //обработка узла.

3.5.1. Практикум по теме

1. Написать и отладить программу для работы с деревьями по предложенному преподавателем варианту индивидуального задания (табл. П.2.2). Программа должна выводить на экран изображение дерева с разметкой его вершин, сделанной заданным способом, а под ним — последовательность меток вершин при обходе дерева и результат вычисления заданного параметра. Можно взять за основу учебный пример, убрав из него всё лишнее.

2. Сделать узел дерева и дерево в целом объектами соответствующих классов, а обходы дерева — функциями-членами для класса «дерево».

3. Объявить в классе «дерево» деструктор и все конструкторы, поддерживаемые по умолчанию. Сделать невозможным использование тех конструкторов, которые на самом деле не нужны. Сделать в тексте программы временные дополнения и убедиться, что это действительно так.

3.5.2. Контрольные вопросы

1. Чем отличаются алгоритмы для разных способов обхода деревьев?

2. Нужно ли сочетать ввод данных для построения дерева с клавиатуры с его обходом?

3. Можно ли считать применённые вами алгоритмы обхода дерева эффективными?

4. Нужно ли создавать отдельные классы для узла и для дерева в целом, или можно ограничиться одним универсальным, рассматривая любой узел как корень некоторого поддерева?

3.6. Отчёт по теме

По теме должен быть оформлен сводный отчёт следующего содержания:

1. Цель работы: исследование алгоритмов для работы с двоичным (троичным) деревом.

2. Задание на работу с деревьями.

3. Обоснование выбора способа представления деревьев в памяти ЭВМ. Здесь следует сделать ссылку на выводы в отчётах по темам 1 и 2.

4. Тестовый пример: изображение дерева и порядок его ввода с клавиатуры.

5. Результаты прогона программы с генерацией случайного дерева (скриншоты).

6. Оценки временной сложности для каждой функции обхода дерева, использованной в программе: создание дерева, обработка, вывод.

7. Выводы о результатах испытания алгоритмов обхода деревьев.

8. Список использованной литературы.

9. Приложение: исходный текст программы для работы с деревьями.

Тема 4. ГРАФЫ

Граф — это пара множеств $G = \langle V, E \rangle$, где V — произвольное множество, а $E = \{\{u, v\}: u, v \in V, u \neq v\}$ — множество пар из элементов множества V. Если пара $\{u, v\}$ представляет собой множество мощностью 2, граф называется неориентированным, а если это последовательность $\langle u, v \rangle$ — ориентированным. Будем обозначать мощность множества вершин $|V| = n$, а мощность множества рёбер $|E| = m$. Очевидно, что справедливо ограничение $m = O(n^2)$.

Вершины $\{u, v\}$, образующие ребро, называются *смежными*, а само ребро — *инцидентным* по отношению к образующим его вершинам, а вершины, в свою очередь, *инцидентны* ребру. Количество рёбер, инцидентных вершине, называется её *степенью*. Вершина, не входящая ни в одно ребро, имеет степень 0 и называется изолированной. В ориентированном графе различают также количество рёбер, входящих в вершину — *полустепень захода* — и количество выходящих рёбер — *полустепень выхода*.

Последовательность попарно смежных вершин образует *путь* в графе. *Длина пути* равна количеству входящих в него рёбер. Если в последовательности, образующей путь, все вершины различны, путь называется *элементарным*. Путь, начало и конец которого совпадают, называется *циклом*. Связный граф без циклов называется *деревом*, несвязный — *лесом*.

Если любая пара вершин графа связана путём, граф называется *связным*. Если для любой пары вершин находятся, по крайней мере, два пути, множества вершин которых не пересекаются, граф — *двусвязный*.

В ориентированном графе (орграф) путь между некоторыми вершинами может быть только в одну сторону. Если же любая пара вершин орграфа связана путями в обе стороны, такой граф называется *сильно связным*.

Граф с пустым множеством вершин называется *пустым*, а граф, в котором имеются все возможные рёбра, — *полным* графом или *кликой*.

Граф, множество вершин которого можно разбить на два непустых непересекающихся подмножества таким образом, что концы любого ребра будут принадлежать разным подмножествам, называется *двудольным*.

Если граф каким-либо из перечисленных свойств не обладает, можно ставить задачу отыскания компонент — максимальных подграфов, обладающих нужным свойством, например, компонент связности, двусвязности, максимальных клик и т. п.

Графы $G = < V, E >$ и $G' = < V', E' >$ называются изоморфными, если существует биекция $f: E \to E'$ такая, что для любой пары вершин $\{u, v\} \in E \Leftrightarrow \{f(u), f(v)\} \in E'$.

На свойстве изоморфизма строятся все возможные способы хранения графа в памяти. Перечислим наиболее употребительные из них:

1. Вершины хранятся в массиве, каждый элемент которого — множество рёбер в форме вектора битов. Единичные биты соответствуют рёбрам, инцидентным данной вершине. Альтернатива — массив рёбер, каждое из которых задано вектором инцидентных вершин, которых может быть ровно две. Это — матрица

инциденций размером $m \times n$. Это расточительный способ, потому что матрица большей частью состоит из нулей. Но он достаточно компактен и удобен для некоторых задач, например для отыскания вершинного или рёберного покрытия. Способ является естественным для неориентированного графа. Для орграфа следует различать начала и концы рёбер, например, так: «–1» — ребро выходит из вершины, «+1» — ребро входит, «2» — и входит, и выходит (петля).

2. Вершины хранятся в массиве, каждый элемент которого — множество смежных вершин в форме вектора битов. Это матрица смежности размерами $n \times n$, она может содержать 0 и 1 в любой пропорции. Так, полному графу соответствует единичная матрица. Способ удобен для орграфов. Неориентированные графы хранятся как дважды ориентированные, т. е. их матрица смежности всегда симметрична; она может храниться только верхним треугольником.

3. Вершины хранятся в массиве, каждый элемент которого — множество смежных вершин в форме списка. Каждый элемент списка содержит поле с номером смежной вершины — индексом массива вершин. Это — списки смежности. Они удобны, если количество рёбер в графе не очень велико, и требуют памяти порядка $O(n + m)$.

4. Массив рёбер, каждое из которых задано парой номеров инцидентных вершин, — массив пар. Требует $2 \times m$ ячеек памяти.

5. Разветвляющийся список из вершин, в котором рёбра реализованы посредством указателей на смежную вершину. Этот способ применяется главным образом для ациклических орграфов (деревьев), а в общем случае малопригоден без каких-либо дополнений (структура Вирта).

4.1. Обходы графов

Обход вершин графа может быть выполнен теми же способами, что и обход дерева. Однако следует учитывать, что граф общего вида отличается тем, что он может быть не связен и может содержать циклы. Поэтому нужно создавать дополнительную структуру данных — массив битов и отмечать в нём пройдённые вершины. Если по завершении алгоритма обхода часть осталась не пройдённой, алгоритм перезапускается до тех пор, пока таковых не останется. Количество запусков алгоритма равно количеству компонент связности графа.

4.2. Некоторые задачи на графах

Далее в качестве примера ввода и обработки графа приводится программа для отыскания компонент двусвязности в неориентированном графе, основанная на алгоритме обхода в глубину. Граф представлен пользовательской структурой

данных, формирующейся в конструкторе.

В исходных данных вершины графа размечены латинскими буквами. Для каждой вершины графа вводится строка букв — меток смежных вершин, т. е. множество смежности. Строка, начинающаяся не буквой, завершает ввод.

Далее из введённых массивов формируется матрица смежности. Она делается симметричной с нулевой главной диагональю. Тем самым устраняется дублирование и возможная неполнота ввода.

Затем из матрицы смежности формируются списки смежности *LIST*, рекомендованные для использования в алгоритме *DBL*( ).

Поскольку граф в общем случае может быть не связным, рекурсивная функция *DBL*( ) нуждается в оболочке *DBL_EXEC*( ), обеспечивающей её запуск для каждой компоненты связности. Результат выводится на экран.

Пример 4.1. Программа отыскания компонент двусвязности

```
#include <iostream>
#include <vector>
#include <list>
#include <stack>
using namespace std;
const int MaxV = 26;
char Ch(int c) { return c+'a'; }
class GR {
  int num, n, m;
  vector <list<int>> LIST;
  vector <int> NUM, L;
  stack<pair<int, int>> STACK;
  void DBL (int v, int p);
public:
  GR(int);
  void DBL_Exec( );
  ~GR( ) = default;
};
GR :: GR(int MaxV) : num(0), n(0), m(0)
    { int G[ MaxV ][ MaxV ]; string s;
     for(int i = 0; i < MaxV; ++i)
      for(int j = 0; j < MaxV; ++j) G[ i ][ j ] = 0;
     cout <<"\n Введите мн-ва смежности (строки букв a до z)\n";
```

```
    do{
       cout << "v[" << Ch(n) << "]="; cin >> s;
      for (auto i : s)  if (isalpha( i )){
        int j = (tolower( i ) - 'a');
        G[ n ][ j ] = G[ j ][ n ] = 1;
      }
      ++n;
    } while(isalpha(s[0]) && n < MaxV);
    n = m = 0; LIST.resize(MaxV);
    for (int i=0; i<MaxV; ++i)
    { int f = 0;
      cout << '\n' << Ch(i) << ": ";
       for (int j = 0; j < MaxV; ++j)
       if(G[ i ][ j ])
        { ++f;
          LIST[ i ].push_back(j);
          cout << Ch( j ) << ' ';
        }
        else cout << "- ";
        m += f;
        if( f ) ++n;
        else break;
   }
   cout << "\n|V|=" << n <<  " |E|=" << m/2;
}
void GR :: DBL_Exec() //Обход графа в целом, возможно, несвязного
{
  NUM.resize(n, 0); L.resize(n, 0);
  for (int i = 0; i < n; ++i)
       if (!NUM[ i ]) DBL( i, -1 );
}
void GR :: DBL (int v, int p) //Обход компоненты связности
{ using edge = pair<int, int>;
  NUM[ v ] = L[ v ] = ++num;
  for (u : LIST[ v ])
  {   if (!NUM[ u ])
```

```
      {  edge e0(u,v), e1(0,0);
         STACK.push(e0); // ребро -> в стек
         DBL(u, v);
         L[ v ] = L[ u ] < L[ v ] ? L[ u ] : L[ v ];
         if (L[ u ] >= NUM[ v ])
         {
            cout << "\n ребро <" << Ch(v) << '-' << Ch(u)
            << "> замыкает компоненту [";
          do { //выдача компоненты двусвязности
            e1 = STACK.top(); STACK.pop();
           cout << Ch(e1.first) << '-' << Ch(e1.second) << ';' ;
          } while ((e1 != e0) && !STACK.empty());
          cout << "] ";
        }
      } // if (NUM…
      else if ((u != p) && (NUM[ u ] < NUM[ v ]))
      { STACK.push(make_pair(u, v));
        L[ v ] = NUM[ u ] < L[ v ] ? NUM[ u ] : L[ v ];
      }
   } //  for…
}
int main( )
{
  setlocale(LC_ALL, "Russian");
  cout << "\nDBL test ============== (C)lgn, 10.10.03;24.02.20";
  GR Gr(MaxV);
  Gr.DBL_Exec( );  //Тестирование функции DBL
  cout << "\n ===== Конец =====\n";
}
```

4.3. Переборные алгоритмы на графах

Для решения задачи, для которой нет эффективного алгоритма, можно применить следующие подходы:

1. Задача, решением которой является некоторая перестановка элементов полного множества. Примеры: проверка графов на изоморфизм, отыскание гамильтонова цикла и т. п. Решение такой задачи может быть сведено к генерации перестановок и проверке каждой перестановки, не является ли она решением.

Альтернатива — генерация случайных перестановок до тех пор, пока решение не будет обнаружено или не закончится отведённое для этого время.

2. Задача, решением которой является подмножество. Здесь можно использовать генератор всех подмножеств.

В обоих случаях можно попытаться применить алгоритм перебора с возвратом, который, начиная с пустого вектора, пытается расширить его до решения, отбрасывая заведомо негодные альтернативы. Способ применим, если для каждого *k*-го элемента вектора решений X можно указать конечное множество допустимых значений A_k и, возможно, простую функцию $f(x)$, $x \in A_k$, принимающую значение *false* для заведомо непригодных значений x. Так, в задаче отыскания гамильтонова пути (элементарного пути через все вершины графа) в качестве A_k можно взять множество вершин, смежных с x_{k-1}, а в качестве дополнительного критерия для отбора — множество свободных вершин *NEW*.

Пример 4.2. Программа отыскания гамильтонова пути

```
#include <iostream>
#include <vector>
#include <list>
#include <stack>
using namespace std;
const int MaxV = 26;
char Ch(int c) { return c+'a'; }
class GR {
   int n, m;
   vector <list<int>> LIST;
   vector <int> NEW, X;
   void GAM(int);
public:
   GR(int); //Эта функция определяется аналогично примеру 4.1.
   void GAM_test();
   ~GR( ) = default;
};
void GR::GAM (int k)
{
  if(k == n) //Найдено решение, выводим
  {  cout << "<";
     for (auto x : X) cout << Ch(x) << ' ';
```

```
      cout << ">\n"; cin.get();
    }
    else
    { for (auto u : LIST[ X[ k-1 ] ]) //Перебираем множество A_k
      { if (NEW[ u ]) //Очередная вершина свободна?
          { NEW[ u ] = true; //Объявляем вершину занятой
            X.push_back(u); // и добавляем её в решение
            GAM(k+1); //Продолжаем расширять вектор
            X.pop_back(); //Убираем проверенную вершину…
            NEW[ u ] = false; // и объявляем её снова свободной
          }
      }
     }
}
void GR::GAM_test(){
  NEW.resize(n, 1);
  cout << "\nГамильтонов путь:\n";
  X.push_back(0); NEW[0] = false;
  GAM(1);
  cin.get( );
}
int main( )
{ GR G(MaxV);
  setlocale(LC_ALL, "Russian");
   cout <<
"\nGAM test ====================== (C)lgn, 16.10.03;24.02.20";
  //Тестирование функции GAM
  G.GAM_test();
}
```

Перебор с возвратом работает значительно быстрее полного перебора, но временная сложность алгоритма всё равно остаётся экспоненциальной.

Поэтому на практике часто используются приближённые алгоритмы полиномиальной сложности, которые теоретически не могут дать точного решения. Проверить этот факт можно прямым сравнением алгоритмов на опыте.

Пример 4.3. Испытания эмпирического алгоритма отыскания максимальной клики в произвольном неориентированном графе. Исходный граф представлен

матрицей смежности, заполняемой с помощью датчика случайных чисел таким образом, чтобы граф получался плотным. Мощность множества вершин графа задана константой в программе. Если её значение невелико, матрица смежности выводится на экран для возможности визуального контроля результата.

Испытания программы показывают, что эмпирический алгоритм находит оптимальное решение, только если количество вершин не очень велико. Алгоритм перебора с возвратом всегда находит все решения.

```
#include <time.h>
#include <iostream>
using namespace std;
const int N=10;     //Количество вершин
int a[ N ][ N ], i, j, maxv = 0, k, st, ans[ N ], i1, num, K[ N+1 ], U[ N ];
void G(int k)    //Перебор с возвратом
{  int i, i0;
    if(k == 1) i0 = 0; else i0 = K[ k-2 ] + 1;
    for( i = i0; i < N; i++)
        if (U[ i ]) {
          K[ k-1 ] = i; j = 0;
          while ((j < k) && a[ K[ j ] ][ i ]) j++;
          if (j+1 == k) { //Найдена клика...
              if (k > maxv) {//больше предыдущей, зафиксировать решение
                    maxv = k;
                    for (auto i1 = 0; i1 < k; ++i1)
                        ans[ i1 ] = K[ i1 ] + 1;
                    }
              if (k == maxv) { //... и выдать
                    cout << '\n' << " max=" << maxv << " : ";
                    for(auto i1 = 0; i1 < maxv; ++i1)
                        cout << (K[i1] + 1) << " ";
                    cin.get( );
                    }
              U[ i ] = 0; //Вершина (i) теперь занята
              G(k+1); // Попробовать расширить решение
              U[ i ] = 1; //Возврат: (i) снова свободна
          }
    }
```

```
}
int main( )
{
     setlocale(LC_ALL, "Russian");
     srand(time(nullptr));
//Генерация матрицы смежности неорграфа
     for(auto i = 0; i < N; ++i)
     {  U[ i ] = 1;
          for (auto j = i; j < N; ++j)
               if(j == i) a[ i ][ j ] = 0;
               else      a[ i ][ j ] = a[ j ][ i ] = rand() % 15 > 2;
     }
     if (N<21) { //Вывод на экран, если помещается
     cout << "\nМатрица смежности";
cout << "\n      1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 0";
     cout << "\n-------------------------------------------------------------";
     for (auto i = 0; i < N; ++i)
          { cout << "\n "<< i+1 << " |";
          for(auto j = 0; j < N; ++j)
               cout << " " << a[ i ][ j ] << " ";
          }
     }
     //Эмпирический алгоритм — полиномиальной сложности
     for (auto i = 0; i < N; ++i)
     {  K[0] = i;
          for(auto st = i + 1; st < N; ++st)
               if(a[ i ][ st ])
               { k = 1;
                    for(auto j = st; j < N; ++j)
                    {    num = 1;
                         while((a[ K[ num-1 ] ][ j ]) && (num <= k)) ++num;
                         if ((num - 1) == k)
                              { ++k; K[ k-1 ] = j; }
                    }
                    if (k > maxv) //Зафиксировать решение
                         { maxv = k;
```

```
                    for(auto i1 = 0; i1 < k; ++i1) ans[ i1 ] = K[ i1 ] + 1;
                    }
                if (k == maxv) { //... и выдать
                    cout << "\n max=" << maxv << " : ";
                    for (auto i1 = 0; i1 < maxv; ++i1)
                        cout << (K[i1] + 1) << " ";
                    cin.get();
                    }
                }
            }
            cout << "\n Клика мощностью " << maxv <<" из вершин: ";
            for(auto i = 0; i < maxv; ++i)
                cout << ans[i] << " ";
            cout << "\n Контроль перебором";
            maxv = 0;
            G(1);
            cout << "\nИТОГ: мощность " << maxv <<", вершины: ";
            for(auto i = 0; i < maxv; ++i)
                cout << ans[i] << " ";
            cout << endl;
            cin.get( );
        }
```

Результаты работы программы:

Вариант 1. Граф из 9 вершин. Результаты работы двух алгоритмов совпадают.

```
Матрица смежности
      1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
-------------------------------------------------------------------
  1 | 0  1  0  1  1  1  1  1  0
  2 | 1  0  0  1  1  1  1  1  1
  3 | 0  0  0  0  1  1  1  1  0
  4 | 1  1  0  0  1  1  1  1  1
  5 | 1  1  1  1  0  1  1  1  1
  6 | 1  1  1  1  1  0  1  1  1
  7 | 1  1  1  1  1  1  0  1  1
  8 | 1  1  1  1  1  1  1  0  0
  9 | 0  1  0  1  1  1  1  0  0
 max=7 : 1 2 4 5 6 7 8
 Клика мощностью 7 из вершин: 1 2 4 5 6 7 8
 Контроль перебором
 max=1 : 1
 max=2 : 1 2
 max=3 : 1 2 4
 max=4 : 1 2 4 5
 max=5 : 1 2 4 5 6
 max=6 : 1 2 4 5 6 7
 max=7 : 1 2 4 5 6 7 8
ИТОГ: мощность 7, вершины: 1 2 4 5 6 7 8
```

Вариант 2. Количество вершин увеличено до 20. Эмпирический алгоритм начинает проигрывать.

```
Матрица смежности
      1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
-------------------------------------------------------------------
  1 | 0  1  1  1  1  1  1  0  0  1  1  0  1  1  1  1  1  1  1  1
  2 | 1  0  1  1  1  1  1  0  0  0  1  1  1  1  1  1  1  1  1  1
  3 | 1  1  0  1  0  1  1  1  1  1  1  1  1  1  1  1  1  1  1  0
  4 | 1  1  1  0  1  1  1  1  1  1  1  1  1  0  1  1  0  1  0  0
  5 | 1  1  0  1  0  1  1  1  0  1  1  1  1  1  1  1  1  1  1  1
  6 | 1  1  1  1  1  0  0  1  1  0  1  1  0  1  1  1  1  1  1  1
  7 | 1  1  1  1  1  0  0  1  0  1  0  1  1  1  1  0  1  1  1  1
  8 | 0  0  1  1  1  1  1  0  1  1  0  1  1  1  1  0  1  1  0  0
  9 | 0  0  1  1  0  1  0  1  0  1  1  1  1  1  1  1  1  1  1  1
 10 | 1  0  1  1  1  0  1  1  1  0  1  1  1  1  1  1  1  1  1  0
 11 | 1  1  1  1  1  1  0  0  1  1  0  0  1  1  1  1  1  1  1  1
 12 | 0  1  1  1  1  1  1  1  1  1  1  0  0  1  1  1  1  1  1  1
 13 | 1  1  1  1  1  0  1  1  1  1  1  1  0  0  1  1  1  1  1  1
 14 | 1  1  1  0  1  1  1  1  1  1  1  1  0  0  1  1  1  1  0  1
 15 | 1  1  1  1  1  1  1  1  1  1  1  1  1  1  0  0  0  1  1  1
 16 | 1  1  1  1  1  1  0  0  1  1  1  1  1  1  0  0  1  1  1  0
 17 | 1  1  1  0  1  1  1  1  1  1  1  1  1  1  0  1  0  1  1  1
 18 | 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  0  1  0
 19 | 1  1  1  0  1  1  1  0  1  1  1  1  1  0  1  1  1  1  0  1
 20 | 1  1  0  0  1  1  1  0  1  0  1  1  1  1  1  0  1  0  1  0
```

```
max=8 : 1 2 3 4 6 11 15 18
max=8 : 3 4 6 8 9 12 15 18
max=8 : 3 6 8 9 12 14 15 18
max=8 : 3 7 8 10 12 13 15 18
max=8 : 3 8 9 10 12 13 15 18
max=8 : 3 9 10 11 13 15 18 19
max=8 : 4 7 8 10 12 13 15 18
max=8 : 4 8 9 10 12 13 15 18
max=8 : 5 7 8 10 12 13 15 18
Клика мощностью 8 из вершин: 1 2 3 4 6 11 15 18
Контроль перебором
max=1 : 1
max=2 : 1 2
max=3 : 1 2 3
max=4 : 1 2 3 4
max=5 : 1 2 3 4 6
max=6 : 1 2 3 4 6 11
max=7 : 1 2 3 4 6 11 15
max=8 : 1 2 3 4 6 11 15 18
max=8 : 1 2 3 4 6 11 16 18
max=8 : 1 2 3 4 7 13 15 18
max=8 : 1 2 3 4 11 13 15 18
max=8 : 1 2 3 4 11 13 16 18
max=8 : 1 2 3 6 11 14 15 18
max=8 : 1 2 3 6 11 14 16 17
```

```
max=9 : 1 2 3 6 11 14 16 17 18
max=9 : 1 2 3 6 11 16 17 18 19
max=9 : 1 2 3 11 13 16 17 18 19
max=9 : 1 2 5 6 11 14 16 17 18
max=9 : 1 2 5 6 11 16 17 18 19
max=9 : 1 2 5 11 13 16 17 18 19
max=9 : 1 3 10 11 13 16 17 18 19
max=9 : 1 5 10 11 13 16 17 18 19
max=9 : 3 4 7 8 10 12 13 15 18
max=9 : 3 4 8 9 10 12 13 15 18
max=9 : 3 9 10 11 13 16 17 18 19
max=9 : 3 9 10 12 13 16 17 18 19
max=9 : 4 5 7 8 10 12 13 15 18
ИТОГ: мощность 9, вершины: 1 2 3 6 11 14 16 17 18
```

Полный перебор нашёл целых 13 клик мощностью 9.

Вывод: для тестирования переборного алгоритма нужны графы с достаточным количеством вершин (не менее 10). Если тест генерируется, необходим контрольный заведомо работающий алгоритм, в качестве которого рекомендуется алгоритм перебора с возвратом.

4.3.1. Практикум по теме

Выполнить курсовую работу по предложенному преподавателем индивидуальному заданию (табл. П.2.3). Детальную постановку задачи взять из

рекомендованных литературных источников. Реализовать алгоритм на языке C++ с использованием объектов и возможностей *STL*.

Выбрать оптимальную структуру данных для представления графа в памяти ЭВМ. Реализовать граф как объект, а обработку — как функцию-член для него. Результат обработки может быть или не быть частью объекта, способ его представления выбирается особо.

Для объекта должны быть объявлены все вспомогательные методы (методы по умолчанию) — конструкторы, деструктор и т. п. Использование ненужных методов блокируется на уровне компиляции или выполнения.

Стек и очередь (если нужны) берутся из STL.

Интерфейс программы должен быть удобен для испытаний алгоритма. Следует предусмотреть ввод заранее заготовленных и генерацию произвольных тестовых данных.

Дополнительное требование: оценить возможный объём исходных данных для решения поставленной задачи для следующих ограничений:

— возможность вывода данных на экран;

— доступный объём памяти;

— получение решения за разумное время.

4.3.2. Содержание пояснительной записки к курсовой работе

1. Текст индивидуального задания.
2. Математическая формулировка задачи в терминах теории множеств.
3. Выбор и обоснование способа представления данных.
4. Описание алгоритма и оценка его временной сложности.
5. Набор тестов и результаты проверки алгоритма на ЭВМ.
6. Выводы.
7. Список использованных источников.
8. Приложение: исходный текст программы для ЭВМ (на машинном носителе), файлы с тестами.

4.3.3. Защита курсовой работы

К защите допускаются отчёты, имеющие все перечисленные рубрики с приложением текста программы на машинном носителе в виде, пригодном для компиляции. На защите следует продемонстрировать работу программы, обосновать решения, принятые при реализации алгоритма и выводы о его временной сложности.

Список литературы

Ахо Дж., Хопкрофт А., Ульман Дж. Структуры данных и алгоритмы. — СПб.: И. Д. Вильямс, 2001. — 382 с.

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

Готтшлинг П. Современный C++ для программистов, инженеров и учёных. — М.: И. Д. Вильямс, 2016. — 512 с.: ил.

Гэри М., Джонсон Д. Вычислительные машины и трудно решаемые задачи. — М.: Мир, 1982. — 419 с.

Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. 2-е изд. / пер. с англ. — М.: Мир, 2005. — 1296 с.: ил.

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

Липский В. Комбинаторика для программистов. — М.: Мир, 1978. — 213 с.

Макконелл Дж. Основы современных алгоритмов. 2-е. изд. — М.: Техносфера, 2004. — 368 с.

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

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

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

Страуструп Б. Язык программирования C++. Специальное издание. Пер. с англ. — М.: Изд-во Бином, 2015. — 1136 с.: ил.

Шилдт Г. C++ для начинающих. Пер. с англ. — М.: Экон Паблишерс, 2013. — 640 с.: ил.

Приложения

1. Оценка временной сложности алгоритмов

При проектировании алгоритмов, как правило, не представляет интереса точное число шагов, необходимых для решения задачи на конкретном наборе данных. Гораздо важнее знать, как будет изменяться время решения задачи T, если размер входа n растёт.

Класс алгоритмов, время работы которых растёт, по крайней мере, так же быстро, как некоторая функция $f(n)$, обозначается как $\Omega(f(n))$. Это означает, что при всех n, превышающих порог n_0, $T(n) \geq C \cdot f(n)$ для некоторого положительного числа C. Оценка времени работы снизу может представлять интерес только как теоретическая нижняя граница эффективности любого алгоритма для некоторой задачи, которую преодолеть невозможно.

Класс алгоритмов, время работы которых растёт не быстрее функции $f(n)$, обозначается $O(f(n))$, что означает существование положительных чисел n_0 и C таких, что при $n > n_0$ $T(n) \leq C \cdot f(n)$. Этот класс — важнейшая характеристика алгоритма, его *временная сложность*. По скорости роста этого времени в зависимости от размера входа алгоритмы делятся на следующие классы временной сложности:

— алгоритмы константной сложности — $T(n) \in O(1)$;

— логарифмической сложности — $T(n) \in O(\log n)$;

— линейной сложности — $T(n) \in O(n)$;

— квадратичной сложности — $T(n) \in O(n^2)$;

— кубической сложности — $T(n) \in O(n^3)$;

— полиномиальной сложности — $T(n) \in O(n^k)$, где $k = \text{const}$; $k = 0, 1, 2$ или 3 — это частные случаи классов полиномиальной сложности;

— экспоненциальной сложности — $T(n) \in O(a^n)$.

Очевидно, что классы в этом перечне упорядочены по возрастанию мощности. Так, класс $O(1)$ является подмножеством любого из остальных классов. Задача программиста — найти или разработать алгоритм класса минимально возможной мощности и реализовать его так, чтобы оценка временной сложности не ухудшилась.

Алгоритм, для которого оценки $\Omega(f(n))$ и $O(f(n))$ совпадают, называется *оптимальным*. Так, очевидно, что алгоритм, имеющий на входе некоторое множество, будет оптимальным, если его временная сложность $O(1)$. Такой алгоритм можно попытаться найти, если задача не требует рассмотреть множество целиком. Если же требуется что-то сделать с каждым элементом множества

мощностью n, оптимальный алгоритм будет иметь сложность $O(n)$. Если имеется два множества, и нужно обработать все возможные пары их элементов, можно ожидать сложности $O(n^2)$, для трёх множеств, если обрабатываются все тройки, — $O(n^3)$, и т. д.

Если же для получения результата необходимо рассмотреть все подмножества исходного множества или все перестановки его элементов — это задача на полный перебор, принадлежащая классу экспоненциальной сложности. В таких случаях говорят об отсутствии *эффективного* алгоритма.

Время работы программы часто зависит не только от мощности входных данных, но и от того, какие именно данные поступили на вход. В таких случаях делаются две оценки временной сложности:

— для самого неудобного набора данных — сложность «в худшем случае»;

— для типового набора данных — сложность «в среднем».

Тривиальные входные данные («лучший случай») обычно интереса не представляют.

Для оценки временной сложности по реализации алгоритма (тексту программы) можно руководствоваться следующими соображениями:

— операции присваивания (копирования) и проверки условия для базовых типов данных выполняются за константное время. Если при этом вызывается функция, сложность шага алгоритма определяется сложностью функции (в операциях с объектами функции могут вызываться неявно);

— сложность алгоритма, состоящего из последовательности шагов, определяется по самому сложному шагу;

— сложность выбора по условию определяется по самой сложной из альтернатив. В порядке исключения можно не принимать во внимание альтернативы, выбираемые очень редко. Можно учесть такие альтернативы, как «худший случай»;

— если какие-то шаги являются внутренней частью цикла с количеством повторений, зависящим от размера входа n, в оценку сложности циклического шага добавляется множитель n. Если же количество повторений не зависит от n, цикл игнорируется, поскольку его можно рассматривать просто как повторение некоторого количества одинаковых шагов алгоритма;

— рекурсия рассматривается как тот же цикл. Её сложность определяется как произведение сложности одного вызова функции на количество вызовов.

Пример 1. Вычислить $b = (a \in A)$, где множество A мощностью nA представлено массивом целых чисел.

Решение:

```
b = false; for (i = 0; !b && (i < nA); ++i) b |= (a == A[ i ]);
```

Временная сложность алгоритма — $O(nA)$. Если элемент a найден, алгоритм прекращает работу, выполнив от 1 до nA шагов. В среднем количество шагов будет $nA / 2$, в худшем случае ($a \notin A$) — nA.

Пример 2. Вычислить $C = A \cap B$ для множеств, представленных неупорядоченными массивами.

Решение: проверяем все возможные пары элементов двух множеств и отбираем совпадения.

```
for (i = k = 0; i < nA; ++i)
    for (j = 0; j < nB; ++j) if (A[ i ] == B[ j ]) C[ k++ ] = A[ i ];
```

Проверка на совпадение и присваивание выполняются за константное время, поэтому сложность алгоритма — $O(nA \times nB)$, или $O(n^2)$, где n — средняя мощность множеств.

Пример 3. Вычислить $D = A \cap B \cap C$.

Очевидное решение

```
for (int i = 0, k = 0; A[ i ]; ++i)
    for (int  j = 0; B[ j ]; ++j)
        for (int r = 0;  C[ r ]; ++r)
          if ((A[ i ] == B[ j ]) && (A[ i ] == C[ r ])) D[ k++ ] = A[ i ];
```

имеет временную сложность $O(n^3)$, поскольку перебираются все возможные тройки. Однако перебирать все тройки никакой необходимости нет.

Модифицируем алгоритм:

```
for (int i = 0, k = 0; A [ i ]; ++i)
    for (int j = 0; B[ j ]; ++j)
        if (A[ i ] == B[ j ]) for (int r = 0; C[ r ]; ++r)
            if (A[ i ] == C[ r ]) D[ k++ ] = A[ i ];
```

В алгоритме по-прежнему три вложенных цикла, но внутренний цикл теперь зависит от условия $A[\,i\,] == B[\,j\,]$, которое проверяется n^2 раз, но удовлетворяется не более чем n раз, т. е. рассматриваются только такие тройки, у которых первые два элемента совпадают. Проверка $A[\,i\,] == C[\,r\,]$ выполняется, таким образом, не более n^2 раз, и общая сложность алгоритма — $O(n^2)$.

Пример 4. Вычислить $C = A \cap B$ для множеств, представленных строками

символов.

Решение:

```
for (i = k = 0; i < strlen(A); ++i)
  for (j = 0; j < strlen(B); ++j) if (A[ i ] == B[ j ]) C[ k++ ] = A[ i ];
```

По аналогии с примером 2 можно оценить сложность как $O(n^2)$. Однако это неверно, так как в обоих циклах по n раз вычисляется функция определения длины строки *strlen*( ), которая подсчитывает в строке количество символов до ближайшего нуля перебором, т. е. имеет линейную сложность. Вычисление этой функции — один из шагов внутренней части обоих циклов. Таким образом, внутренняя часть вложенного цикла состоит из трёх шагов, двух константных (проверка и присваивание) и линейного (вычисление функции). С учётом n повторений сложность всего цикла — $O(n^2)$. Внешний цикл добавляет сюда ещё шаг вычисления функции сложностью $O(n)$. Сложность его внутренней части — $O(n^2)$, а всего алгоритма — $O(n^3)$! Это цена экономии двух целых переменных. На самом деле нужно вычислить пределы заранее $nA = strlen(A)$, $nB = strlen(B)$, а затем использовать алгоритм из примера 2. Альтернатива: если известно, что массивы символов ограничены нулём, это можно использовать, как показано в примере 3. Подумайте, как ограничить нулём результат!

2. Задания к зачётным работам

Таблица П.2.1

Индивидуальные задания к темам «Множества» и «Классы»

№ варианта	Универсум	Что надо вычислить
1	Десятичные цифры	Множество, содержащее цифры, общие для множеств *A* и *B*, а также все цифры из множеств *C* и *D*
2	Прописные латинские буквы	Множество, содержащее все символы из множества *A*, за исключением символов, содержащихся в *B* или *C*, а также все символы множества *D*
3	Шестнадцатеричные цифры	Множество, содержащее цифры, имеющиеся в любом из множеств *A*, *B*, *C*, и *D*
4	Строчные латинские буквы	Множество, содержащее все буквы, общие для множеств *A* и *B*, за исключением букв, содержащихся в *C*, а также все буквы из *D*
5	Десятичные цифры	Множество, содержащее все цифры множества *A*, за исключением цифр из *B* и *C*, а также все цифры из *D*
6	Русские буквы	Множество, содержащее все буквы множеств *A* и *B*, за исключением букв, содержащихся в *C*, а также все буквы из *D*
7	Шестнадцатеричные цифры	Множество, содержащее цифры, имеющиеся в каждом из множеств *A*, *B*, *C*, а также все цифры из *D*
8	Латинские буквы	Множество, содержащее буквы, имеющиеся во множестве *A*, но не являющиеся общими для *B* и *C*, и все буквы из *D*
9	Десятичные цифры	Множество, содержащее все цифры из *A*, все цифры, общие для множеств *B* и *C*, а также все цифры из *D*
10	Прописные латинские буквы	Множество, содержащее буквы, имеющиеся в *A* или *B*, но отсутствующие в *C*, кроме того, обязательно встречающиеся также и в *D*
11	Шестнадцатеричные цифры	Множество, содержащее цифры, имеющиеся во множестве *A* или *C*, но отсутствующие в *B* или *D*
12	Строчные латинские буквы	Множество, содержащее буквы, имеющиеся в любом из множеств *A*, *B*, *C*, но отсутствующие в *D*
13	Десятичные цифры	Множество, содержащее цифры, общие для множеств *A* и *B*, но не встречающиеся ни в *C*, ни в *D*
14	Прописные русские буквы	Множество, содержащее все буквы множества *A*, которых нет во множествах *B*, *C* или *D*
15	Шестнадцатеричные цифры	Множество, содержащее цифры, имеющиеся в *A* или *B*, но отсутствующие и в *C*, и в *D*
16	Строчные русские буквы	Множество, содержащее буквы, общие для множеств *A*, *B*, *C* и не встречающиеся *D*
17	Десятичные цифры	Множество, содержащее цифры из *A*, не являющиеся общими для множеств *B* и *C* и не встречающиеся в *D*
18	Русские буквы	Множество, содержащее буквы, имеющиеся в *A* или общие для *B* и *C*, но не встречающиеся в *D*
19	Шестнадцатеричные цифры	Множество, содержащее все цифры, общие для *A* и *B*, а также все цифры, являющиеся общими для *C* и *D*

Продолжение табл. П.2.1

№ варианта	Универсум	Что надо вычислить
20	Латинские буквы	Множество, содержащее буквы из множества *A*, не содержащиеся в *B*, а также все общие для *C* и *D*
21	Десятичные цифры	Множество, содержащее цифры, имеющиеся в *A* или *B* или являющиеся общими для *C* и *D*
22	Прописные русские буквы	Множество, содержащее буквы, общие для *A* и *B*, но не являющиеся общими для *C* и *D*
23	Шестнадцатеричные цифры	Множество, содержащее цифры из *A*, не встречающиеся в *B* и не являющиеся общими для *C* и *D*
24	Строчные русские буквы	Множество, содержащее все буквы из *A* и *B*, но не содержащее букв, являющихся общими для *C* и *D*
25	Десятичные цифры	Множество, содержащее цифры, имеющиеся в каждом из множеств *A*, *B*, *C*, и *D*
26	Прописные латинские буквы	Множество, содержащее все буквы из *A*, не являющиеся общими для *B*, *C* и *D*
27	Шестнадцатеричные цифры	Множество, содержащее цифры, имеющиеся в *A*, и все цифры, являющиеся общими для *B*, *C* и *D*
28	Строчные латинские буквы	Множество, содержащее буквы, общие для множества *A* и любого из множеств *B*, *C* и *D*
29	Десятичные цифры	Множество, содержащее цифры, общие для цифр из *A*, не встречающихся в *B*, с одной стороны, и любых цифр из *C* и *D* – с другой
30	Знаки операций C++	Множество, содержащего операции, имеющиеся в *A*, также одновременно в *B* и *C*, и все операции из *D*
31	Шестнадцатеричные цифры	Множество, содержащее цифры, имеющиеся во множестве *A* или *C*, но отсутствующие в *B* или *D*
32	Русские буквы	Множество, содержащее буквы, имеющиеся в любом из множеств *A*, *B*, *C*, но отсутствующие в *D*
33	Десятичные цифры	Множество, содержащее цифры, общие для множеств *A* и *B*, но не встречающиеся ни в *C*, ни в *D*
34	Латинские буквы	Множество, содержащее все буквы множества *A*, которых нет во множествах *B*, *C* или *D*
35	Шестнадцатеричные цифры	Множество, содержащее цифры, имеющиеся в *A* или *B*, но отсутствующие и в *C*, и в *D*
36	Строчные русские буквы	Множество, содержащее буквы, общие для множеств *A*, *B*, *C* и не встречающиеся в *D*

Окончание табл. П.2.1

№ варианта	Универсум	Что надо вычислить
37	Десятичные цифры	Множество, содержащее цифры из *A*, не являющиеся общими для множеств *B* и *C* и не встречающиеся в *D*
38	Прописные русские буквы	Множество, содержащее буквы, имеющиеся в *A* или общие для *B* и *C*, но не встречающиеся в *D*
39	Шестнадцатеричные цифры	Множество, содержащее все цифры, общие для *A* и *B*, а также все цифры, являющиеся общими для *C* и *D*
40	Латинские буквы	Множество, содержащее буквы из множества *A*, не содержащиеся в *B*, а также все общие для *C* и *D*
41	Десятичные цифры	Множество, содержащее цифры, имеющиеся в *A* или *B* или являющиеся общими для *C* и *D*
42	Русские буквы	Множество, содержащее буквы, общие для *A* и *B*, но не являющиеся общими для *C* и *D*
43	Шестнадцатеричные цифры	Множество, содержащее цифры из *A*, не встречающиеся в *B* и не являющиеся общими для *C* и *D*
44	Латинские буквы	Множество, содержащее все буквы из *A* и *B*, но не содержащее букв, являющихся общими для *C* и *D*
45	Десятичные цифры	Множество, содержащее цифры, имеющиеся в каждом из множеств *A*, *B*, *C* и *D*
46	Строчные латинские буквы	Множество, содержащее все буквы из *A*, не являющиеся общими для *B*, *C* и *D*
47	Шестнадцатеричные цифры	Множество, содержащее цифры, имеющиеся в *A* и все цифры, являющиеся общими для *B*, *C* и *D*
48	Прописные латинские буквы	Множество, содержащее буквы, общие для множества *A* и любого из множеств *B*, *C* и *D*
49	Десятичные цифры	Множество, содержащее цифры, общие для цифр из *A*, не встречающихся в *B*, с одной стороны, и любые цифры из *C* и *D* — с другой
50	Романы о Мегрэ	Множество, содержащее романы, имеющиеся в списке *A*, также одновременно в *B* и *C*, и все романы из *D*

Таблица П.2.2

Индивидуальные задания к теме «Деревья»

№ варианта	Вид дерева	Разметка	Способ обхода	Что надо вычислить
1	Двоичное	Обратная	В глубину	Высоту левого поддерева для корня
2	Двоичное	Прямая	В ширину	Количество листьев
3	Троичное	Обратная	Внутренний	Количество вершин, имеющих хотя бы одного потомка

Продолжение табл. П.2.2

№ варианта	Вид дерева	Разметка	Способ обхода	Что надо вычислить
4	Троичное	Прямая	В ширину	Количество вершин, имеющих предков
5	Двоичное	Симметричная	В ширину	Количество вершин, имеющих не более одного потомка
6	Двоичное	Обратная	Внутренний	Количество вершин на глубине больше 2
7	Троичное	Ширинная	В глубину	Количество вершин, имеющих ровно одного потомка
8	Троичное	Обратная	В ширину	Количество вершин, имеющих хотя бы одного потомка
9	Двоичное	Прямая	Внутренний	Количество вершин на уровне не выше 2
10	Двоичное	Симметричная	В глубину	Количество вершин, имеющих не более одного потомка
11	Троичное	Прямая	В ширину	Высоту среднего поддерева для корня
12	Троичное	Обратная	Внутренний	Количество правых листьев
13	Двоичное	Симметричная	В глубину	Количество вершин на глубине не более 2
14	Двоичное	Симметричная	В глубину	Количество потомков у каждой из вершин
15	Троичное	Прямая	Внутренний	Количество вершин, имеющих не более двух потомков
16	Троичное	Обратная	Внутренний	Высоту среднего поддерева для корня
17	Двоичное	Обратная	В глубину	Количество левых листьев
18	Двоичное	Ширинная	Внутренний	Количество вершин на самом нижнем уровне
19	Троичное	Глубинная	Внутренний	Количество вершин не на самом нижнем уровне
20	Троичное	Обратная	В глубину	Количество вершин, имеющих не более трёх потомков
21	Двоичное	Прямая	Внутренний	Высоту правого поддерева для корня
22	Двоичное	Обратная	Внутренний	Количество листьев на самом нижнем уровне, имеющем листья
23	Троичное	Симметричная	В глубину	Количество средних листьев
24	Троичное	Прямая	В ширину	Количество предков у каждой из вершин
25	Двоичное	Обратная	Внутренний	Количество вершин, имеющих не более двух потомков
26	Троичное	Симметричная	В глубину	Количество листьев не на самом нижнем уровне, имеющем листья
27	Троичное	Прямая	В ширину	Высоту среднего поддерева для корня
28	Двоичное	Обратная	Внутренний	Количество предков у каждой из вершин

Окончание табл. П.2.2

№ варианта	Вид дерева	Разметка	Способ обхода	Что надо вычислить
29	Двоичное	Обратная	В глубину	Количество вершин на глубине не более 3
30	Троичное	Симметричная	В ширину	Количество вершин, имеющих не более двух потомков
31	Двоичное	Симметричная	В глубину	Количество вершин на глубине не более 2
32	Двоичное	Симметричная	В ширину	Количество потомков у каждой из вершин
33	Троичное	Прямая	В ширину	Высоту самого мощного поддерева для корня
34	Троичное	Обратная	Внутренний	Количество правых листьев
35	Двоичное	Обратная	Прямой	Количество левых листьев
36	Двоичное	Симметричная	В ширину	Количество вершин на самом нижнем уровне
37	Троичное	Глубинная	Внутренний	Количество вершин, имеющих не более двух потомков
38	Троичное	Обратная	Внутренний	Высоту левого поддерева для корня
39	Двоичное	Прямая	В ширину	Высоту правого поддерева для корня
40	Двоичное	Обратная	Внутренний	Количество листьев на самом нижнем уровне
41	Троичное	Глубинная	Внутренний	Количество вершин на самом нижнем уровне
42	Троичное	Обратная	В глубину	Количество вершин, имеющих не более трёх потомков
43	Двоичное	Обратная	Внутренний	Количество вершин, имеющих не менее одного потомка
44	Троичное	Симметричная	В глубину	Количество листьев не на самом нижнем уровне
45	Троичное	Симметричная	В глубину	Количество средних листьев
46	Троичное	Прямая	В ширину	Количество предков у каждой из вершин
47	Двоичное	Обратная	В глубину	Количество вершин на глубине не более 3
48	Троичное	Симметричная	В ширину	Количество вершин, имеющих не менее двух потомков
49	Троичное	Прямая	В ширину	Высоту среднего поддерева для корня
50	Двоичное	Обратная	В глубину	Количество вершин на глубине не более 4

Таблица П.2.3

Индивидуальные задания к теме «Графы»

№ варианта	Алгоритм для исследования
1	Получение множества двудольных компонент неориентированного графа
2	Проверка ориентированного графа на ацикличность
3	Поиск кратчайшего пути между заданной парой вершин в *неориентированном* графе с нагруженными рёбрами
4	Обнаружение всех элементарных циклов ориентированного графа.
5	Подсчёт расстояний от произвольной вершины до всех остальных вершин в ориентированном *ненагруженном* графе
6	Получение множества компонент сильной связности в ориентированном графе
7	Поиск изоморфного неориентированного подграфа
8	Отыскание кратчайшего пути между произвольной парой вершин в ориентированном графе с нагруженными рёбрами (веса рёбер неотрицательные)
9	Отыскание клики наибольшей мощности в неориентированном графе
10	Отыскание максимального паросочетания в произвольном неориентированном графе
11	Построение полного множества циклов для ориентированного графа
12	Стягивающее дерево наименьшей стоимости неориентированного графа с нагруженными рёбрами (алгоритм Краскала)
13	Вычисление матрицы расстояний между всеми парами вершин в ориентированном графе с нагруженными рёбрами
14	Поиск изоморфного ориентированного подграфа
15	Построение эйлерова пути в неориентированном графе
16	Раскраска минимальным числом цветов вершин неориентированного графа с соблюдением условия: никакое ребро не соединяет вершины одного цвета
17	Проверка наличия цикла отрицательной стоимости в ориентированном графе с нагруженными рёбрами
18	Отыскание минимального вершинного покрытия неориентированного графа
19	Стягивающее дерево наименьшей стоимости неориентированного графа с нагруженными рёбрами (алгоритм Прима)
20	Отыскание гамильтонова цикла в неориентированном графе
21	Вычисление матрицы расстояний между всеми парами вершин неориентированного графа с нагруженными рёбрами
22	Построение глубинного стягивающего леса для произвольного ориентированного графа
23	Построение фундаментального множества циклов в неориентированном графе
24	Построение эйлерова цикла в ориентированном графе
25	Поразрядная сортировка произвольной последовательности целых чисел
26	Построение ширинного стягивающего леса для неориентированного графа

Окончание табл. П.2.3

№ варианта	Алгоритм для исследования
27	Построение трансверсали максимальной мощности для произвольного набора частично пересекающихся подмножеств
28	Построение глубинного стягивающего леса для неориентированного графа
29	Топологическая сортировка вершин ориентированного ациклического графа
30	Получение ширинного стягивающего леса для ориентированного графа
31	Получение множества компонент двусвязности для произвольного неориентированного графа
32	Построение максимального паросочетания в двудольном неориентированном графе
33	Получение минимального вершинного покрытия для двудольного неориентированного графа
34	Проверка на изоморфизм произвольных корневых деревьев
35	Отыскание кратчайшего пути между заданной парой вершин в произвольном ориентированном графе с нагруженными рёбрами
36	Отыскание кратчайшего пути между заданной парой вершин в произвольном ациклическом ориентированном графе с нагруженными рёбрами
37	Отыскание элементарного кратчайшего пути через все вершины произвольного неориентированного графа с нагруженными рёбрами (задача коммивояжёра)
38	Пирамидальная сортировка произвольной последовательности целых чисел
39	Сортировка слиянием для произвольной последовательности целых чисел
40	Сортировки произвольного набора цепочек (строк) из букв латинского алфавита
41	Оптимальная упаковка рюкзака заданного объёма грузами, объёмы которых заданы произвольной последовательностью целых чисел
42	Оптимальная раскладка по ящикам заданного объёма набора грузов, объёмы которых заданы произвольной последовательностью целых чисел
43	Нахождение в неориентированном графе компоненты, изоморфной заданному графу
44	Нахождение минимального рёберного покрытия неориентированного графа
45	Нахождение максимального независимого множества вершин неориентированного графа
46	Нахождение раскраски вершин неориентированного графа минимальным числом цветов
47	Нахождение минимального множества вершин неориентированного графа, разрезающих циклы
48	Нахождение минимального множества рёбер неориентированного графа, разрезающих циклы
49	Проверка неориентированного графа на планарность
50	Построение множества точек сочленения неориентированного графа

СОДЕРЖАНИЕ

Колинько Павел Георгиевич

Пользовательские структуры данных

Учебно-методическое пособие
по дисциплине «Алгоритмы и структуры данных, часть 1»

Издание публикуется в авторской редакции

Подписано в печать 27.08.2024 . Формат 60×84 1/16.
Бумага офсетная. Печать офсетная. Печ. л. 4,0.
Гарнитура «Times New Roman». Тираж 100 экз.